Corso di Programmazione Concorrente

Programmazione Concorrente
Processi vs Thread

Valter Crescenzi

http://crescenzi.inf.uniroma3.it

Sommario

- Processi vs Thread
  - Caratteristiche, Vantaggi & Svantaggi
  - Cenni al supporto di alcune piattaforme
- Le piattaforme di riferimento
  - Processi/Thread in C sotto Linux/UNIX
  - Java Thread
- Alcune librerie di riferimento
  - Processi UNIX
  - POSIX Thread
  - Java Thread
  - `java.util.concurrent`
- Teoria → Pratica
  - Primitive astratte: fork(), join(), exit()
  - Strumenti: semafori, monitor, variabili condizione

Dalla Teoria alla Pratica

- *flusso di esecuzione*: concetto astratto che trova realizzazione con due diffusi strumenti concreti: *thread* e *processi*
- Introduciamo gli strumenti che useremo in concreto per scrivere programmi concorrenti
  - i processi ed i thread in C sotto UNIX
  - i thread in Java
- Confrontiamo i meccanismi resi disponibili da questi linguaggi per la programmazione concorrente con quanto discusso sinora

Processi vs Thread

- Esistono diversi termini che fanno riferimento a concetti correlati anche se a diversi livelli di granularità ed indipendenza:
  - …
  - Processo: unità di condivisione delle risorse (alcune possono essere inizialmente ereditate dal padre)
  - …
  - Thread (o lightweight process): idealmente, è un flusso di esecuzione indipendente che condivide tutte le sue risorse, incluso lo spazio di indirizzamento, con altri thread
  - …

Concetto di Thread

- Anche chiamati “lightweight process” (processi leggeri) perché possiedono un contesto più snello rispetto ai processi
- Flusso di esecuzione indipendente
  - interno ad un processo
  - condivide lo spazio di indirizzamento con gli altri thread dello stesso processo
- Più semplici ed efficienti rispetto ai processi

Contesto: Thread vs Processi

- Contesto di un thread
  - stato della computazione (codice, registri, stack)
  - attributi (schedulazione, priorità)
  - memoria TSD “privata”
  - descrittore di thread (tid, priorità, maschera dei segnali)
- Ed in più per i processi
  - tabelle di rilocazione (MMU)
  - risorse private (varie tabelle di descrittori)

Vantaggi dei Thread

- Comunicazione inter-thread facile
  - attraverso la memoria condivisa
    - i processi invece richiedono meccanismi di comunicazione espliciti (IPC)
      - ciascuno con la propria API
- Efficienza rispetto ai processi
  - creazione e distruzione più rapidi
  - context switching e quindi scheduling più veloci
  - lo scambio di informazioni tramite uno spazio di indirizzamento comune è il meccanismo di comunicazione più veloce possibile

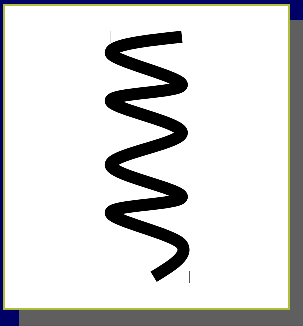
Svantaggi dei Thread

- Minore *robustezza* ai fallimenti in generale: il fallimento di un thread compromette tutto il sistema di f.d.e. di cui fa parte
- In particolare, maggior pericolo di interferenza
  - perché tutta la memoria è condivisa e quindi può essere oggetto di accessi concorrenti
- Minore autonomia
  - difficoltà di creare e gestire risorse private del singolo thread
  - servono appositi meccanismi: TSD
    - ciascuno con la propria API

Relazione Thread / Processi

- Le implementazioni di Thread e Processi oramai si basano quasi sempre su una relazione esplicita tra i due strumenti
  - Solaris, NPTL (Linux), OS X & iOS (Mac)
  - I processi sono visti come “contenitori” di thread
  - Un processo è il raccoglitore di risorse che i suoi thread condividono (in particolare lo spazio di indirizzamento della memoria)
  - Ogni thread ha un processo di riferimento
    - Non ha più senso parlare di un processo senza almeno un thread che ne esegua le istruzioni
    - Come non ha più senso parlare di un thread senza di un processo che ne ospiti le risorse

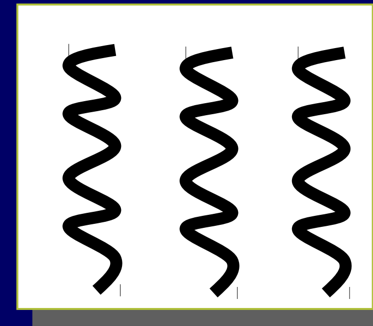
Modello di Thread



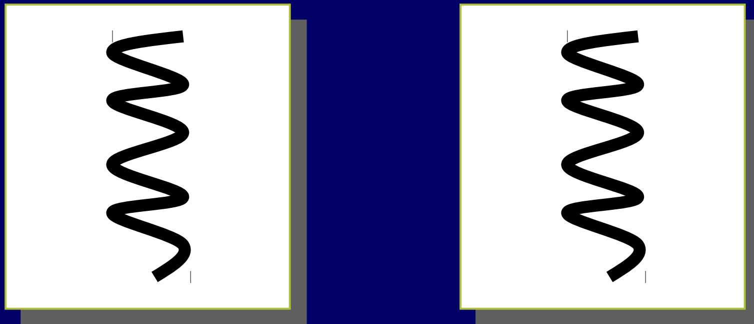
ms-dos

singolo processo
singolo thread

JVM

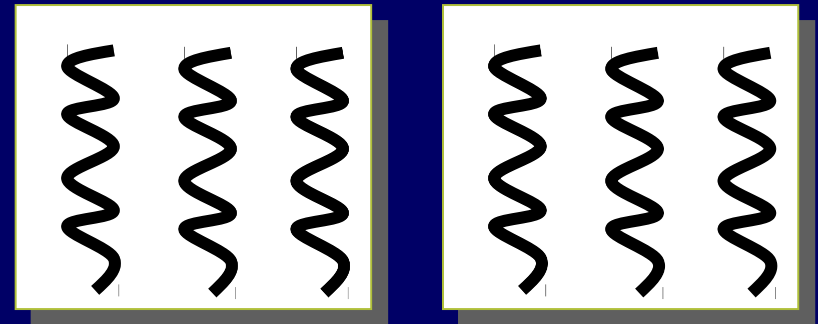


singolo processo
thread multipli



Multi-processo
un thread per processo

Linux + LinuxThread?



Multi-processo
Multi-thread per processo

Linux + NPTL ; OS X & iOS
solaris; Windows NT

POSIX Thread

- Uno standard per i thread: POSIX thread (Pthreads)
  - *https://en.wikipedia.org/wiki/POSIX_Threads*
- Implementationi disponibili per la maggior parte dei sistemi UNIX
  - FreeBSD, NetBSD, OpenBSD, Linux, Mac OS X, Solaris
  - library **`libpthread`**.
  - **`#include <pthread.h>`**
  - Windows (e DOS): librerie esterne al S.O. come pthreads-w32
- Pthreads definisce una API in C (include tipi, funzioni, costanti)
  - Più di 100 funzioni disponibili, prefisso comune **`pthread_`**
  - Approssimativamente raggruppabili in 4 gruppi:
    - Gestione thread: creazione, joining ....
    - Mutex
    - Variabili condizioni
    - Altri strumenti di sincronizzazione (read/write locks ...)

POSIX Thread per Linux

- Due diverse implementazioni per Linux:
  - LinuxThreads – NPTL
- Nessuna è perfettamente aderente allo standard
- LinuxThreads: semplice e parziale implementazione user-level ormai obsoleta che risultò l'unica disponibile sino ai kernel 2.2
  - Non richiese cambiamenti al kernel Linux
  - Largamente basata sulla chiamata di sistema **`clone()`**
- NPTL: evoluzione kernel-level che ambisce ad implementare lo standard POSIX completamente
  - Anche grazie a significativi cambiamenti nel kernel Linux

Modello di Thread: User vs Kernel

- User-Level: thread all’interno del processo utente gestiti da una libreria specifica
  - il kernel non è a conoscenza dell’esistenza dei thread
  - lo switching non richiede chiamate al kernel
- Kernel-Level: gestione dei thread affidata al kernel tramite chiamate di sistema
  - gestione integrata processi e thread dal kernel
  - lo switching è provocato da chiamate al kernel

Thread User Level

- Vantaggi:
  - Lo switching non coinvolge il kernel è quindi non ci sono cambiamenti della modalità di esecuzione
  - Maggiore libertà nella scelta dell'algoritmo di scheduling che può anche essere personalizzato
  - Siccome le chiamate possono essere raccolte in una libreria, c'è una maggiore portabilità tra SO
- Svantaggi:
  - una chiamata al kernel può bloccare tutti i thread di un processo, indipendentemente dal fatto che in realtà solo uno dei suoi thread ha causato la chiamata bloccante
  - In sistemi SMP, due processori non risulteranno mai associati a due thread del medesimo processo

Thread Kernel Level

- Vantaggi:
  - il kernel può eseguire più thread dello stesso processo anche su più processori
  - il kernel stesso può essere scritto multithread
- Svantaggi:
  - lo switching coinvolge chiamate al kernel e questo, soprattutto in sistemi con molteplici modalità di esecuzione, comporta un costo
  - l’algoritmo di scheduling è meno facilmente personalizzabile
  - meno portabile

Programmazione Concorrente: Linguaggi di Programmazione

- Molti costrutti introdotti astrattamente in precedenza si possono trovare realizzati dai linguaggi di programmazione concreta che useremo nel resto del corso
  - Processi UNIX
  - (POSIX) Thread
  - Java Thread

Programmazione Concorrente: Processi UNIX

- Processi UNIX
  - creazione e terminazione
    - `fork(), exec(), exit(), wait(), waitpid()`
  - semafori
    - `semget(), semop(), semctl()`

Programmazione Concorrente: POSIX Threads

- I thread POSIX
  - creazione e terminazione
    - pthread_create(), pthread_join(), pthread_exit()
  - mutex
    - pthread_mutex_init(), pthread_mutex_lock(), pthread_mutex_trylock(), pthread_mutex_unlock(), pthread_mutex_destroy()
  - semafori
    - sem_init(), sem_wait(), sem_trywait(); sem_post(), sem_getvalue(), sem_destroy()
  - variabili condizione
    - pthread_cond_init(), pthread_cond_signal(), pthread_cond_broadcast(), pthread_cond_wait(), pthread_cond_timedwait(), pthread_cond_destroy()

Programmazione Concorrente: Java Thread

- creazione e terminazione
  - classe `java.lang.Thread`;
  - interfaccia `java.lang.Runnable`
  - metodi `run()`, `join()` di Thread
- sincronizzazione
  - modificatore di metodo e di blocco `synchronized`
  - metodi `wait()`, `notify()`, `notifyAll()` di `java.lang.Object`
  - parola chiave `volatile`

Processi UNIX: Creazione e Terminazione

- `fork()`
  - crea un processo figlio di quello corrente per sdoppiamento
- `exec()`
  - sostituisce l'immagine (codice) del corrente processo con un eseguibile specificato
- `exit()`
  - termina il processo corrente
- `wait()`
  - blocca il processo corrente sino alla terminazione di uno qualsiasi dei suoi processi figli
- `waitpid()`
  - blocca il processo corrente sino alla terminazione del figlio con il PID specificato

POSIX Thread: Creazione e Terminazione

- `pthread_create()`
  - crea un thread che esegue la funzione specificata
- `pthread_exit()`
  - termina il thread corrente
- `pthread_join()`
  - termina il thread corrente sino al termine del thread specificato

Java Thread: Creazione e Terminazione

- Creazione di Thread
  - si creano oggetti istanze della classe `java.lang.Thread`.
    - Il thread viene effettivamente creato dalla JVM non appena si richiama il metodo `start()`
    - Il nuovo thread esegue il metodo `run()`
- Terminazione di Thread
  - i thread terminano quando
    - finisce l'esecuzione del metodo `run()`
    - sono stati interrotti con il metodo `interrupt()`
    - eccezioni ed errori
- Join con Thread
  - richiamando il metodo `join()` su un oggetto Thread si blocca il thread corrente sino alla terminazione del thread associato a tale oggetto

Processi UNIX: Semafori

- `semget()`
  - per creare un array di semafori
- `semop()`
  - per effettuare le classiche operazioni dei semafori anche se scalate su un array di semafori
- `semctl()`
  - per effettuare varie operazioni di controllo inclusa quella di deallocazione di un array di semafori

Processi UNIX: Semafori vs Semafori di Dijkstra

- Rispetto alla versione presentata astrattamente, le chiamate di sistema sui semafori
  - lavorano su un array di semafori e non su un singolo semaforo per motivi di efficienza
  - permettono di specificare nelle chiamate stesse di quanto incrementare/decrementare il valore del semaforo
    - semantica “*wait-for-zero*” se si specifica 0
  - … e tanti altri dettagli ancora …

POSIX Thread: Mutex

- `pthread_mutex_init()`
  - inizializza il mutex specificato
- `pthread_mutex_destroy()`
  - dealloca tutte le risorse allocato per gestire il mutex specificato
- `pthread_mutex_lock()`
  - blocca il mutex
- `pthread_mutex_unlock()`
  - sblocca il mutex

POSIX Thread: Mutex vs Semafori di Dijkstra

- Rispetto alla versione presentata astrattamente, i mutex usati con i POSIX thread
- corrispondono ai semafori binari (in stile competitivo)
  - ne esistono di tre tipologie diverse
    - fast: semantica classica, pertanto un thread che esegue due `mutex_lock()` consecutivi sullo stesso mutex causa uno stallo
    - recursive: conta il numero di volte che un thread blocca il mutex e lo rilascia solo se esegue un pari numero di `mutex_unlock()`
    - error-checking: controlla che il thread che rilascia il mutex sia lo stesso thread che lo possiede
- N.B.

  Si possono usare solo con i thread e non con i processi

POSIX Thread: Semafori

- `sem_init()`
  - inizializza un semaforo al valore dato
- `sem_wait()`
  - attende in attesa passiva che un semaforo assuma un valore non nullo; quindi lo decrementa
- `sem_post()`
  - incrementa il valore del semaforo specificato
- `sem_destroy()`
  - dealloca tutte le risorse allocate per gestire il semaforo specificato

POSIX Thread: Semafori vs Semafori di Dijkstra

- I semafori POSIX corrispondono alla versione dei semafori di Dijkstra precedentemente presentata
- Tuttavia si possono usare solo con i thread e non con i processi
- Bisogna considerare anche molti altri dettagli
- Offrono anche altre utilissime primitive
  - `sem_trywait()`

POSIX Thread: Variabili Condizione

- `pthread_cond_init()`
  - inizializza la variabile condizione specificata
- `pthread_cond_signal()`
  - riattiva uno dei thread in attesa sulla variabile condizione scelto non deterministicamente
- `pthread_cond_broadcast()`
  - riattiva tutti thread in attesa sulla variabile condizione
- `pthread_cond_wait()`
  - blocca il thread corrente che resta in attesa passiva di una signal sulla variabile condizione specificata
- `pthread_cond_destroy()`
  - dealloca tutte le risorse allocate per gestire la variabile condizione specificata

POSIX Thread: Variabili Condizione vs Primitive di Hoare

- Le variabili condizione dello standard POSIX risultano una implementazione delle variabili condizione teorizzate da Hoare
- Tuttavia si differenziano perché non inserite nel contesto dei monitor
  - bisogna allocare e gestire esplicitamente il mutex che garantisce contro eventuali corse critiche sulla var. cond. stessa. Altrimenti:
    - un thread crea una var. cond. e si prepara ad attendere
    - un altro thread effettua la segnalazione sulla var. cond.
    - il primo thread ha perso la `signal()` prima ancora che riesca ad effettuare la `wait`

```
int pthread_cond_wait(  pthread_cond_t *cond,
                        pthread_mutex_t *mutex);
```

- N.B. Si possono usare solo con i thread e non coi processi

Java Thread: Competizione

- La JVM associa un semaforo binario (mutex) ad ogni oggetto
- Tutti i metodi marcati con `synchronized` vengono eseguiti in sezione critica sull'oggetto sul quale vengono invocati
- E' possibile scrivere blocchi di codice da eseguire in sezione critica anche con

```
...
synchronized (object) {
  //Sezione Critica sull'oggetto object
}
…
```

Java Thread: Cooperazione

- La JVM associa anche una coda di attesa ad ogni oggetto che possono essere visti alla stregua di variabili condizioni
- I thread che sono interessati alla condizione associata ad un oggetto `object` eseguono

  `object.wait()`

  e vengono sospesi in attesa passiva dopo aver rilasciato il monitor
- Altri thread possono segnalare la condizione `object.notify()` oppure `object.notifyAll()`

Java Thread:
Java Monitor vs Monitor Astratti (1)

- Rispetto a quanto presentato astrattamente, il linguaggio Java utilizza una variante dei monitor ove:
  - le `notify()` e le `notifyAll()`, possono essere eseguite in qualsiasi punto e non solo alla fine dei metodi
  - semantica *signal_and_continue*
  - se un thread possiede il monitor di un oggetto può chiamare direttamente od indirettamente i metodi `synchronized` della stessa istanza senza pericolo di stallo
    - nessun problema delle chiamate annidate!
  - *non* tutti i metodi devono essere eseguiti in sezione critica

Java Thread: Java Monitor vs Monitor Astratti (2)

- La più importante delle differenze
  - è possibile "acquisire" il monitor di un oggetto anche al di fuori dei metodi della classe di cui è istanza
- Con i monitor "astratti" *non* è possibile
  - "Remember, with great power comes great responsibility":
    - **`java.lang.IllegalMonitorStateException`**
  - Dai Javadoc:
    - *Thrown to indicate that a thread has attempted to wait on an object's monitor or to notify other threads waiting on an object's monitor without owning the specified monitor.*
  - Tradotto? Con riferimento ai monitor astratti...

`java.util.concurrent`

- A partire da Java 5, la programmazione concorrente della piattaforma java è stata radicalmente rivista ed affidata a figure altamente specializzate del settore

- **`java.util.concurrent.Semaphore`**
- **`java.util.concurrent.locks.`*`Lock`***
- **`java.util.concurrent.locks.`*`Condition`***

- **`java.util.concurrent.`*`Executor`***
  - lightweight execution framework

- …

Thread Safeness (1)

- Riassunto:
  - Le strutture dati accedute da un programma multithread, dalle più semplici fino a quelle complesse, sono oggetto di aggiornamenti
  - Gli aggiornamenti non avvengono atomicamente (perché in generale le piattaforme non offrono operazioni di aggiornamento atomico), ma sono decomponibili in varie operazioni di modifica intermedie
  - Una generica struttura dati oggetto di un aggiornamento da parte di un thread parte da uno stato consistente e torna in uno stato consistente alla conclusione dell’operazione
  - Durante il transitorio la struttura dati “perde significato”, e passa per una serie di *stati transienti*

Thread Safeness (2)

- Riassunto:
  - Per tutto il transitorio, la struttura si può trovare in uno stato inconsistente e non dovrebbe essere *visibile* a thread diversi da quello che esegue l'aggiornamento
  - Un programma si dice *thread safe*, o anche che gode della proprietà di *thread safeness*, se garantisce che nessun thread possa accedere a dati in uno stato inconsistente
  - Esistono diverse soluzioni al problema. Una di quelle già introdotte:
    - Un programma thread safe protegge l'accesso alle strutture in stato inconsistente da parte di altri thread ad esempio (ma non necessariamente) costringendoli in attesa (passiva) del suo ritorno in uno stato consistente

Strutture Dati Thread Safe

- Il termine thread safeness si applica anche a librerie ed a strutture dati ad indicare la loro predisposizione ad essere inseriti in programmi multithread senza causare fenomeni di interferenza
- Un programma con più thread deve gestire correttamente l'accesso alle strutture dati condivise per evitare interferenze
- Questo è vero anche per le strutture dati ad esempio `errno`
- Ad es. si consideri un processo con due thread:
  - Primo Thread: chiamata, errore, `errno`←ERRY
  - Secondo Thread: chiamata, errore, `errno`←ERRX
  - Quale valore di `errno` permane?

Thread Safeness: Condizioni di Bernstein

- Dato un programma multithread, quali strutture dati bisogna proteggere per garantire la thread safeness?
- ✔ Tutte le strutture dati oggetto di accessi concorrenti che violano le condizioni di Bernstein

  ...in altre parole:

  <u>*le strutture dati oggetto di scritture concorrenti da parte di due o più f.d.e.*</u>

- ✔ Nota bene: se non vengono protette, sarà oggetto di fenomeni di interferenza anche chi si limita a leggere
- ✔ La protezione delle strutture richiede la partecipazione <u>anche</u> dei *lettori* che condividono <u>appieno</u> il problema generato dalla presenza di *scrittori*

Tecniche di Programmazione per la Thread-Safeness

- E' pertanto possibile ideare tecniche di programmazione thread-safe invalidando le condizioni di cui prima
- Ovvero, eliminando dal codice:

le strutture dati oggetto di scritture concorrenti da parte di due o più f.d.e.

- E questo può significare, almeno:
  - Eliminare gli accessi concorrenti
    - Serializzando gli accessi da parte dei thread
    - Concedendo l'uso esclusivo della struttura ad un thread
  - Eliminare le scritture

Conclusioni

- Useremo diversi linguaggi di programmazione concreti ed almeno tre distinti strumenti per esprimere programmi concorrenti
  - Processi UNIX
  - POSIX Thread
  - Java Thread
- Ma i meccanismi di sincronizzazione utilizzati in API e linguaggi anche sensibilmente diversi tra loro si riducono ad un nucleo molto contenuto di concetti, che abbiamo già visto