Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2020-2021

Pietro Frasca

Lezione 11

Martedì 10-11-2020

Sincronizzazione tra thread in POSIX

- Per risolvere problemi di mutua esclusione e sincronizzazione la libreria pthread definisce i **mutex, i semafori** e le **variabili condition**.
- Il mutex è un particolare semaforo il cui valore intero (lo stato del mutex) può essere:

  0 (**occupato**) oppure

  1 (libero).

- Generalmente un mutex è usato per garantire che gli accessi a una risorsa condivisa avvengano in mutua esclusione.
- Nella libreria pthreads il mutex è definito dal tipo **pthread_mutex_t** che rappresenta:
  - **lo stato del mutex**;
  - **la coda di thread** nella quale saranno sospesi i thread in attesa che il mutex sia libero.

- Per definire un mutex *M*:

```
pthread_mutex_t M;
```

- Una volta definito, un mutex, si inizializza con delle proprietà. La funzione per l'inizializzazione è **pthread_mutex_init**, la cui sintassi è:

```
int pthread_mutex_init(
    pthread_mutex_t *M,
    const pthread_mutexattr_t *attr)
```

dove:

  - ***M*** è il mutex da inizializzare e
  - **attr** punta a una struttura che contiene gli attributi del mutex; se il valore di *attr* è **NULL**, il mutex viene inizializzato con i valori di default (con stato posto a ***libero***).

- Sul mutex sono possibili le operazioni: ***lock()*** e ***unlock()***, che sono concettualmente equivalenti rispettivamente alle funzioni ***wait()*** e ***signal()*** dei semafori.
- In particolare, la ***pthread_mutex_lock()*** è la realizzazione della ***wait()*** per il mutex:

```
int pthread_mutex_lock (pthread_mutex_t *M);
```

dove ***M*** rappresenta il mutex.

Se *M* è *occupato,* il thread chiamante si sospende nella coda associata al mutex; altrimenti *acquisisce M* e continua l'esecuzione.

- La ***pthread_mutex_trylock()*** è la versione non bloccante della *lock()*:

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

Si comporta come la *pthread_mutex_lock()*, tranne che nel caso di mutex già occupato non blocca il thread chiamante e ritorna immediatamente con l'errore **EBUSY**.

- La system call:

  ```
  int pthread_mutex_unlock(pthread_mutex_t *M);
  ```

  è la realizzazione della ***signal()***. Infatti, L'effetto della *_unlock()*, dipende dallo stato della coda di thread associata al mutex: se ci sono thread in attesa del mutex, ne risveglia uno, altrimenti libera il mutex.

- La system call:

  ```
  int pthread_mutex_destroy (pthread_mutex_t *M);
  ```

  dealloca tutte le risorse assegnate al mutex *M*.

Esempio 1 - mutex

```
/* Il tread main iniziale crea due thread di nome Th1 e
   Th2. I due thread condividono la variabile intera A alla
   quale inizialmente è assegnato il valore 10. ll thread
   Th1 incrementa di 1 il valore di A e attende 1 secondo,
   mentre Th2 decrementa di 1 il valore di A e attende 1
   secondo. Entrambi i thread eseguono le operazioni
   suddette per 10 volte, quindi terminano.
*/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int A=10; // variabile condivisa
pthread_mutex_t M; /* mutex per mutua esclusione */
```

```
void *codice_Th1 (void *arg){
  int i;
  for (i=0; i<10;i++){
    printf("Thread %s: ", (char *)arg);
    pthread_mutex_lock(&M); /* prologo sez. critica */
    A++;
    printf (" A = %d \n",A);
    pthread_mutex_unlock(&M); /* epilogo sez. critica */
    sleep(1);
  }
  pthread_exit(0);
}
```

```
void *codice_Th2 (void *arg){
  int i;
  for (i=0; i<10;i++){
    printf("Thread %s: ", (char *)arg);
    pthread_mutex_lock(&M); /* prologo sez. critica */
    A--;
    printf (" A = %d \n",A);
    pthread_mutex_unlock(&M); /* epilogo sez. critica */
    sleep(1);
  }
  pthread_exit(0);
}
```

```
int main(){
  pthread_t th1, th2;

   // creazione e attivazione del primo thread
  if (pthread_create(&th1,NULL,codice_Th1, "th1")!=0){
    fprintf(stderr,"Errore di creazione thread 1 \n");
    exit(1);
  }

   // creazione e attivazione del secondo thread
  if (pthread_create(&th2,NULL,codice_Th2, "th2")!=0){
    fprintf(stderr,"Errore di creazione thread 2 \n");
    exit(1);
  }
```

```
  // attesa della terminazione del primo thread
  if (pthread_join(th1,NULL) !=0)
    fprintf(stderr,"join fallito %d \n",ret);
  else
    printf("terminato il thread 1 \n");

  // attesa della terminazione del secondo thread
   if (pthread_join(th2,NULL) !=0)
    fprintf(stderr,"join fallito %d \n",ret);
  else
    printf("terminato il thread 2 \n");
  return 0;
}
```

Esempio 2 - mutex

```
/* Il thread main crea una matrice di numeri interi di dimensione NxM assegnando a ciascun elemento della matrice un valore casuale compreso tra 0 e 255.

      Dopo aver creato la matrice, il thread main crea N thread figli ciascuno dei quali ha il compito di eseguire la somma di una riga della matrice.

      Ciascun thread aggiunge la somma che ha calcolato ad una variabile di nome sommaMat che al termine dell'esecuzione del programma conterrà la somma di tutti gli elementi della matrice. Il valore della variabile sommaMat deve essere stampato su video dal thread main. */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define N 20
#define M 1024
```

```
pthread_mutex_t mut; /* mutex condiviso tra threads */
int a[N][M];
int sommaMat=0;

void *sommaRiga_th(void *arg){
   int i=(int)arg;
   int j;
   int sommaRiga=0;
   for(j=0;j<M;j++)
       sommaRiga+=a[i][j];
   pthread_mutex_lock(&mut); /* prologo sez. critica */
   sommaMat += sommaRiga;
   printf("thread %d: sommaRiga=%d
       somma=%d\n",i,sommaRiga,sommaMat);
   //sleep(1);
   pthread_mutex_unlock(&mut); /* epilogo sez. critica */
   pthread_exit(0);
 }
```

```
int main () {
  int i,j;
  pthread_t th[N];
  pthread_mutex_init (&mut,NULL);
  srand(time(NULL)); // seme per random
  for (i=0;i<N;i++)
    for (j=0;j<M;j++)
      a[i][j]=rand()%256;
   for (i=0;i<N;i++)
    if (pthread_create(&th[i],NULL,sommaRiga_th,(int *)i)
       !=0) {
      fprintf (stderr, "errore create thread i \n");
      exit(1);
     }
  for (i=0;i<N;i++)
    pthread_join(th[i],NULL);
  printf("Somma = %d \n",sommaMat);
}
```

Semafori

- POSIX fornisce due tipi di semaforo: con nome o senza nome. Vedremo l'uso di semafori senza nome.

```
#include <semaphore.h>
sem_t sem;
int sem_init(sem_t *sem, int pshared,
             unsigned value);
```

La funzione ***sem_init()***, crea e inizializza il semaforo, ha tre parametri:

- *sem*, un puntatore al semaforo;
- *pshared,* un intero che indica il **livello di condivisione**;
- *value*, un intero che indica il valore iniziale del semaforo,

Se il livello di condivisione (secondo parametro) vale 0, il semaforo è condivisibile solo dai thread che appartengono allo stesso processo che ha creato il semaforo. Un valore diverso da 0, consente l'accesso al semaforo anche da parte di altri processi. Ad esempio,

```
sem_init(&sem, 0, 4);
```

crea un semaforo e lo inizializza al valore 4.

In questa libreria, le funzioni *wait()* e *signal()* prendono rispettivamente i nomi di ***sem_wait()*** e ***sem_post()***.

Altre utili funzioni sono: ***sem_trywait()*** che è la versione non bloccante di *sem_wait()*; s***em_getvalue()*** che restituisce il valore corrente di un semaforo; ***sem_destroy()*** che dealloca le risorse allocate per il semaforo;

Il frammento di codice seguente mostra l'uso del semaforo.

```
#include <semaphore.h>

   sem_t sem;
   …
   /* crea il semaforo */
   sem_init(&sem, 0, 1);
   …

   sem_wait (&sem); /* acquisisce il semaforo */

   /* SEZIONE CRITICA */

   sem_post(&sem); /* rilascia il semaforo */
   …
```

Variabili condizione (condition)

- La variabile **condizione** è uno strumento di sincronizzazione che permette ai thread di sospendersi nel caso sia soddisfatta una determinata condizione logica.
- Come per il mutex e per il semaforo, a una variabile *condizione* è associata una coda nella quale i thread possono essere sospesi.
- Tuttavia, a differenza del mutex e del semaforo, la variabile condizione non ha uno stato (libero o occupato) ed è quindi rappresentata solo da una coda di thread sospesi.
- Anche per le variabili condizione le operazioni fondamentali sono la **sospensione** e la **riattivazione** di thread.
- Con la libreria pthread la variabile condizione si definisce mediante il tipo di dato ***pthread_cond_t*.**

- La definizione:

  `pthread_cond_t` *`C`*`;`

  crea la variabile condizione ***C***. Una volta definita, una variabile *condition* deve essere inizializzata, ad esempio mediante la SC:

  ```
  int pthread_cond_init (pthread_cond_t* C,
      pthread_cond_attr_t* attr) ;
  ```

  dove:

  - ***C*** è la variabile *condizione* da inizializzare;
  - ***attr*** è un puntatore alla struttura che contiene eventuali *attributi* specificati per la *condition* (se NULL, viene inizializzata ai valori di default).

  Un thread si sospende in una coda di una variabile *condition,* se la condizione logica associata ad essa è verificata.

Ad esempio, nel problema del produttore-consumatore nel caso in cui ci siano più produttori e più consumatori è necessario che i produttori si sospendano se il buffer dei messaggi è pieno; al contrario, i consumatori si devono sospendere se il buffer è vuoto. La sospensione dei produttori si può ottenere associando alla condizione di ***buffer pieno*** una variabile condizione, come nel seguente esempio:

```
/*variabili globali: */
pthread_cond_t C;  /* variabile condizione */
pthread_mutex_t M; /* mutex per accedere in
                      mutua esclusione alla
                      condizione logica*/
int bufferpieno=0; /* variabile per esprimere
                       la condizione logica */
```

```
/* codice produttore: */
   pthread_mutex_lock(&M);
   if (bufferpieno) <sospensione sulla condition C>;
   <inserimento messaggio nel buffer>;
   pthread_mutex_unlock(&M);
```

- La variabile che si usa per stabilire l'accesso alla sezione critica (nell'esempio mostrato *bufferPieno*), essendo condivisa tra tutti i produttori, deve essere acceduta in mutua esclusione.
- Per garantire la mutua esclusione alla variabile condizione si associa un mutex.

  Pertanto, la chiamata *wait()* è realizzata con il seguente formato:

```
int pthread_cond_wait (pthread_cond_t * C,
pthread_mutex_t *M) ;
```

dove:

- *C* è la variabile condizione e
- *M* è il mutex associato ad essa.

Quando un thread *T* chiama la ***pthread_cond_wait()*** si ha che:

- ***il thread T* viene sospeso nella coda associata a C, e**
- **il mutex M viene liberato.**

- Quando il thread ***T*** viene riattivato, il mutex ***M*** è **automaticamente** posto a ***occupato***.
- In breve, un thread che si sospende chiamando ***pthread_cond_wait()*** libera il mutex ***M*** associato alla variabile condition ***C***, per poi rioccuparlo successivamente, quando sarà riattivato.

```
/*variabili globali: */
   pthread_cond_t C;
   pthread_mutex_t M; /* per accedere in  mutua
                         esclusione alla condizione
                         logica */
   int bufferpieno=0;

/* codice produttore: */
   pthread_mutex_lock(&M);
   if (bufferpieno) pthread_cond_wait(&C, &M);
   <inserimento messaggio nel buffer>;
   pthread_mutex_unlock(&M);
```

- Per riattivare un thread sospeso nella coda associata a una variabile condizione C si usa la funzione:

```
int pthread_cond_signal(pthread_cond_t * C)
```

Una chiamata a **pthread_cond_signal()** produce i seguenti effetti:

- **viene riattivato il primo *thread* sospeso nella coda associata a C, se presente;**
- **se non ci sono *thread* sospesi, la *signal()* non ha alcun effetto.**

- Per fare un esempio dell'uso delle variabili *condition,* consideriamo il caso in cui una risorsa può essere acceduta contemporaneamente da, al massimo, **MAX** thread. A tal fine, usiamo la variabile *condition* **PIENO**, nella cui coda saranno sospesi i thread che vogliono accedere alla risorsa quando questa è già usata da un numero MAX di thread.
- Indichiamo con la variabile intera **numTH** il numero di thread che stanno operando sulla risorsa:

```
#define MAX 10
int numTH=0; // numero di thread che usano la risorsa
pthread_cond_t PIENO; /* variabile condition */
pthread_mutex_t M; // mutex per mutua esclusione

void codice_thread () {
   pthread_mutex_lock(&M); /* prologo */
    /* controlla la condizione di accesso */
   if (numTH == MAX) pthread_cond_wait(&PIENO, &M);
   /* aggiorna lo stato della risorsa */
   numTH++;
   pthread_mutex_unlock(&M) ;
   <uso della risorsa>
   pthread_mutex_lock(&M); /* epilogo: */
   /* aggiorna lo stato della risorsa */
   numTH--;
   pthread_cond_signal(&PIENO);
   pthread_mutex_unlock(&M);
}
```