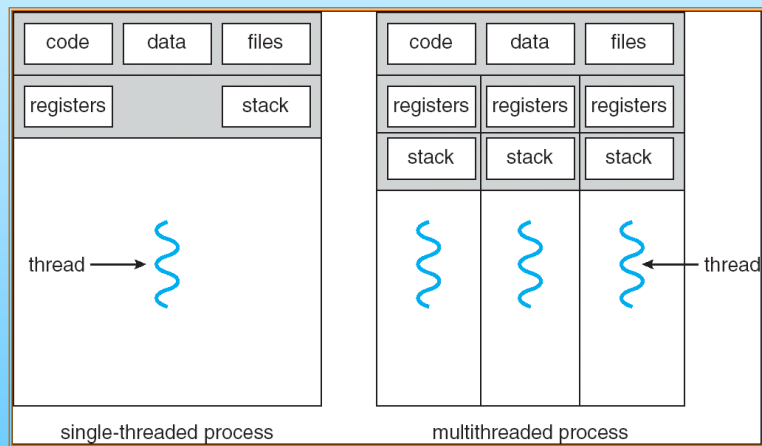


# Threads

## Processi Single e Multi-thread



## Benefici

- **Diminuisce i tempi di risposta di una applicazione:** durante attesa altre attività della applicazione possono essere fatte
- **Condivisione delle risorse:** tutti i thread accedono alla stessa memoria e file
- **Economia:** creazione thread costa meno di creazione processo, context switch ha costo minore
- **Utilizzazione di architettura multi processore:** possibilità di effettivo parallelismo

Sistemi Operativi A.A. 2016/2017

## User/Kernel Thread

- **User Thread:** Gestione dei thread fatto da libreria gestione thread eseguibile a livello utente (no kernel), scheduling fatto dalla applicazione stessa.
- **Kernel Thread:** thread gestiti dal sistema
  - tutti i sistemi operativi recenti supportano il multi-threading

Sistemi Operativi A.A. 2016/2017

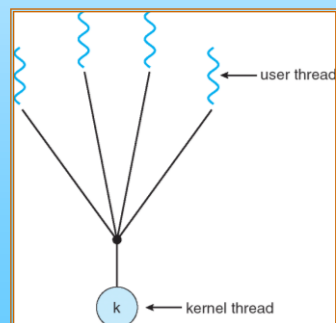
## Modelli di Multithreading

- Differenti tipi di gestione dei thread in base alla relazione tra numero di thread a livello utente e numero di thread a livello kernel
  - **Molti-a-uno**, molti thread utente gestiti da un thread kernel
  - **Uno-a-uno**, un thread utente gestito da un thread kernel
  - **Molti-a-molti**, molti thread utente gestiti da molti thread kernel

Sistemi Operativi A.A. 2016/2017

## Molti-a-uno

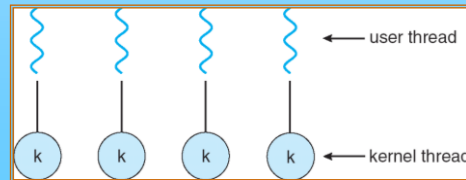
- Molti thread utente gestiti da un singolo thread kernel
- Pro:
  - possono gestire thread in modo efficiente
- Contro:
  - si bloccano su chiamate di sistema bloccanti,
  - non usano più core
  - non pre-empitive, CPU rilasciata esplicitamente,
- Esempi:
  - GNU Portable Threads, Solaris Green Threads



Sistemi Operativi A.A. 2016/2017

## Uno-a-uno

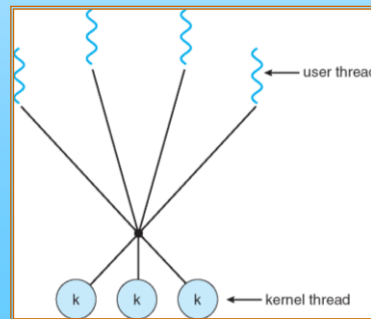
- Ogni thread a livello utente gestito da un thread del kernel
- Pro:
  - gestiti dal sistema non si bloccano su chiamate di sistema bloccanti
  - possono sfruttare parallelismo dei multi-core
- Contro:
  - limitato numero di thread gestibili dal sistema
- Esempi:
  - Windows da NT in poi
  - Linux
  - Solaris 9 e successivi



Sistemi Operativi A.A 2016/2017

## Multi-a-molti

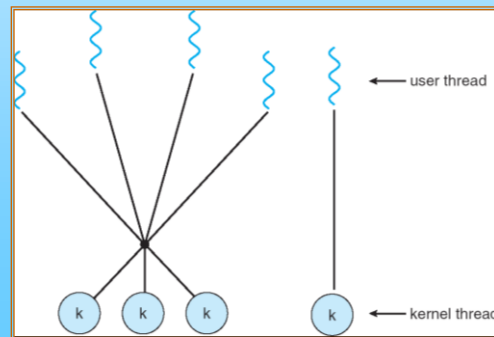
- Permette a molti thread utente di essere gestiti tramite molti thread kernel
- Permette al sistema operativo di creare un numero sufficiente di thread kernel
- Esempi:
  - Solaris prima della versione 9
  - Windows *Fiber* o da Windows 7 User-Mode Scheduling



Sistemi Operativi A.A 2016/2017

## Modello a due livelli

- Simile a modello Multi-a-molti, eccetto che permette ad un thread utente di essere gestito anche con un solo thread del kernel
- Esempi
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 e versioni precedenti



Sistemi Operativi A.A. 2016/2017

## Problematiche programmazione multithread

- Chiamate fork & exec
- Terminazione dei thread
- Dati specifici dei thread

Sistemi Operativi A.A. 2016/2017

## UNIX: Semantica di `fork()` e `exec()`

- La chiamata di sistema **`fork()`** effettuata su un thread di un processo multithread duplica solo il thread chiamante o tutti i thread?
  - **Linux:** duplica solo il thread chiamante, ma può generare errori in quanto gli altri thread possono lasciare memoria del nuovo processo in stato inconsistente e generare problemi → si evita

Sistemi Operativi A.A. 2016/2017

## Terminazione dei Thread

- Terminare un thread prima che abbia terminato
- Due approcci generali:
  - **Terminazione asincrona** termina il thread immediatamente (potrebbe non rilasciare le risorse acquisite dal thread) – da evitare
  - **Terminazione ritardata** il thread controlla periodicamente se deve essere terminato (preferibile)

Sistemi Operativi A.A. 2016/2017

## Dati specifici dei thread

- I thread condividono la memoria 'globale' ma hanno uno stack proprio per le variabili locali
- Può essere utile avere delle variabili globali associate ad ogni thread
- Alcuni sistemi operativi forniscono la memoria TLS (Thread Local Storage), una zona di memoria globale associata al thread

Sistemi Operativi A.A. 2016/2017

## Linux Threads

- Linux usa il termine *task* piuttosto che *thread*
- La creazione di un thread è fatta tramite la chiamata di sistema **clone()** che viene usata anche da **fork()**
- **clone()** permette a un task figlio di condividere lo spazio degli indirizzi del task padre (processo)

Sistemi Operativi A.A. 2016/2017

## Pthreads

- E' una API standard POSIX (IEEE 1003.1c) per la creazione e sincronizzazione di thread
- La API standard specifica il comportamento della libreria per la gestione dei thread, la implementazione è realizzata da chi sviluppa la libreria
- Comune nei sistemi operativi UNIX (Solaris, Linux, Mac OS X)
- <http://pubs.opengroup.org/onlinepubs/007908775/xsh/pthread.h.html>

Sistemi Operativi A.A 2016/2017

## pthread\_create

- Crea un thread associato ad una funzione e lo mette in esecuzione

```
#include <pthread.h>
...
void* thread_function(void* p) { ... } //funzione da eseguire in thread
...
pthread_t tid; //identificatore del task
pthread_attr_t attr; //attributi per la creazione del thread
...
pthread_attr_init(&attr); //inizializza con attributi di default
void* param = ... //parametro passato a funzione thread
int r = pthread_create(&tid, &attr, thread_function, param);
//ritorna 0 se OK
```

Sistemi Operativi A.A 2016/2017



## pthread\_join

- Attende la terminazione di un thread e permette di ottenere il valore ritornato dalla funzione associata al thread.

```
void * ret_val;
```

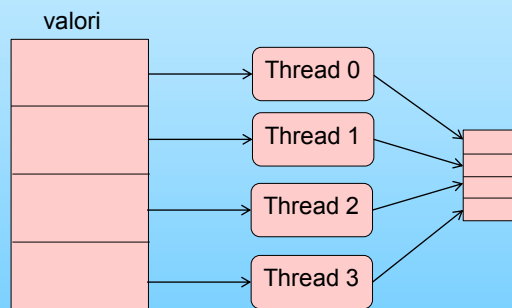
```
pthread_join(tid, &ret_val); //aspetta terminazione di tid e  
mette in ret_val il valore ritornato dalla funzione
```

```
pthread_join(tid, NULL); //aspetta terminazione di tid
```

Sistemi Operativi A.A 2016/2017

## Esempio pthread

- Sommare N valori in un vettore usando 4 thread, ogni thread somma N/4 valori, il risultato si ottiene sommando i 4 valori



Sistemi Operativi A.A 2016/2017

## Esempio Pthreads

```
#include <pthread.h>
#define N 100000
#define NTH 4 //numero thread
int values[N]; //valori da sommare
int sums[NTH]; //somme parziali

void* sum_thread(void* t) {
    int s=0, th, i;
    th=((int*)t); //numero del thread 0, 1, 2, 3
    for(i=th*N/NTH; i<(th+1)*N/NTH; i++)
        s+=values[i];
    sums[th]=s;
    return NULL;
}
```

Sistemi Operativi A.A 2016/2017

## Esempio Pthreads

```
int main() {
    pthread_t tids[NTH];
    int i, s, thn[NTH];
    for(i=0; i<N; i++) // inizializza il vettore con 0, 1, 2, ... N-1
        values[i] = i;
    for(i=0; i<NTH; i++) {
        thn[i] = i;
        pthread_create(&tids[i], NULL, sum_thread, &thn[i]);
    }
    for(i=0; i<NTH; i++) {
        pthread_join(&tids[i], NULL);
        s += sum[i];
    }
    printf("somma: %d\n", s);
}
```

Non usare &i  
Perché?

Sistemi Operativi A.A 2016/2017

## Java Threads

- I thread Java sono gestiti dalla Java Virtual Machine
- I thread java possono essere creati:
  - Estendendo la classe `Thread`
  - Implementando la interfaccia *Runnable*

Sistemi Operativi A.A. 2016/2017

## Classe Thread

- Estendere classe **Thread** e ridefinire metodo **run()** che esegue il codice da eseguire su thread separato
- il metodo **start()** fa partire il thread che eseguirà il metodo `run()`
- il metodo **join()** aspetta che il thread termini, genera eccezione *InterruptedException* se il thread stesso viene interrotto con metodo **interrupt()** (esiste versione `join` che aspetta al massimo un certo numero di millisecondi)
- il metodo statico **Thread.sleep(n)** attende `n` millisecondi, può generare eccezione *InterruptedException* se interrotto

Sistemi Operativi A.A. 2016/2017

## Esempio

```
public class CountDownThread extends Thread {
    public void run() {
        try {
            for(int i=10; i>=0; i--) {
                System.out.println(i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException ex) {}
    }
}

...
CountDownThread cdt = new CountDownThread();
cdt.start(); //parte il thread
cdt.join(); // attende terminazione del thread
```

Sistemi Operativi A.A 2016/2017

## Interfaccia Runnable

- Altrimenti si può creare una classe che implementa l'interfaccia **Runnable** che contiene solo il metodo pubblico astratto run()
- Quindi si crea un oggetto della classe Thread con parametro l'oggetto che implementa l'interfaccia Runnable:

```
public class CountDown implements Runnable {
    ...
    public void run() { ... }
}

...
CountDown countDown = new CountDown();
Thread t = new Thread(countDown);
t.start(); // fa partire il thread ed esegue runObj.run();
...
t.join(); //attende la terminazione del thread
```

Sistemi Operativi A.A 2016/2017

## Terminare un thread

- esiste metodo **stop()** per terminare un thread in modo asincrono ma è deprecato per i problemi che può potenzialmente introdurre (risorse non rilasciate)
- la soluzione più sicura è usare metodo **interrupt()** che non termina il thread ma indica che il thread deve essere terminato.
  - Se il thread è in attesa con `sleep`, `join` etc. viene generata eccezione *InterruptedException* che indica che il thread deve essere interrotto
  - Se invece il thread non entra mai in attesa può essere controllato regolarmente lo stato tramite metodo *Thread.interrupted()*. Di solito la soluzione migliore è lanciare eccezione *InterruptedException*:

```
void run() {  
    while(...) {  
        ...  
        if(Thread.interrupted())  
            throw new InterruptedException();  
    }  
}
```

Sistemi Operativi A.A. 2016/2017

## Esempio terminazione

```
CountDownLatch cdt = new CountDownLatch(1);  
cdt.start();  
Thread.sleep(3500);  
cdt.interrupt();
```

Sistemi Operativi A.A. 2016/2017

## Thread

- il metodo **boolean isAlive()** indica se il thread è vivo o meno.
- Prima di chiamare start si può impostare se il thread è un daemon o no (per default non lo è) tramite metodo `setDaemon(boolean)`, la JVM termina solo quando sono terminati tutti i thread che non sono daemon
- il metodo **yield()** serve a rilasciare la CPU per eseguire il prossimo thread ready.
- Si può impostare la priorità del thread con un valore tra 1 e 10 (1 minimo, 10 massimo e 5 default)
- La politica di esecuzione è che vengono eseguiti prima tutti i thread a più alta priorità quando la coda dei thread pronti a più alta priorità è vuota passa alla coda a priorità subito inferiore etc. (ma questo dipende dalla implementazione della JVM)
- Di solito l'esecuzione dei thread è time-sliced (ma dipende da implementazione della JVM)

Sistemi Operativi A.A. 2016/2017

## Stato Thread

- Ogni thread ha uno stato che può essere:
  - **NEW**, thread creato ma non partito
  - **RUNNABLE**, thread in esecuzione o pronto
  - **BLOCKED**, thread bloccato in attesa lock accesso monitor (vedi dopo)
  - **WAITING**, thread in attesa indefinita
  - **TIMED\_WAITING**, thread in attesa temporizzata (sleep)
  - **TERMINATED**, thread terminato
- Transizioni:
  - NEW → RUNNABLE
  - RUNNABLE ↔ BLOCKED
  - RUNNABLE ↔ WAITING
  - RUNNABLE ↔ TIMED\_WAITING
  - RUNNABLE → TERMINATED

Sistemi Operativi A.A. 2016/2017

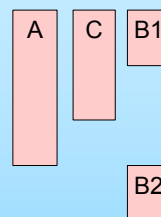
## Esercizio

- Implementare la somma multi-thread di un vettore di valori interi
- Ricerca multithread di un valore in un vettore di interi (non ordinati)

Sistemi Operativi A.A. 2016/2017

## Esercizio

- Tre attività A, B e C
- B è composta da due fasi B1 e B2
- B2 usa risultati di A, C e B1



Sistemi Operativi A.A. 2016/2017

## Esercizio

- A e B sono due attività periodiche
- Quando A/B termina viene avviata nuovamente ma solo se anche l'altra è terminata

