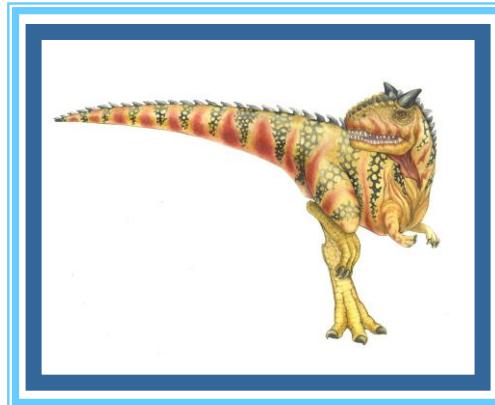


Thread e concorrenza





Obiettivi

- ❖ Introdurre la nozione di thread – l'unità fondamentale di utilizzo della CPU nei computer che supportano il multithreading
- ❖ Descrivere i principali vantaggi e le problematiche più significative della progettazione di processi multithread
- ❖ Discutere le API per le librerie di thread, con particolare riferimento a Pthreads
- ❖ Presentare approcci al threading implicito
- ❖ Definire il supporto di Linux ai thread

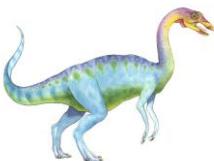




Sommario

- ❖ 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:** allocazione dello spazio degli indirizzi e successiva popolazione
 - **Context–switch:** salvataggio e ripristino degli spazi di indirizzamento (codice, dati, stack) di due processi





Motivazioni – 2

- ❖ 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 ⇒ “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

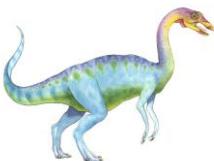




Motivazioni – 3

- ❖ **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
 - ⇒ 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
 - I kernel attuali sono normalmente multithread
 - Il multithreading si adatta perfettamente alla programmazione nei sistemi multicore

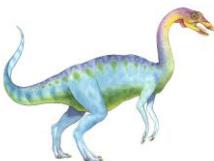




Definizione – 1

- ❖ 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)
- ❖ Un processo tradizionale – *heavyweight process* – è composto da un solo thread
 - Un processo è “pesante”, in riferimento al contesto (spazio di indirizzamento, stato) che lo definisce





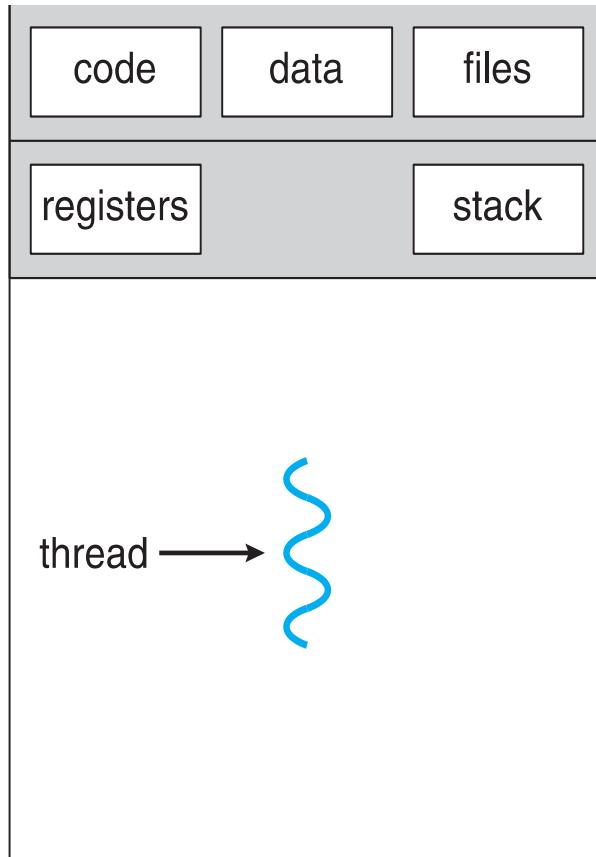
Definizione – 2

- ❖ 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

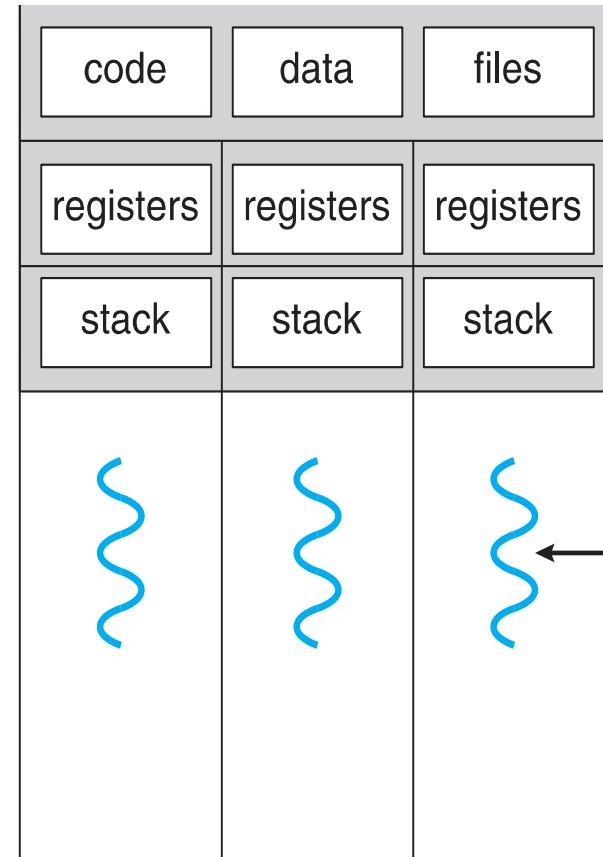




Processi a singolo thread e multithread

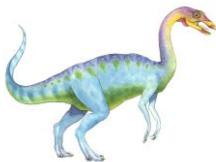


single-threaded process



multithreaded process





Esempi – 1

❖ Browser web

- Un thread per la rappresentazione sullo schermo di immagini e testo
- Un thread per il reperimento dell'informazione in rete

❖ 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





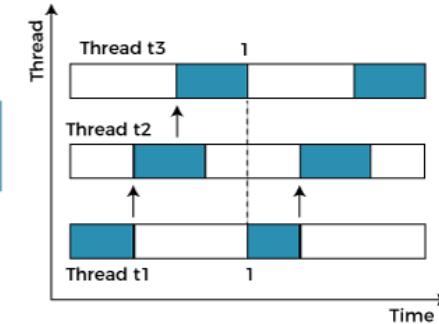
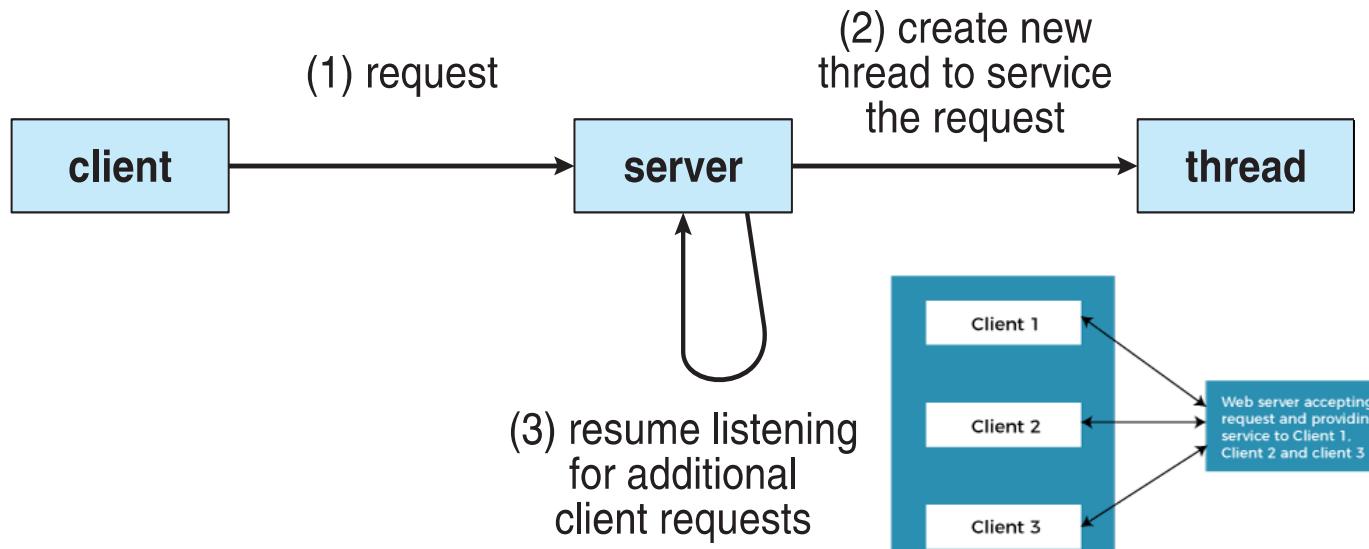
Esempi – 2

❖ Server per RPC

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

❖ Server web

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

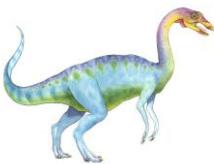




Kernel multithread

- ❖ I kernel dei sistemi operativi attuali sono perlopiù multithread
 - Thread dedicati a servizi specifici
- ❖ Solaris
 - 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

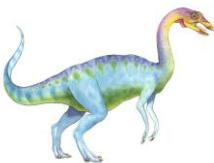
❖ Tempo di risposta

- 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
- Utile per le interfacce utente

❖ 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)



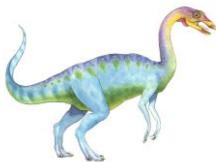


Vantaggi – 2

❖ **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)
 - ▶ **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

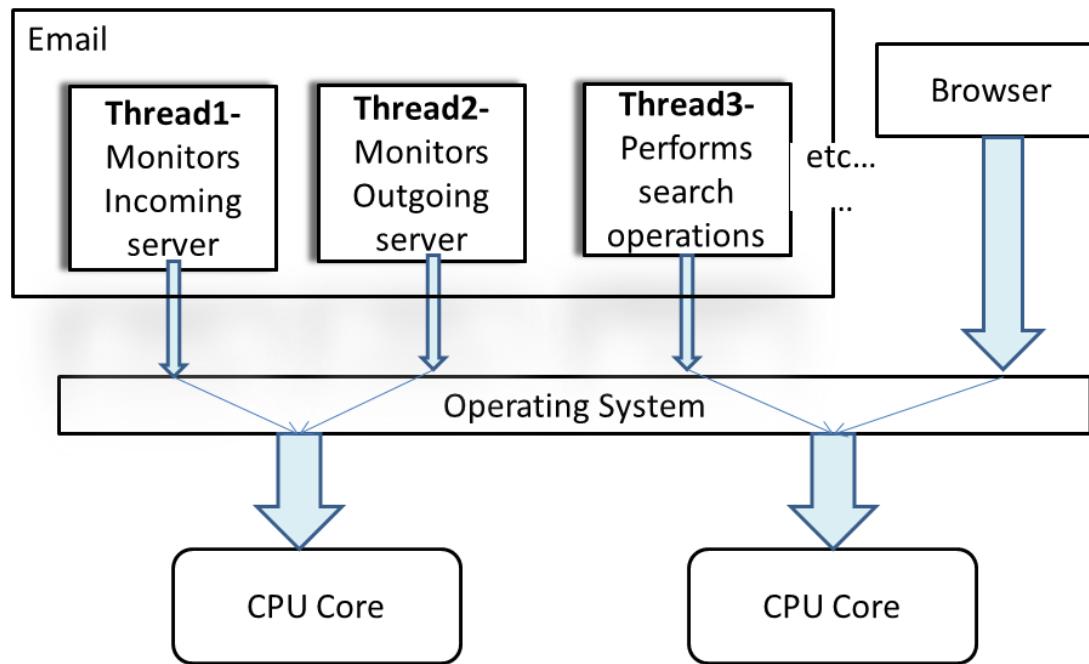




Vantaggi – 3

❖ Scalabilità

- 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





Programmazione multicore – 1

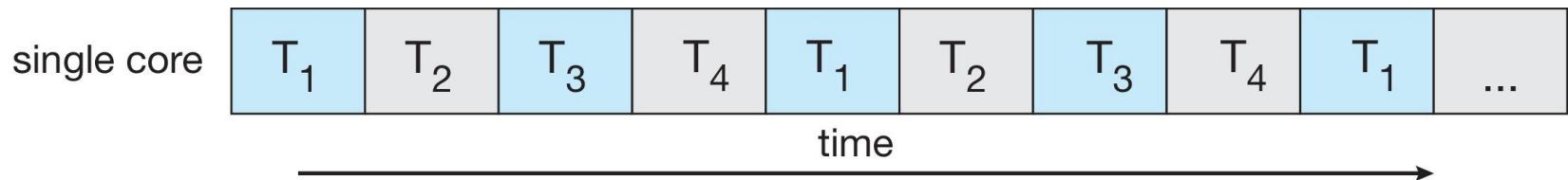
- ❖ Architetture multicore: diverse unità di calcolo sullo stesso chip
 - Ogni unità appare al SO come un processore separato
- ❖ Sui sistemi multicore i thread possono essere eseguiti in parallelo, poiché il sistema può assegnare thread diversi a ciascuna unità di calcolo
- ❖ Distinzione fra **parallelismo** e **concorrenza**
 - Un sistema è parallelo se può eseguire simultaneamente più task
 - Un sistema concorrente, invece, supporta più task, consentendo a tutti di progredire nell'esecuzione grazie al multiplexing della CPU (unica)



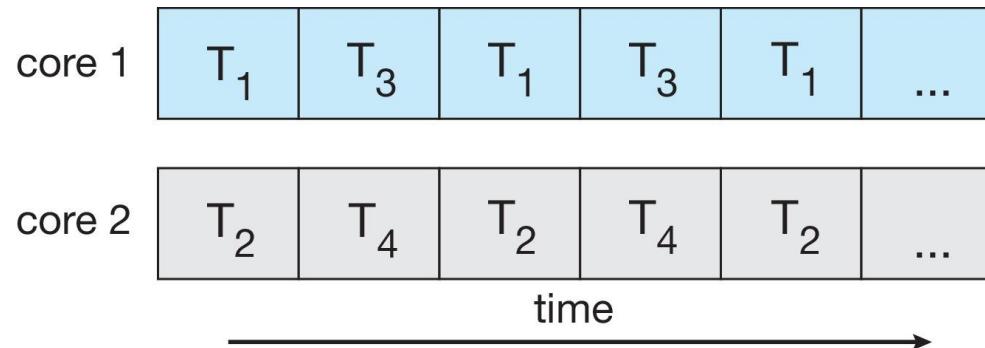


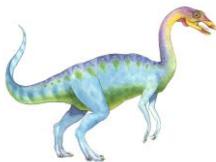
Concorrenza vs Parallelismo

- ❖ Esecuzione concorrente su un sistema single core



- ❖ Parallelismo nei sistemi multicore





Programmazione multicore – 2

❖ Tipi di parallelismo

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

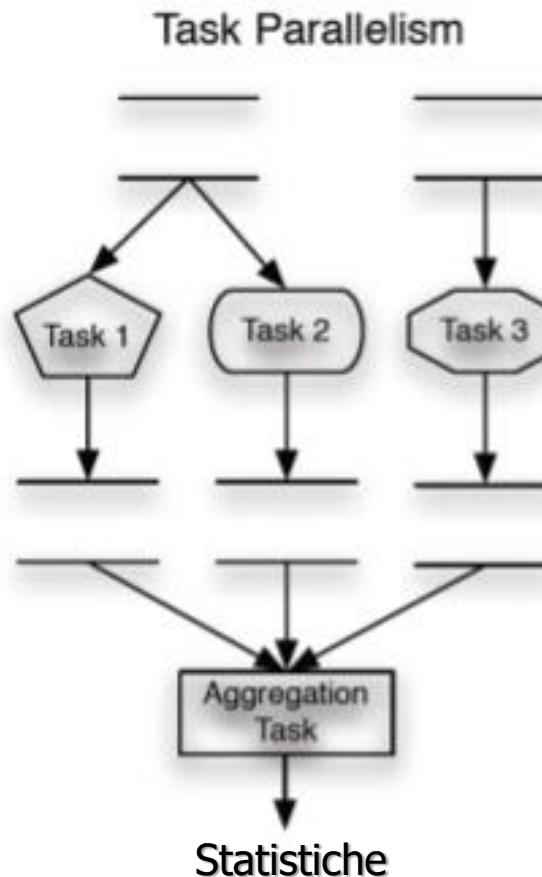
❖ Nella maggior parte dei casi, le applicazioni utilizzano un ibrido delle due strategie



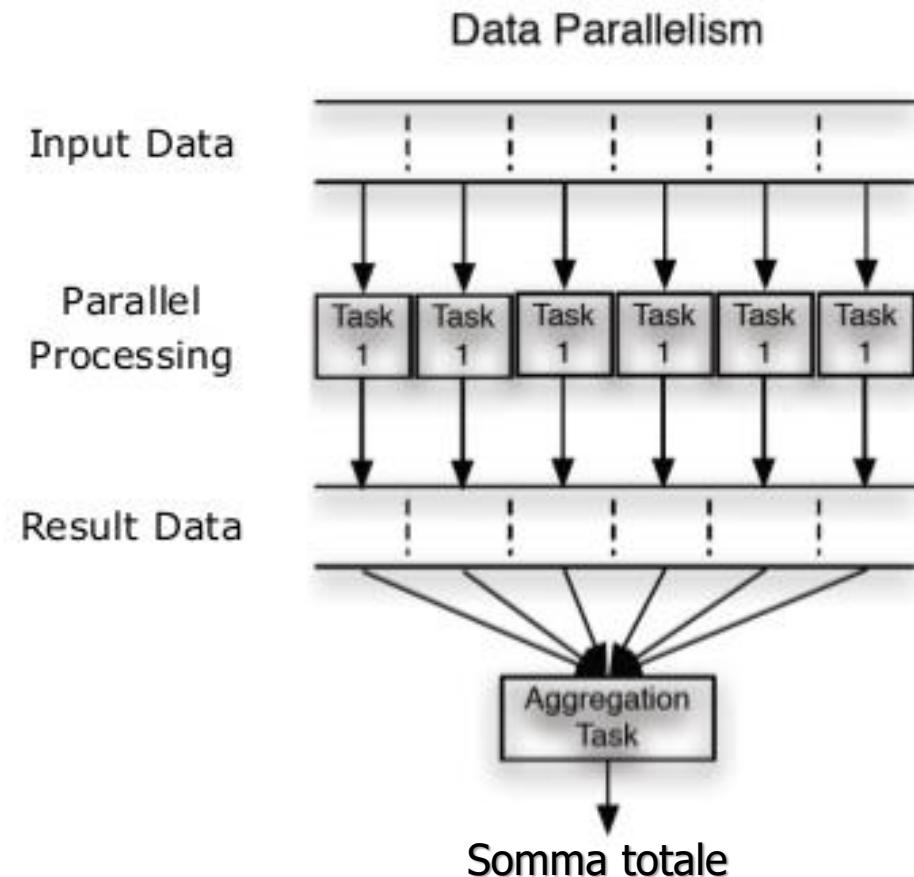


Programmazione multicore – 3

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



Calcolo della somma degli
elementi di un vettore





Programmazione multicore – 4

- ❖ Obiettivi della programmazione nei sistemi multicore
 - **Separazione dei task:** esaminare le applicazioni per individuare aree separabili in task distinti che possano essere eseguiti in parallelo
 - **Bilanciamento:** produrre task che eseguano compiti di mole confrontabile (per carico computazionale)
 - **Suddivisione dei dati:** i dati a cui i task accedono, e che manipolano, devono essere suddivisi per essere utilizzati da unità di calcolo distinte
 - **Dipendenze nei dati:** verificare le dipendenze tra due o più task per sincronizzarne l'esecuzione
 - **Test e debugging:** garantire la correttezza malgrado la possibilità di flussi di esecuzione multipli e di eventuali problemi di concorrenza nell'accesso ai dati

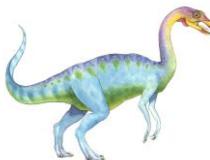




Programmazione multicore – 5

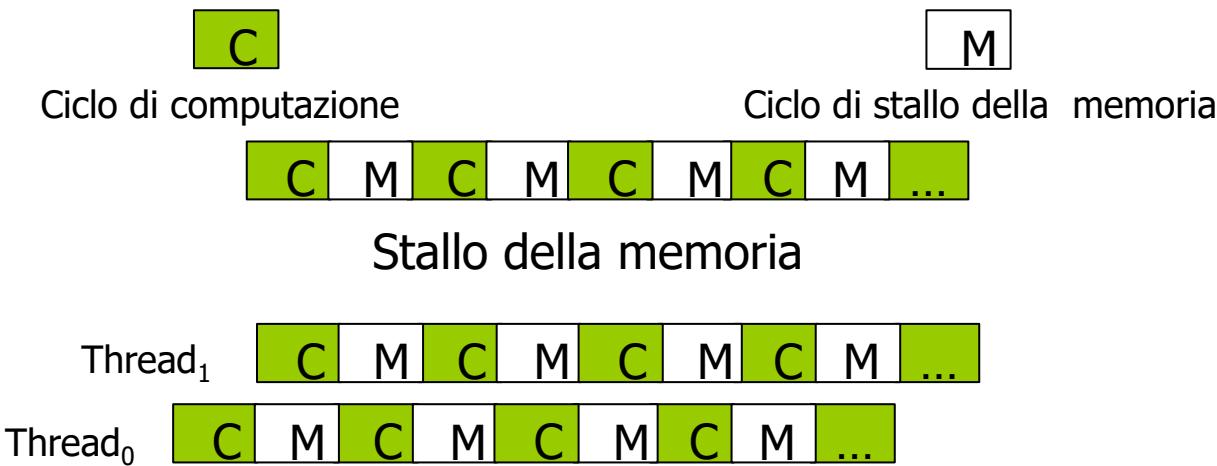
- ❖ In corrispondenza con la proliferazione di programmi multithread, le nuove architetture hardware forniscono sempre maggior supporto al multithreading
- ❖ Progetti hardware recenti (*hyperthreading*) implementano unità di calcolo multithread in cui due o più thread hardware sono assegnati ad una singola CPU
 - 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





Programmazione multicore – 6

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



Sistema con multithread hardware (relativo ad un core)





La legge di Amdahl – 1

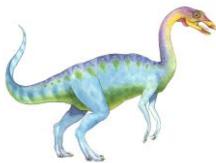
- ❖ Permette di determinare i potenziali guadagni in termini di prestazioni ottenuti dall'aggiunta di core, nel caso di applicazioni che contengano sia componenti seriali (non parallelizzabili) sia componenti parallele

- ❖ S porzione seriale
- ❖ N core

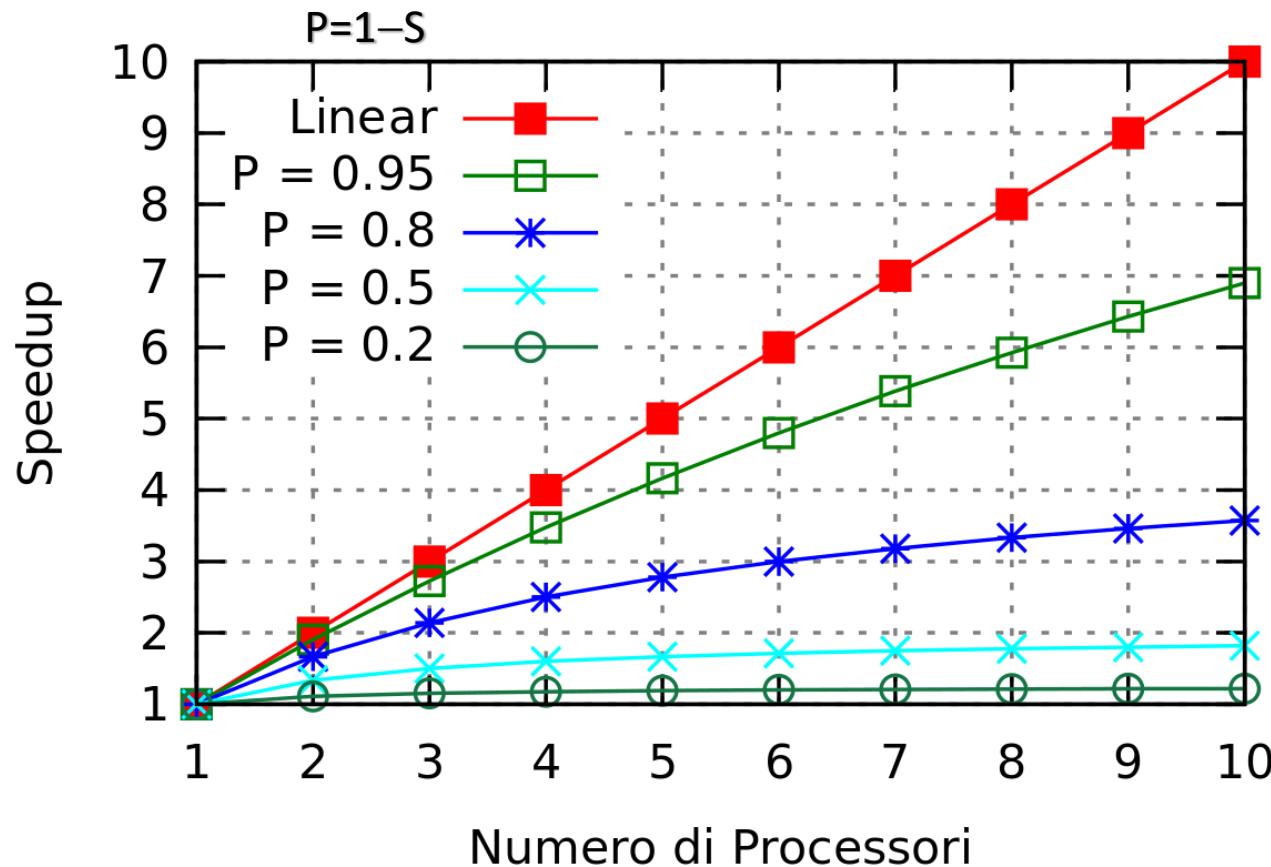
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

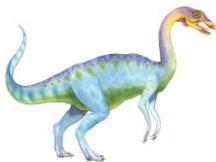
- ❖ **Esempio:** Se un'applicazione è al 75% parallela ed al 25% seriale, passando da 1 a 2 core, si ottiene uno speedup pari a 1.6; se i core sono 4, lo speedup è invece di 2.28
- ❖ In generale... per $N \rightarrow \infty$ lo speedup tende a $1/S$
 - ⇒ Se il 40% di un'applicazione è seriale il massimo aumento di velocità è di 2.5 volte
- ❖ La porzione seriale di un'applicazione ha un effetto dominante sulle prestazioni ottenibili con l'aggiunta di nuovi core





La legge di Amdahl – 2

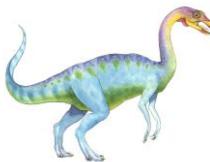




Supporto del SO ai thread – 1

- ❖ Supporto 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)
- ❖ **Thread a livello utente**
 - Sono gestiti come uno strato separato sopra il nucleo del sistema operativo
 - Sono realizzati tramite librerie di funzioni per la creazione, lo scheduling e la gestione dei thread, senza alcun intervento diretto del nucleo
 - **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

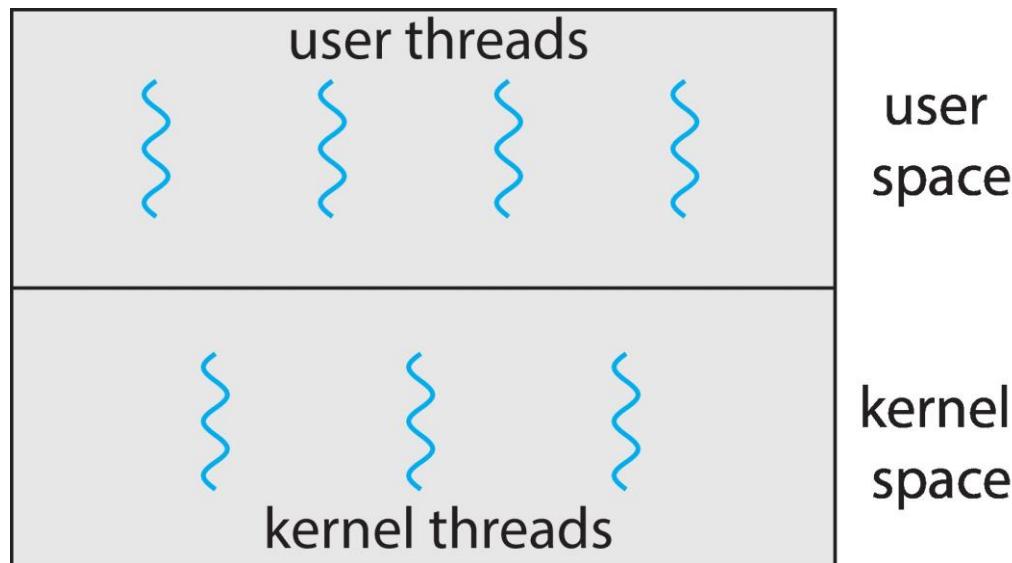
- Sono gestiti direttamente dal SO: il nucleo si occupa di creazione, scheduling, sincronizzazione e cancellazione dei thread nel suo spazio di indirizzi
- Supportati da tutti i sistemi operativi attuali
 - ▶ Windows
 - ▶ Solaris
 - ▶ Linux
 - ▶ Mac OS
 - ▶ iOS
 - ▶ Android





Modelli di programmazione multithread

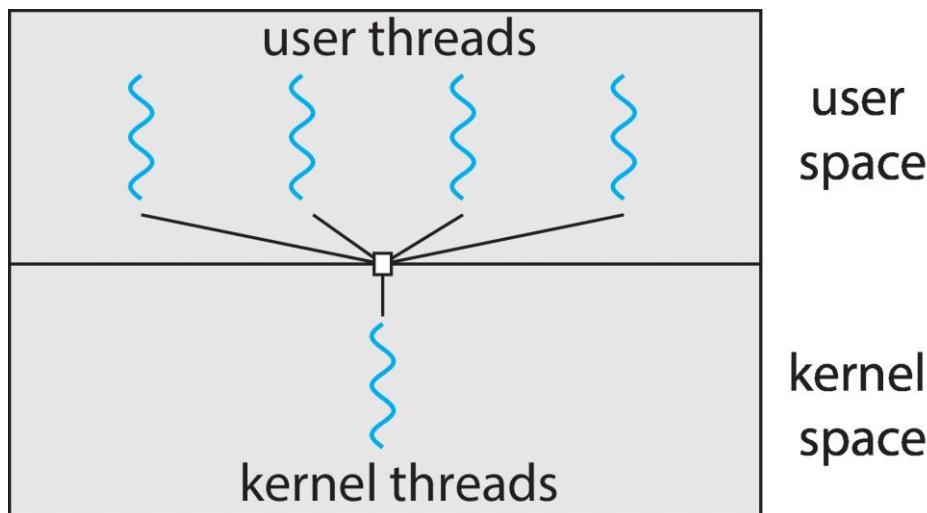
- ❖ Modello multi-a-uno (M:1)
- ❖ Modello uno-a-uno (1:1)
- ❖ Modello multi-a-molti (M:M)





Modello multi-a-uno – 1

- ❖ 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





Modello multi-a-uno – 2

❖ Vantaggi

- Gestione efficiente dei thread nello spazio utente (scheduling poco oneroso)
- Non richiede un kernel multithread per poter essere implementato

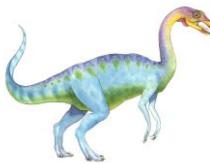
❖ Svantaggi

- 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

❖ Attualmente, pochi SO implementano questo modello

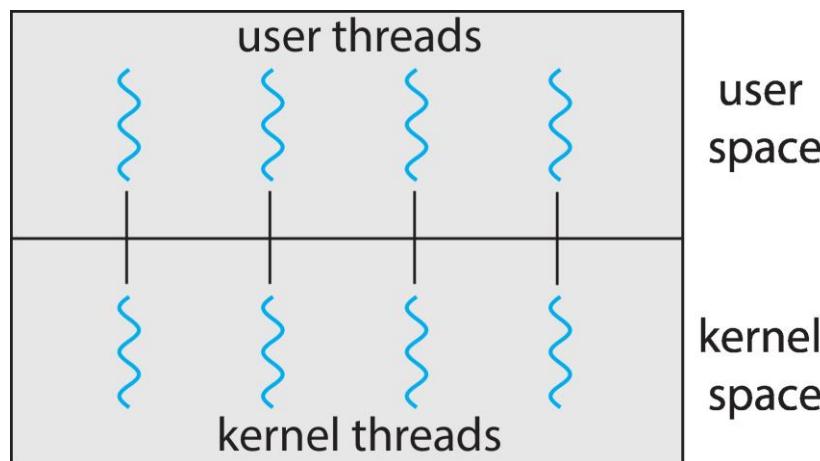
- Solaris Green Threads (2010)
- GNU Portable Threads (2006)

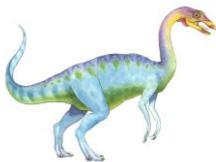




Modello uno-a-uno – 1

- ❖ 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*)





Modello uno-a-uno – 2

❖ Vantaggi

- Scheduling molto efficiente
- Se un thread effettua una chiamata bloccante, gli altri thread possono proseguire nella loro esecuzione
- I thread possono essere eseguiti su processori fisici distinti

❖ Svantaggi

- Possibile inefficienza per il carico di lavoro dovuto alla creazione di molti thread a livello kernel (limiti imposti a priori)
- Richiede un kernel multithread per poter essere implementato

❖ Esempi

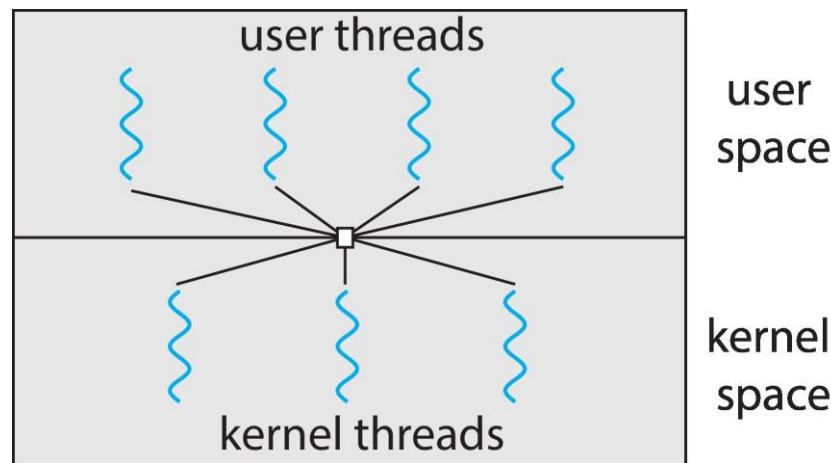
- Windows 95/98/NT/2000/XP/Vista e versioni attuali
- Linux, Solaris (versione 9 e successive)





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
 - 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





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:

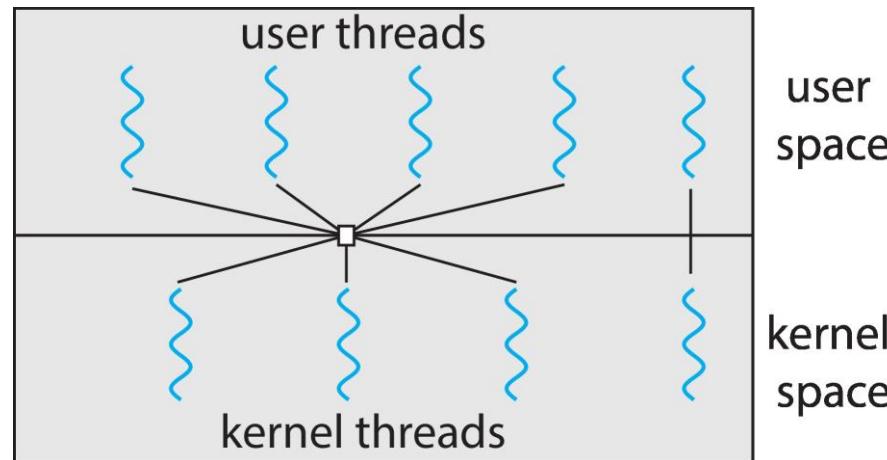
- Windows con il ThreadFiber package
- Solaris (versioni precedenti alla 9)
- Tru64 UNIX

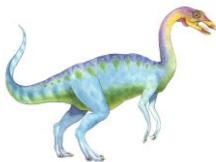




Modello a due livelli

- ❖ 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

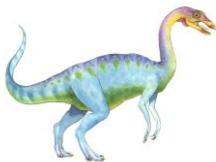




Librerie dei thread – 1

- ❖ Forniscono al programmatore una API per la creazione e la gestione dei thread
- ❖ **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





Librerie dei thread – 2

❖ **POSIX Pthreads**

- Può presentarsi sia come libreria a livello utente sia a livello kernel

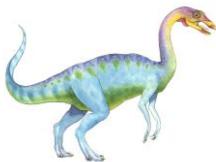
❖ **Win64**

- 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

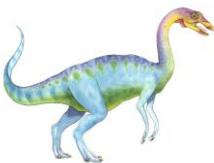




Librerie dei thread – 3

- ❖ 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
- ❖ I dati locali di una funzione sono, normalmente, memorizzati nello stack
- ❖ Ogni thread ha un proprio stack
⇒ ogni thread ha la propria copia di dati locali





Strategie di threading

❖ Threading asincrono

- Il genitore crea un figlio e quindi riprende la propria esecuzione in concorrenza
- Ogni thread viene eseguito in modo indipendente rispetto agli altri ed il thread genitore non ha bisogno di sapere quando terminano i figli
- Scarsa condivisione dei dati fra i thread
- **Esempio:** server web, interfacce utente responsive

❖ Threading sincrono

- Il genitore crea uno o più figli e attende che tutti terminino prima di riprendere l'esecuzione \Rightarrow **fork-join**
- I thread figli si eseguono in concorrenza
- Quando un thread ha terminato il proprio compito, termina e si unisce al genitore
- Significativa condivisione dei dati fra i thread
- Il genitore può combinare i risultati calcolati dai figli

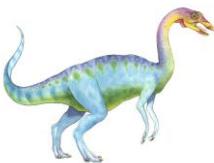




Pthreads

- ❖ È lo standard POSIX (IEEE 1003.1c) che definisce la API per la creazione e la sincronizzazione dei thread (*definizione – no realizzazione*)
- ❖ 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





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) */
void *somma(char *param); /* funzione del nuovo thread */

int main(int argc, char *argv[])
{
    pthread_t tid;          /* identificatore del thread */
    pthread_attr_t attr; /* attributi del thread */
```

if (argc != 2) {
 fprintf(stderr, "Occorreva inserire N!");
 exit(1);
}
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 */
/* il padre attende la terminazione del nuovo thread */
pthread_join(tid, NULL);
printf("somma = %d\n", sum);
exit(0);
}

Dimensione della pila, informazioni per lo scheduling, etc.



Esempio con Pthreads – 2

```
/* 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);
}
```





Join in Pthreads

```
#define NUM_THREADS 10

/* 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);
```



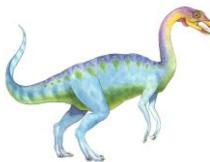


Threading implicito

- ❖ Le applicazioni multithread sono più difficili da programmare ed aumentano in complessità al crescere del numero dei thread
 - ❖ Trasferimento della creazione e della gestione del threading dagli sviluppatori di applicazioni ai compilatori ed alle librerie di runtime
- ⇒ **Threading implicito**

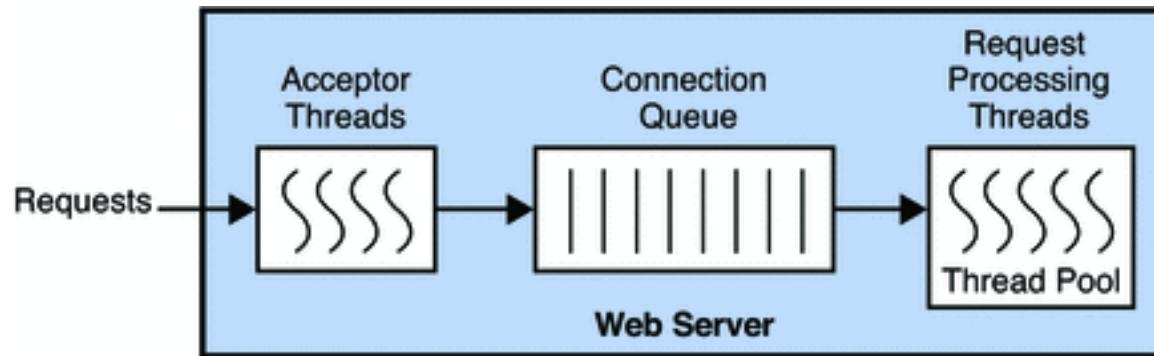
- Gruppi di thread (Thread pool – Windows, Java)
- OpenMP
- Grand Central Dispatch (Mac OS e iOS), Java Fork–Join, Intel Threading Building Blocks





Gruppi di thread – 1

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





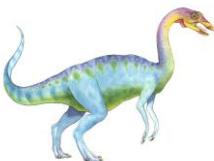
Gruppi di thread – 2

❖ Vantaggi

- 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)
- Separare il task da eseguire dal meccanismo di creazione dello stesso permette strategie diverse di elaborazione
 - ▶ **Esempio:** i task possono essere schedulati per essere eseguiti periodicamente o dopo un dato intervallo di tempo

❖ 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, mentre

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

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





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
    {
        printf("I am a parallel region.");
    }

    /* sequential code */

    return 0;
}
```

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

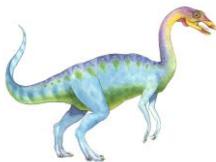




Threading

- ❖ Semantica delle chiamate di sistema **fork()** ed **exec()**
- ❖ Gestione dei segnali (sincroni vs asincroni)
- ❖ Cancellazione dei thread (asincrona vs differita)
- ❖ Dati specifici dei thread
- ❖ Attivazione dello scheduler

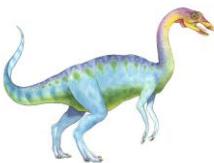




Semantica di **fork()** ed **exec()**

- ❖ In un programma multithread, la semantica della system call **fork()** cambia:
 - 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
 - ▶ Altrimenti, tutti i thread devono essere duplicati
 - Alcune release di UNIX rendono disponibili entrambe le implementazioni della **fork()**

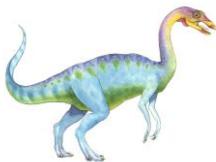




Gestione dei segnali – 1

- ❖ Nei sistemi UNIX si usano **segnali** per comunicare ai processi il verificarsi di determinati eventi
- ❖ I segnali vengono elaborati da un **gestore dei segnali**
 - 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
- ❖ I segnali possono essere...
 - **sincroni**, se vengono inviati allo stesso processo che ha eseguito l'operazione causa del segnale (es.: divisione per 0)
 - **asincroni**, se causati da eventi esterni al processo in esecuzione (es.: scadenza di un timer o ^C)

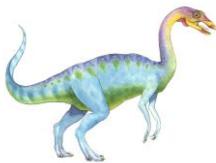




Gestione dei segnali – 2

- ❖ 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
 - 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?



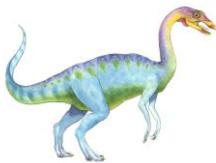


Gestione dei segnali – 3

- ❖ Nei sistemi multithread è possibile...

- inviare il segnale al thread cui il segnale si riferisce
- inviare il segnale ad ogni thread del processo
- inviare il segnale a specifici thread del processo
- definire un thread specifico per ricevere tutti i segnali diretti al processo
 - ▶ 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

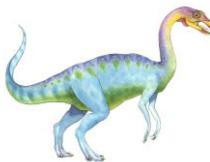




Cancellazione di thread

- ❖ 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
 - ▶ 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)





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);
```

- ❖ 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

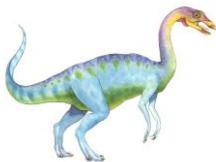




Cancellazione di thread in Pthread – 2

- ❖ 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

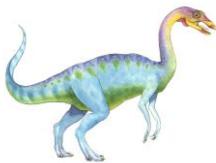




Dati specifici dei thread

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





Attivazione dello scheduler – 1

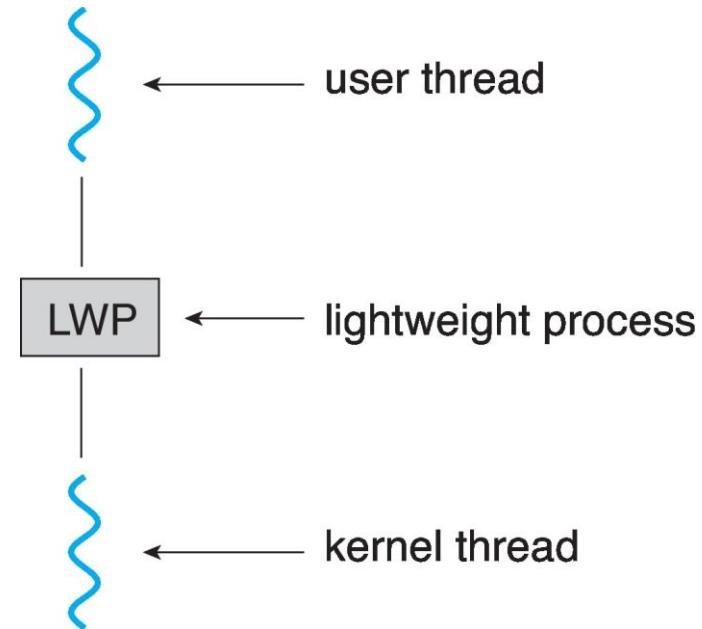
- ❖ Sia il modello M:M che il modello a due livelli richiedono che sia mantenuta una comunicazione costante per garantire un numero sufficiente di thread del kernel allocati ad una particolare applicazione
 - Si alloca una struttura dati intermedia nota come processo leggero o **LWP** (*LightWeight Process*)
 - 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





Attivazione dello scheduler – 2

- ❖ L'attivazione dello scheduler mette a disposizione la procedura di *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



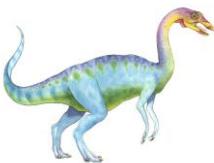


Thread in Linux – 1

- ❖ Linux li definisce **task** (come i processi)
- ❖ La creazione dei task avviene con la chiamata della system call **clone()**
- ❖ 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
 - **clone()** può permettere al task figlio la condivisione totale dello spazio degli indirizzi e delle risorse del task padre: si crea un thread

flag	meaning
CLONE_FS	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.





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()**
- ❖ Condivisione ad intensità variabile 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
 - 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

