Sistemi operativi

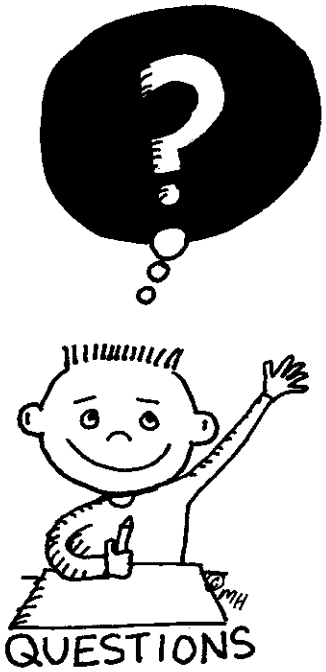
Corso di Laurea Triennale in Ingegneria Informatica

Lezione 7

- `Mutex`
- `Condition`
- `Esempi di utilizzo`

Fabio Del Vigna - fabio.delvigna@for.unipi.it

Domande sulle lezioni passate?



Sommario

- Sincronizzazione
  - Mutex
  - Variabili condition
  - Esempi

Semafori

Cosa sono i semafori?

- I semafori sono primitive fornite dal sistema operativo per permettere la sincronizzazione tra processi e/o thread.

Operazioni sui semafori

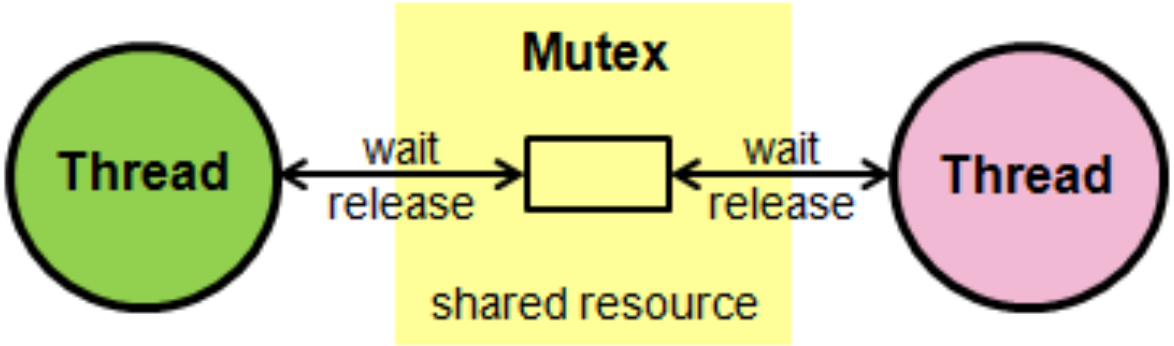
- In genere sono tre le operazioni che vengono eseguite da un processo su un semaforo:
  - Create: creazione e inizializzazione di un semaforo.
  - Wait: attesa su di un semaforo dove si verifica il valore del semaforo

```
while(sem_value <=0);   //wait;
sem_value--;
```

  - Post: incremento del semaforo.

```
sem_value++;
```

Semafori di mutua esclusione



Cosa sono i mutex? (1 di 2)

- Una variabile mutex è una variabile che serve per la protezione delle sezioni critiche:
  - variabili condivise modificate da più thread
  - solo un thread alla volta può accedere ad una risorsa protetta da un mutex
- Il mutex è un semaforo binario cioè il valore può essere 0 (*occupato)* oppure 1 (*libero*)

Cosa sono i mutex? (2 di 2)

- Pensiamo ai mutex come a delle serrature:
  - il primo thread che ha accesso alla coda dei lavori lascia fuori gli altri thread fino a che non ha portato a termine il suo compito.

- I threads utilizzano un mutex nelle sezioni di codice nelle quali vengono condivisi i dati.

Garantire la Mutua Esclusione (1 di 2)

- **Due thread devono decrementare il valore di una variabile globale** `data` **se questa è maggiore di zero**
  - `data = 1`

```
THREAD1
if(data>0)
    data --;
```

```
THREAD2
if(data>0)
    data --;
```

Garantire la Mutua Esclusione (2 di 2)

- **A seconda del tempo di esecuzione dei due thread, la variabile** `data` **assume valori diversi.**

```
Data              THREAD1                    THREAD2
1                 if(data>0)
1                   data --;
0                                            if(data>0)
0                                              // data --;
0 = valore finale di data
-------------------------------------------------------
1                 if(data>0)
1                                            if(data>0)
1                   data --;
0                                              data --;
-1
-1 = valore finale di data
```

Uso dei mutex

- Creare e inizializzare una variabile mutex
- Più thread tentano di accedere alla risorsa invocando l'operazione di `lock`
- Un solo thread riesce ad acquisire il mutex mentre gli altri si bloccano
- Il thread che ha acquisito il mutex manipola la risorsa
- **Lo stesso thread** la rilascia invocando la `unlock`
- Un altro thread acquisisce il mutex e così via
- Distruzione della variabile mutex

Creazione mutex

- Per creare un mutex è necessario usare una variabile di tipo `pthread_mutex_t` contenuta nella libreria pthread
  - `pthread_mutex_t` è una struttura che contiene:
    - ⇨ Nome del mutex
    - ⇨ Proprietario
    - ⇨ Contatore
    - ⇨ Struttura associata al mutex
    - ⇨ La *coda* dei processi *sospesi* in attesa che mutex sia libero.
    - ⇨ … e simili

Inizializzazione mutex

- statica
  - contestuale alla dichiarazione
- dinamica
  - attraverso
    - `pthread_mutex_t mutex;`
    - `pthread_mutex_init (&mutex, NULL);`

Inizializzazione statica

- Per il tipo di dato `pthread_mutex_t`, è definita la macro di inizializzazione `PTHREAD_MUTEX_INITIALIZER`
- Il mutex è un tipo definito "ad hoc" per gestire la mutua esclusione quindi il valore iniziale può essergli assegnato anche in modo statico mediante questa macro.

```
/* Variabili globali */

pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
```

Inizializzazione dinamica

```
pthread_mutex_t mutex;

int pthread_mutex_init( pthread_mutex_t *mutex, const
                        pthread_mutexattr_t *mattr )
```

- `pthread_mutex_t *mutex`
  - puntatore al mutex da inizializzare
- `pthread_mutexattr_t *mattr`
  - attributi del mutex da inizializzare
  - se NULL usa valori default
- Valore di ritorno
  - il valore `0` in caso di successo
  - Il numero dell'errore altrimenti

Interfacce

- Sui mutex sono possibili solo due operazioni: **locking** e **unlocking (**equivalenti a *P* e *V* sui semafori)

Interfaccia: `Lock`

- Ogni thread, prima di accedere ai dati condivisi, deve effettuare il `lock` su una stessa variabile mutex.
- Blocca l'accesso da parte di altri thread.
- Se più thread eseguono l'operazione di `lock` su una stessa variabile mutex, solo uno dei thread termina il `lock` e prosegue l'esecuzione, gli altri rimangono bloccati nel `lock`. In tal modo, il processo che continua l'esecuzione può accedere ai dati (protetti mediante il mutex).

Operazioni: `lock` e `trylock`

- `lock`
  - **bloccante (standard)**
- `trylock`
  - **non bloccante (utile per evitare deadlock)**
  - **come `lock()` ma se si accorge che il mutex è già in possesso di un altro thread (e quindi bloccherebbe il thread) restituisce immediatamente il controllo al chiamante con risultato `EBUSY`**

Una situazione di deadlock si verifica quando uno o più thread sono bloccati aspettando un evento che non si verificherà mai.

lock

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

- `pthread_mutex_t *mutex`
  - puntatore al mutex da bloccare
- Valore di ritorno
  - `0` in caso di successo
  - diverso da `0` altrimenti

trylock

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

- `pthread_mutex_t *mutex`
  - puntatore al mutex da bloccare
- Valore di ritorno
  - `0` in caso di successo e si ottenga la proprietà della mutex
  - EBUSY se il mutex è occupato

Interfaccia: `Unlock`

- Libera la variabile mutex.

- Un altro thread che ha precedentemente eseguito la `lock` della mutex potrà allora terminare la `lock` ed accedere a sua volta ai dati.

unlock

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

- `pthread_mutex_t *mutex`
  - puntatore al mutex da sbloccare
- Valore di ritorno
  - `0` in caso di successo

destroy

```
int pthread_mutex_destroy( pthread_mutex_t *mutex )
```

- Elimina il mutex
- `pthread_mutex_t *mutex`
  - puntatore al mutex da distruggere
- Valore di ritorno
  - `0` in caso di successo
  - `EBUSY` se il mutex è occupato

Esempio 1: uso dei mutex (1 di 2)

```
#include <pthread.h>
int a=1, b=1;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
void* thread1(void *arg) {
   pthread_mutex_lock(&m);
   printf("Primo thread (parametro: %d)\n", *(int*)arg);
   a++; b++;
   pthread_mutex_unlock(&m);
}
void* thread2(void *arg) {
   pthread_mutex_lock(&m);
   printf("Secondo thread (parametro: %d)\n", *(int*)arg);
   b=b*2; a=a*2;
   pthread_mutex_unlock(&m);
}
```

Esempio 1: uso dei mutex (2 di 2)

```
main() {
   pthread_t threadid1, threadid2;
   int i = 1, j=2;
   pthread_create(&threadid1, NULL, thread1, (void *)&i);
   pthread_create(&threadid2, NULL, thread2, (void *)&j);
   pthread_join(threadid1, NULL);
   pthread_join(threadid2, NULL);
   printf("Valori finali: a=%d b=%d\n", a, b);
}
```

Esempio 2: inizializzazione dinamica (1 di 2)

```
#include <pthread.h>
int a=1, b=1;
pthread_mutex_t m;
void* thread1(void *arg) {
   pthread_mutex_lock(&m);
   printf("Primo thread (parametro: %d)\n", *(int*)arg);
   a++; b++;
   pthread_mutex_unlock(&m);
}
void* thread2(void *arg) {
   pthread_mutex_lock(&m);
   printf("Secondo thread (parametro: %d)\n", *(int*)arg);
   b=b*2; a=a*2;
   pthread_mutex_unlock(&m);
}
```

Esempio 2: inizializzazione dinamica (2 di 2)

```
main() {
   pthread_t threadid1, threadid2;
   int i = 1, j=2;
   pthread_mutex_init(&m, NULL);
   pthread_create(&threadid1, NULL, thread1, (void *)&i);
   pthread_create(&threadid2, NULL, thread2, (void *)&j);
   pthread_join(threadid1, NULL);
   pthread_join(threadid2, NULL);
   printf("Valori finali: a=%d b=%d\n", a, b);
   pthread_mutex_destroy(&m);
}
```

Esempio 3 (1 di 3)

```
/* esempio utilizzo dei Mutex */
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mymutex;


void *body(void *arg){
          int i,j;
          for (j=0; j<40; j++) {
                pthread_mutex_lock(&mymutex);
                for (i=0; i<1000000; i++);
                fprintf(stderr,*(char *)arg);
                pthread_mutex_unlock(&mymutex);
          }
          return NULL;
}
```

Esempio 3 (2 di 3)

```
int main(){
        pthread_t t1,t2,t3;
        pthread_attr_t myattr;
        int err;

        pthread_mutexattr_t mymutexattr;
        pthread_mutexattr_init(&mymutexattr);

        pthread_mutex_init(&mymutex, &mymutexattr);
        pthread_mutexattr_destroy(&mymutexattr);

        pthread_attr_init(&myattr);

        …
```

Esempio 3 (3 di 3)

```
    err = pthread_create(&t1, &myattr, body, (void *)".");
    err = pthread_create(&t2, &myattr, body, (void *)"#");
    err = pthread_create(&t3, &myattr, body, (void *)"o");

    pthread_attr_destroy(&myattr);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    printf("\n");
    return 0;
}
```

Variabili condition

Condition vs Semafori

- Le variabili condition sono molto diverse dai semafori di sincronizzazione, anche se semanticamente fanno la stessa cosa

- Le primitive delle condition si preoccupano di rilasciare la **mutua esclusione** prima di bloccarsi e ed riacquisirla dopo essere state sbloccate
- I semafori generali, invece, prescindono dalla presenza di altri meccanismi

Cosa sono le variabili condition

- **Strumento di sincronizzazione**: consente la sospensione dei thread in attesa che sia soddisfatta una condizione logica.
- Una condition variable è utilizzata per sospendere l'esecuzione di un thread in attesa che si verifichi un certo evento.
- Ad ogni condition viene associato un **set** per la sospensione dei thread.
- La variabile condizione non ha uno *stato*, rappresenta solo una ***set di thread***.

Variabili condition

- Attraverso le variabili condition è possibile implementare condizioni più complesse che i thread devono soddisfare per essere eseguiti.
- Una variabile condition è un set di thread in attesa che si verifichi una condizione