POLITECNICO DI MILANO





I Thread

Laboratorio Software 2008-2009
M. Grotto R. Farina

Sommario

I Thread

Introduzione

- Unità di esecuzione a grana più fine rispetto ai processi
  - I thread esistono all'interno di un processo
  - Tutti eseguono lo STESSO programma (una differente sezione) nello STESSO processo
- Ogni thread condivide le stesse risorse
  - File descriptors, zona di memoria, ...
  - Un'azione in un thread ha ripercussioni anche sugli altri
- Se un thread invoca una funzione della famiglia `exec` tutti i thread terminano ed inizia l'esecuzione del nuovo programma

I Thread

Introduzione

- GNU/Linux implementa la API standard di POSIX (Portable Operating System Interfaces) per i thread (`pthread`)
  - Dalla versione 2.4.20 implementa la NPTL (Native POSIX Thread Library)
  - Tutti i tipi di dato e le funzioni per i thread sono definite in `<pthread.h>`
  - Non sono incluse nella libreria standard, ma nella libreria `libpthread.so`
  - Necessario specificare la libreria `lpthread` in fase di linking
- Ogni thread è identificato da un thread ID
  - Usare SEMPRE `pthread_t`

Creazione e terminazione

Utilizzo

- A valle della creazione, ogni thread esegue una funzione specifica (thread function)
  - Una thread function accetta un parametro di tipo `void*` e restituisce un valore di tipo `void*`
- `pthread_create(…);`
  - Crea un nuovo thread. Gli argomenti necessari sono:
    - Un puntatore ad una variabile `pthread_t`
    - Un puntatore ad un oggetto thread attribute
    - Un puntatore alla thread function
    - Un thread argument
- Un thread termina se:
  - termina la thread function
  - il thread invoca `pthread_exit(…);`

Esempio
`thread-create.c`

Creazione e terminazione

Scheduling dei processi

- La chiamata a `pthread_create(…);` termina immediatamente e l'esecuzione riprende dall'istruzione successiva
- Lo scheduling è asincrono
  - Non fare MAI affidamento sull'ordine di esecuzione
- Problema: il thread principale termina prima degli altri
  - Le strutture dati a cui fanno riferimento gli altri thread potrebbero essere eliminate dalla memoria
- Soluzione: `pthread_join(…);`
  - Analoga alla `wait` per i processi

Esempio
`thread-create2.c`

Comunicazione

Passaggio di parametri

- È possibile passare dei parametri al thread tramite l'argomento di tipo `void*`
  - Possibilità di riusare la stessa funzione in più thread
  - Solitamente si passa un puntatore ad una struttura o array di dati
  - Necessario un casting all'interno del thread
- È possibile raccogliere il valore di ritorno di un thread passando un argomento non nullo come secondo parametro di `pthread_join(…);`
  - Necessario un cast esplicito da `void*`

Esempio
`primes.c`

Comunicazione

Funzioni utili

- `pthread_equal(pthread_t t_id_1, pthread_t t_id_2);`
  - per confrontare due thread id
- `pthread_self();`
  - restituisce il thread id del thread nella quale è invocata
  - Utile per evitare che un thread invochi una join su se stesso
    - `EDEADLK`

```
if (!pthread_equal(pthread_self(), other_thread))
{
  pthread_join(other_thread, NULL);
  ...
};
```

Attributi

Descrizione

- Meccanismo per specializzare il comportamento di un thread
- Se il secondo parametro di `pthread_create(…);` è `NULL` si adotta il meccanismo di default
- Nella maggior parte dei casi un solo attributo è di interesse: il detach state
- Gli altri attributi sono tipicamente usati in sistemi real-time

Attributi

Joinable vs detach

- Joinable thread
  - quando la funzione termina non viene pulito automaticamente (simile ai processi zombie)
- Detach thread
  - pulito alla terminazione della funzione
  - Impossibile la sincronizzazione e la lettura del valore di ritorno
  - `pthread_attr_setdetach_state(…);`
    - `PTHREAD_CREATE_DETACHED` come secondo parametro
- È possibile trasformare un joinable thread in detached invocando `pthread_detach(…);`
  - Impossibile il contrario

Attributi

Modifica degli attributi

- Si crea un oggetto del tipo `pthread_attr_t`
- Si invoca `pthread_attr_init(…);` passandogli un puntatore all'oggetto
- Si modifica l'oggetto creato
- Si passa l'oggetto a `pthread_create(…);`
- Si invoca `pthread_attr_destroy(…);` per distruggere l'oggetto
  - La variabile non è deallocata, può essere reinizializzata

Esempio
`detached.c`

Cancellazione

Descrizione

- Definizione: richiesta di terminazione di un altro thread
- Si invoca `pthread_cancel(…);` passando l'id del thread di cui si richiede la terminazione
- È possibile invocare `pthread_join(…);` su un thread cancellato
  - Liberare le risorse
- Il valore di ritorno di un thread cancellato è `PTHREAD_CANCELED`

Cancellazione

Stati di cancellazione

- Un thread può contenere codice che va eseguito in maniera “all-or-nothing”
  - Allocazione delle risorse: una cancellazione non consentirebbe di liberare le risorse allocate
- Possibilità di “controllare” la cancellazione
  - Asynchronously cancelable: può essere cancellato in qualsiasi momento
  - Synchronously cancelable: le richieste di cancellazione sono accodate e processate al raggiungimento di specifici punti nel codice (cancellation points)
  - Uncancelable: le richieste sono ignorate

Cancellazione

Sync vs Async

- `pthread_setcanceltype(…);`
  - Opera sul thread che la invoca
  - `PTHREAD_CANCEL_ASYNCHRONOUS` per impostare la modalità asincrona
  - `PTHREAD_CANCEL_DEFERRED` per impostare la modalità sincrona (ripristina la modalità di default)
- Punti di cancellazione:
  - `pthread_testcancel();`
    - Per processare una richiesta di cancellazione pendente
    - Da invocare periodicamente per computazioni lunghe
  - `man pthread_cancel`
    - per gli altri punti di cancellazione

Cancellazione

Uncancelable

- `pthread_setcancelstate(…);`
  - `PTHREAD_CANCEL_DISABLE`
  - `PTHREAD_CANCEL_ENABLE`
- Possibilità di implementare delle sezioni critiche tramite la disabilitazione della cancellazione
  - Sezioni di codice da eseguire interamente o da non eseguire
- Importante alla fine ripristinare lo stato originale e NON impostare `PTHREAD_CANCEL_ENABLE` in maniera incondizionata
  - L'invocazione potrebbe essere annidata

Esempio
`critical-section.c`

Thread-specific data

Descrizione

- Possibilità di definire uno spazio di memoria “indipendente”
  - Creare copie di variabili per poterle modificare senza influire sul comportamento degli altri thread
- È possibile creare un numero arbitrario di data item
  - Tutti di tipo `void*`
  - Referenziati per mezzo di una chiave
    - Usata da ogni thread per accedere alla copia specifica

Thread-specific data

Creazione ed uso della chiave

- `pthread_key_create(…);` per creare la chiave
  - Puntatore ad una variabile `pthread_key_t` da usare successivamente per accedere alla propria copia
  - Puntatore ad una funzione di pulitura (cleanup)
    - Automaticamente invocata quando il thread termina
    - Invocata anche a fronte di richieste di cancellazione
    - Non invocata se il thread-specific data è `NULL`
    - Viene passata la copia locale della variabile
- `pthread_setspecific(…);`
  - per impostare il proprio valore nella variabile locale
- `pthread_getspecific(…);`
  - per leggere il valore

Cleanup handlers

Descrizione

- Funzione invocata quando un thread termina
  - Non specifica per ogni singolo thread data item
- Accetta un unico parametro di tipo `void*`
  - Specificato all'atto della registrazione dell'handler
  - Utile per deallocare istanze multiple di una risorsa
- Misura temporanea per deallocare risorse quando un thread termina o è cancellato invece di terminare l'esecuzione di una regione particolare di codice
  - In circostanze normali la risorsa va deallocata esplicitamente e l'handler va rimosso

Cleanup handlers

Registrazione e cancellazione

- `pthread_cleanup_push();`
  - registra un cleanup handler
    - Puntatore alla funzione di cleanup
    - Argomento di tipo `void*` per la funzione
- `pthread_cleanup_pop();`
  - cancella la registrazione di un cleanup handler
    - Bilancia la chiamata a `pthread_cleanup_push`
    - Flag intero: se diverso da zero la funzione è eseguita e solo successivamente - la registrazione è cancellata

Esempio
`cleanup.c`

Native POSIX Thread Library

Linux e lo standard POSIX

- I thread di GNU/Linux sono processi che condividono lo spazio degli indirizzi
- Raccomandazioni POSIX aiutano a garantire un certo grado di uniformità nella gestione dei thread di vari SO
- Linux non è completamente POSIX compliant.
  In particolare per quanto riguarda
  - la gestione dei segnali
  - Process Identifier / Thread Identifier
  - ed in generale per tutti gli aspetti in cui la vera natura dei thread di LinuxThreads non può essere nascosta

Native POSIX Thread Library

I Nuovi Thread per GNU/Linux

- Dal 2003, grazie all'interesse di IBM e Red Hat, si è sviluppato il supporto nativo ai thread per GNU/Linux
- NPTL: Native POSIX Thread Linux
- E' distribuita con tutte le versioni più recenti del kernel, ma non tutte le distribuzioni attivano NPTL automaticamente
- Si tratta di un lavoro ormai completo che da vantaggi
  - in termini di prestazioni
  - maggiore aderenza allo standard POSIX

Esempio
`thread-pid.c`

Native POSIX Thread Library

Thread manager

- Creato alla prima chiamata di `pthread_create(…);`
- Se un processo riceve un segnale in quale thread va gestito?
  - In Linux i thread sono realizzati tramite processi…
    - Solitamente i segnali sono inviati al processo che realizza il thread principale
- Es: un processo esegue una `fork();` il figlio tramite una `exec(…);` esegue un programma multithread
- Il padre mantiene come id del figlio il pid del processo che realizza il thread principale
- `pthread_kill(pthread_t t_id, signal int sig);`
  - per inviare segnali ad uno specifico thread