

Algorismes de sincronització amb espera activa

Sistemes Operatius 2

Grau d'Enginyeria Informàtica

Objectius

- Presentar algorismes de sincronització per a 2 fils (i múltiples fils). No tots proveeixen exclusió mútua! Ho sabreu trobar?
- Presentar com s'aconsegueix la sincronització amb semàfors i monitors.

Suposem una estructura d'un codi com aquest, executada per 2 fils:

codi independent per a cada fil

lock(clau)

secció crítica

unlock(clau)



codi independent per a cada fil

En el codi anterior

- La variable `clau` és una variable compartida entre els fils.
- Els fils tenen una part de codi que no interfereix amb l'altre fil.
- Quan un fil vol accedir a la **secció crítica** (el recurs compartit) ha de cridar a la funció `lock`.
- Quan un fil vol sortir de la **secció crítica** ha de cridar a la funció `unlock`.

A continuació presentarem diferents implementacions de la funció **lock** i **unlock** (només per a 2 fils).

Per analitzar els algorismes que anem a veure:

- Primer analitzarem com entra un fil a la secció crítica (sense la competència de l'altre fil).
- Amb el primer fil a la secció crítica, pot entrar el segon a la secció crítica ? 
- Hi ha alguna forma que els dos fils entrin a la secció crítica ? 

Algorisme 1 per a dos fils

variables globals:

```
boolean flag[2] = {false, false};
```

lock fil 0:

```
while (flag[1]) {};  
flag[0] = true;  
return;
```

unlock fil 0:

```
flag[0] = false;  
return;
```

lock fil 1:

```
while (flag[0]) {};  
flag[1] = true;  
return;
```

unlock fil 1:

```
flag[1] = false;  
return;
```

Observar que el “return” permet entrar (lock) o sortir (unlock) de la secció crítica.

Algorisme 1 per a dos fils



Observeu que:

- L'algorisme 1 no proveeix exclusió mútua. Hi ha formes d'aconseguir que els dos fils siguin a l'interior de la secció crítica.
- Al següent algorisme invertim les instruccions de la funció lock.

Algorisme 2 per a dos fils

variables globals:

```
boolean flag[2] = {false, false};
```

lock fil 0:

```
flag[0] = true;  
while (flag[1]) {};  
return;
```

unlock fil 0:

```
flag[0] = false;  
return;
```

lock fil 1:

```
flag[1] = true;  
while (flag[0]) {};  
return;
```

unlock fil 1:

```
flag[1] = false;  
return;
```

Algorisme 2 per a dos fils



Observeu que

- L'algorisme anterior satisfà l'exclusió mútua. Mai dos fils poden ser a la secció crítica.
- En canvi, aquí es pot produir un deadlock.

Al següent algorisme veurem

- Una solució que satisfà exclusió mútua i no provoca deadlock.
- Utilitza una variable `victim` per decidir quin fil no entra a la secció crítica en cas que tots dos vulguin fer-ho.

Algorisme 3 (de Peterson) per a dos fils

variables globals:

```
int victima;  
boolean flag[2] = {false, false};
```

lock fil 0:

```
flag[0] = true;  
victima = 0;  
while (flag[1] &&  
    victima == 0) {};  
return;
```

unlock fil 0:

```
flag[0] = false;  
return;
```

lock fil 1:

```
flag[1] = true;  
victima = 1;  
while (flag[0] &&  
    victima == 1) {};  
return;
```

unlock fil 1:

```
flag[1] = false;  
return;
```

Algorisme 3 (de Peterson) per a dos fils



Observeu que

- L'algorisme proposat satisfà l'exclusió mútua i no provoca deadlock.
- Què passaria si invertim d'ordre les dues instruccions abans del `while` ? Veiem-ho al següent codi...

Algorisme 4 per a dos fils

variables globals:

```
int victima;  
boolean flag[2] = {false, false};
```

lock fil 0:

```
victima = 0;  
flag[0] = true;  
while (flag[1] &&  
    victima == 0) {};  
return;
```

unlock fil 0:

```
flag[0] = false;  
return;
```

lock fil 1:

```
victima = 1;  
flag[1] = true;  
while (flag[0] &&  
    victima == 1) {};  
return;
```

unlock fil 1:

```
flag[1] = false;  
return;
```

Algorisme 4 per a dos fils

Observeu que

- L'algorisme ja no permet obtenir exclusió mútua. Sembla doncs que només s'ha d'anar amb compte a l'hora de programar la funció lock!

Però... l'algorisme 3 (de Peterson) tampoc funciona a les màquines d'avui en dia!

- Els compiladors poden canviar l'ordre de les instruccions si així (creuen que) generen un codi més eficient.
- Les CPUs poden canviar l'ordre de les instruccions per eficiència.
- Amb múltiples CPUs cal tenir en compte el protocol de sincronització de les memòries cau (*cache*). Arquitectures diferents poden tenir protocols diferents!

Instruccions atòmiques

Avui en dia, les màquines multiprocessadores incorporen **instruccions màquina específiques** per assegurar la bona sincronització entre fils i la memòria cau.

Una d'aquestes és la **Get-and-Set**, que es defineix com

boolean Get-and-Set(boolean *variable)

```
<boolean value = *variable;  
  *variable = true;  
  return value;>
```

La funció **executa de forma atòmica** (bloquejant el bus)

- 1 Guarda una còpia del valor original de la memòria.
- 2 Posa la memòria a true.
- 3 Retorna el valor original.

Algorisme 5 (Test-and-Set) per a múltiples fils

Aquest algorisme serveix per a múltiples fils!!!

variables globals:

```
boolean flag = false;
```

lock:

```
while Get-and-Set(&flag) {};  
return;
```

unlock:

```
flag = false;  
buidar memòria cau;  
return;
```

Algorisme 5 (Test-and-Set) per a múltiples fils

L'algorisme anterior

- Satisfà l'exclusió mútua i no provoca deadlock.
- Observeu que en sortir s'executa una instrucció màquina per buidar la memòria cau (i.e. transferir les escriptures a memòria).
- Funciona amb múltiples fils.

Quins defectes té l'algorisme?

- Tots els fils competeixen per entrar a la secció crítica.
- Hi ha alguna forma que els fils entrin a la secció crítica en l'ordre en què criden a lock?

Algorisme 6 per a múltiples fils

Hi ha alguna forma que els fils entrin en l'ordre en què arriben a la secció crítica ?

Sí, fent servir un algorisme similar al que fem servir quan anem al súper i agafem tanda per comprar formatge, per exemple.

- En arribar a la parada hem d'agafar un número.
- Hem d'estar atents (i aturats) esperant que a la pantalla surti el nostre número.

Aquest algorisme funciona si tothom es comporta de forma “ètica”. Si algú es “cola”, l'esquema se'n pot “anar en norris”.

Algorisme 6 per a múltiples fils

variables globals:

```
boolean flag[N] = {false, false, ..., false}; flag[0] = true;  
int torn_actual; /* el número que surt a pantalla */  
int torn = 0; /* el següent número al ticket */
```

lock:

```
int torn_meu;  
<torn_meu = torn; torn++;>  
torn_meu = torn_meu % N;  
while (!flag[torn_meu]) {};  
torn_actual = torn_meu;  
return;
```

unlock:

```
flag[torn_actual] = false;  
flag[(torn_actual + 1) % N] = true;  
return;
```

Algorisme 6 per a múltiples fils

A l'anterior algorisme

- Els fils entren a la secció crítica en l'ordre en què hi arriben.
- Té un defecte: s'ha de fixar N , el nombre de fils a controlar. Si volem fer un algorisme general es pot fer servir una cua de fils.

Tingueu en compte que

- Hem vist només alguns algorismes d'espera activa.
- No hi ha una distinció clara entre variable que es fan servir per realitzar càlculs i variables que es fan servir per bloquejar.
- És ineficient a la majoria de programes multifil, llevat de la programació paral·lela. A la programació paral·lela el nombre de fils és igual al nombre de processadors. En general un procés té molts més fils que processadors té l'ordinador.