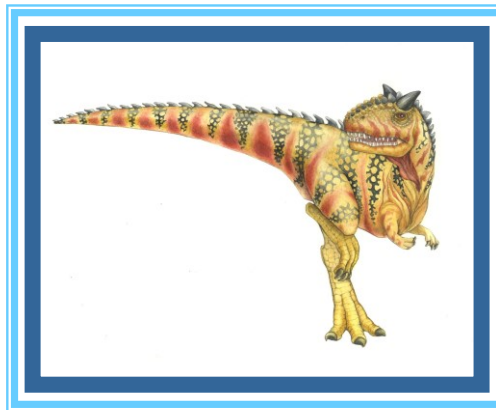


Thread e concorrenza





Paralleli-zei nello Spero Processori

Obiettivi

- ❖ Introdurre la nozione di thread – l'unità fondamentale di utilizzo della CPU nei computer che supportano il multithreading *di base*
- ❖ Descrivere i principali vantaggi e le problematiche più significative della progettazione di processi multithread *di usare Thread e Processi*
difficoltà nell'usare Thread
- ❖ Discutere le API per le librerie di thread, con particolare riferimento a Pthreads *chiedo al compilatore di vedere un programma multithread*
- ❖ Presentare approcci al threading implicito
- ❖ Definire il supporto di Linux ai thread *usando uno syscall clone, fork con portabili parametri*





Sommario

- ❖ Thread: motivazioni e definizione
- ❖ Programmazione multicore
- ❖ Modelli di programmazione multithread
- ❖ Librerie per i thread
- ❖ Threading implicito
- ❖ Problemi nella programmazione multithread
- ❖ Thread di Linux





Motivazioni – 1

❖ La maggior parte delle applicazioni attuali sono multi-thread

❖ I thread vengono eseguiti “all’interno” delle applicazioni

❖ La gestione dei processi può diventare molto onerosa, sia dal punto di vista computazionale che per l'utilizzo di risorse

- Creazione: *di un thread, non devo allocare nuovo spazio degli indirizzi e non vanno allocati DATI e CODICE* allocazione dello spazio degli indirizzi e successiva popolazione
 - Context-switch: *tempo di overhead puro per CPU* salvataggio e ripristino degli spazi di indirizzamento (codice, dati, stack) di due processi
- per i thread molto più veloci, meno tempo sprecato*





Creazione Thread, ottimo se è veloce

Motivazioni – 2

ogni volta che arriva una richiesta per far girare un Thread per gestire la richiesta

- ❖ Con un'applicazione che genera molti processi (es.: server), il SO rischia di passare la maggior parte del tempo a svolgere operazioni interne di gestione, piuttosto che ad eseguire codice applicativo
- ❖ Inoltre, a volte, l'attività richiesta ha vita relativamente breve rispetto ai tempi di gestione
 - **Esempio:** invio di una pagina html da parte di un server Web $\Rightarrow$ "operazione leggera", per motivare un nuovo processo
- ❖ Infine, esistono applicazioni che, presentando un intrinseco grado di parallelismo, possono essere decomposte in attività da eseguire concorrentemente, sulla base di un insieme di dati e risorse comuni

non vogliono che tutti i processi di esecuzione del Thread

tutti i Thread fanno riferimento a dati comuni





1 processo, tanti thread, si usa per codice, dati o file o agenti

Motivazioni – 3

❖ **Esempio:** Applicazioni in tempo reale per il controllo di impianti fisici, in cui si individuano attività quali...

- Controllo dei singoli dispositivi di I/O dedicati a raccogliere i dati del processo fisico (sensori)
- Invio di comandi verso l'impianto

Attivazione a voce multithread

⇒ Le diverse attività devono frequentemente accedere a strutture dati comuni, che rappresentano lo stato complessivo del sistema da controllare

❖ Infine...

- La programmazione multithread può produrre codice più semplice e più efficiente non sempre

- I kernel attuali sono normalmente multithread

Supporto la possibilità di avere thread

- Il multithreading si adatta perfettamente alla programmazione nei sistemi multicore

perché ora su un core posso avere 2 thread dello stesso processo in esecuzione

in Solaris, un core thread costa 1/30 rispetto alla versione mono-processo
1/5 per il 4.6 context switch





Definizione – 1

THREAD, PROCESSO, PC, CONTESTO
E STACK

#DEF THREAD, UNITÀ BASE D'USO della CPU

❖ Un *thread* (trama, filo) è l'unità base d'uso della CPU e comprende un identificatore di thread (TID), un contatore di programma, un insieme di registri ed uno stack

- Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati (globali) e le altre risorse allocate al processo originale (file aperti e segnali)

CONDIVISIONE CODICE, DATI GLOBALI E FILE APERTI

❖ Un processo tradizionale – *heavyweight process* – è composto da un solo thread

PROCESSO PESANTE

- Un processo è "pesante", in riferimento al contesto (spazio di indirizzamento, stato) che lo definisce

Thread
percorsi esecutivi
voglio distinguere
Dati globali

anche processi pesanti hanno sempre 2 punti
dalle risorse
Filo di esecuzione
voglio distinguere le due cose
così da avere più percorsi





Definizione – 2

- ❖ I thread sono detti anche “processi leggeri”, perché hanno un contesto ridotto (condividono lo spazio degli indirizzi)
- ❖ Poiché i thread appartenenti ad uno stesso processo ne condividono codice, dati e risorse...
 - Se un thread altera una variabile non locale al thread, la modifica è visibile a tutti gli altri thread
 - Se un thread apre un file, allora anche gli altri thread possono operare sul file
- ❖ Un processo multithread è in grado di eseguire più compiti in modo concorrente

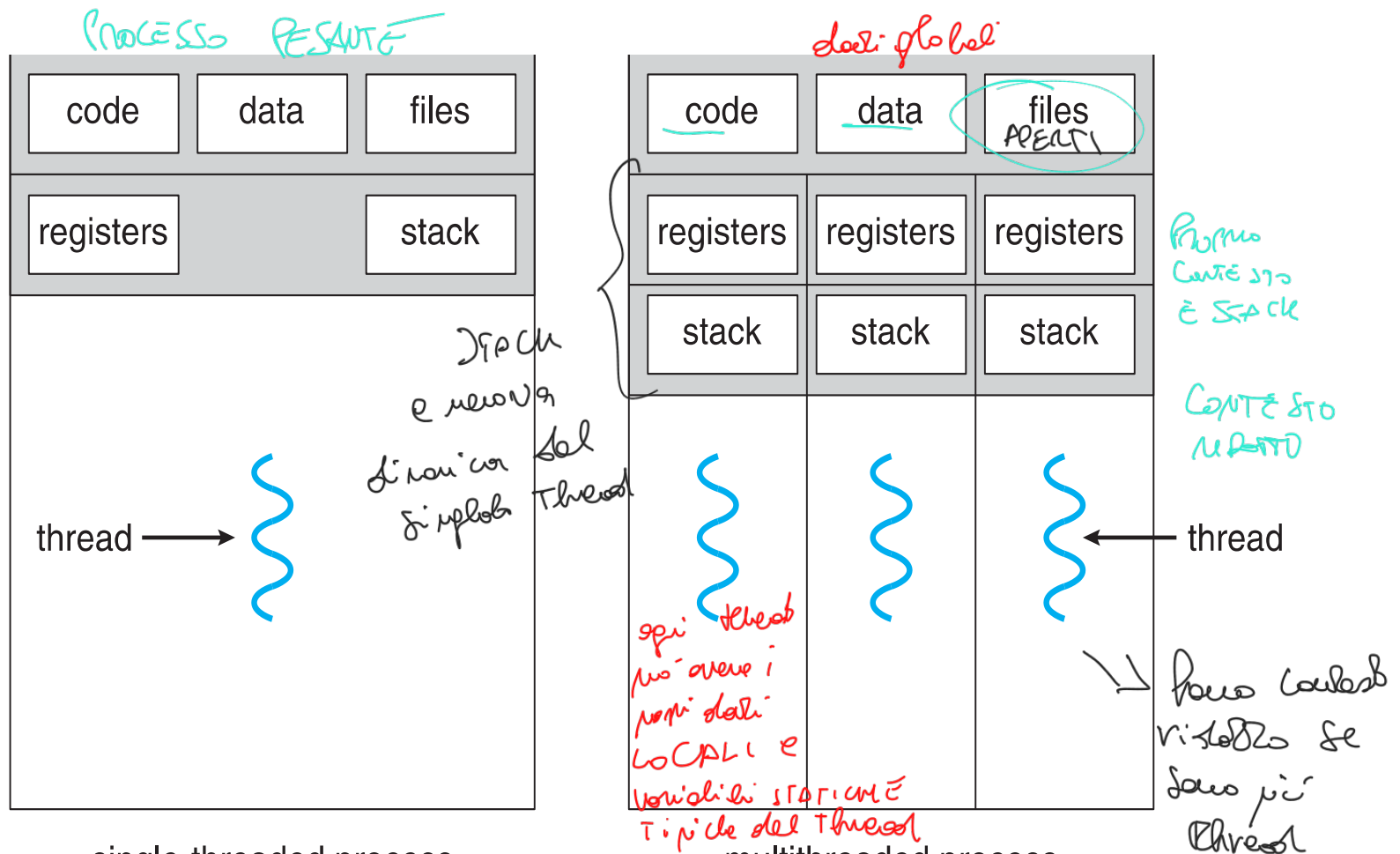
• essendo percorsi esecutivi:
cambiano i contenuti dei
registri della CPU
i THREAD hanno un loro
CONTESTO

hanno un loro TID (non PID), contesto
e PC





Processi a singolo thread e multithread



single-threaded process
CONTEXT SWITCH: ogni processo ha il suo proprio spazio indirizzi.

Processi ora vengono implementati. Rappresentano i processi sono sparsi nella RAM, e per mapparli, c'è una struttura dati (TA BELL delle PAGINE)





PAGINE
1
2

FRANC
1 23
420

Esempi – 1

Ci sono dei registri che devono contenere l'indirizzo della Tabella o della pagina

Thread condividevano PCB, anche info sui file aperti ecc, non o da condividere nel context switch

non se lo più thread non cambia la Tabella
popolo. Popolo da copiare nel context
e' un sottoinsieme dei registri del processore

❖ Browser web

- Un thread per la rappresentazione sullo schermo di immagini e testo *Client web multithread*
- Un thread per il reperimento dell'informazione in rete ** es un thread per gestire tutte queste cose*

❖ Text editor

- Possibile avere un thread per ciascun documento aperto (i thread sono tutti uguali ma lavorano su dati "locali" diversi)
- Relativamente ad ognuno è anche possibile avere...
 - ▶ Un thread per la visualizzazione dei dati su schermo
 - ▶ Un thread per la lettura dei dati immessi da tastiera
 - ▶ Un thread per la correzione ortografica e grammaticale
 - ▶ Un thread per il salvataggio periodico su memoria di massa

Se thread condividono file o questo file, e' verosimile condividerli con gli altri thread

Queste operaz. usate nella stessa cornice





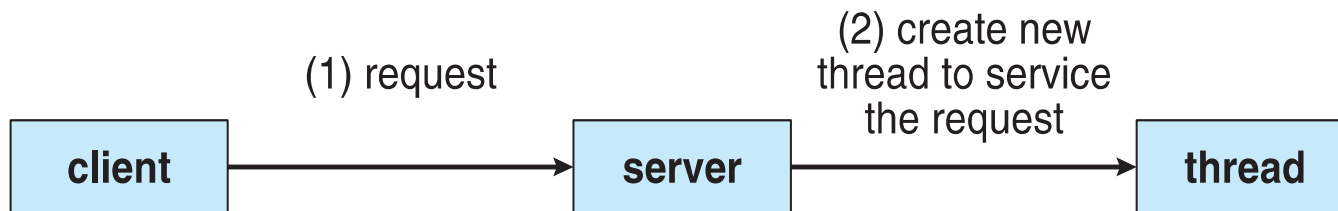
Esempi – 2

❖ Server per RPC

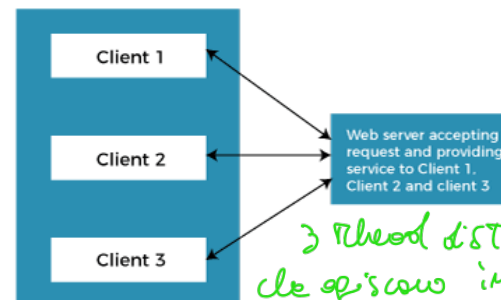
- Ogni invocazione di procedura remota viene delegata ad un thread separato (diverse richieste gestibili in concorrenza)

❖ Server web *1) thread server over in ascolto*

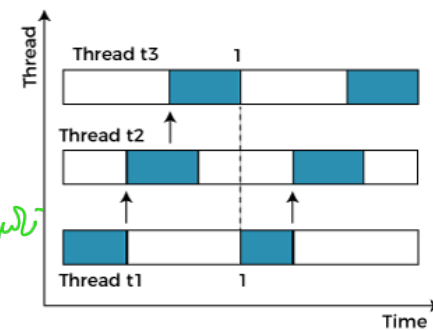
- Un thread per il servizio di ciascuna richiesta da parte dei client



thread server
(3) resume listening for additional client requests



*3 thread distinti
che operano in
CONCORRENZA*





Kernel multithread *funzionale per*

- ❖ I kernel dei sistemi operativi attuali *poterlo estendere sopra alle applicazioni* sono perlopiù multithread
 - Thread dedicati a servizi specifici
- ❖ **Solaris** *l'utente richiama interrupt e lo gestisce*
 - Thread specializzati nella gestione delle interruzioni
- ❖ **Linux**
 - Impiega thread a livello kernel per la gestione dei dispositivi, della memoria e delle interruzioni
 - Si può utilizzare il comando "**ps -ef**" per visualizzare i thread a livello kernel
 - Il thread **kthreadd** (con **pid=2**) funge da genitore di tutti i thread a livello kernel





Vantaggi – 1

Vantaggi Multithreading

❖ Tempo di risposta

Metnica es-tempo addizionale: tempo in cui processore è nella ready queue

*quando ci mette
dal primo al
ultimo output*

■ Rendere multithread un'applicazione interattiva permette ad un programma di continuare la propria esecuzione, anche se una parte di esso è bloccata o sta eseguendo un'operazione particolarmente lunga, riducendo il tempo medio di risposta

Parallelismo Compil.

■ Utile per le interfacce utente

con multithread si riduce il tempo di risposta più cose, anche se un thread è bloccato ci saranno altri più veloci dello stesso processo da produrre output

❖ Condivisione delle risorse

■ I thread condividono la memoria e le risorse del processo cui appartengono; il vantaggio della condivisione del codice e dei dati risiede nel fatto che un'applicazione può consistere di molti thread relativi ad attività diverse, tutti nello stesso spazio di indirizzi (comunicazioni molto più rapide che con memoria condivisa o message passing)

evita scambi messaggi (blocco) e condivisione memoria tra processi (no controllo S.O.)





Vantaggi – 2

CONTEXT SWITCH MOLTO PIÙ RAPIDO

❖ Economia

- Assegnare memoria e risorse per la creazione di nuovi processi è oneroso; poiché i thread condividono le risorse del processo cui appartengono, è molto più conveniente creare thread e gestirne i cambiamenti di contesto (il TCB dei thread non contiene informazioni relative a codice e dati, quindi alla gestione della memoria) *no nuovo spazio di memoria*
- ▶ **Esempio:** in Solaris la creazione di un processo richiede un tempo trenta volte maggiore della creazione di un thread; il cambio di contesto è cinque volte più lungo per i processi rispetto ai thread





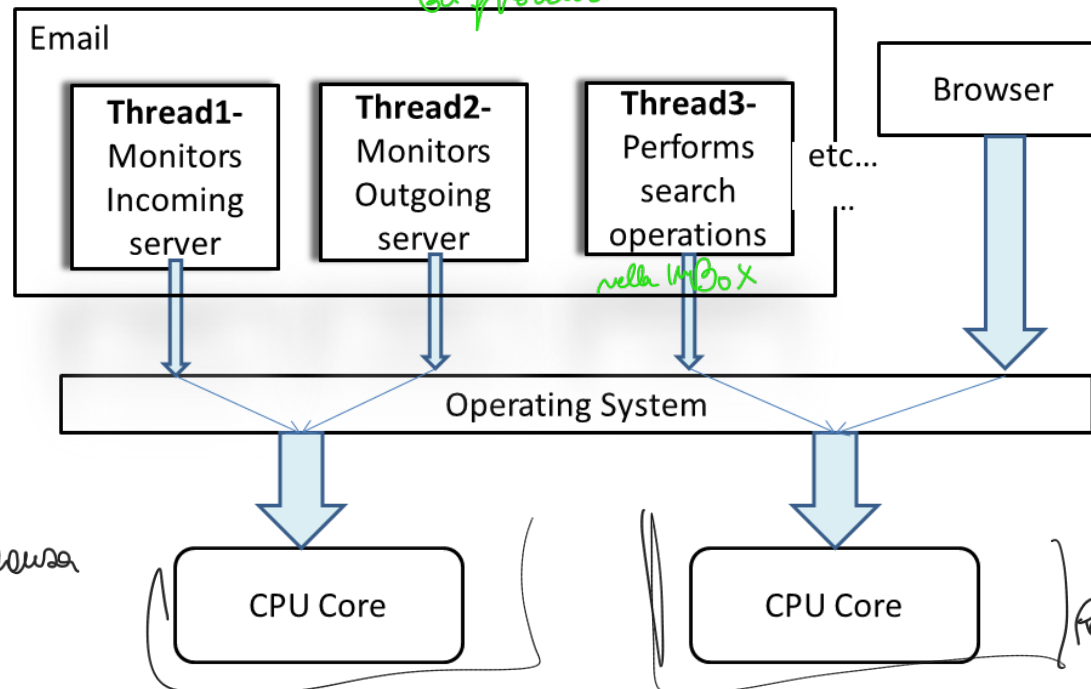
Vantaggi – 3

❖ Scalabilità

multithread più vantaggioso più thread abbiamo a disposizione

- I vantaggi della programmazione multithread aumentano notevolmente nelle architetture multiprocessore, dove i thread possono essere eseguiti in parallelo; l'impiego della programmazione multithread in un sistema con più unità di elaborazione fa aumentare il grado di parallelismo

Scheda di comp. differenti



Server per la gestione della mail

Concorrenza

ho concorrenza e parallelismo

Parallelismo



Programmazione multicore – 1

- ❖ Architetture multicore: diverse unità di calcolo sullo stesso chip *RAM e 1 o 2 livelli di cache condivisa*
- Ogni unità appare al SO come un processore separato
- ❖ Sui sistemi multicore i thread possono essere eseguiti in parallelo, poiché il sistema può assegnare thread diversi a ciascuna unità di calcolo
- ❖ Distinzione fra **parallelismo** e **concorrenza** *SISTEMA CONCORRENTE VS SISTEMA PARALL*
 - Un sistema è parallelo se può eseguire simultaneamente più task *Solo se ci sono più unità di calcolo. V CPU in solo process in esecuzione*
 - Un sistema concorrente, invece, supporta più task, consentendo a tutti di progredire nell'esecuzione grazie al multiplexing della CPU (unica) *scambio di processi in elaborazione*

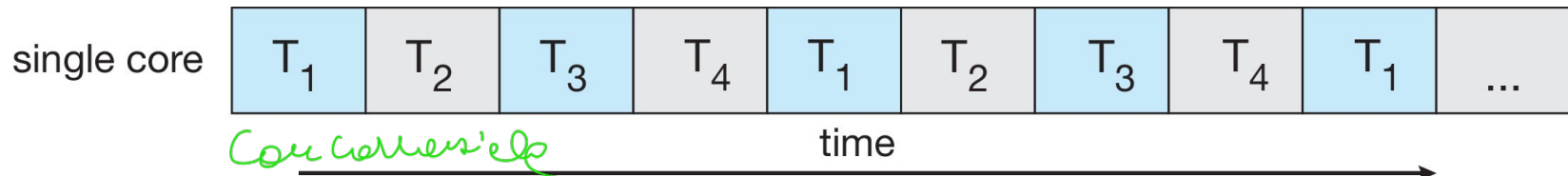
ogni core è un processore a se stante dal pto di vista del SO e dunque uno scheduler V core.
ancora più difficile schedare



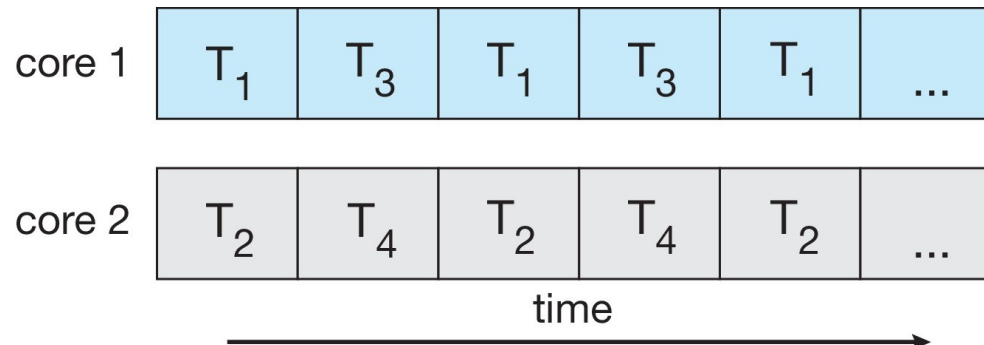


Concorrenza vs Parallelismo

❖ Esecuzione concorrente su un sistema single core



❖ Parallelismo nei sistemi multicore *ho due core ho parallelismo*



*ovviamente
in ogni core
ci sarà
concorrenza*





Programmazione multicore – 2

❖ Tipi di parallelismo

Una stessa operazione può essere eseguita in parallelo su più core su sottoinsiemi di dati.
CALCOLO SU STESSO SET DI DATI

- Con il termine data parallelism ci si riferisce a scenari in cui la stessa operazione viene eseguita contemporaneamente (ovvero in parallelo) su sottoinsiemi dei dati; nelle operazioni in parallelo sui dati, infatti, l'insieme originale viene suddiviso in partizioni in modo che più thread (che compiono la stessa operazione) possano agire simultaneamente su segmenti diversi

ogni core opera su un insieme di dati differenti, in parallelo

- Con il termine task parallelism si indica una forma di parallelizzazione del codice tra più processori in ambienti di calcolo parallelo; il task parallelism si concentra cioè sulla distribuzione di thread diversi in esecuzione nei diversi nodi di calcolo (che possono agire su dati comuni)

→ Si core diff. vanno thread che eseguono op. diversi solo stesso programma su dati uguali o differenti

- ❖ Nella maggior parte dei casi, le applicazioni utilizzano un ibrido delle due strategie

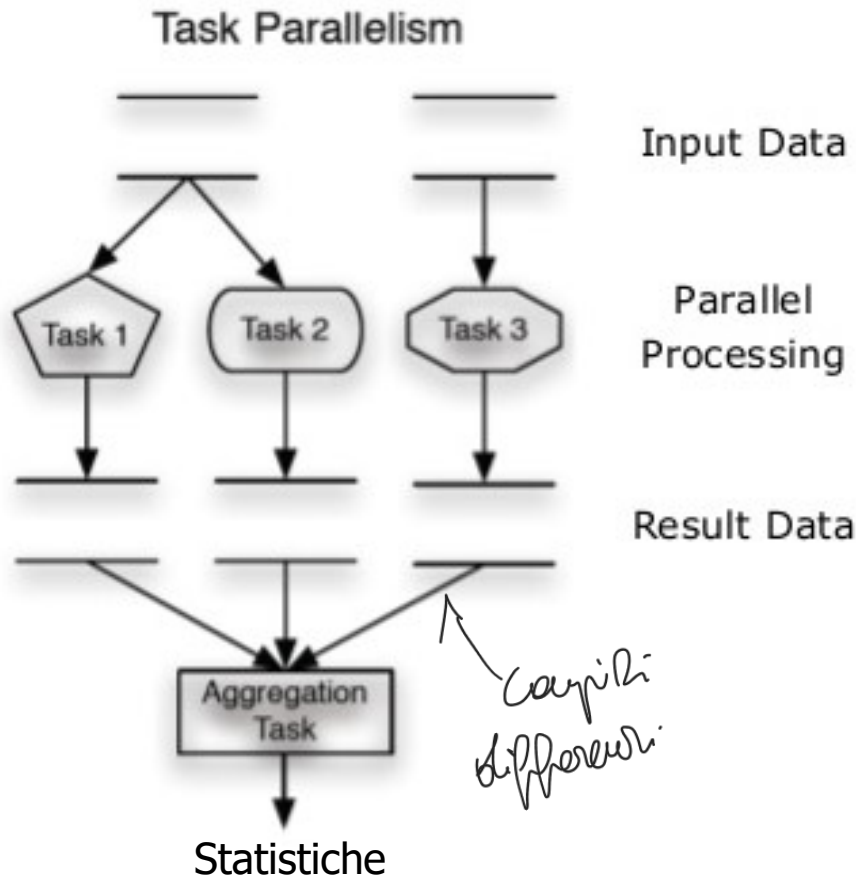
OPERAZIONI DIVERSE



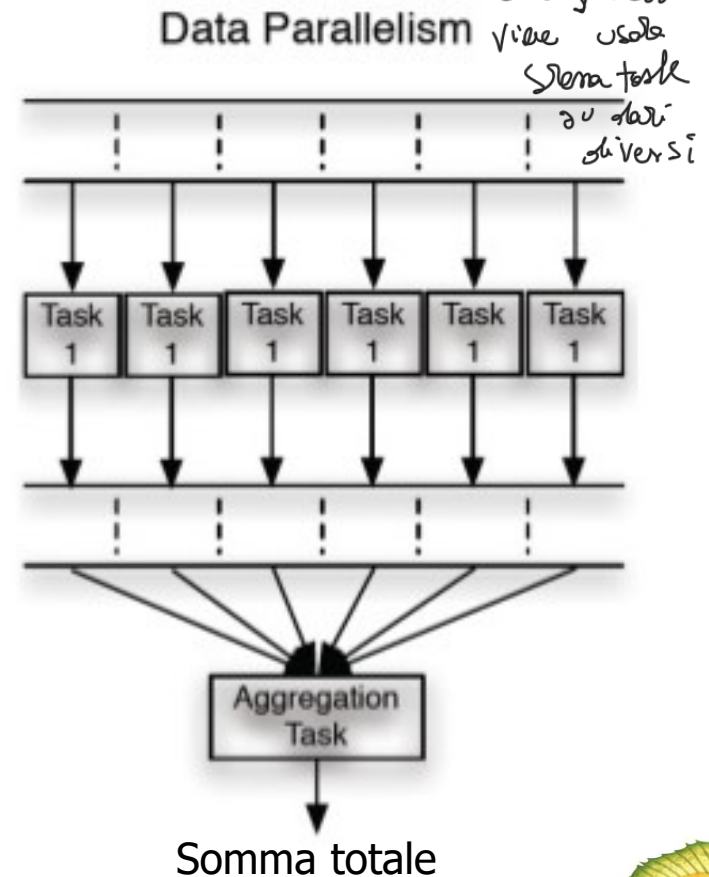


Programmazione multicore – 3

Calcolo di max e min e della media degli elementi di un vettore



Calcolo della somma degli elementi di un vettore



nella realtà, natura ibrida, uso esau...





Programmazione multicore – 4

❖ Obiettivi della programmazione nei sistemi multicore

es. dati con i
e capitoli diversi
e ogni capitolo è
in threads

Separazione dei task: esaminare le applicazioni per individuare aree separabili in task distinti che possano essere eseguiti in parallelo

computazionale dei thread deve essere bilanciato

- Bilanciamento: produrre task che eseguano compiti di mole confrontabile (per carico computazionale)

dati suddivisi
contribuzione, esempio
architettura NUMA,
RAM UNICA

Suddivisione dei dati: i dati a cui i task accedono, e che manipolano, devono essere suddivisi per essere utilizzati da unità di calcolo distinte

porzione cache & CPU

- Dipendenze nei dati: verificare le dipendenze tra due o più task per sincronizzarne l'esecuzione

è possibile che
non tutti i thread
possono accedere
in parallelo e
devono essere
sequenziali

Test e debugging: garantire la correttezza malgrado la possibilità di flussi di esecuzione multipli e di eventuali problemi di concorrenza nell'accesso ai dati

↳ uso programmi, debugger

↳ difficile verificare
correttezza dati





Programmazione multicore – 5

ogni core può essere visto come più core logici distinti in cui fare scheduling

multithreading Hardware

- ❖ In corrispondenza con la proliferazione di programmi multithread, le nuove architetture hardware forniscono sempre maggior supporto al multithreading
- ❖ Progetti hardware recenti (*hyperthreading*) implementano unità di calcolo multithread in cui due o più thread hardware sono assegnati ad un singolo core
 - Quando un processore accede alla memoria, infatti, una quantità significativa di tempo (fino al 50%) trascorre in attesa della disponibilità dei dati: *stallo della memoria*
- ❖ Dal punto di vista del SO, ogni thread hardware appare come un processore logico in grado di eseguire un thread software

Un processore si trova ad eseguire una istruzione in un dato momento in memoria e non nei registri (cache) (sono in RAM); entro lo stallone della memoria

Mentre un core è in stallone, S.O. vede questo core come 2 CPU, e scheduler separa i thread in stallone, però al thread è e' col HW





Programmazione multicore – 6

- ❖ **Esempio:** il microprocessore Oracle SPARC T4 (2011) ha otto core, ciascuno dotato di otto thread hardware
⇒ 64 unità di calcolo distinte che il SO deve gestire

*few silberschatz 64 processi logici
così sempre CPU usata*

C

M

Ciclo di computazione

Accesso alla memoria



Stallo della memoria

Thread₁



Thread₀



Sistema con multithreading hardware (relativo ad un core)





Quanti più Spreadup prestazioni core
paralleli possibile

quantando meno core, come avere
trasmissioni?

La legge di Amdahl – 1

non tutto il programma può
essere parallelizzato

- ❖ Permette di determinare i potenziali guadagni in termini di prestazioni ottenuti dall'aggiunta di core, nel caso di applicazioni che contengano sia componenti seriali (non parallelizzabili) sia componenti parallele

- ❖ S porzione seriale

- ❖ N core quanto moltiplica
il tempo di esecuzione?

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Questo perché
non è lineare

$$S = \frac{1}{\frac{1}{p} + 1 - p}$$

- ❖ **Esempio:** Se un'applicazione è al 75% parallela ed al 25% seriale, passando da 1 a 2 core, si ottiene uno speedup pari a 1.6; se i core sono 4, lo speedup è invece di 2.28

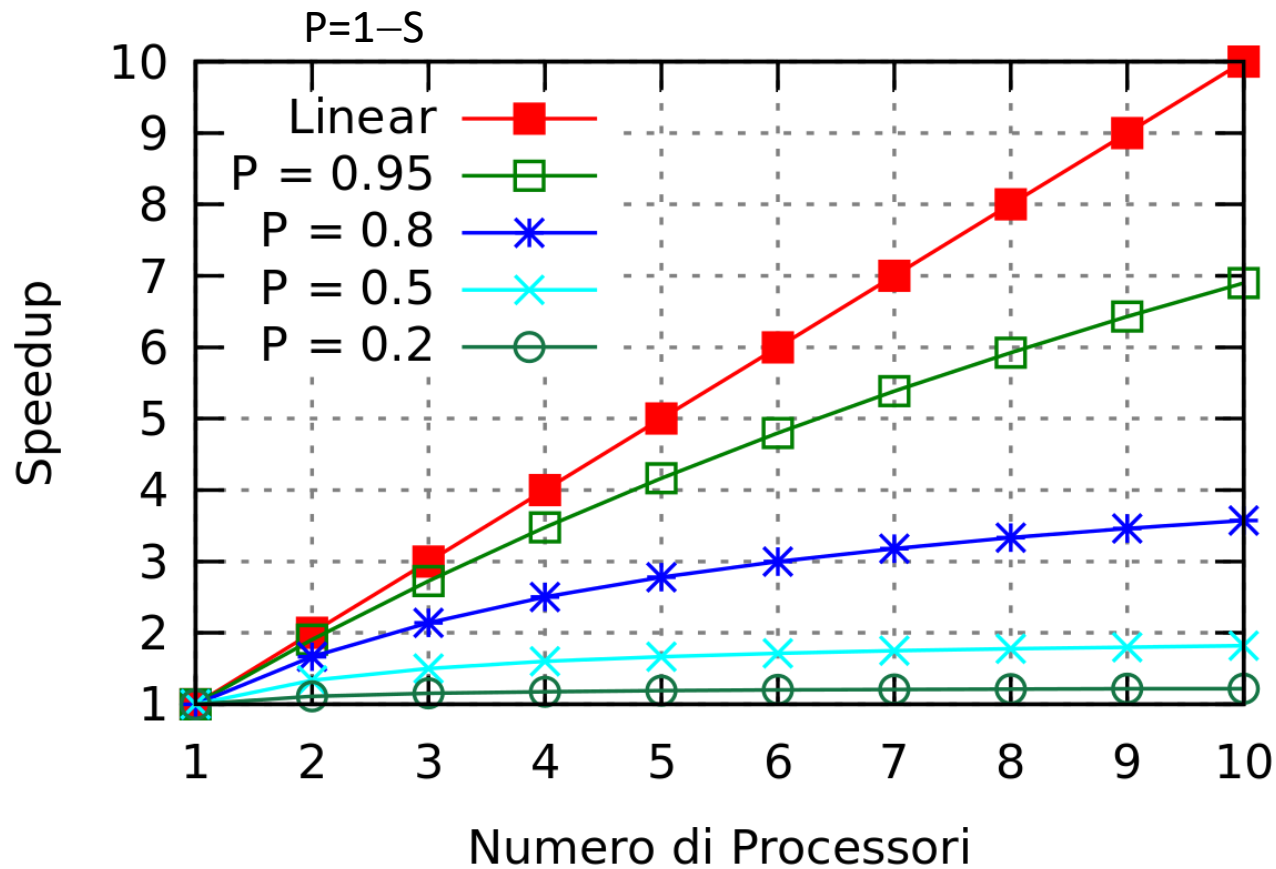
- ❖ In generale... per $N \rightarrow \infty$ lo speedup tende a $1/S$ es. qui $1-p = 0.25$
 ⇒ Se il 40% di un'applicazione è seriale il massimo aumento di velocità è di 2.5 volte speedup limitato dalla parte seriale, non ripartibile

- ❖ La porzione seriale di un'applicazione ha un effetto dominante sulle prestazioni ottenibili con l'aggiunta di nuovi core





La legge di Amdahl – 2





Supporto del SO ai thread – 1

Supporto a livello kernel.

- 2 soluzioni
- ❖ Supporto user-level, tramite librerie di funzioni (API) per gestire n thread in esecuzione
 - ❖ Supporto kernel-level, tramite "un'astrazione di traccia di esecuzione" (su un processore virtuale)
- kernel si occupa del suo scheduling
- kernel interviene pesantemente
- ## Thread a livello utente

- Sono gestiti come uno strato separato sopra il nucleo del sistema operativo no intervento kernel
- Sono realizzati tramite librerie di funzioni che formano API per la creazione, lo scheduling e la gestione dei thread, senza alcun intervento diretto del nucleo no intervento kernel, create e gestite dalle API
- POSIX Pthreads è la libreria per la realizzazione di thread utente in sistemi UNIX-like
- Windows threads per sistemi Windows e Java threads per la JVM





Supporto del SO ai thread – 2

❖ Thread a livello kernel *GESTITE dal KERNEL*

- Sono gestiti direttamente dal SO: il nucleo si occupa di *creazione, scheduling, sincronizzazione e cancellazione* dei thread nel suo spazio di indirizzi
- Supportati da tutti i sistemi operativi attuali
 - ▶ Windows
 - ▶ *Solaris* *obsoleto*
 - ▶ Linux
 - ▶ Mac OS
 - ▶ iOS
 - ▶ Android





Modelli di programmazione multithread

una possibilità
→ ❖

Modello multi-a-uno (M:1)

Non c'è vero parallelismo tra thread della stessa app

Come si agitano i thread a livello utente su thread a livello utente

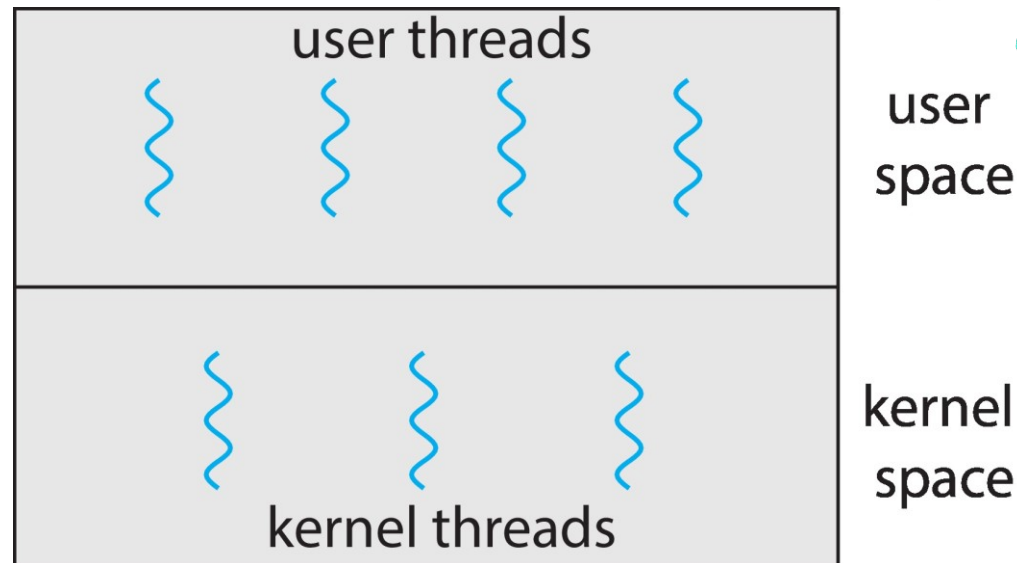
❖ **Modello uno-a-uno (1:1)**

COMUNE. Corrisponde 1:1 e parallelismo

❖ **Modello multi-a-molti (M:M)**

IBMD, non implementato

Voglio MAPPER THREAD
da livello utente
a KERNEL



Kernel gestisce thread, non processi



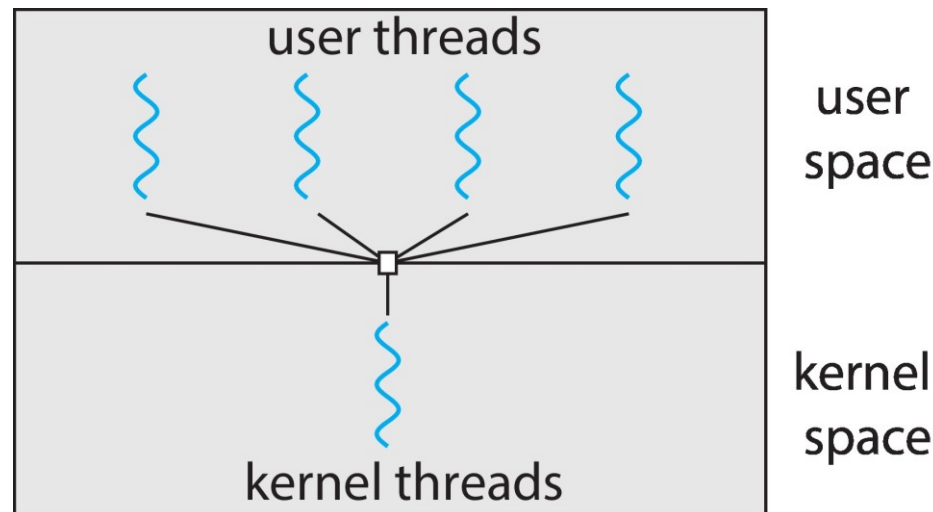


Modello multi-a-uno – 1

*libreria utente schedula i thread
ma kernel schedula un solo thread*

- ❖ Molti thread a livello utente vanno a corrispondere ad un unico thread a livello kernel
 - Il kernel “vede” una sola traccia di esecuzione
- ❖ In altre parole: i thread sono implementati a livello di applicazione, il loro scheduler non fa parte del SO, che continua ad avere solo la visibilità del processo

a livello utente deciso un unico thread che avrebbe nella READY QUEUE



*KERNEL vede
solo un processo*





Modello multi-a-uno – 2

❖ Vantaggi

- Gestione efficiente dei thread nello spazio utente (scheduling poco oneroso) *efficiente e poco pericoloso*
- Non richiede un kernel multithread per poter essere implementato *kernel vede una processa unica, poco usare kernel unico*

❖ Svantaggi *Tempo di risposta lento, no parallelismo*

- L'intero processo rimane bloccato se un thread invoca una chiamata di sistema di tipo bloccante
- I thread sono legati allo stesso processo a livello kernel e non possono essere eseguiti su processori fisici distinti *NO PARALLELISMO SU CORE DIFF.*

❖ Attualmente, pochi SO implementano questo modello; sono esempi storici:

- GNU Portable Threads (2006)
- Solaris Green Threads (2010)

è qm ha un unico thread in esecuzione

No parallelismo anche con più core, solo su processori differenti.

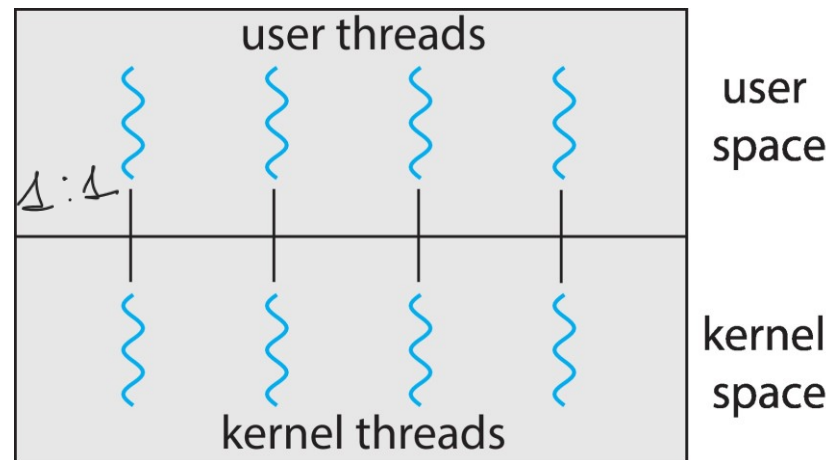




Modello uno-a-uno – 1

*tutti i thread di tutte le app partecipano alla contesa
azione della CPU*

- ❖ Ciascun thread a livello utente corrisponde ad un thread a livello kernel
 - Il kernel “vede” una traccia di esecuzione distinta per ogni thread
 - I thread vengono gestiti dallo scheduler del kernel (come se fossero processi; si dicono *thread nativi*)



*scheduling fatto
dal S.O.*





Modello uno-a-uno – 2

❖ Vantaggi

- Scheduling molto efficiente *perché il SO è in grado di gestire il I/O*
- Se un thread effettua una chiamata bloccante, gli altri thread possono proseguire nella loro esecuzione
- I thread possono essere eseguiti su processori fisici distinti *PARALLELISMO della stessa app*

❖ Svantaggi

- Possibile inefficienza per il carico di lavoro dovuto alla creazione di molti thread a livello kernel (limiti imposti a priori) *PIÙ CARICO e KERNEL DEVE SUPPORTARLO*
- Richiede un kernel multithread per poter essere implementato *molto carico nel sistema, scheduling difficile*
posso avere limiti posti a priori
↳ ho thread nativi

❖ Esempi

- Windows 95/98/NT/2000/XP/Vista e versioni attuali
- Linux, Solaris (versione 9 e successive)



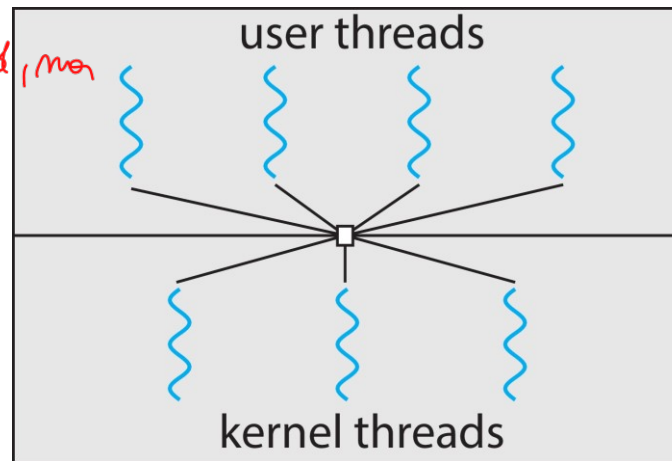


Modello multi-a-molti – 1

- ❖ Si mettono in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel
molti bob. utente > molti. kernel
 - In altre parole, il sistema dispone di un pool di thread – *worker* – ognuno dei quali viene assegnato di volta in volta ad un thread utente

*C'è Scheduler a livello utente
posso schedulare più thread, ma
un sottosistema*

*Posso regolare il carico
sul sistema*



*SCHEDULO UN SOTTOINSIEME DI
THREAD*

*SCHEDULO
a livello utente
e KERNEL*





Modello multi-a-molti – 2

❖ Vantaggi

- Possibilità di creare tanti thread a livello kernel quanti sono necessari per la particolare applicazione (e sulla particolare architettura), eseguibili in parallelo su architetture multiprocessore
- Se un thread invoca una chiamata di sistema bloccante, il kernel può fare eseguire un altro thread

❖ Svantaggi

- Difficoltà nel definire la dimensione del pool di worker e le modalità di cooperazione tra i due scheduler

❖ Esempi:

- Windows con il ThreadFiber package
- Solaris (versioni precedenti alla 9)
- Tru64 UNIX

poco usato

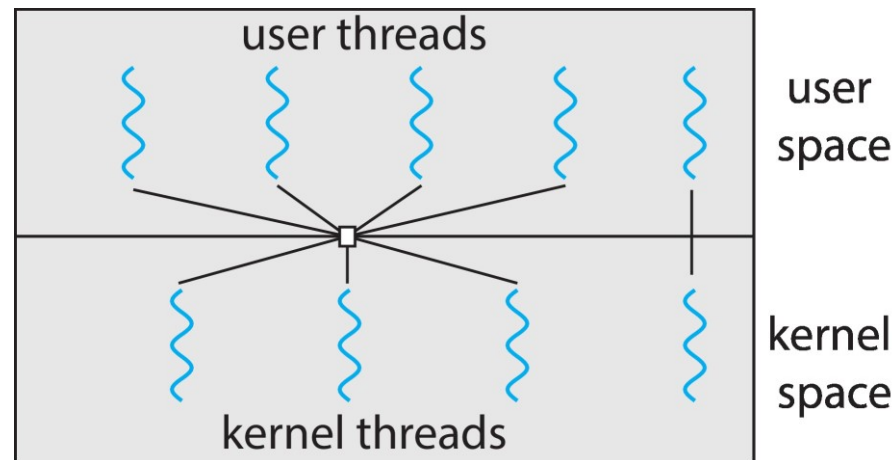




Modello a due livelli

- ❖ Simile al modello M:M, offre però la possibilità di vincolare un thread utente ad un thread del kernel
- ❖ Esempi:
 - IRIX
 - HP-UX
 - Solaris (versioni precedenti alla 8)
 - Tru64 UNIX

4cx 1:1 e N:M





Librerie dei thread – 1

per Neelzone thread

- ❖ Forniscono al programmatore una API per la creazione e la gestione dei thread *Applic. Programming Interface*

- ❖ Librerie a livello utente

- Codice e strutture dati nello spazio utente
- Invocare una funzione di libreria si traduce in una chiamata locale a funzione, non in una system call

- ❖ Librerie a livello kernel

- Codice e strutture dati nello spazio del kernel
- Invocare una funzione della API provoca, generalmente, una chiamata di sistema

↳ chiama f. libreria => syscall





Librerie dei thread – 2

Standard IEEE

- ❖ **POSIX Pthreads** = Specifico per implementare Specifiche
 - Può presentarsi sia come libreria a livello utente sia a livello kernel
 - ❖ **Win64**
 - Libreria di thread a livello kernel per sistemi Windows
 - ❖ **Java**
 - API gestibile direttamente dai programmi Java
 - La API di Java per i thread è solitamente implementata per mezzo della libreria dei thread del sistema che ospita la JVM
- principale a livello kernel*
- Java VM fa riferimento all'HW sottostante
es. se c'è sotto WINDOWS, saranno gestiti da LW*





Librerie dei thread – 3

- ❖ Nel threading di POSIX e di Windows tutti i dati dichiarati a livello globale, ovvero al di fuori da ogni funzione, sono condivisi fra tutti i thread appartenenti allo stesso processo

Dati globali: dati che un processo condivide con i thread

Dato globale e' tutto cio' che e' dichiarato fuori da MAIN e altre FUNZIONI

- ❖ I dati locali di una funzione sono, normalmente, memorizzati nello stack

Memoria Globale non sono in stack, occupano spazio

- ❖ Ogni thread ha un proprio stack

⇒ ogni thread ha la propria copia di dati locali

Thread Local STORAGE; variabili che nel C sono variabili STATIC

→ durata e' globale, ma visibilita' e' LOCALE, visibile solo dalla funzione che la dichiara





Strategie di threading

CONCURRENZA PADRE-FIGLIO

❖ Threading asincrono

Un processo crea un thread e prosegue con la sua esecuzione in concorrenza

- Il genitore crea un figlio e quindi riprende la propria esecuzione in concorrenza *es. nel server: continua a fare il suo e delega il thread a fare un'altra operazione*
- Ogni thread viene eseguito in modo indipendente rispetto agli altri ed il thread genitore non ha bisogno di sapere quando terminano i figli *Simulano e esecuzioni, segue diverso*
- Scarsa condivisione dei dati fra i thread *ogni thread ha solo bisogno di un po' di dati comuni*
- Esempio:** server web, interfacce utente responsive

Simulano se possono padre e figlio se eseg. in concorrenza

OGNI THREAD È INDIPEND.

❖ Threading sincrono

- Il genitore crea uno o più figli e attende che tutti terminino prima di riprendere l'esecuzione ⇒ **fork-join** *No CONCURRENZA PADRE-FIGLIO*
- I thread figli si eseguono in concorrenza *ma non con il padre*
- Quando un thread ha terminato il proprio compito, termina e si unisce al genitore *lo rende noto al genitore che lo termina*
- Significativa condivisione dei dati fra i thread
- Il genitore può combinare i risultati calcolati dai figli

→ genitore deve combinare i dati prodotti dai figli





Pthreads

- ❖ È lo standard POSIX (IEEE 1003.1c) che definisce la API per la creazione e la sincronizzazione dei thread (definizione – no realizzazione) → dipende dai S.O.
- ❖ Viceversa, è la API che specifica il comportamento della libreria dei thread, la cui implementazione dipende dal particolare sistema operativo (può essere a livello utente o a livello kernel)
- ❖ Comune nei SO UNIX-like (Solaris, Linux, Mac OS)
- ❖ **Esempio:** threading sincrono per il calcolo della somma dei primi N interi

Programma principale delega i thread.
Sottoprogramma è un programma a parte





Esempio con Pthreads – 1

```
/* Programma per il calcolo della somma dei primi N interi */
#include <pthread.h> /* include per la libreria Pthreads */
#include <stdio.h>
#include <stdlib.h>

int sum; /* variabile globale (condivisa) */ dato condiviso del process
void *somma(char *param); funzione che darà vita ai thread /* funzione del nuovo thread */
int main(int argc, char *argv[])
{
    pthread_t tid; /* identificatore UNICO del thread */
    pthread_attr_t attr; /* attributi del thread */ Struttura dati che ha più attributi

    if (argc != 2) {
        fprintf(stderr, "Occorreva inserire N!");
        exit(1); devo inserire un numero come argomento da linea di comando
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "%d deve essere >=0\n", atoi(argv[1]));
        exit(1);
    }

    pthread_attr_init(&attr); /* reperisce attributi predefiniti */
    pthread_create(&tid, &attr, somma, argv[1]); /* crea il thread */ NOME ARGOMENTO
    /* il padre attende la terminazione del nuovo thread */
    pthread_join(tid, NULL); ricomprico figlio attraverso tid
    printf("somma = %d\n", sum);
    exit(0); Se c'è qualcosa da fare, recupero del processo padre, uso la shell
}
```

es. dati dello
stack

Se riesco lo
stack, finisco
spazio degli
iniziali

controlli

inizializzo gli
attrib. del thread con
quelli predefiniti

Dimensione della pila,
informazioni per lo
scheduling, etc.

*è una stringa
caratteristica*



↳ Sono nome sottoprogramma che viene eseguito come corpo del Thread

Esempio con Pthreads – 2

Padre ha un solo figlio

```
/* Il nuovo thread assume il controllo da questa funzione */  
void *somma(char *param);  
{  
    int i, upper=atoi(param);  
    sum = 0;  
  
    for (i=1, i<=upper; i++)  
        sum += i;  
  
    pthread_exit(0);  
}
```

Se ho più figli, devo fare il join di più figli.





Join in Pthreads

```
#define NUM_THREADS 10
```

*creo 3 threads
LAVORATORI*

```
/* an array of threads to be joined upon */  
pthread_t workers[NUM_THREADS];
```

```
for (int i = 0; i < NUM_THREADS; i++)  
    pthread_join(workers[i], NULL);
```

*Si fa vestizione Petri e lo i tibi prima di
lavorare l'esecuzione*





Threading implicito

affidare threading
al compilatore
libreria

interazione thread difficile

- ❖ Le applicazioni multithread sono più difficili da programmare ed aumentano in complessità al crescere del numero dei thread *Difficile da programmare, difficile prendere errori di sincronizzazione*
- ❖ Trasferimento della creazione e della gestione del threading dagli sviluppatori di applicazioni ai compilatori ed alle librerie di runtime

⇒ Threading implicito

- Gruppi di thread (Thread pool – Windows, Java)
- OpenMP C, C++... insieme di direttive al compilatore *affidare il codice*
- Grand Central Dispatch (Mac OS e iOS), Java Fork-Join, Intel Threading Building Blocks *sia parallelo che*





Gruppi di thread – 1

*non come thread di un web server
che si creano in base alle richieste client*

❖ Un numero illimitato di thread presenti nel sistema potrebbe esaurirne le risorse

→ già creati, esistono già

⇒ Creare un certo numero di thread (worker) alla creazione del processo ed organizzarli in un gruppo in cui attendano il lavoro che verrà loro successivamente richiesto

Per thread definiti a priori, non caricano il server stesso, max 10 thread, per non max 10 richieste.

• Se ho lo più pronto,

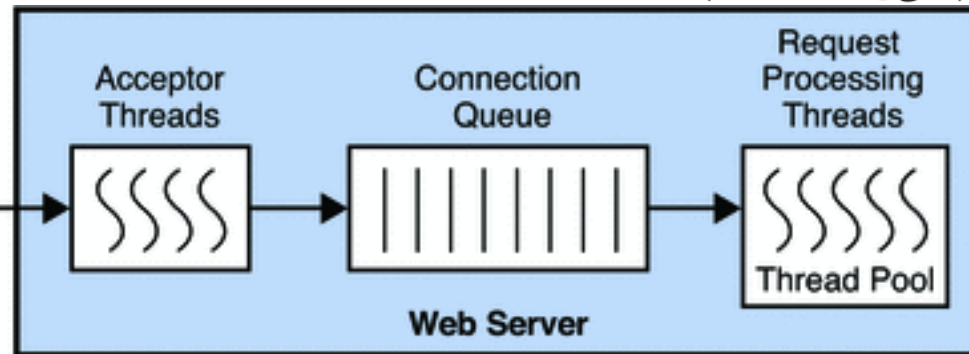
non posso tempo per

crearlo, devo Requests

Se proprio tardi

per eseguire il lavoro,

Meno tempo res.





Gruppi di thread – 2

alloca N
thread allo
cassa del
processo e li
alloca al
occorre

❖ Vantaggi

Separo creazione da esecuzione -
=> thread più in memoria

- Di solito il servizio di una richiesta tramite un thread esistente è più rapido, poiché elimina l'attesa della creazione di un nuovo thread
- Si limita il numero di thread relativi a ciascun processo alla dimensione prestabilita (rilevante per sistemi che non possono sostenere un numero elevato di thread concorrenti) *capito di creazione diverso dal capito di esecuzione*
- Separare il task da eseguire dal meccanismo di creazione dello stesso permette strategie diverse di elaborazione *poss o periodici*
 - ▶ **Esempio:** i task possono essere schedulati per essere eseguiti periodicamente o dopo un dato intervallo di tempo *per un tempo ben preciso*

❖ La API per i thread di Windows supporta i thread pool





OpenMP – 1

- ❖ **OpenMP** è un insieme di direttive del compilatore ed una API per programmi scritti in C, C++ e FORTRAN
- ❖ Fornisce supporto per la programmazione parallela in ambienti a memoria condivisa
- ❖ Identifica le **regioni parallele**, cioè i blocchi di codice eseguibili in parallelo

- ❖ **Esempio**

#pragma omp parallel

crea tanti thread quanti sono i core^{disponibili}, mentre
L'è processori virtuali

#pragma omp parallel for

```
for(i=0; i<N; i++) {  
    c[i]=a[i]+b[i]; }  
}
```

avuto paralleli sono dei dati, non solo task

divide il compito di esecuzione del ciclo **for** fra i diversi thread (tanti quanti sono i core)

Se non viene specificato #thread = #CPU virtuali





OpenMP – 2

- ❖ **Esempio:** si producono tante stampe quanti sono i core

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel una oper. di 5 steps
    {
        printf("I am a parallel region.");
    } come dire qual era

    /* sequential code */

    return 0;
}
```

- ❖ OpenMP permette inoltre di impostare manualmente il numero di thread ed il livello di condivisione dei dati fra i thread

Dov'è sapere quale è il codice parallelizzabile





Threading

Come si regolano processi di pari livello in relazione a:

❖ Semantica delle chiamate di sistema **fork()** ed **exec()** *cosa succede se thread riceve segnale o fa fork?*

❖ Gestione dei segnali (sincroni vs asincroni)

❖ Cancellazione dei thread (asincrona vs differita)

per terminare processo è facile, PERIL PID FULLO

❖ Dati specifici dei thread

*ma ora con i thread condiviso dati
in nome ad altri thread*

❖ Attivazione dello scheduler

es. se sono a MOLT-MOLT

Thread in LINUX





Semantica di `fork()` ed `exec()`

l'ipotesi exec,
 / sovraccarico
 sono

di solito si copia sono involontario no come padre, l'ipotesi eredita.

❖ In un programma multithread, la semantica della system call `fork()` cambia: CAMBIA COMPORTAMENTO FORK e EXEC

Se thread
 dopo fork fa una
 exec, non duplica
 risorse,

■ Se un thread in un programma invoca la `fork()`, il nuovo processo potrebbe, in generale, contenere un duplicato di tutti i thread oppure del solo thread invocante

■ La scelta fra le due opzioni dipende dalla immediatezza o meno della successiva chiamata ad `exec()` – la cui modalità di funzionamento non cambia

▶ Se la chiamata di `exec()` avviene immediatamente dopo la chiamata alla `fork()`, ~~la duplicazione dei thread non è necessaria~~, poiché il programma specificato nei parametri della `exec()` sostituirà in toto il processo ⇒ si duplica il solo thread chiamante

Se si duplicano
 tutti i thread
 del processo

Se thread dice fork e poi exec
 duplica solo un thread
 altrimenti duplica
 tutti i thread

▶ Altrimenti, tutti i thread devono essere duplicati

■ Alcune release di UNIX rendono disponibili entrambe le implementazioni della `fork()`

Solo se si duplicano tutti i processi o no.

So. deve sapere dopo cosa verrà eseguito, se c'è un exec





Gestione dei segnali – 1 *es. interazioni*

- ❖ Nei sistemi UNIX si usano segnali per comunicare ai processi il verificarsi di determinati eventi
- ❖ I segnali vengono elaborati da un gestore dei segnali *servizio del S.O.*
 - All'occorrenza di un particolare evento, si genera un segnale
 - S'invia il segnale ad un processo
 - Una volta ricevuto, il segnale deve essere gestito da un gestore, che può essere:
 - ▶ predefinito (gestito dal SO)
 - ▶ definito dall'utente ← *difficile da verificare implementati*
- ❖ I segnali possono essere...
 - sincroni, se vengono inviati allo stesso processo che ha eseguito l'operazione causa del segnale (es.: divisione per 0) *genera e riceve il segnale*
 - asincroni, se causati da eventi esterni al processo in esecuzione (es.: scadenza di un timer o $\uparrow C$)

Control - Z invece sospende, con fg, programma lo ripreso, serve

Control - C, uccide processo

4.50 Silberschatz, Galvin and Gagne





intero

Gestione dei segnali – 2

- invio un segnale* *kill - 9 = Ctrl-C*
- ❖ Nei sistemi UNIX-like, la funzione standard per l'invio di un segnale è kill(pid_t pid, int signal)
 - ❖ ~~Per ogni segnale esiste un gestore predefinito del segnale~~, che il kernel esegue in corrispondenza del particolare evento *opportune occas. richieste dal segnale*
 - Il gestore definito dall'utente, se presente, si sostituisce al gestore standard
 - Nei processi a singolo thread, i segnali vengono inviati al processo
 - ❖ Cosa accade nel caso di processi multithread?





Gestione dei segnali – 3

❖ Nei sistemi multithread è possibile...

- inviare il segnale al thread cui il segnale si riferisce *es. segnale SIGUSR1. problema, uccido un solo thread*
- inviare il segnale ad ogni thread del processo *es. Ctrl-C, uccido anche il thread*
- inviare il segnale a specifici thread del processo
- definire un thread specifico per ricevere tutti i segnali diretti al processo *thread che cattura segnali e gestisce*

*all'ho alle
vices. di
altri segm solo
altri thread*

*gestione deve
intervenire una
sola volta*

- ▶ I segnali sincroni devono essere recapitati al thread che ha generato l'evento causa del segnale
- ▶ Alcuni segnali asincroni (per esempio il segnale di terminazione di un processo, ^C) devono essere inviati a tutti i thread
- ⇒ Possibilità di specificare, per ciascun thread, quali segnali accettare e quali bloccare (`pthread_kill()`)
- ⇒ Tuttavia, poiché i segnali vanno gestiti una sola volta, il segnale è recapitato al primo thread che non lo blocca





Cancellazione di thread

devo stare attento
a non vedere dati
INCONGENTI

- ❖ È l'operazione che permette di terminare un thread prima che completi il suo compito
 - **Esempio:** pressione del pulsante di terminazione di un programma di consultazione del Web per interrompere il caricamento di una pagina
- ❖ Il thread da cancellare viene definito thread bersaglio
- ❖ La cancellazione di un thread bersaglio può avvenire...
 - ...in modalità asincrona: terminazione immediata del thread bersaglio *si trova in cui si trova, cancella*
 - ▶ Problemi se si cancella un thread mentre sta aggiornando dati che condivide con altri thread
 - ...in modalità differita: il thread bersaglio può periodicamente controllare se deve terminare, in modo da riuscire al momento opportuno (*cancellation point*, in Pthreads) *controlla in tempi ben definiti. tutti in cui se thread viene cancellato, i dati sono corrotti.*

STANDARD

PER EVITARE PROBLEMI
DI
CORRUZIONE
CHECKPOINT

devo fare una read, lo cancella prima della lettura





Cancellazione di thread in Pthread – 1

- ❖ Codice Pthread per creare e cancellare un thread

```
pthread_t tid;
```

```
/* create the thread */  
pthread_create(&tid, 0, worker, NULL);
```

```
. . .
```

```
/* cancel the thread */  
pthread_cancel(tid);
```

*← non è solo che venga fatto sul thread
dipende dallo stato*

- ❖ La chiamata della **pthread_cancel()** comporta solo una richiesta di cancellazione del thread bersaglio: l'effettiva cancellazione dipende dallo stato del thread

Mode	State	Type
Off	Disabled	-
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

*Un certo thread viene cancellato solo se non ha i flag
su stati cancellabili*





Cancellazione di thread in Pthread – 2

- ❖ In caso di cancellazione disabilitata, l'operazione rimane pendente finché il thread non procede all'abilitazione
- ❖ Per ogni thread, lo stato di default è "differita"
 - La cancellazione avviene solo al momento in cui il thread raggiunge un punto di cancellazione
 - ▶ Il punto di cancellazione viene creato con una invocazione alla funzione `pthread_testcancel()`
 - ▶ Si richiama quindi il *cleanup handler*, che permette di rilasciare tutte le risorse del thread
- ❖ Nei sistemi Linux, la cancellazione dei thread è gestita tramite segnali





Dati specifici dei thread

Dati LOCALI al thread, analogo a Dati statici

- ❖ Il Thread Local Storage (TLS) permette a ciascun thread di possedere una propria copia dei dati (non necessariamente tutti i dati del processo di origine)

*Utile
Solo del
thread
che li
porta in*

- **Esempio:** In un sistema per transazioni, si può svolgere ciascuna transazione tramite un thread distinto ed un identificatore unico per ogni transazione *es. Banca,*

*vi è del dato
va dove quello
del singolo
thread*

- Lo stesso TID può essere considerato un dato TLS
- ❖ Utili quando non si ha controllo sul processo di creazione dei thread (per esempio, nel caso dei gruppi di thread)

- ❖ Diversamente dalle variabili locali (che sono visibili solo durante una singola chiamata di funzione), i dati TLS sono visibili attraverso tutte le chiamate

- ❖ Simili ai dati dichiarati **static** in C

- I dati TLS sono però unici per ogni thread

*Dati LOCALI alla
singola funzione
del thread ma
a tutto il
thread*





Attivazione dello scheduler – 1

thread utente > # thread kernel

❖ Sia il modello M:M che il modello a due livelli richiedono che sia mantenuta una comunicazione costante per garantire un numero sufficiente di thread del kernel allocati ad una particolare applicazione *il SO usa una metafora del*

- Si alloca una struttura dati intermedia nota come processo leggero o **LWP** (*LightWeight Process*) *processo virtuale*
- Per la libreria dei thread a livello utente, LWP è un processore virtuale a cui l'applicazione può richiedere lo scheduling di un thread a livello utente
- Corrispondenza fra LWP e thread a livello kernel: quanti LWP dovranno essere creati per una data applicazione?
- I kernel thread vanno in esecuzione sui processori fisici
 - ▶ Se un thread del kernel si blocca, l'effetto si propaga attraverso l'LWP fino al thread al livello utente ad esso associato

Kernel SO mette a disposizione di ogni App un # di processi virtuali = # thread a livello kernel che possono eseguire a quella App





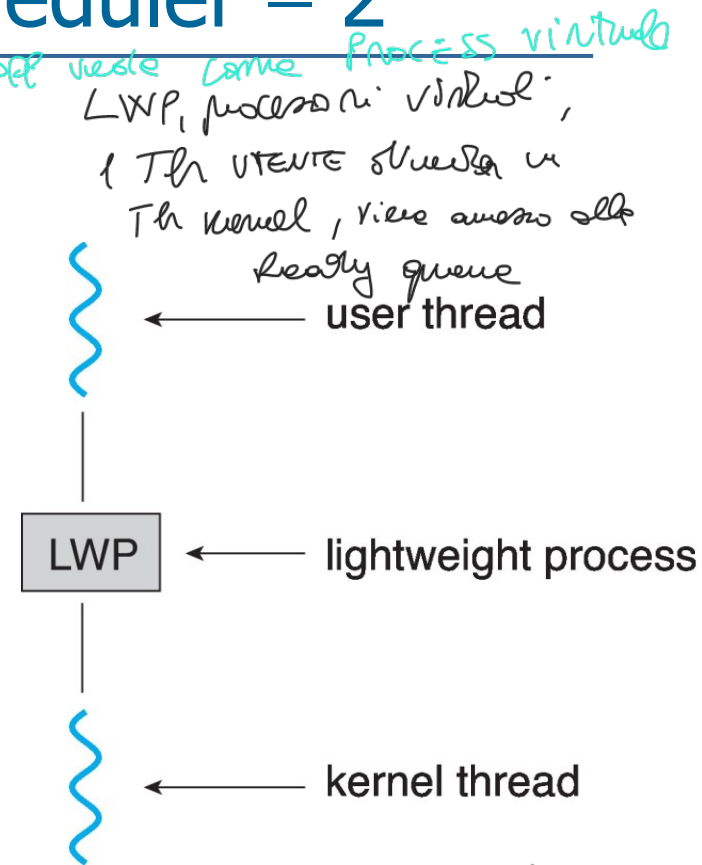
* Thread che aspetta come SCHEDULABILI non si perdono toller
archi Retruva e del program

Attivazione dello scheduler – 2

- KERNEL MENTE A DISPOSIZIONE LWP che per veste come process virtuali*
- ❖ L'attivazione dello scheduler mette a disposizione la procedura di *Cl'azione in su* upcall – un meccanismo di comunicazione dal kernel alla libreria di gestione dei thread

- Si informa l'applicazione che un suo thread sta per bloccarsi
- Si assegna all'applicazione un nuovo processore virtuale su cui gestire il segnale (salvataggio del contesto del thread bloccante) e scheduling di un nuovo thread

- ❖ Tale comunicazione garantisce all'applicazione la capacità di mantenere attivi un numero opportuno di thread a livello kernel



Ci si accorge che

IN CASO DI BLOCCO un thread sta per bloccarsi
KERNEL AVVISA APP
COSI' PUO' ASSEGNARE
NUOVO THREAD UTENTE AL LWP

quello kernel si rende conto che un thread è importante e richiede operos. / 10,
per avere che si sta per bloccare e
che un altro LWP, manda un messaggio per averne che si sta per bloccare e
4.58 deve scoprirne un altro





L'utente si divide in altro thread e non LWP. così lo segue in base merito di thread kernel admin

Thread in Linux – 1

Implementazione LINUX

sta facendo che thread

- ❖ Linux li definisce task (come i processi) usa una Task struct con struttura dati.
- ❖ La creazione dei task avviene con la chiamata della system call clone() rispetto alla fork si prende dalle flag
- ❖ All'invocazione, clone() riceve come parametro un insieme di indicatori (flag), che definiscono quante e quali risorse del task genitore saranno condivise dal task figlio
USO TASK STRUCT
nella Task struct, dati non sono condivisi, ma lo partizioni ai dati
- clone() può permettere al task figlio la condivisione totale dello spazio degli indirizzi e delle risorse del task padre: si crea un thread
può decidere cosa condividere

flag	meaning
CLONE_FS = 1	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Se flag arriva
Se Flag è arrivato vuol dire
ho creato un thread, che
condivide tutto.
Se non condivide nulla
è un processo non condiviso nulla

non appena creato
condivide FS con padre
non Task condivide spazio
memoria con chi
l'ha creato





Thread in Linux – 2

- ❖ Se nessun flag è impostato al momento dell'invocazione di `clone()`, non si ha alcuna condivisione: si ottiene una situazione analoga all'invocazione di una `fork()` *duplice
nessuna*
- ❖ Condivisione ad intensità variabile possibile perché:
 - Ogni task è rappresentato da una struttura dati nel kernel (TCB, `struct task_struct`, che punta alle strutture dati del processo, che siano esse uniche o condivise)
 - La struttura non contiene dati, ma puntatori alle strutture contenenti dati
 - La chiamata a `fork()` duplica tutte le strutture dati del task genitore *deciso se condividere puntatori o no*
 - Con `clone()` il nuovo task non riceve una copia di tutte le strutture dati, ma solo alcuni puntatori a tali strutture (quelle del padre), in base ai parametri di chiamata di `clone()`



Fine del Capitolo 4

