Architettura dei Calcolatori e Sistemi Operativi
Threads

Chair

Prof. C. Brandolese

e-mail: carlo.brandolese@polimi.it
phone: +39 02 2399 3492
web: home.dei.polimi.it/brandole

Politecnico di Milano

Teaching Assistant

A. Canidio

e-mail: andreA. Canidio@mail.polimi.it
material: github.com/acanidio/polimi_cr_acso_2018

Outline

- **Threads**
  - Creazione – *create*
  - Attesa della terminazione – *join*
  - Sincronizzazione
    - Semafori
    - Mutex

Threads - Creazione

- **In LINUX/POSIX per utilizzare i thread è necessario includere la libreria pthread.h**

```
#include <pthread.h>
```

- **Il comportamento del thread è determinato da una funzione, che prende come argomento un puntatore a void, e ritorna un puntatore a void.**

```
void* thread_function(void* param){
                                /* the void* argument needs a cast to the correct type*/
                                …
                        }
```

- **Ogni thread è contraddistinto da un thread ID: per gestire tale ID è definito il tipo pthread_t**

```
pthread_t thread_ID;
```

- **Per creare un thread si utilizza la funzione pthread_create, con quattro argomenti:**

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*),
void *arg);
```

  - Il primo argomento è un puntatore ad una variabile di tipo pthread_t che contiente l'ID del thread
  - Il secondo argomento è un puntatore ad un oggetto di tipo thread_attribute; se NULL il thread viene creato con i suoi parametri di default
  - Il terzo argomento è un puntatore alla thread function;
  - Il quarto argomento di tipo `void*` è il parametro da passare alla thread function.

Thread – Creazione

- **Esempio:**

```
#include <pthread.h>


void* thread_func (void *){

    printf("This is the thread\n");

}

int main(){
    pthread_t t;
    int i;
    pthread_create(&t, NULL, &thread_func,NULL);
    for(i=0;i<10000<i++){
      … /*do something*/
    }
}
```

Thread – Terminazione e attesa

- **Un thread termina con la funzionae** `pthread_exit` **o con la normale** `return`

```
void pthread_exit(void *value_ptr);
```

Il parametro passato alla pthread_exit, opportunamente castato a void*, è il return value del thread

- **Una chiamata a** `exit(int)` **all'interno del thread causa la terminazione del processo padre e di conseguenza di tutti gli altri thread**
- **Se il processo padre termina prima di uno dei suoi thread possono nascere problemi in quanto la memoria cui tali thread fanno accesso viene deallocata e tali thread vengono terminati insieme al padre**
- **Per prevenire tale effetto, nonché per attendere un thread all'interno di un altro thread, si usa la funzione** `pthread_join`

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Il primo parametro è il thread ID da attendere, il secondo è un puntatore a void che prende il valore di uscita del thread

- **Un thread non dovrebbe mai attendere se stesso, per evitare tale circostanza è opportuno controllare il proprio thread ID attraverso la funzione pthread_self**

```
pthread_t pthread_self(void);
```

- **Per controllare l'uguaglianza di due thread ID si usa la funzione pthread_equal**

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

Thread – Terminazione e attesa

- **Esempio:**

```
#include <pthread.h>

void* thread_func (void * arg){

      printf("This is thread %d\n",(int)arg);
      … /*do something*/
}

int main(){
    pthread_t t1,t2;
    int t_num;
    t_num=1;
    pthread_create(&t1, NULL, &thread_func,(void*)t_num);
    t_num=2;
    pthread_create(&t2, NULL, &thread_func,(void*)t_num);
    pthread_join(t1,(void*)&t_num);
    printf("Received thread %d",t_num);
    pthread_join(t2, (void*)&t_num);
    printf("Received thread %d",t_num);
    return 0;
}
```

Thread - Sincronizzazione

- Poiché i thread vengono schedulati dal sistema operativo in maniera non prevedibile, è opportuno utilizzare opportuni meccanismi di sincronizzazione per evitare *race condition* nell'utilizzo di dati condivisi

Esempio di race condition:

```
#include <pthread.h>
int me;

void* thread_func (void * arg){
      … /*do something*/
      me=(int)arg;
      …/*do something*/
      printf("My ID: %d", me);
}

int main(){
    pthread_t t1,t2;
    int t_num;
    t_num=1;
    pthread_create(&t1, NULL, &thread_func,(void*)t_num);
    t_num=2;
    pthread_create(&t2, NULL, &thread_func, (void*)t_num);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    return 0;
}
```

A run-time, il sistema operativo in questo punto interrompe l'esecuzione di t1 e passa a t2, il quale resetta la variabile me al suo argomento, poi sempre in questo punto interrompe t2.
Rimette dunque in esecuzione t1 che stampa la variabile me, ma tale variabile non varrà più l'argomento di t1, ma bensì quello di t2.

Vedremo due meccanismi di sincronizzazione: mutex e semafori

Thread - Sincronizzazione

Mutex

- Un mutex, abbreviazione per MUTual EXclusion lock, è una primitiva di sincronizzazione che fa leva su un concetto molte semplice:
  - *Solo un thread alla volta può detenere il lock sul mutex: se qualche altro thread tenta di effettuare il lock viene messo in attesa e bloccato finche il thread che detiene il lock non lo rilascia attraverso un'operazioen di unlock.*
- Per creare un mutex si dichiara una variabile del tipo pthread_mutex_t, tale variabile detiene l'identificativo del mutex
- Per inizializzare il mutex si utilizza la funzione pthread_mutex_init, che prende come primo argomento la variabile di tipo pthread_mutex_t e come secondo argomento una variabile di tipo mutex attribute (se il secondo argomento è posto a NULL il mutex viene inizializzato con gli attributi di default)
- In alternativa per inizializzare un mutex si può assegnare alla variabile di tipo pthread_mutex_t il valore speciale PTHREAD_MUTEX_INITIALIZER

```
/*primo metodo*/
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL);

/*secondo metodo*/
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Thread - Sincronizzazione

Mutex

- **Su un mutex sono possibili due operazioni fondamentali: l'operazione di lock e l'operazione di unlock, tramite le seguenti funzioni:**

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- **È inoltre possibile un'altrra operazione, la trylock, che non è bloccante:**

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- **Tipi di mutex:**
  - *fast mutex*: modalità di **default,** è sempre bloccante, anche se lo stesso thread chiama in sequenza due lock sullo stesso mutex ,in tal caso si ha un **deadlock** irrisolvibile
  - *recursive*: se il thread che detiene il lock effettua altre operazioni di lock, l'operazione non risulta bloccante e pertanto non si ha deadlock
  - *error checking*: ritorna un errore nel caso in cui il thread che detiene il lock su un mutex tenti di effettuare una nuova operazione di lock in sequenza
  - Noi utilizzeremo di default i fast mutex, per settare la tipologia recursive o error checking è opportuno inizializzare convenientemente gli attributi mutex . Tale pratica si adatta ad un utilizzo più avanzato di tali strumenti ,si rimanda alla bibliografia di cui alla slide 14 per approfondimenti.

Thread - Sincronizzazione

Mutex – Esempio

```
#include <pthread.h>
int me;
pthread_mutex_t mutex= PTHREAD_MUTEX_INITIALIZER;

void* thread_func (void * arg){
      … /*do something*/
      pthread_mutex_lock(&mutex);
      me= (int)arg;
      …/*do something*/
      printf("My ID: %d", me);
      pthread_mutex_unlock(&mutex);
}

int main(){
    pthread_t t1,t2;
    int t_num;
    t_num=1;
    pthread_create(&t1, NULL, &thread_func,(void*)t_num);
    t_num=2;
    pthread_create(&t2, NULL, &thread_func, (void*)t_num);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    return 0;
}
```

In questo modo viene evitato l'ipotetico caso di race condition di cui alla slide 26

Thread - Sincronizzazione

Mutex – Deadlock

- Occorre evitare situazioni di deadlock distribuito, ovvero un intreccio delle condizioni di attesa che rende impossibile il proseguire del flusso di esecuzione di due o più thread, bloccandoli perennemente.

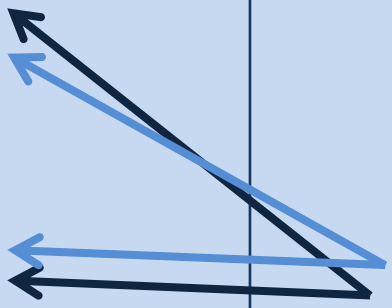
Esempio:

```
#include <pthread.h>
pthread_mutex_t mutex1=
PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2=
PTHREAD_MUTEX_INITIALIZER;

void* thread_func_1 (void *){
      … /*do something*/
      pthread_mutex_lock (&mutex1);
      pthread_mutex_lock (&mutex2);
      …
}

void* thread_func_2 (void *){
      … /*do something*/
      pthread_mutex_lock (&mutex2);
      pthread_mutex_lock (&mutex1);
      …
}
```

```
int main(){
    pthread_t t1,t2;
    pthread_create(&t1, NULL, &thread_func_1,NULL);
    pthread_create(&t2, NULL, &thread_func_2,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    return 0;
}
```



Se il sistema operativo schedula t1 fino al lock di mutex1, poi passa l'esecuzione a t2 che blocca mutex2, poi ritorna a t1 si avrà un deadlock: t1 non potrà predere il lock su mutex2 e t2 non potrà prendere più il lock su mutex1 in quanto entrambi sono già bloccati. Tale condizione di attesa durerà indefinitamente.

Una buona prassi per evitare deadlock di questo tipo è quella di richiedere i lock, nonché di rilasciarli, nello stesso ordine in ogni thread.

Thread - Sincronizzazione

Semafori

- **Un semaforo è un meccanismo di sincronizzazione basato su un contatore, se tale contatore è maggiore di zero, i thread che hanno effettuato una condizione di attesa sono autorizzati a procedere nel loro flusso di esecuzione. Se il contatore è zero, tali thread si bloccano finché il contatore non viene incrementato ad un valore positivo.**
- **Su un semaforo sono possibili due operazioni fondamentali:**
  - *Wait*: questa operazione decrementa di uno il contatore del semaforo. Se il contatore è a zero si blocca nell'attesa che il contatore venga incrementato. Una volta che il contatore è incrementato, la wait si risveglia e decrementa di uno il contatore.
  - *Post*: incrementa di uno il contatore del semaforo. Se il contatore era a zero, oltre ad incrementare il contatore, la post risveglia uno dei thread che erano rimasti bloccati sulla wait.
- **Per utilizzare i semafori occorre includere la libreria <semaphore.h>**
- **Per creare un semaforo si dichiara una variabile di tipo sem_t, poi si invoca la funzione sem_init:**

```
#include <semaphore.h>
sem_t semaforo;
sem_init(&semaforo, 0, 5); /*inizializza il semaforo con un valore iniziale pari a 5*/
```

La funzione **int sem_init(sem_t **sem*, int *pshared*, unsigned int *value*)**, prende come primo parametro un puntatore alla variabile sem_t, come secondo parametro sempre zero e come terzo parametro il valore a cui inizializzare il contatore del semaforo

Thread - Sincronizzazione

Semafori

- **Sui semafori sono possibili, come detto operazioni di wait e di post, con il significato descritto nella slide precedente.**

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
```

La trywait non è bloccante, nel caso il contatore sia zero va avanti, senza decrementare il contatore.

- **Infine, quando un semaforo non serve più, occorre deallocarlo per mezzo della funzione sem_destroy**

```
int sem_destroy(sem_t *sem);
```

- **Di seguito un tipico esempio di produttore-consumatore…**

Thread - Sincronizzazione

Semafori – Esempio

```
#include <pthread.h>
#include <semaphore.h>
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
sem_t sem;
int a; /*this should not be negative*/

void* producer (void *){
      pthread_mutex_lock(&mutex);
      a++;
      sem_post(&sem);
      pthread_mutex_unlock(&mutex);
}

void* consumer (void *){
      … /*do something*/
      sem_wait(&sem);
      pthread_mutex_lock(&mutex);
      a--;
      /*do something*/
      …
     pthread_mutex_unlock(&mutex);
}
```

```
int main(){
   pthread_t p[3], c[3];
   int i;
   sem_init(&sem,0, 0);
   for(i=0;i<3;i++)
    pthread_create(&p[i], NULL, &producer,NULL);
   for(i=0;i<3;i++)
    pthread_create(&c[i], NULL, &consumer,NULL);
   /*do something for sometime*/
   …
   for(i=0;i<3;i++)
    pthread_join(c[i],NULL);
   for(i=0;i<3;i++)
    pthread_join(p[i],NULL);

   sem_destroy(&sem);

   return 0;
}
```

Thread - Approfondimenti

- **I concetti affrontati sono da considerarsi base per quanto riguarda i thread**

- **Per approfondimenti si consiglia il seguente libro, al capitolo 4:**

  Mitchell, M., Oldham, J., and Samuel, A., Advanced Linux Programming. Boston, MA: New Riders, 2001.