Programació amb fils (2a part)

Lluís Garrido – lluis.garrido@ub.edu

Novembre 2014

Resum

En aquesta fitxa ens centrarem en les diverses formes de què disposem per sincronitzar fils: els semàfors i els monitors. Veurem com utilitzar-los amb fils i per a cada un d'ells veurem alguns exemples que poden ser d'utilitat per programar de forma concurrent i paral·lela.

Índex

1 Semàfors

En aquesta secció descriurem breument l'ús de semàfors per a sincronització de fils. Per utilitzar els semàfors cal incloure al codi C la següent instrucció

```
#include <semaphore.h>
```

Per compilar amb semàfors serà necessari compilar amb la llibreria *pthreads*

```
$ gcc programa.c -o programa -lpthread
```

Un semàfor es declara de la següent forma

```
sem_t mutex;
```

on *sem_t* és el tipus associat al semàfor.

Un semàfor es pot inicialitzar a un determinat valor amb la instrucció *sem_init*. Per exemple, la següent instrucció inicialitza la variable *mutex* a 1

```
sem_init(&mutex, 0, 1);
```

El segon paràmetre és, per defecte, 0. En cas que es posi a un valor diferent de 0 el semàfor pot ser compartit entre diversos processos; en cas contrari només pot ser compartit entre els fils que pertanyen al mateix procés. Cal tenir en compte però que per utilitzar semàfors entre processos es més recomanable utilitzar la funció *sem_open* que s'ha vist a les transparències.

El tercer paràmetre és el valor a què volem inicialitzar el semàfor. En aquest cas el semàfor s'inicialitza al valor 1.

Tal com s'ha vist a classe de teoria, hi ha dues funcions per manipular un semàfor: *sem_wait* i *sem_post*. Una forma de protegir una secció crítica de codi es basa en utilitzar el semàfor de la següent forma

```
sem_wait(&mutex);  /* protocol d'entrada */
// seccio critica
sem_post(&mutex);  /* protocol de sortida */
```

Les funcions anteriors inclouen les barreres de memòria necessàries perquè no hi hagi problemes d'accés a dades entre múltiples fils.

Al següent exemple es mostra una implementació d'un productor i un consumidor fent servir semàfors. El productor genera una sèrie de nombres que emmagatzema a un *buffer*; el consumidor agafa cadascun d'aquests nombres i els suma. El codi correspon a `sem_productor_consumidor.c`, veure Figura 1.

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

sem_t buit, ocupat; // semafors
int buffer;         // buffer compartit
int numIters;

void *productor(void *arg)
{
  int i;

  for(i = 1; i <= numIters; i++) {
    sem_wait(&buit);
    printf("Productor produeix la data %d\n", i);
    buffer = i;
    sem_post(&ocupat);
  }
}

void *consumidor(void *arg)
{
  int i, total = 0;

  for(i = 1; i <= numIters; i++) {
    sem_wait(&ocupat);
    printf("Consumidor agafa la dada %d\n", buffer);
    total = total + buffer;
    sem_post(&buit);
  }

  printf("El total es %d\n", total);
}

int main(int argc, char **argv)
{
  pthread_t prod, cons;

  if (argc != 2) {
    printf("Us: %s <numIters>\n", argv[0]);
    exit(1);
  }

  sem_init(&buit, 0, 1);   // buit = 1
  sem_init(&ocupat, 0, 0); // ocupat = 0

  numIters = atoi(argv[1]);
  pthread_create(&prod, NULL, productor, NULL);
  pthread_create(&cons, NULL, consumidor, NULL);
  pthread_join(prod, NULL);
  pthread_join(cons, NULL);
}

```

Figura 1: Codi `sem_productor_consumidor.c`, veure secció 1.

2 Monitors

En aquesta secció ens centrarem en els monitors; tant Java com C tenen suport per monitors. De fet tots dos els simulen ja que el concepte de monitor és lleugerament diferent al que hi ha implementat en aquests llenguatges. Formalment, un monitor s'associa a la programació orientada a objectes. Tots els membres d'un monitor disposen d'exclusió mútua implícita, és a dir, només un fil pot executar una de les funcions membre en un instant determinat. En el llenguatge Java i C l'exclusió mútua és explícita, és a dir, hem d'introduir a mà les instruccions per adquirir i alliberar la clau.

En el llenguatge C la llibreria *pthreads* (que hem vist a la fitxa anterior) ens proveeix també dels mecanismes per implementar els monitors. Això vol dir que per fer servir els monitors hem d'incloure al codi C la següent instrucció

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

A l'hora de compilar, ho hem de fer així

```
$ gcc programa.c -o programa -lpthread
```

Observeu l'opció "`-lpthread`" que indica al compilador que ha d'enllaçar amb la llibreria de fils.

2.1 Exclusió mútua

La llibreria *pthreads* ens ofereix de funcions per adquirir i alliberar una clau i així assegurar que només un fil pot executar la secció crítica. En aquest context una variable *mutex* és la variable associada a la clau. A la llibreria *pthreads* les claus es declaren amb el tipus *pthread_mutex_t*. Abans de utilitzar-la s'ha d'inicialitzar amb la funció *pthread_mutex_init*. Si la variable *mutex* es reserva dinàmicament (amb *malloc*, per exemple) s'haurà de cridar a *pthread_mutex_destroy* abans d'alliberar la memòria associada. En cas que la variable *mutex* es declari de forma estàtica, en comptes d'inicialitzar-la amb *pthread_mutex_init*, es pot inicialitzar a la constant *PTHREAD_MUTEX_INITIALIZER*.

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                                        pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Aquestes funcions retornen 0 en cas que l'operació s'hagi realitzat amb èxit, i un valor diferent de 0 en cas contrari. Per inicialitzar un *mutex* amb els paràmetres per defecte, hem de posar *attr* a *NULL*. En el cas de *Linux*, per exemple, un dels atributs per defecte associat a una variable *mutex* és que no és *reentrant*. És dir, un fil es bloquejarà (*deadlock*) en cas que el fil intenti adquirir dues vegades seguides la mateixa clau sense alliberar la clau abans d'intentar adquirir-la per segon cop. A la secció 3 es comentarà com fer que la variable *mutex* estigui associada a una clau *reentrant*.

Per adquirir una clau s'utilitzarà *pthread_mutex_lock*. En cas que algun altre fil ja hagi adquirit la clau sobre la variable, el fil que fa la crida a *pthread_mutex_lock* haurà d'esperar-se fins que l'altre fil hagi alliberar la clau. Per alliberar una clau es fa servir la funció *pthread_mutex_unlock*.

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

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

#define MAXFILS 500
#define NITERS  1000

pthread_t ntid[MAXFILS];
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int a;

void *thr_fn(void *arg)
{
  pthread_mutex_lock(&mutex);
  a++;
  pthread_mutex_unlock(&mutex);

  return ((void *)0);
}

int main(void)
{
  long int i, j;
  int err;

  for(j = 0; j < NITERS; j++) {
    a = 0;

    for(i = 0; i < MAXFILS; i++) {
      err = pthread_create(ntid+i, NULL, thr_fn, NULL);
      if (err != 0) {
        printf("no puc crear el fil numero %d.", i);
        exit(1);
      }
    }

    for(i = 0; i < MAXFILS; i++) {
      err = pthread_join(ntid[i], NULL);
      if (err != 0) {
        printf("error pthread_join al fil %d\n", i);
        exit(1);
      }
    }

    if (a != MAXFILS)
      printf("fil principal: iteracio j = %d, a = %d\n", j, a);
  }
}

```

Figura 2: Codi `mon_suma_fils.c`, veure secció 2.1.

Aquestes funcions retornen 0 si l'operació s'ha realitzat amb èxit, i un valor diferent de 0 en cas contrari. La funció *pthread_ mutex_ trylock* serveix per provar si la clau pot ser adquirida. Si un fil crida a *pthread_ mutex_ trylock* i la clau no està adquirida per un altre fil, el fil que fa la crida a *pthread_ mutex_ trylock* podrà adquirir la clau i la funció retornarà 0. Si la clau ha estat adquirida per un altre fil, la funció retornarà *EBUSY* sense esperar que l'altre fil alliberi la clau. Aquesta funcionalitat pot ser útil en cas que no ens puguem permetre que un fil estigui esperant que un altre fil alliberi la clau. D'aquesta forma, en detectar que la clau ha estat adquirida, el fil pot dedicar-se a fer altres coses i provar d'agafar la clau més endavant.

De forma similar als semàfors, les funcions d'adquisició i alliberament de claus inclouen les funcions necessàries de barreres de memòria.

En el següent codi es mostra un exemple que incrementa una variable global, veure la figura 2. Observar que la secció crítica, *a++*, es protegeix mitjançant les instruccions de *lock* i *unlock* corresponents. La clau que s'utilitza per protegir aquesta secció crítica està associada a la variable *mutex*, que és una variable global i per tant està compartida entre tots els fils. D'aquesta forma ens assegurem que a l'interior de la secció crítica només hi ha un fil.

La solució proposada aquí no es gaire eficient, ja que els fils agafaran i alliberaran múltiples vegades la clau *mutex*. Això pot provocar que els fils es bloquegin mútuament entre sí per entrar a la secció crítica. Una solució millor es basa en fer que cada fil realitzi primer les operacions d'increment amb una variable local i que només al final accedeixin a la variable global.

2.2 Variables condicionals

Les variables condicionals són les que ens permeten implementar la sincronització condicional. Les variables condicionals s'utilitzen per bloquejar un fil, posar-lo a una cua d'espera fins que es compleix una determinada condició. De la mateixa forma que amb l'exclusió mútua disposem de dues funcions per controlar els fils – una funció per adquirir la clau i una altra per alliberar-la –, amb les variables condicionals també en disposem de dues funcions: una funció per bloquejar els fils i una altra per desbloquejar-los.

Una variable condicional té el tipus *pthread_ cond_ t* i s'ha d'inicialitzar amb la funció *pthread_ cond_ init*. Si la variable condicional ha estat inicialitzada dinàmicament s'haurà de cridar a *pthread_ cond_ destroy* abans d'alliberar la memòria associada. En cas que la variable condicional es declari de forma estàtica, en comptes d'inicialitzar-la amb *pthread_ cond_ init*, es pot inicialitzar a la constant *PTHREAD_ COND_ INITIALIZER*.

```
int pthread_cond_init(pthread_cond_t *cond,
                                    pthread_cond_attr_t *attr);
int pthread_cond_destroy(ptrhead_cond_t *cond);
```

Aquestes funcions retornen 0 si l'operació s'ha executat amb èxit i un valor diferent de 0 en cas contrari. Per crear una variable condicional amb atributs per defecte, posem *attr* a *NULL* en el moment de cridar a la funció d'inicialització.

Utilitzarem *pthread_ cond_ wait* per fer que un fil es bloquegi

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

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

#define N 100

int comptador;
pthread_t ntid[N];
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond  = PTHREAD_COND_INITIALIZER;

void barrera()
{
  pthread_mutex_lock(&mutex);
  comptador = comptador - 1;
  if (comptador == 0) {
    comptador = N;
    pthread_cond_broadcast(&cond);
  } else {
    pthread_cond_wait(&cond, &mutex);
  }
  pthread_mutex_unlock(&mutex);
}

void *thr_fn(void *arg)
{
  int i, j;

  j = 0;
  for(i = 0; i < 10; i++)
  {
    j++;
    barrera();
    printf("%d ", j);
  }

  return ((void *)0);
}

int main(void)
{
  int i, err;

  comptador = N;

  for(i = 0; i < N; i++) {
    err = pthread_create(ntid+i, NULL, thr_fn, NULL);
    if (err != 0) {
      printf("no puc crear el fil numero %d.", i);
      exit(1);
    }
  }

  for(i = 0; i < N; i++) {
    err = pthread_join(ntid[i], NULL);
    if (err != 0) {
      printf("error pthread_join al fil %d\n", i);
      exit(1);
    }
  }
}

```

Figura 3: Codi `mon_barrera.c`, veure secció 2.2.

```
                                pthread_mutex_t *mutex,
                                struct timespec *timeout);
```

Aquestes funcions retornen 0 si l'operació s'ha executat amb èxit i un valor diferent de 0 en cas contrari. Observar que la funció *pthread_ cond_ wait* té dos paràmetres: el primer és la variable de condició sobre la qual el fil es bloquejarà; el segon és la variable *mutex* associada a la condició. Un fil no pot cridar a *pthread_ cond_ wait* sense adquirir abans la clau sobre la variable *mutex* corresponent. En el moment de cridar a *pthread_ cond_ wait* el fil que fa la crida allibera la clau associada a *mutex*. D'aquesta forma altres fils poden adquirir-la. De la mateixa forma, en desbloquejar un fil associada a la variable de condició *cond* aquest s'esperarà a adquirir la variable *mutex*.

La funció *pthread_ cond_ timedwait* és similar a *pthread_ cond_ wait* amb l'afegit que permet que un fil es bloquegi durant un temps especificat al tercer argument. Si el fil no es desbloqueja en aquest temps la funció retornarà amb un codi d'error *ETIMEDOUT* (abans però haurà d'adquirir la clau sobre *mutex*). El temps d'espera s'especifica mitjançant l'estructura *timespec*. Amb aquesta estructura el temps s'ha d'especificar de forma absoluta en comptes de relatiu al moment en què es crida.

Disposem de dues funcions per desbloquejar els fils: d'una banda, la funció *pthread_ cond_ signal* permet desbloquejar només un fil associat a una variable condicional mentre que la funció *pthread_ cond_ broadcast* permet desbloquejar tots els fils associats a una variable condicional.

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

A continuació mostrem un exemple d'implementació de barrera fent servir monitors. El codi correspon a `mon_barrera.c` i es pot veure a la Figura 3.

Finalment tornem a veure l'exemple del productor-consumidor de la secció 1 re-escrit fent servir variables condicionals. Observar que utilitzem "*if*" en comptes de "*while*" per bloquejar un fil. En aquest cas no hi ha problema ja que només hi ha un fil productor i només hi ha un fil consumidor. El codi correspon a `mon_productor_consumidor.c`. Observar que aquest codi utilitza les funcions *pthread_ mutex_ init* i *pthread_ cond_ init* per inicialitzar les variables (s'hauria pogut inicialitzar també el *mutex* com a l'exemple anterior).

2.3 Lectors i escriptors

La llibreria *pthreads* ens proporciona unes funcions especialitzades per implementar el paradigma dels lectors i escriptors. Amb una variable de *mutex* només un fil pot executar la secció crítica. Per tant, només un fil pot adquirir la clau. Tota la resta hauran d'esperar que el fil alliberi la clau per poder adquirir-la. En una clau associada a lectors i escriptors hi ha tres estats possibles: clau adquirida en mode de lectura, clau adquirida en mode d'escriptura i clau alliberada. Només un fil pot adquirir la clau en mode escriptura, però múltiples fils poden adquirir la clau en mode lectura.

Quan un fil té adquirida la clau en mode d'escriptura tota la resta de fils s'hauran d'esperar que el primer alliberi la clau per poder adquirir la clau. Quan un fil té adquirida la clau en mode lectura tots els fils que vulguin adquirir-la també en mode lectura seran admesos, però qualsevol fil que vulgui adquirir-la en mode d'escriptura haurà d'esperar-se fins que tots els fils lectors alliberin la seva clau. La implementació d'aquest tipus de clau pot variar de la plataforma, però usualment un lector es queda esperant si hi ha algun escriptor que també està esperant a agafar la clau. Això impedeix que un escriptor esperi de forma indefinida que els lectors alliberin la seva clau.

```c
pthread_mutex_t mutex;
pthread_cond_t cua_cons, cua_prod;
int buffer;  // buffer compartit
int buit;    // 1 si buit, 0 si ple
int numIters;

void *productor(void *arg)
{
  int i;

  for(i = 1; i <= numIters; i++) {
    pthread_mutex_lock(&mutex);
    if (!buit)
      pthread_cond_wait(&cua_prod, &mutex);
    printf("Productor produeix la data %d\n", i);
    buffer = i;
    buit   = 0;
    pthread_cond_signal(&cua_cons);
    pthread_mutex_unlock(&mutex);
  }
}

void *consumidor(void *arg)
{
  int i, total = 0;

  for(i = 1; i <= numIters; i++) {
    pthread_mutex_lock(&mutex);
    if (buit)
      pthread_cond_wait(&cua_cons, &mutex);
    printf("Consumidor agafa la dada %d\n", buffer);
    total = total + buffer;
    buit  = 1;
    pthread_cond_signal(&cua_prod);
    pthread_mutex_unlock(&mutex);
  }

  printf("El total es %d\n", total);
}

int main(int argc, char **argv)
{
  pthread_t prod, cons;

  if (argc != 2) {
    printf("Us: %s <numIters>\n", argv[0]);
    exit(1);
  }

  buit = 1;

  pthread_mutex_init(&mutex, NULL);
  pthread_cond_init(&cua_cons, NULL);
  pthread_cond_init(&cua_prod, NULL);

  numIters = atoi(argv[1]);
  pthread_create(&prod, NULL, productor, NULL);
  pthread_create(&cons, NULL, consumidor, NULL);
  pthread_join(prod, NULL);
  pthread_join(cons, NULL);
}

```

Figura 4: Codi `mon_productor_consumidor.c`, veure secció 2.2.

Tot i que els lectors i escriptors es poden implementar mitjançant l'exclusió mútua i les variables condicionals, la llibreria *pthreads* ofereix a l'usuari una implementació d'aquesta funcionalitat. El tipus associat a una clau d'aquest tipus és *pthread_ rwlock_ t*. De forma similar a abans, aquest tipus de claus s'han d'inicialitzar abans de ser utilitzades i destruir abans d'alliberar la memòria associada.

```
int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                                    pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Aquestes funcions retornen 0 en cas d'èxit, i un valor diferent de 0 en cas contrari. Per inicialitzar els atributs per defecte hem de passar un punter *NULL* als seus atributs.

Per adquirir i alliberar una clau lector-escriptor disposem d'aquestes funcions

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Com ja indica el nom de la funció, la primera permet adquirir la clau en mode lector, la segona permet adquirir la clau en mode escriptor mentre que la tercera permet alliberar la clau. Les funcions retornen 0 si s'han executat amb èxit, i un valor diferent de 0 en cas contrari. En general les implementacions poden tenir un límit superior al nombre de lectors que poden tenir adquirida a la vegada la clau i per tant es recomanable comprovar el codi d'error en cas que vulguem adquirir la clau en mode lector. Per a les dues altres funcions no es necessari comprovar el codi de retorn sempre que l'algorisme que programem estigui ben fet. Els errors que poden sorgir són per exemple utilitzar la clau sense inicialitzar-la abans o bé quan un fil intenta adquirir una clau que ja té adquirida.

La llibreria *pthreads* també pot tenir implementada aquestes funcions

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

Si la clau pot ser adquirida les funcions retornen un 0. En cas contrari retornen un *EBUSY* sense esperar a adquirir la clau.

3 Deadlocks i claus reentrants

Un fil es bloquejarà de forma indefinida si intenta agafar la mateixa variable de *mutex* dues vegades seguides, però també hi ha altres formes en què els fils es poden bloquejar. Per exemple, si utilitzem més d'una variable *mutex*, per exemple *mutex1* i *mutex2*, podem bloquejar els fils de forma indefinida si es produeix aquesta successió d'esdeveniments: el primer fil adquireix *mutex1* i el segon fil adquireix *mutex2*. A continuació el primer fil intenta adquirir *mutex2* mentre que el segon fil intenta adquirir *mutex1*. En aquesta situació tots dos fils es bloquejaran de forma indefinida esperant que un altre fil alliberi la clau corresponent. En anglès aquesta situació s'anomena *deadlock*.

Els deadlock es poden evitar si a l'hora de programar es controla de forma curosa l'ordre en què s'agafen les claus. Per evitar *deadlocks* una bona recomanació és que tots els fils agafin les claus sempre en el mateix ordre. A l'exemple anterior això vol dir que el segon fil hauria d'agafar *mutex1*

abans d'agafar *mutex2*. D'aquesta ens assegurem que no es puguin produir problemes d'aquests tipus.

En certes situacions però es pot fer difícil organitzar el codi de forma que l'adquisició de claus estigui ordenada. En cas que hi hagi moltes claus i estructures pot ser complicat organitzar una jerarquia de claus. En aquest cas hem d'utilitzar altres funcions com per exemple *pthread_mutex_trylock* (veure secció 2.1). Si en cridar a la funció es pot adquirir la clau la funció retornarà un 0 i per tant podrem procedir com usualment. En cas contrari es poden alliberar les claus que hem adquirit (assegurant abans que deixem les dades en un estat consistent) i provar d'agafar les claus més endavant.

Per poder fer una clau *reentrant* (és dir, adquirir dues vegades seguides la mateixa clau sense alliberar la clau abans d'intentar adquirir-la per segon cop) cal inicialitzar la clau tal com es mostra al següent exemple. La clau només s'ha d'inicialitzar d'aquesta forma en cas que realment sigui necessari agafar la clau dues vegades seguides.

```
pthread_mutexattr_t attr;
pthread_mutex_t mutex;

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

4 Bibliografia

1. Rochkind, M.J. Advanced Unix Programming. Addison Wesley, 2004.

2. Stevens, W.R. Advanced Programming in the Unix Environment. Addison Wesley, 2005.