Systèmes d'exploitation, 2ème année

Méthodes de synchronisation

Yves Stadler

Université Paul Verlaine - Metz

20 octobre 2011

Agenda

Plan du cours

- Pourquoi la synchronisation
- Problèmes de synchronisation
- Solutions de synchronisation

Objectifs

Pourquoi synchroniser

- Hors du cadre temps réel
- Les ressources sont limités et ne peuvent/doivent pas être utilisées en même temps ;
- Éviter l'instabilité du système/programme résultant d'un mauvais usage des ressources partagées.

Race conditions

Pourquoi synchroniser

- L'ordonnancement peut arrêter un processus pendant son exécution ;
- Processus plus prioritaire, quantum expiré, ...

Processus 1

```
x++;
y = x;
```

Processus 2

```
x++;
z = x;
```

Quel résultat

- Selon l'ordre d'exécution, les résultats seront différents.

Exécutions

P1 avant P2

```
P1: x++; // x == 1
P1: y = x; // y == 1
P2: x++; // x == 2
P2: z = x; // z == 2
```

Incrément en premier

```
P1: x++; // x == 1
P2: x++; // x == 2
P1: y = x; // y == 2
P2: z = x; // z == 2
```

P2 avant P1

```
P2: x++; // x == 1
P2: z = x; // z == 1
P1: x++; // x == 2
P1: y = x; // y == 2
```

Résumé

- Cas 1 : x == z == 2 ;
- Cas 2 : x == 2, z == 1 ;
- Cas 3 : x == 1, z == 2 ;

Autre exemple

Pourquoi synchroniser

```
next = next_available;
*next = something;
next_available++;
```


Constatations

Inconsistance

- Les exécutions sont inconsistantes
- Les résultats sont dépendants de l'ordonnanceur
- Dépendant d'un facteur qui n'est pas contrôlable ;
- On ne peut pas se fier à de tels programmes.

Mauvaise vision de ce qui se passe

- Notons qu'ici on a pris une instruction `x++`
- Cette instruction n'est pas atomique
- Une lecture et une écriture
- Encore plus de cas différents si l'ordonnanceur interrompt après lecture et avant écriture.

Section critique

Inconsistance

- On définit des zones de programmes pendant lesquelles les autres processus ne doivent pas utiliser la ressource partagée.
- Le programme en section critique à le contrôle exclusif de la ressource.

Comment mettre en œuvre

- Masquer les interruptions
- Utiliser des variables pour signaler un verrouillage
- Alternance des processus sur une ressource
- Moyens matériels
- Sémaphores.

Solutions

Exclusion mutuelle

- Masquage des interruptions
- Le problème ne sera pas coupé
- Il effectue sa section critique
- Il réactive les interruptions

Exclusion mutuelle

- Ne fonctionne que pour les architecture mono-processeur ;
- Si le programme fait une erreur, on bloque le système.

Solutions

Variable de verrouillage

- Chaque ressource est associé à un variable qui indique si elle est libre ou occupée
- Les processus consulte cette variable avant d'utiliser la ressource

Variable de verrouillage

- La race condition est déplacée sur la variable de verrouillage
- Problème de l'atomicité des opérations, le processus peut-être interrompu entre le test de la variable et l'accès à la ressource.

Solutions

Exclusion mutuelle

```
while(resource_status == BUSY)
    ;
resource_status = BUSY;
//Critical section
resource_status = FREE;
```

Exclusion mutuelle

- Réquisition entre le test `while` et la ligne suivante

Test set and Lock

Instruction TSL

- Test Set and Lock
- L'instruction garantit l'atomicité de l'opération de test et d'affectation.
- 0 : bloqué, 1 : libre.

Instruction TSL

- Perte de temps CPU
- Inversion de priorité
- Une fois qu'un processus est prêt il doit être élu.

Test set and Lock

Instruction TSL

```
entry:
  tsl reg, flag -- Opération atomique lect. flag + écrit 1
  cmp reg, #0   -- Reg == ancienne valeur de flag
  jnz entry     -- Si non zéro on recommence
  ret           -- Si 0 on a le droit de continuer

quit:
  mov flag, #0  -- On remet 0 dans le flag
  ret
```


Test set and Lock

Autres méthodes

- Méthode de l'alternance
- Algorithme de Dekker
- Algorithme de Peterson

Alternance

Alternance version 1

```
while(turn != t)
    ;
//Critical section
turn = 1 - t; //1 or 0
```

Alternance version 1

- Si un processus à plus de tâches que l'autre
- Si un processus à une plus grande priorité que l'autre
- Risque de blocage

Dekker's algorithm

Alternance version 2

```
(flag[0] = false; flag[1] = false; other = 1 -t;) //init
flag[t] = true;
while(flag[other])
    ;
//Critical section
flag[t] = false;
```

Alternance version 2

- Si un processus à plus de tâches que l'autre
- Si un processus à une plus grande priorité que l'autre
- Risque de blocage

Peterson

Peterson's algorithm

```
int other = i -t;
flag[t] = true; // Process interessted in CS
turn = other;
while((flag[other] == true) && (turn == other))
    ;
//CS
flag[t] = false;
```

Peterson's algorithm

- Si tous intéressés : chacun son tour
- Si un processus n'est pas intéressé, il ne fait participe pas au choix du processus qui va entrer en section critique ;
- L'attente est bornée

Les problèmes classiques

Lecteurs - rédacteurs

- Producteurs et consommateurs
- Des processus partagent une base
- Les lecteurs peuvent partager une même données
- Les rédacteurs doivent gérer l'accès.

Producteurs - consommateurs

Autres méthodes

```
while(count == SIZE)
    ;
++count;
buffer[in_curs] = I;
in_curs = (in_curs+1) % SIZE;
```

Autres méthodes

```
while(count == 0)
    ;
--count;
I = buffer[out_curs];
out_curs = (out_curs+1)%SIZE;
```


Producteurs - consommateurs

Autres méthodes

- Les philosophes mangent ou pensent.
- Pour manger il a besoin de deux baguettes (une à droite, une à gauche) ;
- Comment faire manger les philosophes ? (1 philosophe $=$ 1 processus)

Les philosohpes - implémentation

```
while(true) {
    think();
    get_chopstick(i); // gauche
    get_chopstick(i+1); // droite
    eat();
    release_chopstick(i);
    release_chopstick(i+1);
```


Le barbier

- N chaises, 1 barbier
- Le client s'installe s'il y a une chaise libre, sinon il repart
- Problème : faire que le barbier serve les client de manière juste.

Idée

- Actuellement nous avons de l'attente active (`while(...);`
- Primitives `sleep()` et `wakeup()`
- Quand on a rien à faire : `sleep()`
- Cela libère le processeur,
- Quand on a fini d'utiliser une ressource critique : `wakeup` permet de réveiller un processus en attente.

Producteur

```
while(true) {
    produce(item);
    if(cpt == N) sleep();
    place(item);
    cpt++;
    if(cpt == 1)
        wakeup(consumer)
}
```

Producteur

```
while(true) {
    if(cpt == 0) sleep();
    retrieve(item);
    cpt--;
    if(cpt == N-1)
        wakeup(producer)
}
```

Éviter l'attente active

Problème

- Conflit sur cpt
- On a toujours un problème si l'ordonnanceur interrompt juste après le test.
- Le wakeup est perdu ! (Processus pas encore endormi)
- On veut mémoriser le `wakeup` : sémaphore.

Le sémaphore

- Il faut que l'appel aux sémaphores soient atomiques
- Non interruptible
- Nécessite donc le mode système.

Sémaphore

Description

- Un sémaphore est une variable de contrôle d'accès
- Il indique le nombre d'éléments disponible
- Il tient à jour une liste de processus bloqués qui attendent l'accès à cette ressources
- S'il la disponibilité de la ressource ne dépasse pas 1, mutex (sémaphore binaire)
- >0 : ressources dispo ; <0 : nombre de processus bloqués.
- Fonctions P(roberen) et V(erhogen).
- La file n'est pas nécessairement FIFO.

Attention

- On initialise les sémaphores hors des processus à sections critiques.

Sémaphore

Producteurs - consommateurs

- Un mutex pour contrôler les accès critiques au buffer
- Un sémaphore pour la disponibilité en consommation (init : 0 ; `filled`)
- Un sémaphore pour la disponibilité en production (init : N ; `empty`)

Producteurs - consommateurs

- Tous les processus utilisent le mutex
- Les consommateurs P sur `filled` et V sur `empty` (consomment une case occupée et créent une case vide)
- Les producteurs P sur `empty` et V sur `filled` (consomment une case vide et créent une case occupée)

Sémaphore

Producer

```
P(empty)
P(acces)
add(item)
V(access)
V(filled)
```

Consumer

```
P(filled)
P(acces)
add(item)
V(access)
V(empty)
```

Autres méthodes

- Primitive de haut niveau
- Procédures, variables, structures de données
- Un seul processus actif dans un moniteur (sinon il est suspendu)
- Accès externes aux variables interdit.
- `wait/signal`

Autres méthodes

- `signal(i)` réveil un processus qui attend sur `i`

Autres méthodes

- On peut remplacer notre schéma producteur consommateur avec le sémaphore `access` en moins.
- Dépendant du langage : géré par le compilateur.