Sincronizzazione threads in Linux

Se più threads si contendono una risorsa, deve esistere un meccanismo che consenta di sincronizzarli, che quindi preveda che un thread, che aspetta il rilascio di una risorsa da parte di un altro thread, vada in uno stato di attesa o di wait.

Gli oggetti di sincronizzazione hanno lo scopo di consentire al programmatore di attivare tale meccanismo, sotto linux con con la libreria Pthread in C/C++ questi oggetti sono i Mutex e i Semafori.

Mutex

Mutex è una abbreviazione di "mutual exclusion", e le variabili Mutex sono principalmente utilizzate per la sincronizzazione dei thread e per la protezione di dati condivisi quando si prevedono scritture multiple.

Il concetto base di come una variabile mutex agisce in un programma multithreads è che un solo un thread può bloccare "lock" (o possedere) un mutex, anche se più threads provano a bloccare lo stesso mutex.

Quindi nessun altro thread può possedere un mutex fin quando esso non è rilasciato "unlocks" del thread che lo possiede.

Tipica l’uso di un mutex prevede:

- Creare e inizializare un mutex
- Più threads provano a bloccare il mutex
- Solo una ha sucessoed il thread possiede il mutex
- Il thread che possiede il mutex continua la sua elaborazione
- Il thread che possiede il mutex sblocca il mutex
- Un altro thread acquisisce il mutex continua la sua elaborazione
- E cosi via
- Infine il mutex è cancellato

Le funzioni per la gestione dei mutex dei pthread sono :

- pthread_mutex_init (mutex,attr)
- pthread_mutexattr_init (attr)
- pthread_mutexattr_settype (attr)
- pthread_mutexattr_destroy (attr)
- pthread_mutex_destroy (mutex)
- pthread_mutex_lock (mutex)
- pthread_mutex_trylock (mutex)
- pthread_mutex_unlock (mutex)

Una varabile Mutex è dichiarata di tipo **`pthread_mutex_t`** ed è possibile inizializzarla o in maniera statica o dinamica con la funzione pthread_mutex_init(), comunque il mutex è inizialmente unlocked.

pthread_mutex_init (mutex,attr)

La funzione pthread_mutex_init ha la seguente sintassi:

```
pthread_mutex_init(
pthread_mutex_t *MUTEX,
const pthread_mutexattr_t *MUTEXATTR
)
```

Essa inizializza di un oggetto di tipo pthread_mutex_t puntato dalla variabile MUTEX con attributi individuati da MUTEXATTR

pthread_mutexattr_init/settype(attr)

Gli attributi del mutex possono essere settati utilizzando le funzione pthread_mutexattr_init e pthread_mutexattr_settype per inizializzare e settare una variabile MUTEXATTR di tipo `pthread_mutexattr_t` passata per parametro ad uno dei seguenti valori:

fast valore di default in cui si preferisce la velocità alla correttezza, in questo caso non si ha nessun check sulla proprietà del mutex, ed si può rilasciare il mutex anche se non ci appartiene, e ogni thread può sbloccare un fast mutex. Inoltre non viè nessun controllo se il mutex è già stato “locked”, quindi è possibile che capiti un deadlock su se stessi, e non c’è controllo sulla corretta inizializzazione del mutex.

Per dichiarare un fast mutex si può utilizzare questa sintassi

```
pthread_mutex_t mutex;
...
pthread_mutex_init (&mutex, NULL);
```

oppure in maniera statica

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

error checking se si preferisce la correttezza rispetto ala velocità bisogna dichiarare un error checking mutex. In questo caso se si prova a bloccare lo stesso mutex due volte viene ritornato l'errore EDEADLK, e se si blocca un mutex non in proprio possesso viene ritornato l'errore EPERM. Per creare un

error checking mutex si deve inizializzare una variabile di tipo mutex_attr e passarla alla funzione pthread_mutex_init()

```
pthread_mutex_t mutex;
pthread_mutexattr_t attr;

pthread_mutexattr_init (&attr);
pthread_mutexattr_settype (&attr, PTHREAD_MUTEX_ERRORCHECK_NP);
pthread_mutex_init (&mutex, &attr);
```

oppure in maniera statica

```
pthread_mutex_t mutex =PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

recursive I mutex ricorsivi si comportano come un error checking mutex ad eccezione del fatto che si può bloccare lo stesso mutex più volte. Si tiene conto del numero di volte che esso è stato bloccato e si deve sbloccarlo tante volte quante si è

bloccato prima di sbloccarlo realmente. Per creare un recursive mutex si deve inizializzare una variabile di tipo mutex_attr e passarla alla funzione pthread_mutex_init()

```
pthread_mutex_t mutex;
pthread_mutexattr_t attr;

pthread_mutexattr_init (&attr);
pthread_mutexattr_settype(&attr,PTHREAD_MUTEX_RECURSIVE_NP);
pthread_mutex_init (&mutex, &attr);
```

Oppure in maniera statica

```
pthread_mutex_t mutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
```

pthread_mutexattr_destroy

La funzione pthread_mutexattr_destroy ha la seguente sintassi:

```
pthread_mutexattr_destroy(
const pthread_mutexattr_t *MUTEXATTR);
```

e rilasciar tutte le risorse occupate da MUTEXATTR

pthread_mutex_destroy

La funzione pthread_mutex_destroy ha la seguente sintassi:

```
pthread_mutex_destroy(pthread_mutex_t *MUTEX)
```

ed è utilizzata per eliminare e rilasciare le risorse allocate per la varibile MUTEX. La chiamata ha successo solamente se il mutex è in stato unlocked ed in questo caso viene restituito 0, in caso contrario la funzione ritorna l'error code EBUSY

pthread_mutex_lock

La funzione pthread_mutex_lock ha la seguente sintassi:

```
pthread_mutex_lock(pthread_mutex_t *MUTEX)
```

La funzione pthread_mutex_lock blocca un mutex è aggisce sul mutex a secondo del suo stato:

- Libero (unlocked)

  il thread chiamante ne prende possesso bloccandolo immediatamente e la funzione ritorna subito

- Bloccato (locked) da un altro thread

  il thread chiamante viene sospeso sino a quando il possessore non lo rilascia

- Bloccato (locked) dallo stesso thread

  dipende dal tipo mutex

  fast: stallo, perché è il chiamante stesso, che possiede il mutex, viene sospeso in attesa di un rilascio che non avverrà mai

  error checking: la chiamata fallisce

  recursive: la chiamata ha successo, ritorna subito, incrementa il contatore del numero di lock eseguiti dal thread chiamante

pthread_mutex_unlock

La funzione pthread_mutex_unlock ha la seguente sintassi:

```
pthread_mutex_unlock(pthread_mutex_t *MUTEX)
```

La funzione pthread_mutex_unlock sblocca un mutex che si assume fosse bloccato.

Ad ogni modo l'azione esatta dipende dal tipo di mutex

- fast: il mutex viene lasciato sbloccato e la chiamata ha sempre successo
- recursive: si decrementa il contatore del numero di lock eseguiti dal thread chiamante sul mutex, e lo si sblocca solamente se tale contatore si azzera

- error checking: sblocca il mutex solo se al momento della chiamata era bloccato e posseduto dal thread chiamante, in tutti gli altri casi la chiamata fallisce senza alcun effetto sul mutex

Consideriamo come esempio un programma dotprod_mutex.c che esegue il prodotto scalare, il dato principale è reso disponibile a tutti i thread attraverso una struttura globalmente accessibile, e ciascun thread lavora su una parte diversa dei dati. Il thread principale attende per tutti i thread per completare i loro calcoli, e poi la stampa, la somma risultante.

dotprod_mutex.c

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

/* structure that contains input ad output for the function
"dotprod"*/

typedef struct
 { double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;

/* Define globally accessible variables and a mutex */
#define NUMTHRDS 4
#define VECLEN 100
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;
```

```
/*The function dotprod activated when the thread is created.*/

void *dotprod(void *arg)
{  int i, start, end, len ;
   long offset;
   double mysum, *x, *y;
   offset = (long)arg;

   len = dotstr.veclen;
   start = offset*len;
   end   = start + len;
   x = dotstr.a;
   y = dotstr.b;

/* Perform the dot product and assign result to the appropriate
variable in the structure. */

   mysum = 0;
   for (i=start; i<end ; i++)
    {mysum += (x[i] * y[i]);}
```

```
/* Lock a mutex prior to updating the value in the shared
structure, and unlock it upon updating.*/
   pthread_mutex_lock(&mutexsum);
   dotstr.sum += mysum;
   pthread_mutex_unlock (&mutexsum);

   pthread_exit((void*) 0);
}

int main (int argc, char *argv[])
{
long i;
double *a, *b;
void *status;
pthread_attr_t attr;

/* Assign storage and initialize values */
a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
  for (i=0; i<VECLEN*NUMTHRDS; i++) {
  a[i]=1;b[i]=a[i];}
```

```
dotstr.veclen = VECLEN;
dotstr.a = a;
dotstr.b = b;
dotstr.sum=0;

pthread_mutex_init(&mutexsum, NULL);

/* Create threads to perform the dotproduct  */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

for(i=0;i<NUMTHRDS;i++)
  pthread_create(&callThd[i], &attr, dotprod, (void *)i);
pthread_attr_destroy(&attr);
/* Wait on the other threads */
for(i=0;i<NUMTHRDS;i++)
  pthread_join(callThd[i], &status);
/* After joining, print out the results and cleanup */
printf ("Sum =  %f \n", dotstr.sum);
free (a);free (b);
pthread_mutex_destroy(&mutexsum);
pthread_exit(NULL);}
```

Altro esempio Problema dei 5 filodofi a cena

filosofi.c

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define NPHILS 5
#define ITER 3
char *progname;
/* the chopsticks */
pthread_mutex_t chopstick[NPHILS];

/* print error message and die */
void error(char *f)
 {extern char *progname;
  if (progname) fprintf(stderr, "%s: ", progname);
  perror(f);exit(1);
}
 /* suspend execution of the calling thread */
```

```
void waiting(int min, int max)
 {sleep(rand()%(max-min+1) + min);
 }
void rightChopstick(int id)
{if (pthread_mutex_lock(&chopstick[id]))
  error("pthread_mutex_lock");
 printf("#%d got right chopstick\n", id);
}
void leftChopstick(int id)
{if(pthread_mutex_lock(&chopstick[(id+1) % NPHILS]))
  error("pthread_mutex_lock");
 printf("#%d got left chopstick\n", id);
}
void *philosopherRoutine(void *idp)
{ int id=*(int *)idp, i;
  for (i=0; i<ITER; i++)
  { printf("#%d is thinking\n", id);
   waiting(1, 10);
   printf("#%d is hungry\n", id);
   if (id % 2)
   {rightChopstick(id);
     waiting(1,2);
```

```
      leftChopstick(id);
     }
    else
     {leftChopstick(id);
      waiting(1,2);
      rightChopstick(id);
      }
    printf("#%d is eating\n", id);
    waiting(1, 10);
    if (pthread_mutex_unlock(&chopstick[id]))
      error("pthread_mutex_unlock");
    printf("#%d left right chopstick\n", id);
    if (pthread_mutex_unlock(&chopstick[(id+1) % NPHILS]))
     error("pthread_mutex_unlock");
    printf("#%d left left chopstick\n", id);
   }
 return NULL;
}
int main(int argc, char *argv[])
{int i;
  struct { int id;pthread_t thread_id;}
    philosopher[PHILS];
```

```
  progname=argv[0];
  srand(time(NULL));
  /* create mutex semaphores */
  for (i=0; i<NPHILS; i++)
   if (pthread_mutex_init(&chopstick[i], NULL))
    error("pthread_mutex_init");
  /* create and run the threads */
  for (i=0; i<NPHILS; i++)
   {philosopher[i].id=i;
     if (pthread_create(&philosopher[i].thread_id, NULL,
      philosopherRoutine, &philosopher[i].id))
      error("pthread_create");
   }
  /* wait for the threads to terminate */
for (i=0; i<NPHILS; i++)
 if (pthread_join(philosopher[i].thread_id, NULL))
  error("pthread_join");
 return 0;
}
```

Semafori

I semafori sono dei semplici contatori che, attraverso opportune funzioni, può essere atomicamente incrementato o decrementato. Per poter utilizzare i semafori occorre includere la libreria semaphores.h e dichiarare una variabile di tipo puntatore a sem_t. Le funzioni di gestione dei semafori sono:

- sem_init
- sem_destroy
- sem_wait
- sem_post
- sem_getvalue
- sem_trywait

sem_init

La funzione sem_init ha la seguente sintassi:

```
int sem_init (sem_t *SEM, int PSHARED, unsigned int VALUE)
```

Essa inizializza la variabile di tipo sem_t puntato dal primo argomento al valore espresso dalla variabile VALUE. Il valore PSHARED indica invece se il semaforo è utilizzato nell'ambito dello stesso processo (in questo caso ha valore 0) o se utilizzato anche da processi esterni (valore diverso da 0). Il valore restituito in caso di successo è 0 mentre in caso di errore la variabile ERRNO viene settata a EINVAL quando il contatore ha raggiunto il massimo espresso dalla variabile SEM_VALUE_MAX, o a ENOSYS quando l'argomento PSHARED è diverso da zero. Infatti alcune versioni più vecchie del kernel Linux non supporta semafori condivisi tra più processi.

sem_destroy

La funzione sem_destroy ha la seguente sintassi

```
int sem_destroy (sem_t *SEM)
```

Essa dealloca le risorse allocate per il semaforo puntato da SEM. Se la chiamata non ha successo la variabile ERRNO viene settata a EBUSY.

sem_wait

La funzione sem_wait ha la seguente sintassi

```
int sem_wait (sem_t *SEM)
```

Essa sospende il thread chiamante finché il valore del semaforo puntato dall'argomento è diverso da zero e viene decrementato automaticamente ed atomicamente, il contatore.

sem_post

La funzione sem_post ha la seguente sintassi

```
int sem_post (sem_t *SEM)
```

Essa al contrario della precedente incrementa il valore del semaforo passato come parametro. Se il semaforo ha già raggiunto il massimo numero consentito viene ritornato -1 mentre la variabile ERRNO viene settata ad EINVAL. In caso di successo, invece, viene restituito 0.

sem_getvalue

La funzione sem_getvalue ha la seguente sintassi

```
int sem_getvalue (sem_t *SEM,int *SVAL)
```

Essa imposta il valore della variabile puntata da SVAL al valore corrente del semaforo passato come primo parametro.

sem_trywait

La funzione sem_trywait ha la seguente sintassi

```
int sem_trywait (sem_t *SEM,int *SVAL)
```

Questa funzione è una variante non bloccante della funzione sem_wait ed è utilizzata principalmente per la decrementare il valore del semaforo.

Se un semaforo non è già uguale a zero una chiamata a sem_trywait fa decrementare di uno il valore del semaforo. In caso di successo viene ritornato 0, altrimenti (nel caso il valore del semaforo fosse già 0) viene restituito immediatamente -1 ed il valore della variabile ERRNO viene settato a EAGAIN.

Come esempio consideriamo il problema del barbiere che dorme. In un negozio di barbiere abbiamo

Un barbiere, una poltrona per il cliente servito, N sedie per clienti in attesa

Se non vi sono clienti nel negozio il barbiere dorme sulla poltrona, il primo cliente che entra nel negozio vuoto sveglia il barbiere.

I clienti che entrano trovando la poltrona occupata si mettono in attesa su una sedia.

Il cliente che non trova una sedia libera va via.

sleepbarber.c

```
// Compile gcc SleepBarber.c -o SleepBarber -lpthread -lm
#define _REENTRANT
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
// The maximum number of customer threads.
#define MAX_CUSTOMERS 25
void *customer(void *num);
void *barber(void *);
void randwait(int secs);
// Define the semaphores.
//waitingRoom Limits the # of customers allowed
// to enter the waiting room at one time.
sem_t waitingRoom;
// barberChair ensures mutually exclusive access to
// the barber chair.
sem_t barberChair;
// barberPillow is used to allow the barber to sleep
```

```
// until a customer arrives.
sem_t barberPillow;
// seatBelt is used to make the customer to wait until
// the barber is done cutting his/her hair.
sem_t seatBelt;
// Flag to stop the barber thread when all customers
// have been serviced.
int allDone = 0;
int main(int argc, char *argv[])
{pthread_t btid;
 pthread_t tid[MAX_CUSTOMERS];
 long RandSeed;
 int i, numCustomers, numChairs;
 int Number[MAX_CUSTOMERS];
 // Check to make sure there are the right number of
 // command line arguments.
 if (argc != 4)
  {printf("Use: SleepBarber \n"); exit(-1); }
   // Get the command line arguments and convert them
   // into integers.
   numCustomers = atoi(argv[1]);
   numChairs = atoi(argv[2]);
```

```
    RandSeed = atol(argv[3]);
  // Make sure the number of threads is less than the number of
  // customers we can support.
    if (numCustomers > MAX_CUSTOMERS)
     {printf("The maximum number of Customers is %d.\n",
      MAX_CUSTOMERS);
      exit(-1);
     }
    printf("\nSleepBarber.c\n\n");
    srand48(RandSeed);
    // Initialize the numbers array.
    for (i=0; i<MAX_CUSTOMERS; i++) Number[i] = i;
    sem_init(&waitingRoom, 0, numChairs);
    sem_init(&barberChair, 0, 1);
    sem_init(&barberPillow, 0, 0);
    sem_init(&seatBelt, 0, 0);
    // Create the barber.
    pthread_create(&btid, NULL, barber, NULL);
    // Create the customers.
    for (i=0; i<numCustomers; i++)
    {pthread_create(&tid[i], NULL,
     customer, (void *)&Number[i]);}
```

```
   // Join each of the threads to wait for them to finish.
   for (i=0; i<numCustomers; i++)
   { pthread_join(tid[i],NULL); }
   // When all of the customers are finished, kill the
   // barber thread.
   allDone = 1;
   sem_post(&barberPillow);
   // Wake the barber so he will exit.
   pthread_join(btid,NULL);
 }
void *customer(void *number)
 { int num = *(int *)number;
   // Leave for the shop and take some random amount of
   // time to arrive.
   printf("Customer %d leaving for barber shop.\n", num);
   randwait(5);
   printf("Customer %d arrived at barber shop.\n", num);
   // Wait for space to open up in the waiting room
   sem_wait(&waitingRoom);
   printf("Customer %d entering waiting room.\n", num);
   // Wait for the barber chair to become free
   sem_wait(&barberChair);
```

```
    // The chair is free so give up your spot in the
    // waiting room.
    sem_post(&waitingRoom);
    // Wake up the barber...
    printf("Customer %d waking the barber.\n", num);
    sem_post(&barberPillow);
    // Wait for the barber to finish cutting your hair
    sem_wait(&seatBelt);
    // Give up the chair.
    sem_post(&barberChair);
    printf("Customer %d leaving barber shop.\n", num);
 }


void *barber(void *junk)
  {// While there are still customers to be serviced...
   // Our barber is omnicient and can tell if there are
   // customers still on the way to his shop.
   while (!allDone)
   {// Sleep until someone arrives and wakes you..
    printf("The barber is sleeping\n");
    sem_wait(&barberPillow);
```

```
    // Skip this stuff at the end...
    if (!allDone)
    {// Take a random amount of time to cut the
     // customer's hair.
     printf("The barber is cutting hair\n");
     randwait(3);
     printf("The barber has finished cutting hair.\n");
     // Release the customer when done cutting
     sem_post(&seatBelt);
    }
   else
     {printf("The barber is going home for the day.\n");
     }
   }
}

void randwait(int secs)
  {int len; // Generate a random number...
   len = (int) ((drand48() * secs) + 1);
   sleep(len);
  }
```