Architecture et Systèmes

Stefan Schwoon

Cours L3, 2016/17, ENS Cachan

Programmation concurrente

Dans la programmation concurrent on se trouve devant les problèmes suivants :

- Coordonner l'accès aux ressources partagés (exclusion mutuelle)
- Signaler qu'une ressource est prête à consummer
- Assurer qu'un processus/thread n'est pas interrompu pendant une partie critique de son code.

Exemples:

- Éviter les situations de compétition (deux processus accèdent à une donnée, au moins un accès en écriture)
- Opérations en temps réel (communication avec périphérique)
- Transactions complexes (de multiple modifications doivent être effectuées pour assurer la cohérence des données)

Premier exemple: Section critique

Modèle abstract :

On a un ensemble de processus/threads qui possède tous des sections critiques (une partie du code).

On doit assurer qu'au plus un seul processus est dans une section critique en même temps.

La solution dépend du contexte : mémoire partagé, communication asynchrone, autorité centrale, . . .

Algorithme de Peterson

Solution pour deux processus avec mémoire partagé (trois bits)

On suppose que la lecture/écriture d'un bit est atomique.

Variables:

- `flag[0]`: premier processus veut entrer dans une section critique
- `flag[1]`: deuxième processus veut entrer dans une section critique
- `victim`: pour résoudre des conflits

Algorithme de Peterson

Au départ : `flag[0] = flag[1] = 0;`

Code du processus i=0,1 (autour de la section critique) :

```
autre = 1-i;
flag[i] = 1;
victim = i;
while (victim == i && flag[autre]);
...  critical section ...
flag[i] = 0;
```

Remarque: La conjonction (`&&`) peut être non-atomique et évaluée dans n'importe quel ordre.

En supposant que les processus terminent toujours leurs section critiques, l'algorithme de Peterson est ...

correct (un seul processus peut être critique à la fois) ;

équitable (tout processus réussit finalement à entrer dans sa section critique);

libre de blocages.

Il est possible de généraliser le principe à n participants qui font $n - 1$ tours d'élimination.

Problèmes

Du code compliqué à écrire autour de chaque accès.

Il est facile de se tromper dans la programmation.

Nécessite la mémoire partagé.

Assez lourd pour plusieurs processus.

Semaphores

Un sémaphore est une structure de donnée gérée par le noyau qui offre une solution si tous les processus sont dans un même ordinateur.

Gère un compteur de *créneaux* disponibles, avec les opérations suivantes :

Init(n), où n est un nombre de *créneaux* initiaux

Wait: si compteur positif, décroître et renvoyer;
sinon on attend qu'il devient positif pour le faire

Post: augmenter le compteur

Exemple : Sémaphore pour sections critiques

```
                    Init(1);

while (1) {                    while (1) {
    ...;                           ...;
    Wait();                        Wait();
    Critical1();                   Critical2();
    Post();                        Post();
    ...;                           ...;
}                              }
```

Mettre Wait et Post autour des accès.

Sémaphores dans Unix

Supporté par le noyau, voir `sem_overview`(7):

Sémaphores anonymes (entre threads/processus pére et fils):
`sem_init`, `sem_wait`, `sem_post`

Sémaphores nommés (dans tout le système):
`sem_open`, `sem_unlink`

Implémentation d'une sémaphore

Naïvement:

```
Init(n) { ctr = n; }

Wait() { while (ctr == 0); ctr = ctr-1; }

Post() { ctr = ctr+1; }
```

Deux problèmes :

Atomicité: aucune interruption doit avoir lieu entre lecture de `ctr` et écriture de la nouvelle valeur !

Attente: active ou sommeil/reveil par noyau ?

Pour les sémaphores POSIX, l'atomicité est assuré en bloquant les interruptions (ce qui est possible dans le noyau, mais pas permis à l'utilisateur)

Attente : méthode sommeil/reveil utilisé par les opérations proposées

Les spinlock utilisent l'attente active.