Corso di Laboratorio di Sistemi Operativi
A.A. 2016–2017

Lezione 18

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

A.A. 2016–2017 - Primo Semestre - 14/12/2016

Multithreading

Parte 2

Sincronizzazione di thread

In applicazioni con più thread concorrenti è spesso necessario che i diversi thread **comunichino** tra loro:

- La comunicazione può avvenire direttamente (senza pipe, socket, ecc.) perchè i thread condividono lo stesso spazio di indirizzi.
- Tuttavia, l'accesso a risorse condivise va sincronizzato per evitare **race condition**.
- Il sistema operativo fornisce una serie di primitive per la sincronizzazione:
  - Mutex
  - Condition variables
  - Semafori
- La sincronizzazione è necessaria spesso anche in assenza di multithreading, nella gestione di **segnali**.

Mutex

Il Mutex (da **mut**ual **ex**clusion) è forse il meccanismo di sincronizzazione più comune:

- È un meccanismo utile per proteggere strutture dati condivise da modifiche concorrenti (es., in modo da implementare sezioni critiche).
- Un mutex può essere **bloccato** (locked), oppure **libero** (unlocked).
- Un thread viene **sospeso** se prova a bloccare un mutex già bloccato, e riprende l'esecuzione quando il mutex si libera.

Mutex

Utilizzo

Per utilizzare un mutex va dichiarata una variabile di tipo `pthread_mutex_t` e inizializzata in modo particolare come segue:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Successivamente il codice compreso fra le chiamate `pthread_mutex_lock()` e `pthread_mutex_unlock()` potrà essere eseguito soltanto da un thread alla volta:

```
pthread_mutex_lock(&mutex);
// sezione critica
pthread_mutex_unlock(&mutex);
```

Se un thread non riesce a bloccare un mutex (perché già bloccato da un altro thread), viene sospeso fintanto che il thread che lo sta bloccando non lo rilascia.

Mutex

Problematiche

L'uso di mutex può avere conseguenze di cui bisogna tener conto:

- L'utilizzo errato può portare a **deadlocks**: situazione in cui due o più thread rimangono bloccati indefinitamente aspettando l'uno un'azione dell'altro.
- L'abuso di mutex può degradare le performance del programma:
  - Bloccare e sbloccare i mutex comporta due chiamate di sistema, per cui almeno due context switch.
  - Se le sezioni critiche sono troppo numerose o troppo pesanti il grado di parallelismo del programma diminuisce, e non si riesce a sfruttare il numero di CPU a disposizione.
  - Occorre quindi progettare il codice in modo che l'esecuzione di sezioni critiche sia ridotto al minimo.

Mutex

Esempio: sincronizzazione dell'accesso ad un contatore

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

int n = 0;
pthread_mutex_t mutex =
  PTHREAD_MUTEX_INITIALIZER;

void *count(void *);

int main() {
  pthread_t th1, th2;
  pthread_create(&th1, NULL, count, NULL);
  pthread_create(&th2, NULL, count, NULL);

  pthread_join(th1, NULL);
  pthread_join(th2, NULL);

  printf("n: %d\n", n);

  return 0;
}
```

```c
void *count(void *arg) {
  int local_n = 0;
  do {
    usleep(500000);
    pthread_mutex_lock(&mutex);
    n += 1;
    local_n = n;
    pthread_mutex_unlock(&mutex);
    printf("n: %d\n", local_n);
  } while(local_n < 42);

  return NULL;
}
```

Vedere `mutexnoglobals.c` per una versione più corretta senza var. globali.

Mutex

Esempio: gestione di un segnale

Questa è la versione corretta dell'esempio `annoying.c` della lezione 16.

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <pthread.h>

int signaled = 0;
pthread_mutex_t m =
  PTHREAD_MUTEX_INITIALIZER;

void handler(int x) {
  pthread_mutex_lock(&m);
  signaled = 1;
  pthread_mutex_unlock(&m);
}
```

```c
int main() {
  signal(SIGINT,  handler);
  signal(SIGTERM, handler);

  while(1) {
    pthread_mutex_lock(&mutex);
    if(signaled)
      printf("You cannot terminate me!\n");
    signaled = 0;
    pthread_mutex_unlock(&mutex);

    printf("Hey apple!\n");

    sleep(1);
  }
  return 0;
}
```

Condition variables

Una **condition variable** è una primitiva di sincronizzaziones che viene utilizzata per sospendere l'esecuzione di un thread in attesa che si verifichi un certo evento.

- Un thread può **sospendersi** su una condition variable, per aspettare che avvenga qualcosa.
- Un altro thread può **risvegliare** uno o più dei thread in attesa per segnalare.

Condition variables

Utilizzo

Per utilizzare una condition variable va dichiarata una variabile di tipo `pthread_cond_t` e inizializzata in modo particolare come segue:

```
pthread_cond_t var = PTHREAD_COND_INITIALIZER;
```

Successivamente è possibile:

- attendere la segnalazione della variabile con la funzione `pthread_cond_wait()`.
- risvegliare uno o tutti i thread in attesa su una variable con `pthread_cond_signal()` o `pthread_cond_broadcast()`, rispettivamente.

Condition variables

Perchè il mutex?

Per bloccarsi su una condition variable, si usa la funzione:

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

Come si può vedere, è necessario fornire anche un mutex, che va precedentemente **bloccato**. Perchè?

- La condition variable è un meccanismo per segnalare un evento, ma non c'è un meccanismo per comunicare dati oltre al solo evento.
- Servirà un mutex per sincronizzare l'accesso ai dati condivisi utilizzati per tale comunicazione.
- Il mutex assicura inoltre che un thread non "perda" una segnalazione mentre ne sta gestendo un'altra.

Condition variables

Perchè il mutex?

Per bloccarsi su una condition variable, si usa la funzione:

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

Come funziona?

- La funzione `pthread_cond_wait()` si aspetta un mutex già precedentemente **bloccato**.
- Atomicamente **sblocca** il mutex e si mette in attesa.
- Quando la condition variable viene segnalata, la funzione al risveglio torna a **bloccare** il mutex. Il thread non si risveglia finchè non trova il mutex libero.
- Anche il thread segnalante deve bloccare e sbloccare lo stesso mutex prima e dopo la segnalazione.
- **Attenzione**: Potrebbero verificarsi **risvegli spuri**: occorre controllare che l'evento segnalato sia successo davvero.

Condition variables

Comune schema di utilizzo

Thread segnalante:

```
pthread_mutex_lock(&mutex);

write_data();
event = 1;
pthread_cond_signal(&cond);

pthread_mutex_unlock(&mutex);
```

Thread in attesa:

```
pthread_mutex_lock(&mutex);

while(!event)
  pthread_cond_wait(&cond, &mutex);
read_data();

pthread_mutex_unlock(&mutex);
```

Esempio

Guardie e ladri multithread

Il file `guardieladri_th.c` allegato contiene un esempio completo di utilizzo di thread in un'applicazione **concorrente**.

- Equivalente all'esempio `guardieladri.c` visto nella Lezione 15, ma con l'utilizzo di thread multipli, comunicanti tramite una variabile condivisa, invece di processi multipli comunicanti tramite pipe.
  - Un thread gestisce l'input dell'utente spostando la guardia.
  - Un altro gestisce il ladro, scegliendo casualmente le sue mosse.
  - Il thread principale riceve le mosse da ognuno e le mostra a schermo.
- Un mutex viene utilizzato per proteggere l'accesso alla variabile condivisa utilizzata per comunicare le mosse, e una condition variable per segnalare al thread di controllo la presenza di una nuova mossa.