Threads

I thread

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Processi: Caratteristiche

- Un processo può eseguire altri processi attraverso
  - Clonazione (UNIX, **fork**)
  - Sostituzione del processo attuale con un altro programma (UNIX, **exec**)
  - Chiamata esplicita (Windows, **CreateProcess**)
- Ogni processo ha
  - Uno spazio di indirizzamento proprio
  - Una singola traccia di esecuzione (un unico program counter)

Processi: Limiti

- La clonazione implica
  - Un aumento sensibile della memoria utilizzata
  - Tempi di creazione relativamente elevati
  - Sincronizzazione e trasferimento di dati
    - Costi nulli o minimi per processi indipendenti o quasi
    - Elevati per processi cooperanti
  - La gestione di processi multipli richiede
    - Scheduling complesso
    - Operazioni di context-switching costose (con intervento del kernel)

Processo (standard) = processo **pesante**
(heavyweight process) = task con un solo thread

Dai processi ai thread

- Esistono diversi casi in cui sarebbe utile avere
  - Costi minori di creazione, gestione, etc.
  - Un unico spazio di indirizzamento
  - Tracce multiple di controllo/esecuzione (concorrenti) all'interno di tale spazio di indirizzamento
- Esempio
  - Applicazione WEB
    - Un server risponde a innumerevoli richieste di accesso
    - Le richieste sono contemporanee, simili, richiedono manipolazione degli stessi dati, etc.

Dai processi ai thread

- Lo standard IEEE/ANSI POSIX 1003.1c [1996] introduce il concetto di **thread**
  - Un thread è una sezione di un processo che viene **schedulata e eseguita indipendentemente** dal processo (thread) che l'ha generata
  - Un thread **condivide** il proprio **spazio di indirizzamento** con gli altri thread

Processo = unità che raggruppa risorse
Thread = unità di schedulazione della CPU

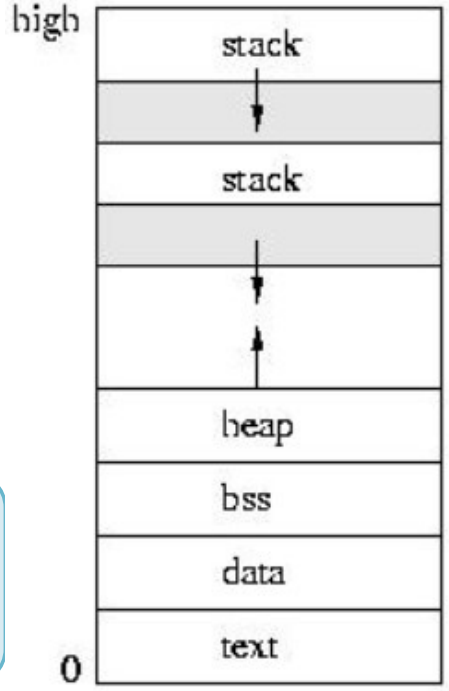
Thread = processo **leggero (ligthweight process)**

Dai processi ai thread

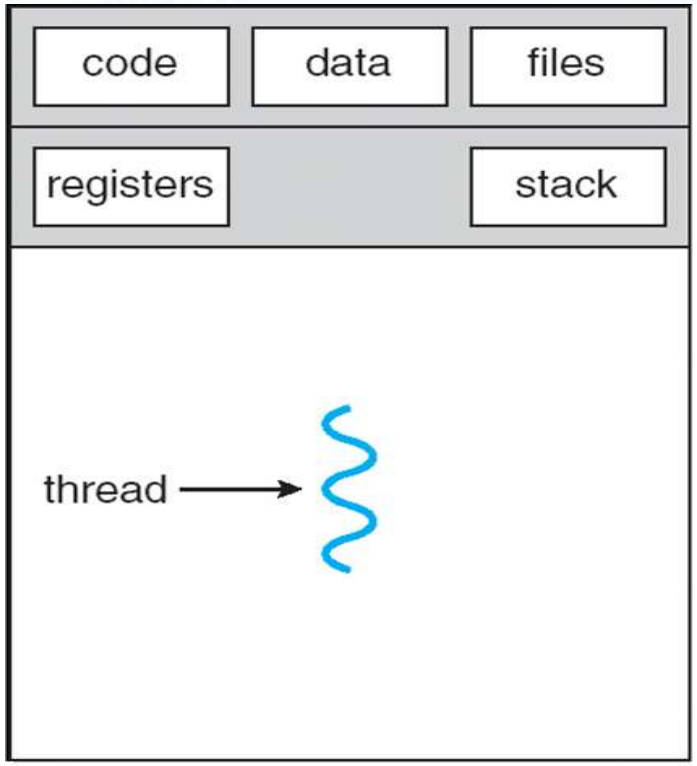
... con gli altri thread dello stesso processo ...

- Dati condivisi
  - Sezione di codice
  - Sezione di dati (variabili, descrittori di file, etc.)
  - Risorse del sistema operativo (e.g., segnali)
    - Ovvero, dati condivisi: variabili statiche, esterne, dinamiche (heap)
- Dati privati
  - Program counter e registri hardware
  - Stack, ovvero, variabili locali e storia dell'esecuzione

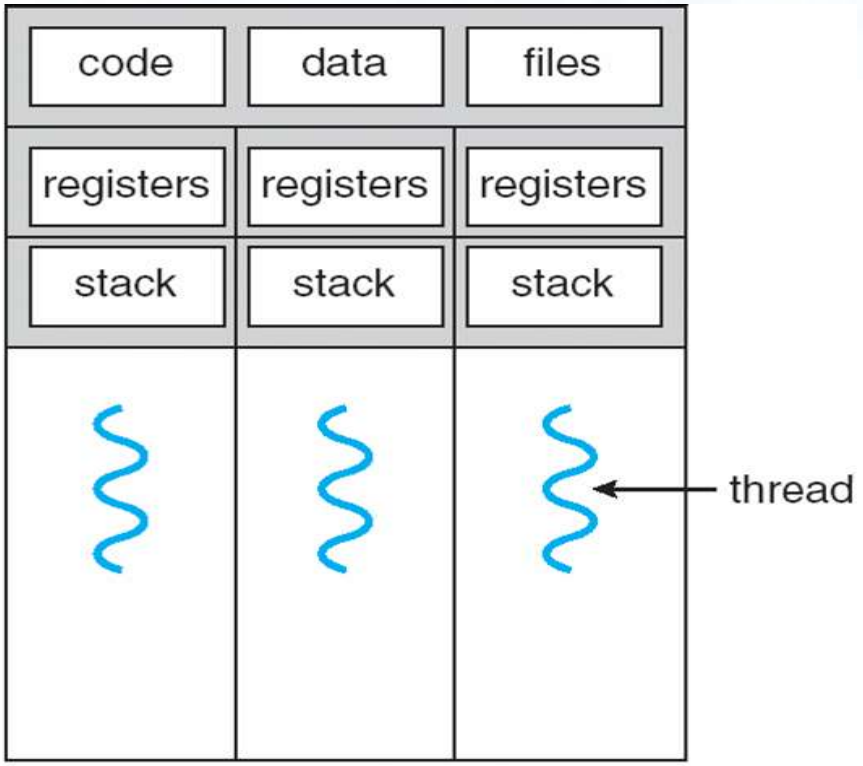
Un thread implica un flusso di esecuzione a se stante (all'interno dello stesso processo)



Dai processi ai thread



single-threaded process



multithreaded process

Un processo con un singolo thread

Un processo con tre thread
Condivisione senza protezione !

Thread: Vantaggi

- L'utilizzo di thread consente
  - Tempi di risposta ridotti
    - Creare un thread è 10-100 volte più veloce che generare un processo
    - Esempio
      - Per creare 50000 processi (**fork**) occorrono 10 secondi (real time)
      - Per creare 50000 thread (**pthread_create**) occorrono 0.8 secondi (real time)
  - Risorse condivise
    - I processi possono condividere dati solo con tecniche particolari
    - I thread condividono dati in maniera automatica

Thread: Vantaggi

- Costi minori per la gestione delle risorse
  - Assegnare la memoria a un processo è costoso
  - Con i thread una sezione di codice e/o dati può servire più clienti
- Maggiore scalabilità
  - I vantaggi della programmazione multi-thread aumentano nei sistemi multi-processore
  - Nei sistemi multi-core (diverse unità di calcolo per processore) i thread permettono più semplicemente paradigmi di programmazione concorrente basati su
    - Separazione dei compiti (Multiple Instruction SD/MD)
    - Suddivisione dei dati (SI/MI Multiple Data)

Thread: Svantaggi

- Non esiste protezione tra thread
  - Tutti sono eseguiti nello stesso spazio degli indirizzi e la protezione da parte del SO è impossibile oppure non necessaria
  - Se i thread **non** sono sincronizzati l'accesso a dati condivisi non è **thread safe**
- Tra thread non esiste relazione gerarchica padre-figlio
  - Al thread creante viene normalmente restituito l'identificatore del thread creato ma questo non implica una relazione gerarchica
  - Tutti i thread sono uguali

Concorrenza e thread

- Si voglia ottimizzare il seguente tratto di codice mediante l'utilizzo di processi o thread?

Prodotto di vettori (v1, v2, v3, v4) di grandi dimensioni

```
#define N 100000000
...
for (i=0; i<N; i++) {
  v[i] = v1[i] * v2[i] + v3[i] * v4[i];
}
```

Con i processi la condivisione dei dati sarebbe costosa e ne preverrebbe l'utilizzo

Con i thread la condivisione dei dati è automatica e la concorrenza immediata

Esempio

Pseudo-codice

Suddividiamo la procedura utilizzando un strategia divide-and-conquer

```
mult (int l, int m) {
  for (i=l; i<m; i++)
    v[i] = v1[i] * v2[i] + v3[i] * v4[i];
}
...
CreateThread (mult, 0, N/2);
CreateThread (mult, N/2, N);
```

Attenzione all'utilizzo di procedure non rientranti, all'accesso non in mutua eslusione a variabili comuni, alle funzioni di libreria, etc.

Su ciascuna partizione si esegue un thread

Modelli di programmazine multi-thread

- Esistono tre modelli di programmazione multi-thread
  - Kernel-level thread
    - Implementazione dei thread a livello kernel
  - User-level thread
    - Implementazione dei thread a livello utente (nello spazio utente)
  - Soluzione mista o ibrida
    - Implementazione dei thread a livello sia kernel sia user

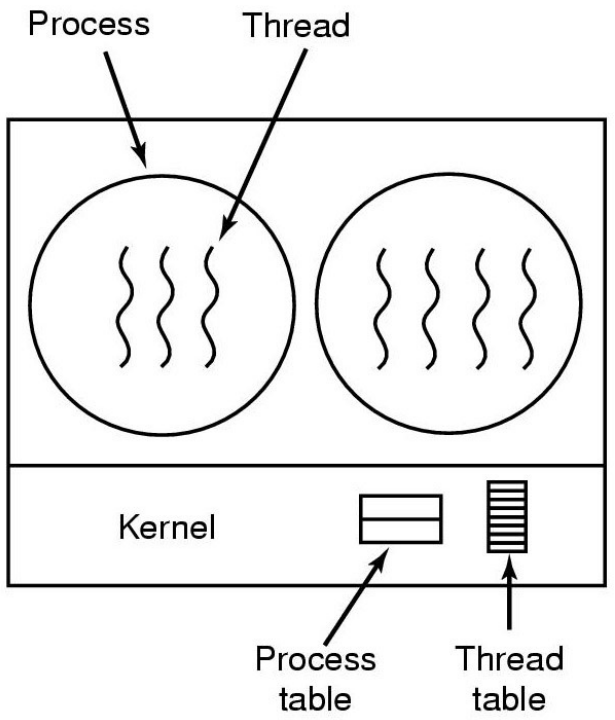
La scelta è moderatamente controversa

Kernel-level thread

- I thread sono gestiti dal kernel
- Il sistema operativo
  - Manipola tanto processi quanto thread
  - È a conoscenza dell'esistenza dei thread
  - Fornisce un supporto adeguato per la loro manipolazione
    - Tutte le operazioni sui thread (creazione, sincronizzazione, etc.) sono effettuate mediante **system call**

Kernel-level thread

- Il sistema operativo mantiene per ciascun thread informazioni simili a quelle che mantiene per ogni processo
  - Tabella dei thread
  - Thread Control Block (TCB) per ogni thread attivo
- Le informazioni gestite sono "globali" all'interno dell'intero sistema operativo



Kernel-level thread

- Vantaggi
  - Dato che il sistema operativo è a conoscenza dei thread può decidere
    - Quale thread di quale processo schedulare
      - Visione globale di tutti i thread di tutti i processi
    - Come suddividere il tempo tra i vari processi
      - Eventualmente allocare più tempo di CPU a processi con molti thread rispetto a processi con pochi thread

Kernel-level thread

- Efficace nelle applicazioni che si bloccano spesso (e.g., read bloccante)
  - I thread ready possono essere schedulati anche se appartengono allo stesso task di un thread che ha chiamato una system call bloccante
  - Ovvero se un thread si blocca è sempre possibile eseguirne un altro nello stesso processo o in un altro processo perchè il sistema operativo controlla tutti i thread di tutti i processi
- Permette un effettivo parallelismo
  - In un sistema multiprocessore si possono eseguire thread multipli

Kernel-level thread

- Svantaggi
  - A causa del passaggio al modo kernel la gestione è relativamente lenta e inefficiente
    - Context switch costoso
    - Tempi di manipolazione centinaia di volte più lenti del necessario
  - Limitazione nel numero massimo di thread
    - Il sistema operativo deve controllare il numero di thread generati
  - Il SO deve mantenere informazioni costose (tabella dei thread e TCB)

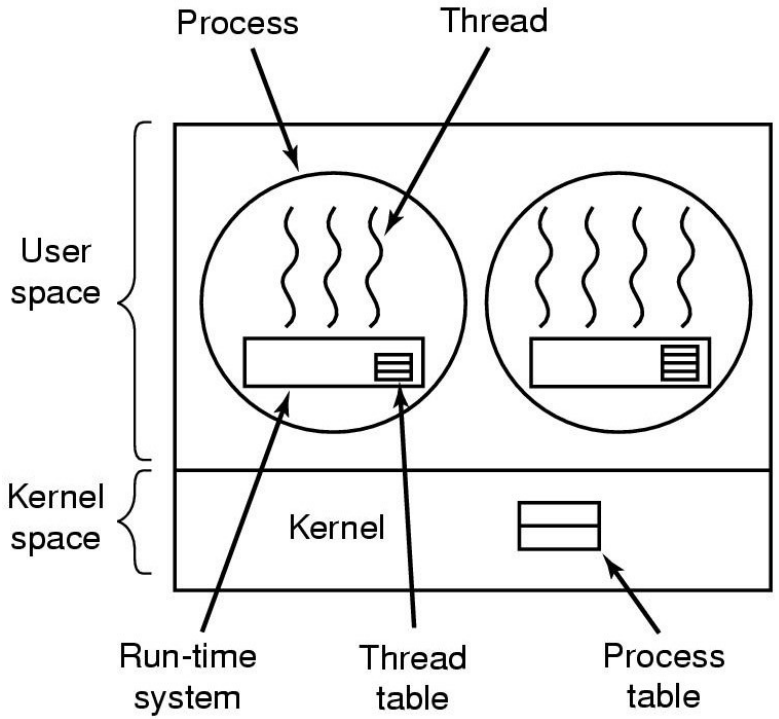
Già affermato per i processi
(motivazione per la nascita dei thread)

User-level thread

- Il pacchetto dei thread è inserito completamente nello spazio dell'utente
  - Il kernel **non** è a conoscenza dei thread e gestisce solo processi
  - I thread sono gestiti run-time tramite una **libreria**
    - Supporto mediante insieme di chiamate a funzioni, a livello utente
    - Creare un thread, sincronizzare thread, schedularli, etc., non richiede l'intervento del kernel
    - Si utilizzano funzioni **non** system call

User-level thread

- Ogni processo ha bisogno di una tabella personale dei thread in esecuzione
  - Le informazioni necessarie sono minori che nel caso di gestione kernel-level
    - TCB di dimensioni ridotte
    - Visibilità locale delle informazioni all'interno del processo cioè visibilità limitata



User-level thread

- Vantaggi
  - Si possono implementare in tutti i kernel, anche nei sistemi che non supportano thread in maniera nativa
  - Non richiedono modifiche al sistema operativo
  - Gestione efficiente
    - Context switch veloce
    - Manipolazione efficiente dei dati
    - Centinaia di volte più veloci dei thread kernel
  - Al programmatore è consentito generare tutti i thread desiderati
    - Al limite potrebbe essere possibile pensare a scheduling/gestioni personalizzate dei thread all'interno di ciascun processo

User-level thread

- Svantaggi
  - Il sistema operativo non sa esistano i thread
  - Possono essere fatte scelte inopportune oppure poco efficienti
    - Il sistema operativo potrebbe schedulare un processo il cui thread in esecuzione potrebbe fare una operazione bloccante
    - In questo caso l'intero processo potrebbe rimanere bloccato anche se al suo interno diversi altri thread potrebbero essere eseguiti

User-level thread

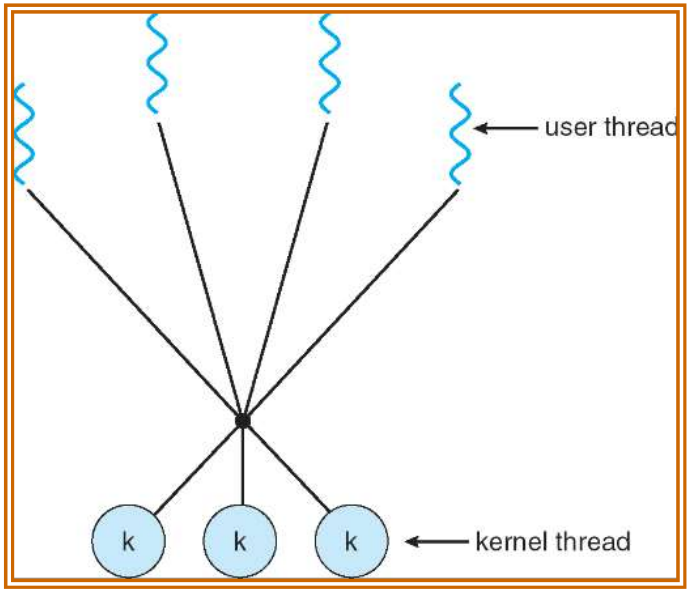
- Occorrer comunicare informazioni tra il kernel e il manager dell'utente run-time
- Senza questo meccanismo di comunicazione
  - Esiste solo un thread in run per task anche in un sistema multiprocessore
  - Non esiste scheduling all'interno di un processo singolo ovvero non esistono interrupt all'interno di un singolo processo
  - Se un thread in esecuzione non rilascia la CPU non è bloccabile

User-level thread

- Lo scheduler deve mappare i thread utente sull'unico thread kernel
  - Se il thread kernel si blocca, tutti i thread utente si bloccano
  - Non esiste parallelismo vero a livello di thread senza la gestione di thread multipli a livello di kernel

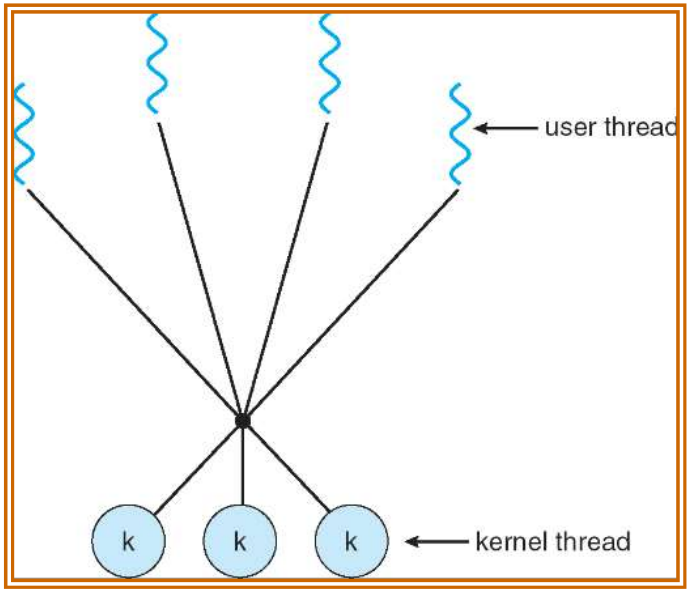
Implementazione ibrida

- Uno dei problemi della programmazione multi-thread è quella di definire il rapporto tra thread utente e thread kernel
- Praticamente tutti i sistemi operativi moderni dispongono di kernel thread
  - Windows, UNIX, Linux, MAC OS X, Solaris
  - L'idea base è di avere m thread utente e di accoppiarli a n thread kernel
  - In generale n<m



Implementazione ibrida

- L'implementazione mista tenta di combinare i vantaggi di entrambi gli approcci
  - L'utente decide quanti thread utente eseguire e su quanti thread kernel mapparli
  - Il kernel è a conoscenza solo dei thread kernel e gestisce solo tali thread
  - Ogni thread kernel può essere utilizzato a turno da diversi thread utente



Coesistenza di processi e thread

- La coesistenza di processi e thread provoca problematiche di vario genere legate alla
  - Gestione dei segnali
    - Quale T definisce il comportamento alla ricezione di un segnale e quale T riceve i segnali?
  - Alla clonazione (fork) di un processo
    - Una fork effettua la duplicazione di tutti i thread (forkall) oppure duplica solo il thread che la invoca (fork1)?
  - Alla sostituzione del programma con un altro (exec)
    - Una exec sostituisce solo il thread che effettua la exec oppure duplica tutti i thread del processo?

Coesistenza di processi e thread

- Gestione dei segnali
  - Il comportamento di un processo alla ricezione di un segnale specifico è **condiviso** tra tutti i thread
    - Ogni thread può definire il comportamento del processo alla ricezione di un segnale
    - Thread diversi possono definire azioni contrastanti
  - Ogni segnale è consegnato a un thread singolo all'interno del processo
    - Se un segnale è specificatamente associato a un thread, esso è consegnato a tale thread
    - Segnali generici sono consegnati a un thread arbitrario

Coesistenza di processi e thread

- Utilizzo della system call fork
  - In un processo multi-thread, una fork duplica solo il thread che richiama la fork
  - I dati privati posseduti dai thread non duplicati dalla fork, possono **non** essere "gestibili" dall'unico thread duplicato
    - Per gestire correttamente lo stato del processo dopo una fork, UNIX mette a disposizione system call specifiche, e.g., **pthread_atfork**

Coesistenza di processi e thread

- Utilizzo della system call exec
  - Una **exec** effettuata da un thread sostituisce con il nuovo programma l'intero processo (non solo il thread che effettua la exec)
    - Se il thread duplicato esegue una exec subito dopo aver eseguito una fork, lo stato degli altri thread presenti prima della fork non ha più importanza