Thread

Introduzione

Creazione, distruzione

Sincronizzazione

Parallelismo

- **Il tipo di meccanismo a supporto del parallelismo dipende dal grado di cooperazione tra le varie entità che costituiscono un'applicazione**
  - Scambio d'informazione
  - Sincronizzazione
  - Dipendenza funzionale
- **Il processo**
  - Realizza il concetto di macchina virtuale
  - Implementa forte isolamento
  - Prevede meccanismi per lo scambio di informazioni
  - Permette di implementare attività parallele indipendenti
- **Il thread**
  - Realizza un'attività specifica e leggera
  - Consente un livello di cooperazione elevato
  - Lo scambio d'informazione è naturale molto facilitato
  - Permette di realizzare attività parallele fortemente interagenti

Thread e processi

	Processi	Thread
Creazione Distruzione	Richiedono allocazione, copia e deallocazione di grandi quantità di memoria	Richiedono solamente la creazione di uno stack per il thread
Errore	Non può danneggiare altri processi	Può danneggiare altri thread e l'intero processo cui appartiene
Codice	Un processo può modificare il proprio codice mediante il cambiamento di eseguibile	Il codice di un thread è fissato e presente nella sezione text del processo cui appartiene
Condivisione	E' onerosa e deve essere implementata dal programmatore	E' automaticamente garantita poiché tutti i thread condividono la memoria del processo cui appartengono
Mutua esclusione	La mutua esclusione è garantita automaticamente dall'isolamento proprio dei processi	Deve essere realizzata dal programmatore mediante semafori, mutex, ecc
Prestazioni	Limitate dall'overhead di gestion	Elevate
Concorrenza	Limitata dalla difficoltà di comunicazione	Elevate

Sequenzialità e concorrenza

- **Sequenzialità**
  - Due attività A e B sono sequenziali se, conoscendo il codice del programma, possiamo sapere con certezza l'ordine in cui esse vengono svolte, cioè
    - Se A è svolta prima di B, indicato con A < B
    - Se B è svolta prima di A, indicato con B < A
- **Concorrenza**
  - Due attività A e B sono concorrenti se non possiamo determinare tale ordinamento in base al solo codice del programma
  - In un sistema monoprocessore risulterà sempre A < B oppure A > B
- **Parallelismo**
  - E' un sinonimo di concorrenza
- **Parallelismo reale**
  - Riguarda i sistemi multiprocessore
  - In tal caso le attività A e B possono anche essere svolte nello stesso momento
    - Quando A e B sono svolte simultaneamente, scriveremo A = B
  - Il parallelismo reale implica la concorrenza (o paralellismo semplice) ma non viceversa

Thread

- **Flusso di controllo**
  - E' una esecuzione sequenziale di istruzioni
  - L'esecuzione è determinata unicamente dalle istruzioni
- **Un thread realizza un flusso di controllo**
  - Che può essere svolto in parallelo con altri
  - Che convive con altri all'interno di uno stesso processo
- **Il flusso di controllo di un thread**
  - E' realizzato mediante una funzione
  - La funzione viene eseguita all'atto della creazione del thread
- **Le funzioni nei vari thread**
  - Convivono nello stesso processo
  - Condividono lo spazio di indirizzamento
  - Condivisono i dati del processo
  - Sono eseguite in parallelo
- **In questo modo si ottiene**
  - Parallelismo
  - Facilità di scambio di informazioni

Thread

- **Esistono diversi modelli di thread**
  - Differiscono tra loro nei dettagli
  - Condividono l'idea generale

- **In questo corso considereremo i thread definiti dallo standard POSIX**
  - Generalmente detti "pthread"
  - Analizzeremo solo gli aspetti fondamentali
  - Per un approfondimento si veda, per esempio:

    Larrie Benton Zacharie,
    "Native POSIX Thread Library: Software Feature",
    VerPublishing, 2011

- **POSIX: Portable Operating System Interface for Computing Environments**
  - Insieme di standard per le interfacce applicative (API) dei sistemi operativi
  - Obiettivo di tale standard è la portabilità dei programmi in ambienti diversi
    - Un programma applicativo utilizza solamente i servizi previsti dalle API di POSIX può essere portato su tutti i sistemi operativi che implementano tali API

Thread

- **Tra thread e processi esistono le seguenti relazioni**

- **Un thread è attivato nell'ambito di un processo**
  - Il codice di un thread è raccolto in una funzione
  - In un processo possono coesistere più thread
  - I thread devono essere creati esplicitamente
    - Attraverso opportune funzioni delle API dello standard POSIX

- **Quando il processo termina**
  - Tutti i suoi thread terminano forzatamente
  - I thread possono terminare in un punto qualsiasi del loro flusso di controllo

- **Durante la vita di un processo**
  - I thread vengono attivati e terminati dal programmatore
  - E' necessario garantire una terminazione coerente di tutti i thread

Thread

- **Ogni thread ha un identificatore di thread**
  - Il tipo di tale identificatore è pthread_t
  - L'identificatore di thread identifica il thread univocamente
  - L'identificatore di thread è diverso dall'identificatore di processo

- **Tutti i thread di un processono**
  - Condividono lo stesso process id
  - La funzione getpid( ), eseguita all'interno dei thread di un medesimo processo, restituisce sempre il PID del processo stesso

- **Le operazioni che si possono compiere sui thread sono**
  - Creazione
  - Distruzione
  - Sospensione in attesa di un evento

Creazione

- **La creazione di un thread avviene mediante la funzione**

```
#include <pthread.h>
int pthread_create( pthread_t* thread,
                    const pthread_attr_t* attr,
                    void *(*function)(void *),
                    void * arg );
```

- **Argomenti**
  - thread
    - E' l'identificatore del thread, passato per indirizzo in quanto è un parametro di uscita
  - attr
    - E' una struttura dati che raccoglie gli attributi che si vogliono assegnare al thread
  - start_routine
    - E' un puntatore alla funzione che il thread deve eseguire
  - arg
    - E' l'argomento che deve essere passato alla funzione eseguita dal thread
- **Valore di ritorno**
  - In caso di successo viene restituito 0, altrimenti un codice di errore diverso da zero

Creazione

- **La creazione di un thread ricorda la creazione di un processo**
  - La funzione pthread_create() ha un comportamento logicamente analogo a fork( )
- **Tuttavia**
  - La creazione di un thread non comporta la copia di memoria
    - Le sezioni text, data, bss eccetera sono le stesse del processo
    - Ogni thread tuttavia dispone di un proprio stack
  - Il codice eseguito dal thread è raccolto in una funzione specifica
    - Non coincide con l'intero codice del processo
  - L'esecuzione del codice del thread inizia sempre dalla prima istruzione della funzione
    - A differenza dei processi in cui l'esecuzione nel processo figlio inizia dall'istruzione immediatamente successiva al ritorno della fork()
  - Ogni thread ha totale visibilità del codice e delle variabili del processo in cui esiste
    - Dal punto di vista della visibilità e del valore dei simboli e dei dati la situazione è identica al caso in cui la funzione eseguita dal thread fosse direttamente chiamata dal processo
  - Una modifica ad una variabile del processo
    - E' immediatamente visibile nel processo e quindi in tutti gli altri thread
    - Nel caso di un processo figlio, la modifica è invece locale alla sua copia della sezione dati

Creazione

- **Ogni thread viene creato nell'ambito di un processo**
  - Il processo ha un suo flusso di controllo
  - Il flusso di controllo del processo inizia dalla funzione main()
- **In un contesto di multithreading**
  - La funzione main() costituisce il thread principale o "main thread"
  - Quando viene eseguito un programma
    - Viene creato un nuovo processo
    - All'interno del processo viene creato il main thread
    - Il main thread esegue il codice della funzione main()
    - Quando il thread principale termina, il processo termina
- **Creazione**
  - Il thread principale può creare altri thread
  - Ogni thread può creare altri thread
    - In questo senso il thread principale non ha nulla di speciale
  - Nel momento in cui coesistono più thread
    - Inizia l'esecuzione concorrente del loro codice

Creazione

- **Un thread è eseguito nello spazio d'indirizzamento del processo che lo ha creato**
  - Tutti i thread di un processo condividono la stessa memoria dati
- **Ogni thread**
  - Ha un proprio stack privato, indipendente da quello degli altri thread
  - Ogni stack è allocato dal sistema operativo
    - Nel contesto del processo
    - Ad indirizzi differenti per ogni thread
  - Ogni thread ha quindi un proprio stack pointer per la gestione dello stack
- **Ne consegue che**
  - Le variabili locali della funzione eseguita dal thread sono allocate sul suo stack privato
    - Tali variabili quindi sono locali e appartengono solo a quel thread
  - Le variabili statiche e globali non sono allocate sullo stack
    - Pertanto sono condivise da tutti i thread
- **Questo tipo di situazione**
  - Rende facile e naturale lo scambio d'informazione tra thread
  - Rende più difficile garantire la correttezza del programma multithreading

Creazione

- **La funzione pthread_create() richiede come argomento un puntatore alla funzione che deve essere eseguita nel thread che verrà creato**
- **Tale funzione deve avere come prototipo**

```
void* function( void* argument );
```

  - La funzione accetta un unico argomento di tipo puntatore a void
  - La funzione restituisce un puntatore a void
  - I puntatori a void possono essere utilizzati per puntare a qualsiasi tipo di dato
    - E' lecito eseguire cast tra puntatore a void e qualsiasi altro tipo
    - Non è possibile dereferenziare un puntatore a void, per cui il casting è sempre necessario
- **Ne consegue che il tipo dell'argomento da passare a pthread_create() è**

```
void* (*function)(void*);
```

- **Può essere utile definire tale tipo, per esempio come**

```
typedef void* (*task_t)(void*);
```

Creazione

- **Un puntatore a void è un intero**
  - Si può convertire direttamente tra questi due tipi

```
void* task( void* a ) {
  // Catches the "int" argument. Must know that it is an integer. Type checking is broken
  int value = (int)a;

  // Uses value as an integer
  printf( "value = %d\n", value );

  // Terminates
  return (void*)0
}

int main() {
  // Thread id
  pthread_t* tid;

  // Data to pass to the thread function
  int data = 123;

  // Converts to void*
  void* arg = (void*)data;

  // Creates the thread
  pthread_create( tid, NULL, task, arg );
  ...
}
```

Creazione

- **Dati di un tipo più grande di un intero devono essere passati per indirizzo**
  - Si deve eseguire una conversione tra puntatori

```
void* task( void* a ) {
  // Catches the "double" argument.
  double* pvalue = (double*)a;
  double  value  = *pvalue;
  // Uses the value
  printf( "value = %f\n", value );
  return (void*)0
}

int main() {
  // Thread id
  pthread_t* tid;

  // Data to pass to the thread function
  double data = 1.23;

  // Takes the address of data and converts it to void*
  void* arg = (void*)(&data);

  // Creates the thread
  pthread_create( tid, NULL, task, arg );
  ...
}
```

Creazione

- **Lo stesso ragionamento vale per il valore di ritorno della funzione eseguita all'interno del thread**
  - Si puo convertire direttamente il dato in void* se il valore da restituire è
    - Un intero
    - Un tipo più piccolo o della stessa dimensione di un intero
  - Si deve convertire il puntatore al dato in void* se il valore da restituire
    - Ha una dimensione maggiore di quella di un intero

- **La soluzione che prevede la conversione sempre tra puntatori**
  - E' più elegante
  - E' più portabile
    - Non dipende in alcun modo dalla dimensione dei vari tipi sulla macchina specifica
  - E' quindi da preferirsi

Terminazione

- **La terminazione di un thread avviene mediante la funzione**

```
#include <pthread.h>
void pthread_exit( void* retval );
```

- **Argomenti**
  - retval
    - E' il valore che deve essere restituito alla funzione eseguita dal thread
- **Valore di ritorno**
  - La funzione non ritorna mai

Terminazione

- **Dal momento che la funzione pthread_exit() non ritorna**
  - Il codice successivo alla chaimata di questa funzione è codice morto
- **L'istruzione return nella funzione eseguita da un thread**
  - Ha lo stesso effetto della chiamata della funzione pthread_exit()
  - E' analogo al caso dell'istruzione return nel main() e la funzione exit()
- **Il valore che si desidera restituire deve essere**
  - Convertito in void*
  - Passato come argomento alla funzione pthread_exit()
  - Tale valore sarà recuperato dalla corrispondente funzione pthread_join()
    - Vedi oltre
- **Se la funzione pthread_exit() viene chiamata nel main thread**
  - Termina l'esecuzione del main thread
  - Non termina l'esecuzione dei thread ancora esistenti nel processo
- **Se la funzione exit() viene chiamata nel main thread**
  - Termina l'esecuzione di tutti i thread
  - E' equivalente ad eseguire una return dalla funzione main()

Attesa

- **L'attesa della terminazione di un thread avviene mediante la funzione**

```
#include <pthread.h>
int pthread_join( pthread_t thread, void** value_ptr );
```

- **Argomenti**
  - thread
    - E' l'identificatore del thread di cui si vuole attendere la terminazione
  - value_ptr
    - E' l'indirizzo di memoria in cui viene memorizzato il valore di ritorno
    - Il valore di ritorno è quello passato a pthread_exit() o alla return nel thread specificato
    - Se value_ptr è NULL, la funzione non recupera il valore di ritorno del thread
- **Valore di ritorno**
  - La funzione restituisce 0 in caso di successo, altrimenti un codice di errore diverso da 0

Attesa

- **Dati di tipo intero possono essere restituiti in modo semplice**
  - Si deve eseguire una conversione tra void* e intero

```
void* task( void* a ) {
  pthread_exit( (void*)12 );
}

int main() {
  // Thread id
  pthread_t* tid;

  // Return value
  void* pretval;
  int   retval;

  // Creates the thread
  pthread_create( tid, NULL, task, NULL );

  // Waits (joins) the thread
  pthread_join( tid, &pretval );

  // Uses the return value
  retval = (int)(pretval);
  printf( "Thread returned %d\n", retval );
}
```

Attesa

- **Passare dati diversi da interi da un thread al chiamante richiede attenzione**
  - Un tale dato non può essere convertito in puntatore a void
- **E' quindi necessario**
  - Prenderne l'indirizzo
  - Convertire l'indirizzo in void*
  - Restituire tale valore, passandolo a pthread_exit() o all'istruzione return
- **Ciò richiede che il dato da restituire non sia contenuto in una variabile automatica della funzione in esecuzione nel thread**
  - Le variabili automatiche risiedono sullo stack del thread
  - Al termine della funzione, lo stack del thread viene rilasciato
  - L'indirizzo restituito quindi punta ad una zona di memoria non più esistente
- **In casi di questo tipo si può**
  - Modificare una variabile globale invece di restituire un valore
  - Dichiarare la variabile da restituire come static nella funzione del thread
  - Allocare memoria dinamica e restituire il puntatore all'area allocata
    - Pericoloso, poiché il chiamante deve farsi carico di rilasciare tale memoria