Monitors

Sistemes Operatius 2

Grau d'Enginyeria Informàtica

Introducció

Característiques

- Poden ser utilitzats per sincronitzar fils o processos.
- Permeten més graus de llibertat que els semàfors a l'hora de programar. Això és perquè hi ha funcions específiques per:
  - Controlar l'entrada i sortida d'una secció crítica.
  - Sincronitzar fils adormint o despertant-los d'una cua.
- Permeten gestionar escenaris de *priority inversion*.
- La gran majoria dels llenguatges (C, Java, ...) disposen de monitors per sincronitzar fils..

Introducció

Tingueu en compte que

- Les instruccions (en C) per utilitzar semàfors o monitors són diferents depenent de si treballem amb Windows o sistemes Unix. Els sistemes Unix (Linux, Mac, ...) utilitzen l'estàndard POSIX.
- Els monitors en C es coneixen també amb el nom de POSIX threads.

Sintaxi i semàntica

A l'hora de programar en C, una clau es declara com

```
pthread_mutex_t mutex;
```

- La variable mutex és la clau! La paraula "mutex" significa "Mutual exclusion" (exclusió mutua).
- Aquesta variable és la que s'ha d'utilitzar per entrar i sortir de la secció crítica. Inclou una cua sobre la qual els fils s'adormen.
- Per inicialitzar una variable de mutex es pot utilitzar la funció `pthread_mutex_init`. Per defecte la variable de mutex s'inicialitza desbloquejada.

Sintaxi i semàntica

En els monitors el mutex és binari: està bloquejat o desbloquejat. No existeixen mutex no binaris com amb els semàfors.

Per entrar i sortir d'una secció crítica

- `pthread_mutex_lock(mutex)`: Agafa el bloqueig sobre la clau. S'utilitza perquè un fil entri a la secció crítica. Només un fil hi pot entrar. La resta que vol entrar a la secció crítica es queda esperant (generalment de forma passiva) a la cua associada a la clau.
- `pthread_mutex_unlock(mutex)`: Allibera el bloqueig sobre la clau. S'utilitza per sortir de la secció crítica. El mateix fil que crida al lock ha de cridar al unlock. En fer unlock es deperten els fils que estan esperant a la clau.

Sintaxi i semàntica

Els monitors incorporen a més funcions addicionals per adormir o despertar fils d'una cua per realitzar la sincronització entre fils.

La cua sobre la qual els fils s'adormen és

```
pthread_cond_t cond;
```

- Aquesta variable s'anomena variable condicional. Només es pot utilitzar amb els monitors, no pas els semàfors.
- Intuïtivament només és una cua on els fils s'hi poden adormir de forma "voluntària".

Sintaxi i semàntica

Per utilitzar aquesta cua tenim aquestes tres funcions en C. Cal tenir agafada la clau `mutex` per cridar-les.

- `pthread_cond_wait(cond, mutex)`: el fil que ho executa s'adorm (al final de) la cua `cond` i allibera la clau `mutex`.
- `pthread_cond_signal(cond)`: el fil que ho executa desperta al primer fil de la cua `cond`. El fil que es desperta haurà d'agafar la clau `mutex` (que havia alliberat en fer el `wait`) per continuar executant. Si la clau `mutex` està agafada, el fil espera que s'alliberi la clau per agafar-la.
- `pthread_cond_broadcast(cond)`: el fil que ho executa desperta a tots els fils de la cua `cond`. Tots els fils es posaran a la competir per agafar la clau `mutex`. Només un ho aconseguirà així que s'alliberi. La resta de fils esperen que s'alliberi la clau per agafar-la.

Sintaxi i semàntica

```
pthread_mutex_t mutex;
pthread_cont_t cond;

int adormit = 0;

void *thread_fn(void *arg)
{
   // Cada fil te un identificador sencer
   int fil = (int *) arg;

   // Agafem la clau
   pthread_mutex_lock(&mutex);

   // Si soc el fil 0 ens adormim i
   // alliberem la clau
   if (fil == 0) {
       adormit = 1;
       pthread_cond_wait(&cond, &mutex);
   }

   // Si soc el fil 1 i el fil 0 esta
   // dormint el desperto de la cua
   if (fil == 1) && (adormit)
       pthread_cond_signal(&cond);

   pthread_mutex_unlock(&mutex);

}
```

- Suposem només dos fils: fil 0 i fil 1.
- Les funcions wait i signal es criden a l'interior de la secció crítica. No fer-ho pot portar a un comportament del programa no previsible.

Sintaxi i semàntica

Suposem que el fil 0 és el primer en entrar a la secció crítica

- El fil 0 s'adormirà al wait i alliberarà la clau `mutex` (equivalent a fer un unlock).
- El fil 1 podrà entrar a la secció crítica i veurà el fil 0 adormit. El fil 1 despertarà el fil 0 de la cua `cond` amb el signal.
- El fil 0 es desperta, intentarà agafar la clau `mutex` però no pot ja que el fil 1 la té. Immediatament el fil 0 s'adorm a la cua de la clau `mutex`.
- Així que el fil 1 surti de la secció crítica allibera la clau `mutex` i desperta tots els fils que estan dormint en aquesta clau. Això fa que el fil 0 es desperti i pugui continuar l'execució!
- Important! El fil que fa unlock només desperta els fils que estan esperant per agafar el `mutex`, no desperta els fils adormits a la variable `cond`.

Sintaxi i semàntica

```
pthread_mutex_t mutex1, mutex2;
pthread_cont_t cond;

int adormit = 0;

void *thread_fn(void *arg)
{
   // Cada fil te un identificador sencer
   int fil = (int *) arg;

   pthread_mutex_lock(&mutex1);
   pthread_mutex_lock(&mutex2);

   // Si soc el fil 0 ens adormim i
   // alliberem la clau
   if (fil == 0) {
       adormit = 1;
       pthread_cond_wait(&cond, &mutex2);
   }

   // Si soc el fil 1 i el fil 0 esta
   // dormint el desperto de la cua
   if (fil == 1) && (adormit)
       pthread_cond_signal(&cond);

   pthread_mutex_unlock(&mutex2);
   pthread_mutex_unlock(&mutex1);
}
```

- Aquest codi no funcionarà en determinats casos!
- En cridar al wait el fil 0 només alliberarà la clau mutex2, no pas la mutex1. Per tant, tal com està el codi, el fil 1 no podrà entrar a la secció crítica de mutex1 si el fil 0 hi entra abans.

Algorismes amb monitors

Anem a veure els següents algorismes amb monitors

- Implementació d'un semàfor amb monitors
- Implementació d'una barrera (paradigma paral·lelisme iteratiu)
- Productors i consumidors
- Lectors i escriptors

A l'hora d'analitzar els algorismes no us espanteu!

- Identifiqueu primer les seccions crítiques (les funcions lock i unlock)
- Identifiqueu després on són els wait i el signal/broadcast

Implementació d'un semàfor amb monitors (algorisme 1)

variables globals:

```
pthread_mutex_t mutex;   pthread_cond_t cond;
int s = M; // semàfor general
```

sem_wait:

```
pthread_mutex_lock(&mutex);
while (s == 0) {
  pthread_cond_wait(&cond, &mutex);
}
s = s - 1;
pthread_mutex_unlock(&mutex);
```

sem_post:

```
pthread_mutex_lock(&mutex);
s = s + 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

Implementació d'un semàfor amb monitors (algorisme 1)

Observeu que

- Mitjançant monitors podem implementar el protocol d'entrada i sortida dels semàfors.
- Amb el signal a la funció sem_post despertem només el primer fil adormit a la variable cond. També es pot utilitzar un broadcast.
- Recordar de les transparències dels semàfors que mitjançant aquesta solució es pot limitar l'accés dels fils a la secció crítica. En particular, en aquesta codi hi pot haver a tot estirar M fils a la secció crítica. Així s'aconsegueix limitar l'accés als recursos.

Implementació d'un semàfor amb monitors (algorisme 1)

És **molt important utilitzar el while**. La raó és que en despertar-se el fil adormit al while cal tornar a comprovar la condició s == 0 ja que és possible que passi el següent

1. Suposem que un fil A ha cridat (i retornat) de sem_wait i que s és 0. Suposem també hi ha un altre fil B adormit per entrar al lock del sem_wait i un altre fil C adormit al wait.
2. Quan el fil A crida a sem_post es posa s a 1, es fa un signal i un unlock.
3. El fil C es desperta de cond i competeix amb B per agafar la clau mutex. Qualsevol dels dos pot obtenir-la primer.
4. Suposem que l'agafa primer el fil B. Aquest posa s a 0. En fer el fil B l'unlock del sem_wait, el fil C agafa la clau.
5. Si només fem un "if" (i no un while) el fil B decrementarà s i el posarà a -1!! Fatal!!! És per això que cal fer un while: amb un while el fil B es tornarà a dormir a la variable condicional si s és 0.

Els "whiles"

Regla molt important! Típicament els "waits" es realitzen quan es compleix una condició

```
if (condicio)
        wait(cond, mutex)
```

Això generalment no funcionarà! En despertar-se un fil d'un wait, cal que torni a comprovar la condició que el va fer adormir

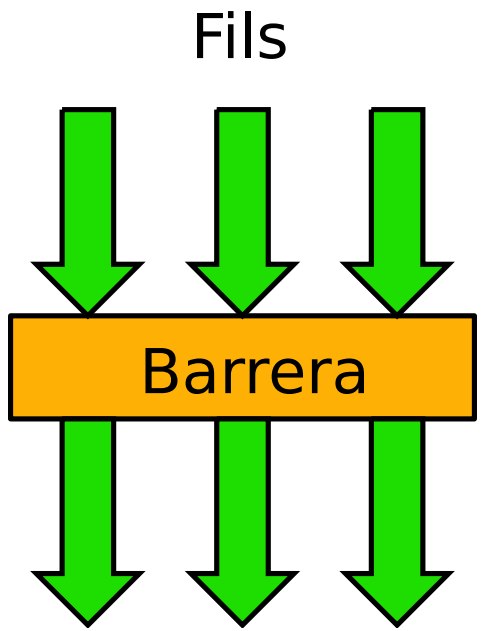
```
while (condicio)
        wait(cond, mutex)
```

És important que torni a comprovar la condició que el va fer adormir perquè, en general, en continuar el fil l'execució creurà que la condició es compleix. En general tots els algorismes fan servir whiles!

Implementació d'una barrera (algorisme 2)

Una barrera és un mecanisme de sincronització de fils molt utilitzat en computació paral·lela (paradigma de paral·lelisme iteratiu)

- Cada fil realitza la seva part del processament.
- En acabar el fil crida a la funció `barrera`: el fil s'adorm per esperar que els altres acabin.
- El darrer fil que crida a barrera ha de despertar a tota la resta perquè els fils puguin continuar.



Implementació d'una barrera (algorisme 2)

Veure codi `monitor_barrera.c`.

variables globals:

```
pthread_mutex_t mutex;  pthread_cond_t cond;
int comptador = N; // nombre de fils
```

barrera:

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

Implementació d'una barrera (algorisme 2)

Observeu que

- Tot fil que entri a la barrera i no sigui el darrer es posarà a dormir a la cua `cond`.
- El darrer fil desperta tots els fils adormits a la cua cond amb broadcast. En alliberar el darrer fil la clau mutex, els fils despertats agafen un a un la clau mutex al wait i l'alliberen a l'unlock.

Múltiples cons. i prod. amb un buffer de mida M (algo. 3)

variables globals:

```
typeT buffer[M];   int w = 0, r = 0;
int comptador = 0; // nombre d'elements ocupats
pthread_mutex_t mutex;    pthread_cond_t condP, condC;
```

productor:

```
while (true) {
  generar "data"
  lock(&mutex);
  while (comptador == M) {
      wait(&condP, &mutex);
  }
  transferir "data" a "buffer[w]"
  w = (w + 1) % M;
  comptador++;
  signal(&condC);
  unlock(&mutex);
}
```

consumidor:

```
while (true) {
  lock(&mutex);
  while (comptador == 0) {
    wait(&condC, &mutex);
  }
  transferir "buffer[r]" a "data"
  r = (r + 1) % M;
  comptador--;
  signal(&condP);
  unlock(&mutex);
  consumir "data"
}
```

Múltiples consumidors i productors amb un buffer de mida M (algorisme 3)

Comparar aquesta solució amb la dels semàfors[1]. Veure codi `monitor_1prod1cons.c` i `semafor_1prod1cons.c`.

El productors

- En col·locar un element al buffer avisa amb signal als consumidors esperant per agafar una dada.
- En emplenar-se el buffer esperen al wait (amb un while!)

Els consumidors

- En agafar una dada del buffer avisa amb signal als productors esperant per col·locar-hi una dada.
- Si el buffer es buit els consumidors esperen al wait (amb un while!)

[1] A l'algorisme anterior no s'utilitzen les funcions amb el nom complet per falta d'espai.

Lectors i escriptors amb preferència sobre els lectors (algorisme 4)

variables globals:

```
int nr = 0;    boolean w = false;
pthread_mutex_t mutex;    pthread_cond_t cond;
```

read_lock:

```
lock(&mutex);
while (w) {
      wait(&cond, &mutex);
}
nr++;
unlock(&mutex);
```

read_unlock:

```
lock(&mutex);
nr--;
if (nr == 0) broadcast(&cond);
unlock(&mutex);
```

write_lock:

```
lock(&mutex);
while (nr > 0 || w) {
      wait(&cond, &mutex);
}
w = true;
unlock(&mutex);
```

write_unlock:

```
lock(&mutex);
w = false;
broadcast(&cond);
unlock(&mutex);
```

Lectors i escriptors amb preferència sobre els lectors (algorisme 4)

Observeu que

- La variable w s'utilitza per saber si l'escriptor està escrivint.
- S'utilitza una única variable condicional cond sobre la qual dormen lectors i escriptors.
- Els lectors tenen preferència sobre els escriptors: si hi ha un lector llegint poden entrar nous lectors encara que hi hagi escriptors esperant.
- En fer un lector un broadcast (al read_unlock) a tot estirar hi ha escriptors dormint al wait. En fer un escriptor un broadcast (al write_unlock) hi pot haver lectors i escriptors dormint al wait.

Lectors i escriptors justos (algorisme 5)

variables globals:

```
int nr = 0;    boolean w = false;
pthread_mutex_t mutex;    pthread_cond_t cond;
```

read_lock:

```
lock(&mutex);
while (w) {
    wait(&cond, &mutex);
}
nr++;
unlock(&mutex);
```

read_unlock:

```
lock(&mutex);
nr--;
if (nr == 0) broadcast(&cond);
unlock(&mutex);
```

write_lock:

```
lock(&mutex);
while (w) {
    wait(&cond, &mutex);
}
w = true;
while (nr != 0) {
    wait(&cond, &mutex);
}
unlock(&mutex);
```

write_unlock:

igual que abans

Lectors i escriptors justos (algorisme 5)

Observeu que

- La variable w s'utilitza per indicar que un escriptor desitja escriure (no pas que està escrivint).
- S'utilitza una única variable condicional cond sobre la qual dormen lectors i escriptors, igual que abans.
- El lector, abans de llegir, comprova si un escriptor vol escriure. Si és així es posa a dormir.
- L'escriptor, abans d'escriure, comprova si un lector està llegint. Si és així es posa a dormir.
- En fer un broadcast els lectors i escriptors competeixen per adquirir la clau mutex. Si w és *true*, només un escriptor podrà accedir al recurs.