Systèmes et Réseaux (ASR 2) - Notes de cours
Cours 5

Anne Benoit

February 25, 2015

1 Structure des systèmes d'exploitation

2 Gestion des processus

3 Les threads

4 Synchronisation des processus

4.1 Sections critiques et actions atomiques

4.2 Solutions matérielles et logicielles

4.2.1 Solution de Peterson

4.2.2 Synchronisation matérielle

Preuve que l'algorithme utilisant TestAndSet pour n processus avec attente bornée vérifie les 3 conditions de solution au problème de la section critique:

1. Exclusion mutuelle: P_i entre en SC uniquement si $waiting[i] := false$ ou $key := false$; key passe à $false$ seulement si TestAndSet() est exécuté, et le premier processus à exécuter TestAndSet() trouvera $key = false$, les autres attendent. $waiting[i]$ peut passer à $false$ uniquement si un autre processus quitte sa SC, et un seul $waiting[i]$ est mis à $false$, garantissant ainsi l'exclusion mutuelle.

2. Progression: mêmes arguments que précédemment: lorsqu'un processus sort de SC, il met soit $lock$ à $false$, soit un $waiting[j]$ à $false$, permettant à un autre processus de rentrer en SC.

3. Attente bornée: lors d'une sortie de SC, le tableau est parcouru circulairement dans l'ordre $(i+1, i+2, \ldots, n-1, 0, \ldots, i-1)$, et le premier processus dans cet ordre qui est prêt à rentrer en SC ($waiting[j] = true$) peut entrer. Chaque processus en attente rentrera dans sa SC après au plus $n-1$ tours.

Par contre, le TestAndSet atomique est dur à implémenter sur multiprocesseurs, et ces solutions matérielles sont difficiles à utiliser pour le programmeur ... d'où l'intérêt des sémaphores.

4.2.3 Sémaphores

Outil de synchronisation sans attente active. Sémaphore:

- variable entière S, et deux opérations atomiques:
- opération $wait(S)$ { while $S \leq 0$ { // boucle vide }; $S--$; }
- opération $signal(S)$ { $S++$; }

Deux processus ne peuvent pas modifier simultanément la valeur d'un sémaphore.

Mutex = sémaphore binaire (valeur 0 ou 1): peut être plus simple à implémenter et fournit l'exclusion mutuelle.

Sémaphore S initialisé à 1; $wait(S)$ avant d'entrer en section critique, et $signal(S)$ en sortie de section critique.

Autres utilisations des sémaphores.

- Sémaphore pour contrôler l'accès à une ressource qui a un nombre fini d'instances: initialisée au nombre de ressources; $wait(S)$ pour utiliser une ressource, et $signal(S)$ pour relâcher une ressource. $S = 0$ quand toutes les ressources sont utilisées.
- Sémaphore pour synchroniser deux processus: P_1 execute une instruction S_1, et P_2 a une instruction S_2 qui doit avoir lieu après S_1: initialiser un sémaphore $synch$ à 0, puis P_1: S_1; $signal(synch)$; et P_2: $wait(synch)$; S_2;

Implémentation des sémaphores. Doit garantir l'atomicité de $wait()$ et $signal()$: problème de la section critique, où $wait$ et $signal$ sont placés en SC. Solution avec attente active: *spinlock semaphore*, le processus attend activement le sémaphore. Avantage: pas de changement de contexte, utile si les verrous sont pris pour des durées très courtes. Mais les applications passent beaucoup de temps en SC → ce n'est pas une bonne solution en général.

Solution: modifier les opérations wait et signal, en utilisant une file d'attente des processus bloqués sur un sémaphore. Sémaphore S: deux données: $S.value$, un entier avec la valeur du sémaphore, et $S.list$, une liste de processus.

Deux opérations, implémentées avec deux appels systèmes:

- $block()$ place le processus dans la file d'attente appropriée;
- $wakeup(P)$ défile un processus de la file et le place en état ready.

On peut maintenant écrire le code de wait et signal:

```
wait (semaphore *S) {
  S.value --;
  if (S.value < 0) {
    block(); // met le processus dans S.list
  }
}

signal (semaphore *S) {
  S.value ++;
  if (S.value <= 0) {
    wakeup(P); // enlève un processus P de la liste S.list
  }
}
```

Problèmes possibles avec les sémaphores:

- Utilisation non correcte: signal(mutex) suivi de wait(mutex) (pas d'exclusion mutuelle!); ou wait(mutex) suivi de wait(mutex) (blocage); oubli d'un wait ou d'un signal...
- Interblocages: deux processus ou plus attendent indéfiniment un événement qui peut être causé uniquement par l'un des processus bloqué. Exemple avec deux sémaphores S et Q initialisés à 1:
  P_1 effectue wait(S); wait(Q); ...; signal(S); signal(Q);
  P_2 effectue wait(Q); wait(S); ...; signal(Q); signal(S);
- Famine: un processus peut rester dans la file d'attente de son sémaphore et ne jamais en sortir.

4.3 Problèmes classiques de synchronisation

4.3.1 Producteurs/consommateurs

C'est le problème qu'on a déjà rencontré, avec un buffer de messages de taille fixée N, géré circulairement. Solution avec des sémaphores?

- Sémaphore mutex initialisé à 1 (entrées en SC);
- Sémaphore full initialisé à 0 (nombre de données);
- Sémaphore empty initialisé à N (nombre de cases vides).

Codes symétriques pour le producteur et le consommateur: le producteur produit un item puis wait(empty); wait(mutex); ajoute son item au buffer; signal(mutex); signal(full); et recommence.
Le consommateur effectue wait(full); wait(mutex); retire un item du buffer; signal(mutex); signal(empty); et recommence.

4.3.2 Lecteurs/rédacteurs

Autre problème classique, avec des lecteurs qui ne font que lire une donnée (partagée), et des rédacteurs qui peuvent lire et écrire. On peut avoir plusieurs lectures simultanées, mais un seul rédacteur à la fois. Solution avec des sémaphores?

- Sémaphore mutex initialisé à 1 (entrées en SC);
- Sémaphore wrt initialisé à 1 (nombre de rédacteurs);
- Entier nbLect initialisé à 0 (nombre de lecteurs).

Le rédacteur attend simplement wrt avant de faire son écriture: while(true) { wait(wrt); écrit dans le fichier; signal(wrt)}.

Code du lecteur:

```
while(true) {
  wait(mutex); nbLect ++;
  if (nbLect == 1) wait(wrt);
  signal(mutex);
  // Lecture
  wait(mutex); nbLect --;
  if (nbLect == 0) signal(wrt);
  signal(mutex);
```

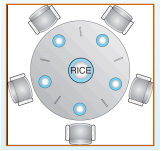
Le premier lecteur se bloque sur wrt si un rédacteur est en SC, et les suivants se bloquent sur mutex. Lors d'un signal(wrt), on peut reprendre l'exécution d'un rédacteur en attente, ou des lecteurs. Le dernier lecteur à sortir de sa lecture fait aussi signal(wrt).

Risque de *famine*: le rédacteur doit attendre que tous les lecteurs aient fini de lire avant de prendre la main, mais le nombre de lecteurs n'est pas borné... Les lecteurs peuvent monopoliser les ressources au détriment des rédacteurs.

Solution plus juste: rajouter un sémaphore rw (read-write) initialisé à 1. Objectif: si un rédacteur est en attente, ne pas autoriser de nouveaux lecteurs à accéder le fichier. Ainsi, le rédacteur commence par wait(rw) et termine par signal(rw). Code du lecteur?

4.3.3 Le dîner des philosophes

Des philosophes sont installés autour d'une table ronde, et un philosophe a besoin de 2 baguettes pour manger. Le philosophe alterne le mode "penser" et le mode "manger".



Solution avec des sémaphores? Tableau de sémaphore baguette[N] (pour N philosophes), initialisés à 1. Le code du philosophe i est le suivant:

```
while(true) {
  wait(baguette[i] );
  wait(baguette[(i+1) mod N]);
  // MANGE
  signal(baguette[i] );
  signal(baguette[(i+1) mod N]);
  // PENSE
```

Il se peut que chaque philosophe tienne une baguette, et on se retrouve en interblocage...

Solutions:

- Ne pas avoir plus de 4 philosophes à table;
- Permettre à un philosophe de manger uniquement si les 2 baguettes sont disponibles;
- Solution asymétrique: les philosophes de numéro impair prennent d'abord la baguette $i + 1$.

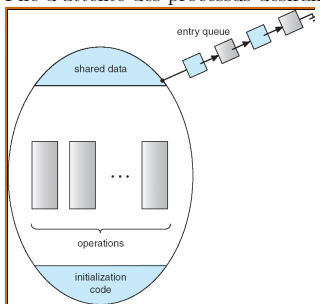
Toute solution satisfaisante doit garantir qu'il n'y ait pas famine, i.e., un des philosophes ne va pas mourir de faim. L'absence d'interblocages ne garantit pas l'absence de famine.

4.4 Les moniteurs

Autre outil de synchronisation: abstraction haut niveau. Dans un moniteur, un seul processus peut être actif à la fois. Procédures définies dans le moniteur.

```
monitor monitor-name {
  // déclaration de variables partagées
  procedure P1 (...) {...}
  ...
  initialisation (...) {...}
}
```

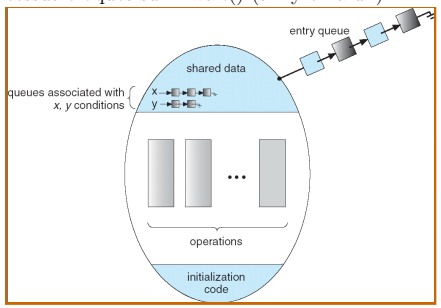
File d'attente des processus désirant entrer dans le moniteur:



En plus, possibilité d'utiliser des variables de condition:

```
condition x,y;
```

Deux opérations: x.wait() suspend le processus, et x.signal() reprend l'exécution d'un des processus bloqués sur x.wait() (s'il y en a un).



Et voici le code pour le dîner des philosophes (le philosophe i appelle dp.pickup(i), il mange, puis il appelle dp.putdown(i).

```
monitor DP
{
   enum {THINKING, HUNGRY,
         EATING} state [5] ;
   condition self [5];

   void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait;
   }

    void putdown (int i) {
        state[i] = THINKING;
            // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

   void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING ;
                self[i].signal () ;
            }
    }

   initialization_code() {
        for (int i = 0; i < 5; i++)
        state[i] = THINKING;
        }

   }
```

Implémentation des moniteurs. On peut facilement implémenter les moniteurs à l'aide de sémaphores: un sémaphore mutex initialisé à 1, et un sémaphore next initialisé à 0. On utilise aussi une variable partagée int next-count = 0.

Les procédures sont remplacées par:

```
wait (mutex);
  // code de la procédure
if (next-count > 0) signal(next);
else signal(mutex);
```

L'exclusion mutuelle est garantie, et next-count indique le nombre de processus qui veulent rentrer dans le moniteur suite à un x.signal.

Pour chaque variable de condition x, on a un sémaphore x-sem initialisé à 0, et int x-count = 0 (nombre de processus bloqués sur x). On a donc:

```
x.wait:
     x-count ++;
     if (next-count > 0) signal (next);
     else signal (mutex);
     wait(x-sem);
     x-count --;

x.signal:
     if (x-count >0) {
        next-count ++;
        signal(x-sem);
        wait(next);
        next-count --;
     }
```

4.5 Conclusion

- Processus séquentiels coopérants partageant des données:
  - Nécessité de fournir un mécanisme d'exclusion mutuelle;
  - Sections critiques: un seul processus/thread à la fois;
  - Plusieurs algorithmes pour résoudre le problème;
- Problème de l'attente active: utiliser les sémaphores. Permet de résoudre les problèmes classiques, utilisés pour tester presque tous les nouveaux schémas de synchronisation;
- Erreurs faciles: solutions de haut-niveau comme les moniteurs et leurs variables de condition;
- Chaque OS fournit un support pour la synchronisation. Par exemple, Linux fournit des sémaphores et spinlocks, et les interruptions sont désactivées pour de petites sections critiques. Pthreads fournit des verrous mutex, des variables de condition et des verrous lecture/écriture, et une extension (POSIX SEM) contient les sémaphores.

Pour tester ses connaissances

1. Proposer une implémentation de l'exclusion mutuelle avec temps d'attente borné à l'aide de l'instruction Swap().
2. Montrer comment implémenter un sémaphore à l'aide d'un moniteur.
3. Quelle est la différence entre le signal() d'un sémaphore et celui d'un moniteur?