4. I Thread: motivazioni

- *Questo capitolo va studiato solo dopo il capitolo 9 sulla memoria centrale (N.B.: ai fini del programma del corso e dell'esame fanno riferimento queste slide, il cui contenuto è molto semplificato rispetto al capitolo 4 del libro di testo)*
- Consideriamo due processi che **devono lavorare sugli stessi dati**. Come possono fare, se ogni processo ha la propria area dati (ossia, gli spazi di indirizzamento dei due processi sono separati)?
  - I due processi possono richiedere al SO un'area di memoria condivisa, o scambiarsi i dati usando messaggi
  - i dati possono essere tenuti in un file che viene acceduto a turno dai due processi.

4. I Thread: motivazioni

- Sarebbe comodo poter avere processi in grado di lavorare sugli stessi dati, senza usare meccanismi espliciti di condivisione/comunicazione, e senza usare file, che sono memorizzati su un supporto relativamente lento.
  Ad esempio, in un editor di testo:
  - un processo gestisce l'input e i comandi di formattazione dell'utente
  - un altro processo esegue il controllo automatico di errore
- I due processi dovrebbero lavorare sullo stesso testo, la cui copia corrente è mantenuta in MP, ma come fanno i due processi a lavorare sulla stessa copia, se ogni processo ha un diverso spazio di indirizzamento?

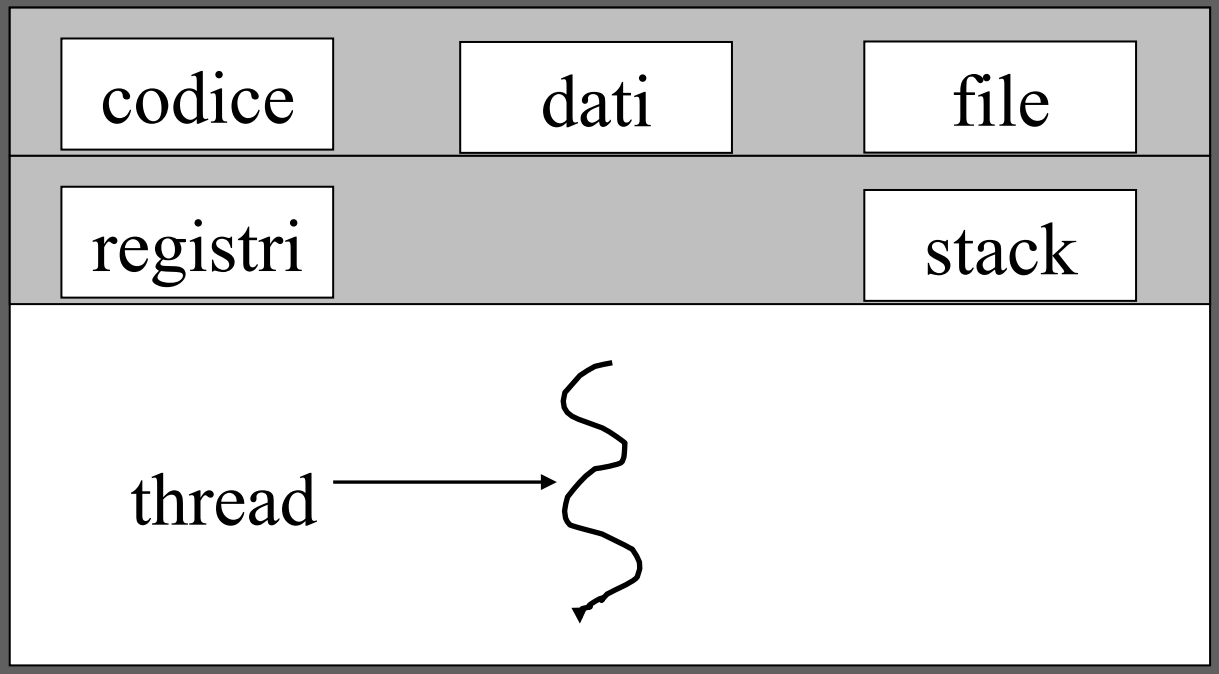
4. I Thread: motivazioni

- Non solo: nel context switch tra processi, occorre disattivare le aree dati e di codice del processo uscente, e attivare quelle del processo entrante.
- Inoltre, le cache fisiche della CPU contengono i dati del processo uscente, e il processo che entra in esecuzione genera inizialmente molti miss cache.
- Se due (o più) processi potessero condividere dati e codice, il context switch fra di loro sarebbe molto meno oneroso.
- Da queste considerazioni nasce il concetto di **thread:** un gruppo di **peer thread** è un insieme di "processi" che condividono lo spazio di indirizzamento (codice e dati)

chiarific.

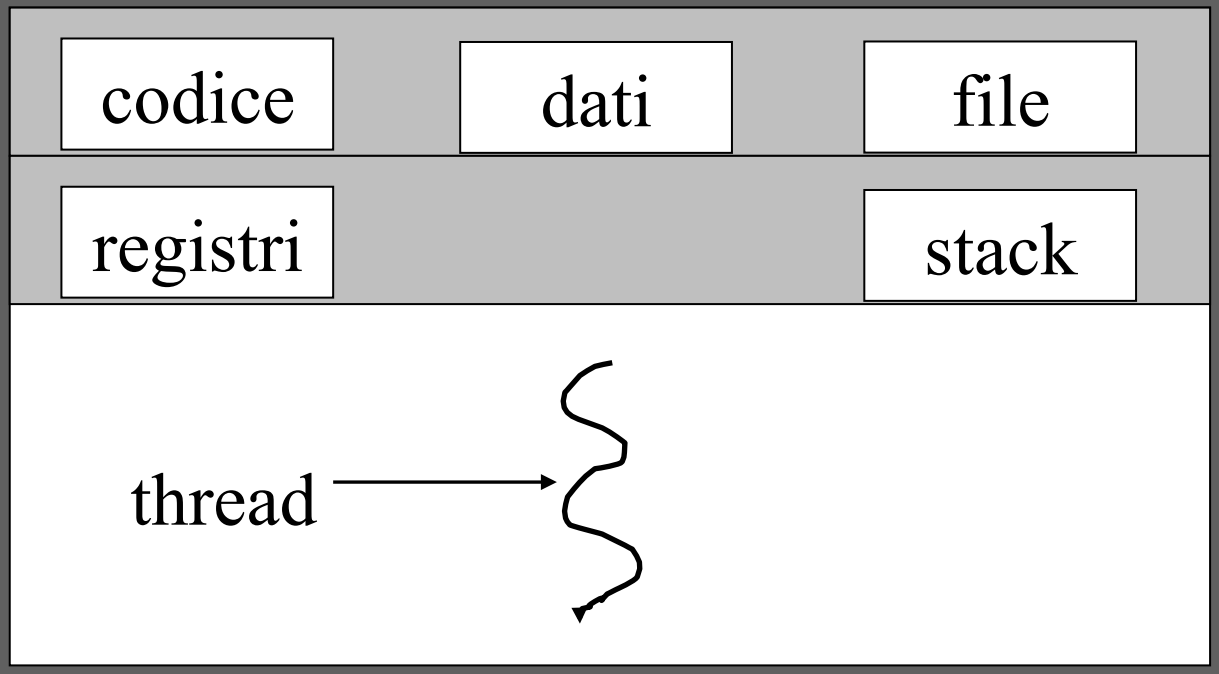
4. I Thread

- Precisiamo meglio la terminologia usata: un processo P (per come li abbiamo studiati fino ad ora) è contraddistinto da un unico **Thread (filo)** di computazione: la sequenza di istruzioni eseguite, che ovviamente può cambiare da un'esecuzione all'altra, se cambiano, ad esempio, i dati di input. (Fig. 4.1a).



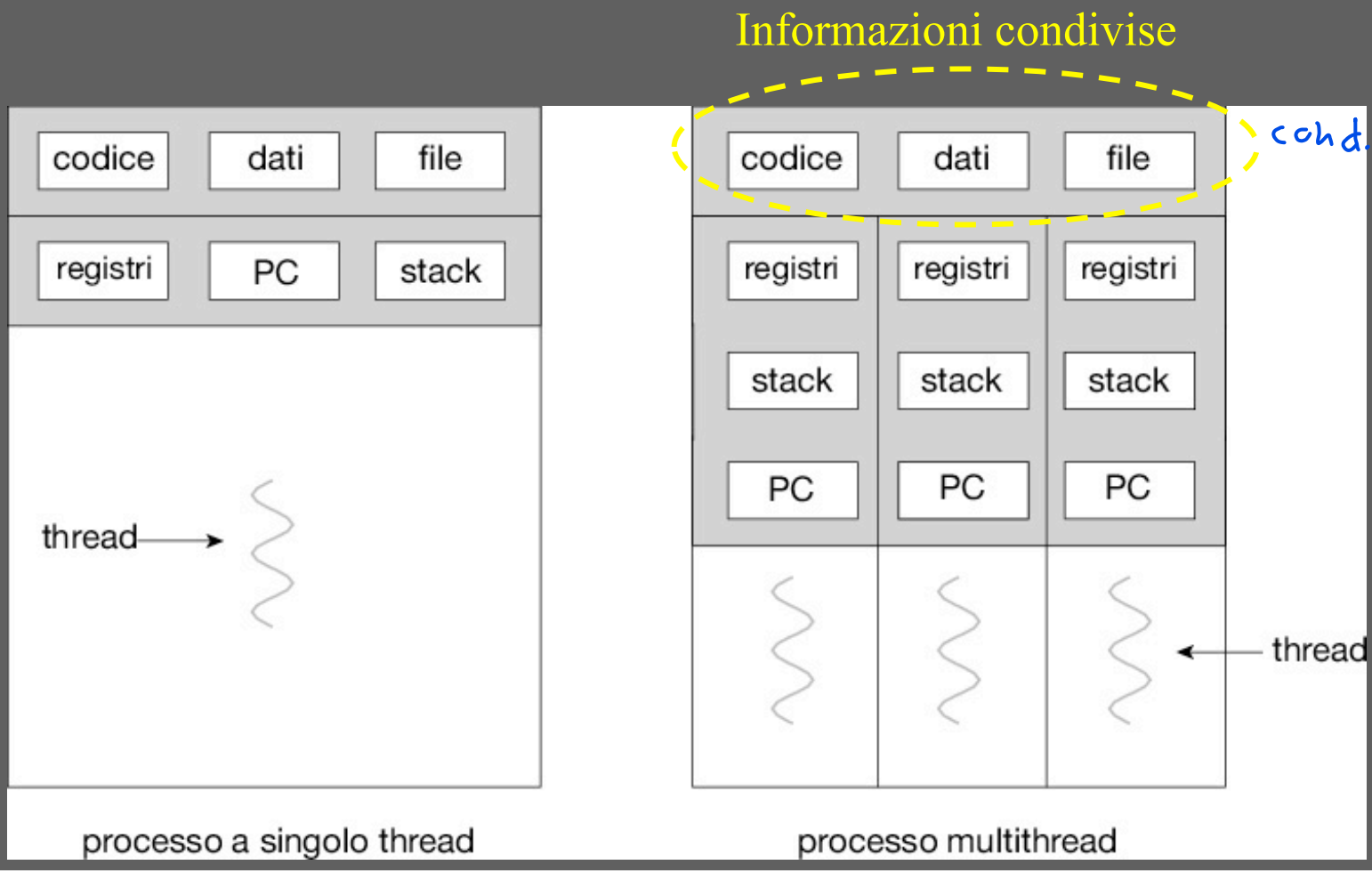
4. I Thread

- Nessun altro processo ha accesso allo spazio di indirizzamento logico di P, dunque in particolare nessun processo utente all'infuori di P può accedere alle aree dati di P. Altri processi possono eseguire lo stesso codice di P, **ma in uno spazio di indirizzamento logico separato da quello di P, e quindi con dati diversi** (Fig. 4.1a).



4. I Thread

- Un processo **Multi-Thread (o Multi-Threaded)** è invece composto da più thread di computazione, detti **peer thread**. Un processo multi-threaded è anche detto **task** (Fig. 4.1).



4. I Thread

- Ad ogni peer thread è assegnata l'esecuzione di codice che, di solito, è diverso da quello degli altri suoi peer, e quindi:
- Ogni thread ha un suo stato della computazione, fatto da: **Program Counter e registri della CPU + stack**
- Ma un insieme di peer thread condivide il codice in esecuzione e soprattutto: **le AREE DATI**
- Quindi, il codice conterrà l'indicazione di quale peer thread deve eseguire quale parte del codice (proprio come in un programma che usa la fork possiamo specificare quale porzione di codice deve eseguire il processo padre e quale il processo figlio).

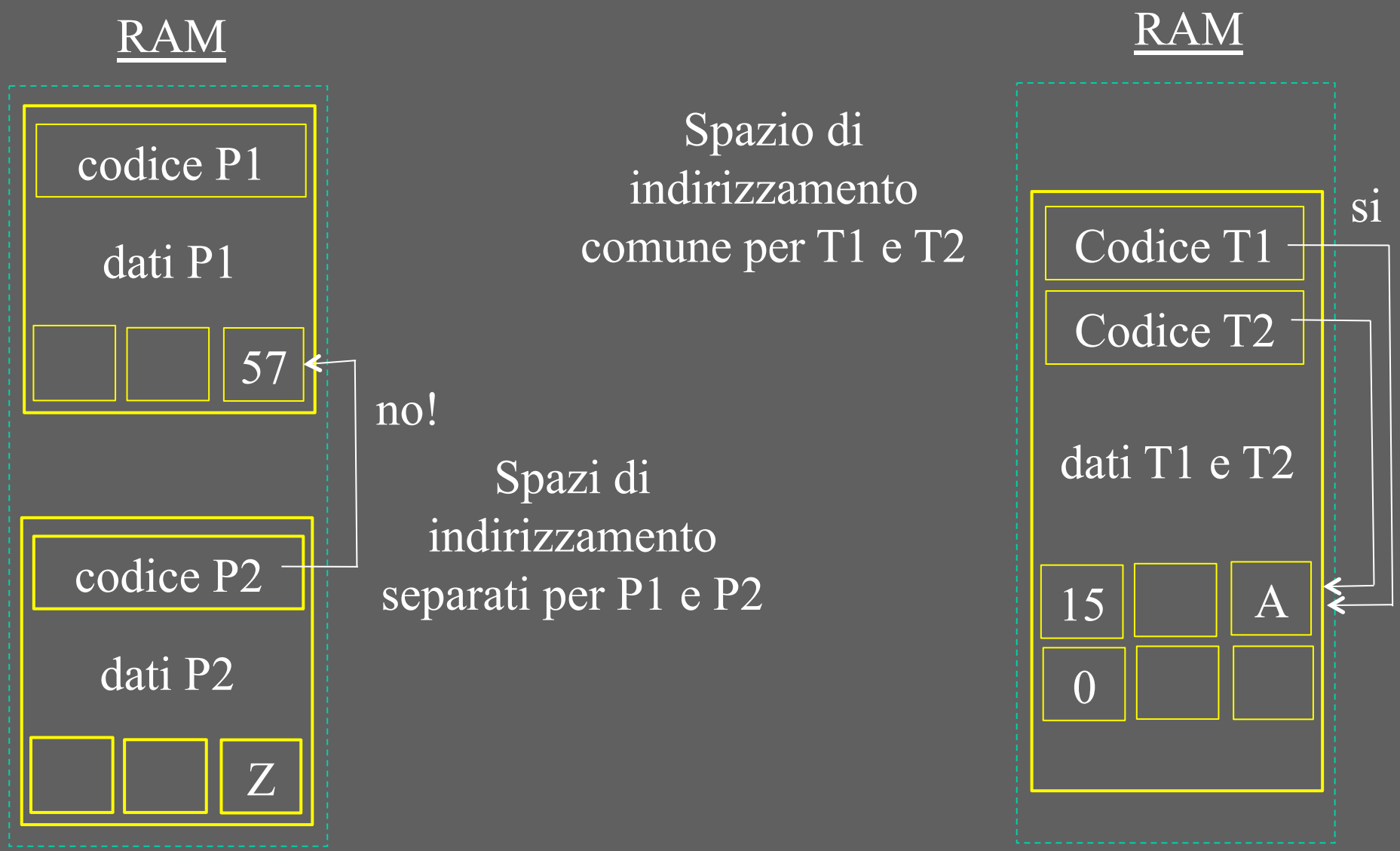
4. I Thread

- Il context switch avviene anche tra ognuno dei peer thread di un processo multi-thread, in modo che ciascuno possa portare avanti l'esecuzione del codice assegnatogli (per ora assumiamo sempre una CPU single core)

- Ma il context switch fra peer thread richiede il salvataggio e il ripristino **solo del Program Counter, dei registri della CPU, e dello stack** (diversi per ogni thread)

- Il codice e i dati, ossia lo spazio di indirizzamento logico, è lo stesso per tutti i peer thread, e non deve essere cambiato. In altre parole **al context switch tra due peer thread non è necessario cambiare la tabella delle pagine del processo multi-thread a cui appartengono quei peer thread.**

4. I Thread


RAM
codice P1
dati P1
57
codice P2
dati P2
Z
no!
Spazi di
indirizzamento
separati per P1 e P2
Spazio di
indirizzamento
comune per T1 e T2
RAM
Codice T1
Codice T2
dati T1 e T2
15
A
0
si

4. I Thread

- Il context switch tra processi è dunque molto più oneroso per il SO del context switch tra peer thread, perché nel primo caso sono coinvolte più informazioni da cambiare.
- Inoltre, a causa del sistema di caching, il context switch tra processi genera inizialmente molti più cache miss del context switch tra peer thread.
- Per questa ragione, i normali processi (quelli di cui abbiamo parlato fino ad ora) sono spesso chiamati: **heavy-weight process (HWP)**
- Mentre per un peer-thread è usato il termine: **light-weight process (LWP)**

4. I Thread

- In un task nuovi peer thread vengono creati con opportune system call, e a ciascun thread si può assegnare del codice da eseguire: un meccanismo simile a quello visto con fork ed exec (in Linux un nuovo thread viene creato con la system call *clone*, in Windows con *CreateThread*)

- Altre system call permetteranno ai peer-thread di sincronizzarsi fra di loro (proprio come i processi possono sincronizzarsi fra di loro usando i semafori), cosa fondamentale per un accesso ordinato ai dati, che sono sempre condivisi da tutti i peer-thread

- Molti linguaggi moderni poi (ad esempio Java), forniscono varie primitive per scrivere programmi multi-threaded

4. Scheduling dei Thread

- In tutti i SO moderni si implementa lo **scheduling dei thread a livello kernel**: il SO mantiene strutture dati per gestire sia i normali processi che tutti i peer thread di un task multi-threaded

- Quando un thread si blocca volontariamente, o termina il suo quanto di tempo, è il SO che assegna la CPU:
  - a un altro peer-thread dello stesso task
  - a uno dei peer-thread di un altro task
  - a un processo normale

Vantaggi nell'uso dei thread

- **Efficienza:** in Solaris, creare un nuovo LWP richiede circa 30 volte meno tempo che creare un HWP. Il context switch tra peer thread richiede 5 volte meno tempo del context switch tra processi

- **Condivisione di dati e risorse:** più thread possono lavorare su dati condivisi in maniera efficiente (ma devono comunque sincronizzarsi in maniera adeguata)

- **Architetture multi-core:** i thread sono particolarmente adatti per girare su processori multi-core, e ancora di più sulle architetture **multithreaded**.

Thread e architetture multi-core

- In un processore single core, tutti i peer thread di un task si alternano in esecuzione esattamente come un insieme di processi (figura 4.3)

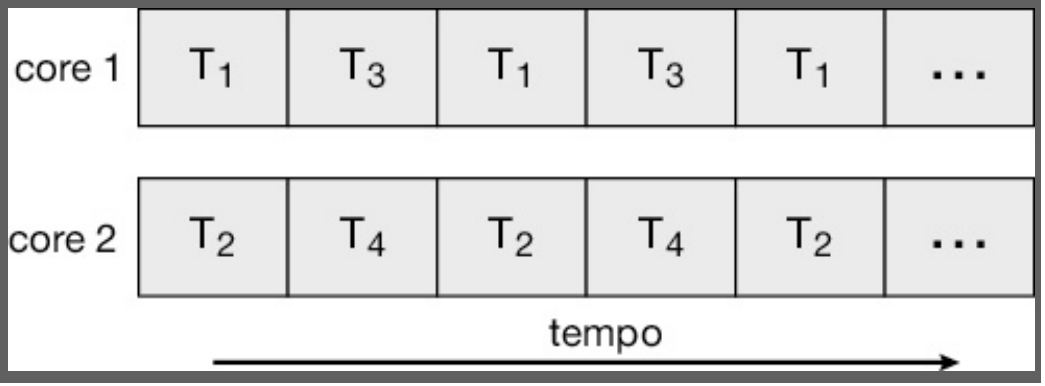
- E come abbiamo già osservato, il context switch tra i vari peer thread sarà meno pesante del context switch tra normali processi.

- Ma il context switch con un altro processo sarà comunque oneroso, e le prestazioni degraderanno anche a causa dei miss cache inizialmente generati dal processo entrante

Thread e architetture multi-core

- Le architetture multi-core invece sono particolarmente adatte a gestire task multi-threaded. Consideriamo un sistema dual-core in cui sono attivi due task multi-threaded: se si assegna un task a ciascun core, nei due core si verificheranno solo context switch tra peer-thread.

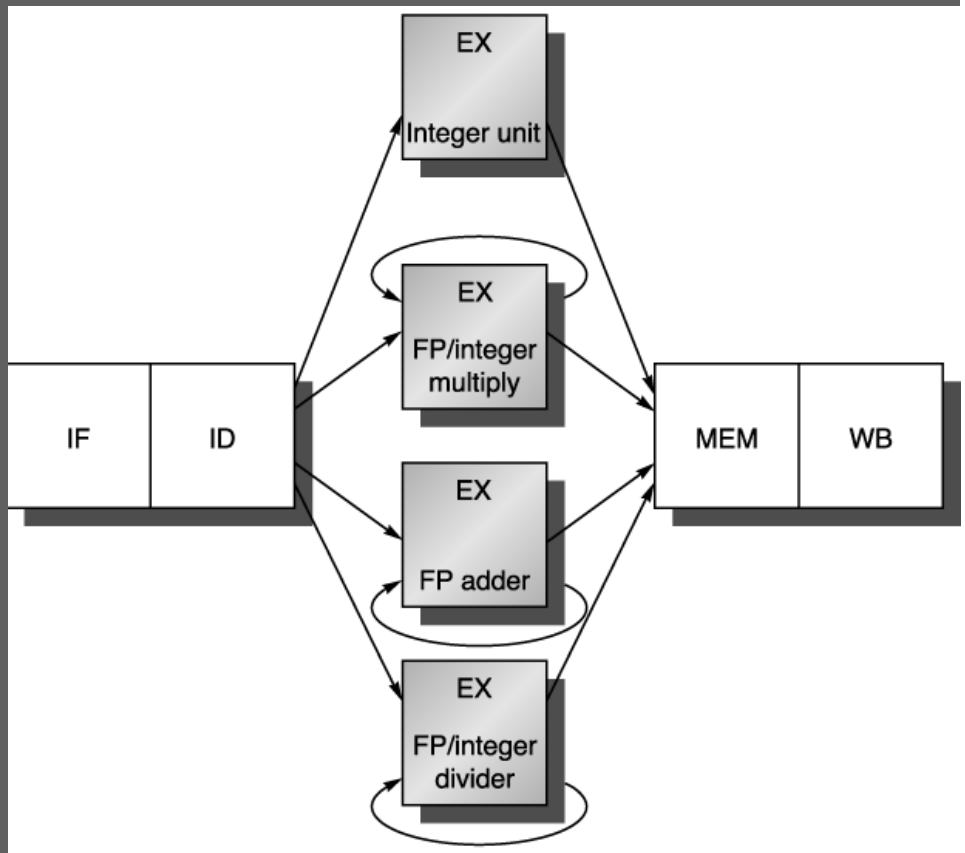
- In realtà di solito il numero totale di task multi-threaded (e quindi anche il numero di peer thread) attivi nel sistema è molto maggiore del numero di core disponibili, e il SO può dover distribuire i peer thread di uno stesso task su core diversi per bilanciare al meglio il carico (fig. 4.4: un task con 4 peer thread eseguito su due core)



CPU/CORE multithreaded

- Ma i processori moderni offrono all'esecuzione dei thread una opportunità in più. Infatti, ogni singolo core di un processore multi-core è in grado di eseguire in parallelo fino a 4 o 5 istruzioni del programma in esecuzione, una tecnica chiamata **multiple issue**.

- Per poter eseguire in parallelo più istruzioni, ogni singolo core deve essere dotato di più unità funzionali: ALU e unità floating point. Si parla di processori con **architettura superscalare**. (in realtà è ogni singolo core ad avere una struttura superscalare e a poter eseguire in parallelo fino a 4 o 5 istruzioni)



CPU/CORE multithreaded

- Tuttavia, ogni singolo core, superscalare e multiple issue, non riesce sempre ad avviare in parallelo il massimo numero di istruzioni possibile. Infatti, in un programma le istruzioni non sono indipendenti fra loro, ma in molti casi una istruzione B deve usare il risultato di una istruzione A. Di conseguenza A e B non possono essere eseguite in parallelo: B deve essere eseguita dopo A.

- Il risultato di queste dipendenze fra le istruzioni di un programma è che spesso il numero di istruzioni lanciate in parallelo è inferiore al massimo consentito da una certa architettura, e alcune unità funzionali rimangono inutilizzate (ad esempio, si useranno solo 2 di 4 ALU disponibili).

- Tuttavia, le istruzioni appartenenti ai diversi peer thread di un task sono quasi certamente indipendenti (infatti appartengono a thread diversi, anche se vedono lo stesso spazio di indirizzamento)

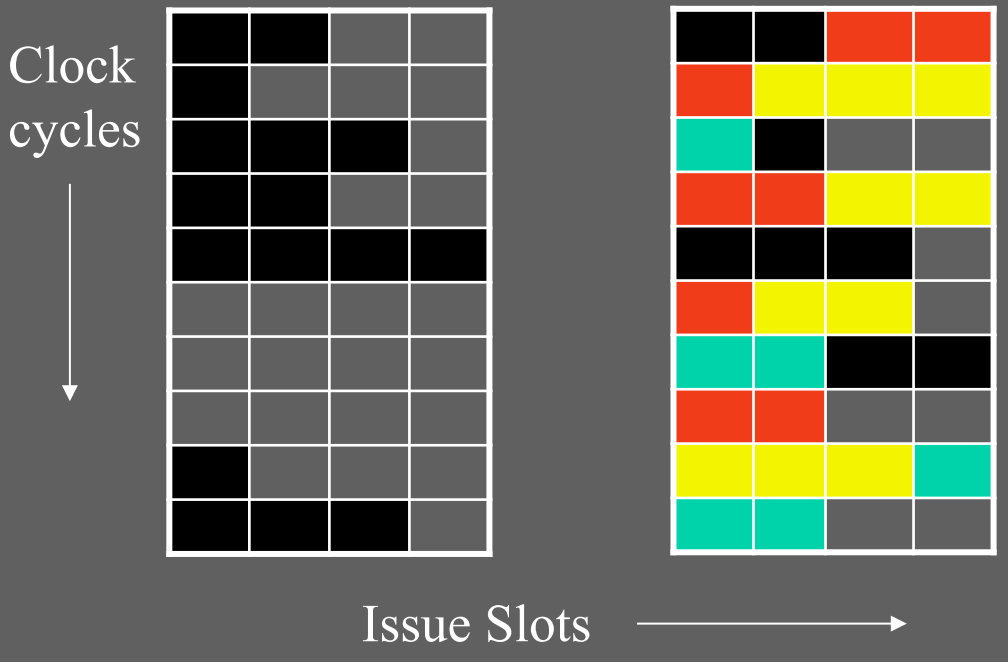
CPU/CORE multithreaded

- Per sfruttare questa caratteristica ogni core di una CPU moderna è **multi-threaded:** può eseguire in parallelo istruzioni appartenenti a peer thread diversi, aumentando così la velocità di esecuzione dei programmi. È la tecnica del **Simultaneous Multi-Threading (SMT)**

Ad ogni ciclo di clock inizia l'esecuzione di un nuovo gruppo di istruzioni. Se si possono eseguire in parallelo istruzioni che appartengono a diversi peer thread (nella figura di destra rappresentate da colori diversi), la produttività di ciascun core della CPU aumenta

core superscalare: fino a 4 istruzioni avviate per ciclo di clock

Core superscalare con Simultaneous Multi-Threading

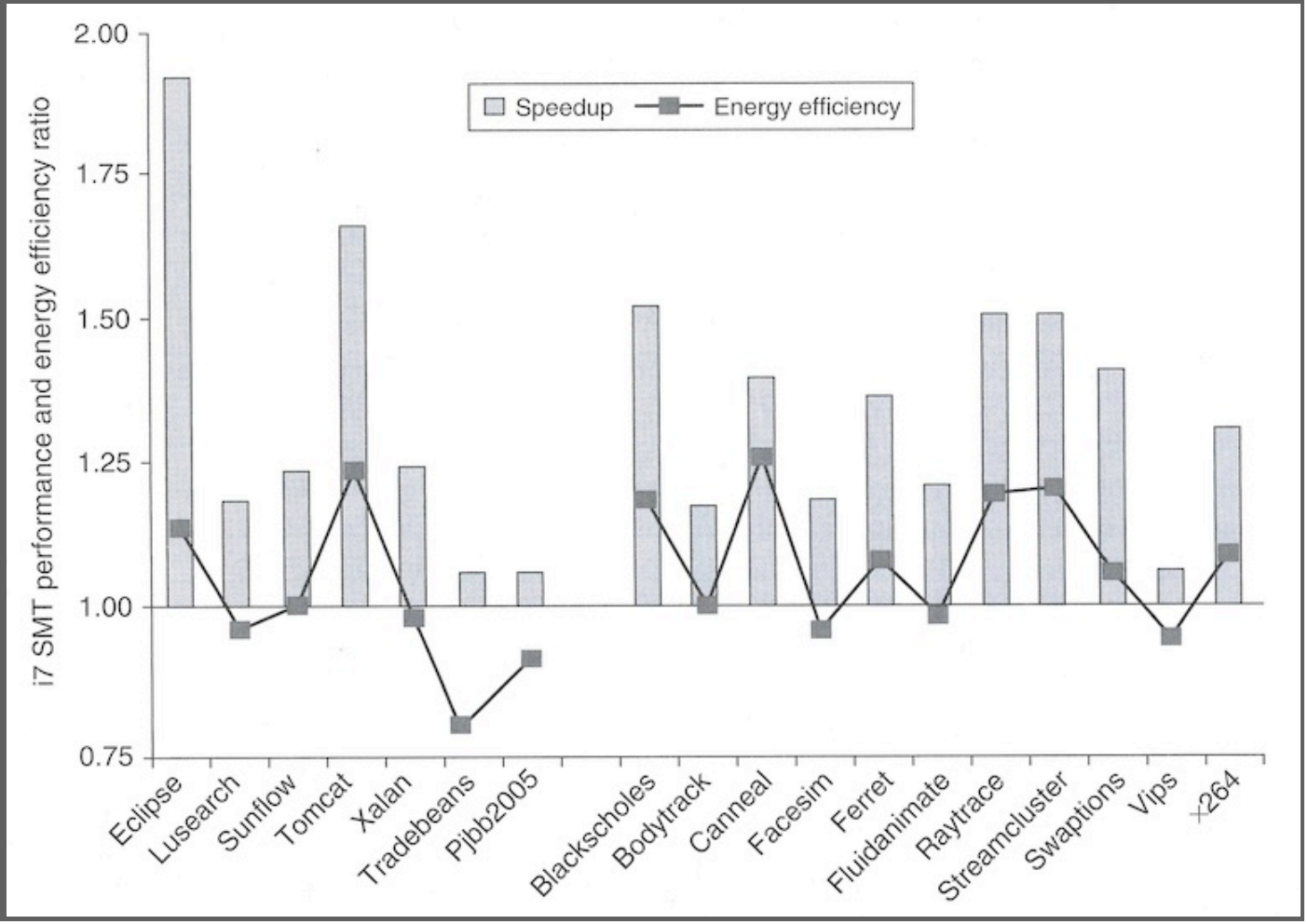


CPU/CORE multithreaded

- Nei moderni processori, ogni core supporta l'SMT, e può eseguire in parallelo istruzioni appartenenti a due (o più) peer thread.
- Questo a volte genera un po' di confusione nella definizione della CPU: il singolo core che implementa l'SMT viene indicato come dual-threaded o come dual-core proprio per la sua capacità di eseguire in parallelo istruzioni appartenenti a thread diversi.
- Ma: "dual-threaded" significa che **il singolo core** può eseguire in parallelo istruzioni appartenenti a due (o più) peer thread distinti. "dual core" significa che **i due core possono** eseguire in parallelo istruzioni appartenenti a due processi distinti, su un core vengono eseguire le istruzioni di un processo, e sull'altro core le istruzioni dell'altro processo.
- Naturalmente ciascun core può essere a sua volta multi-threaded

Ma l'SMT conviene?

Si: ecco lo speed-up ottenuto su uno dei core di una CPU i7 nel passare da un solo thread a 2, per diversi benchmarks. Il dato "Energy efficiency" ci dice se l'introduzione dell'SMT è vantaggiosa dal punto di vista dell'energia consumata. Un valore superiore a 1.0 significa che l'SMT riduce il tempo di esecuzione più di quanto aumenti i consumi.



CPU/CORE multithreaded

- Processori come quelli della famiglia Intel i3 – i9 sono CPU multi-core, ciascuno dei quali è dual-threaded, cioè in grado di eseguire in parallelo istruzioni appartenenti a 2 peer thread (è la tecnologia che Intel chiama Hyper-threading).

- Altri processori sono progettati per favorire esplicitamente l'esecuzione di applicazioni a thread: il SUN SPARC T3 "Rainbow Fall" è formato da 16 core, ciascuno dei quali è in grado di eseguire in parallelo le istruzioni di 8 peer thread.

- Domanda un po' difficile. Nessuna CPU implementa una forma di Simultaneous Multi-Processing, in cui cioè istruzioni appartenenti a processi diversi vengono eseguite in parallelo all'interno dello stesso core. Perché?

Per chi vuole approfondire

- Capitolo 4: Thread e Concorrenza