TD 3 : Sémaphores

Université Paris 7 — LMI&EIDD — Systèmes et Réseaux

3 mars 2014

Dans ce TD on utilise des sémaphores (beaucoup). Avec un sémaphore `sem` on peut :
- Initialiser `sem` avec une certaine valeur, avant tout autre usage. Valeur par défaut : 1.
- `sem.P( )` attend que le sémaphore ait une valeur positive, puis décrémente le sémaphore
- `sem.V( )` incrémente le sémaphore

Ces trois services sont offerts par le système et donc supposés fiables.

Exercice 1 – Les philosophes Chinois

Ce grand classique a été proposé par Dijkstra [1971], l'inventeur des sémaphores pour *THE operating system.*

Cinq philosophes chinois [1] participent à un dîner. Ils sont assis à une table ronde. Chaque philosophe passe successivement dans les états :
- penser, puis après un temps idéfini :
- être affamé. Il prend alors deux baguettes. Dès qu'il a ses deux baguettes :
- manger. Après quoi il est rassasié, il laisse tomber ses baguettes et recommence à penser, etc

Le problème est qu'il n'y a que cinq baguettes : une entre chaque convive.

Que penser des résolutions suivantes du problème? Le code donné est celui du philosophe i, $i \in \{1..5\}$.

```
*** solution 1 ****
repeter:
   penser( )
   baguette[i].P( )
   baguette[i+1 mod 5].P( )
   manger( )
   baguette[i].V( )
   baguette[i+1 mod 5].V( )
fin
```

```
*** solution 2 ***
baguettes est un semaphore initialisé à 5
repeter:
   penser( )
   baguettes.P(2)
   baguette[i].P( )
   baguette[i+1 mod 5].P( )
   baguette[i].V( )
   baguette[i+1 mod 5].V( )
   baguettes.V(2)
   manger( )
fin
```

```
*** solution 3 ***
baguette : tableau de 5 booléens
repeter:
   penser( )
   affaméSansBaguette = 1
   tant que affaméSansBaguette
       mutex.P( )
       si baguette[i]==0 et baguette[i+1]==0
          baguette[i] = 1
          baguette[i+1] = 1
          affaméSansBaguette = 0
       fin si
       mutex.V( )
   fin tant que
   manger( )
   baguette[i] = 0
   baguette[i+1] = 0
fin
```

```
*** solution 4 ***
mangeurs est un semaphore initialisé à 2
repeter:
   penser( )
   mangeurs.P( )
   baguette[i].P( )
   baguette[i+1 mod 5].P( )
   manger( )
   baguette[i].V( )
   baguette[i+1 mod 5].V( )
   mangeurs.V( )
fin
```

1. le détail a son importance

Exercice 2 – Sémaphores généralisés [examen 2012]

Résoudre le problème des philosophes chinois grâce à l'opération `P()` généralisée, qui est la suivante. Elle prend en argument un ou plusieurs sémaphores suivant la syntaxe `P(sem1,sem2,sem3)`. Puis :

- Si tous les sémaphores ont une valeur $\geqslant 1$ ils sont tous décrémentés en même temps
- Sinon le système attend que tous les sémaphores aient une valeur $\geqslant 1$ pour effectuer l'opération

NB : c'est faisable grâce à l'appel système `semop()`.

Exercice 3 – Imprimantes

Il y a une imprimante sur le réseau. Elle est accessible directement. Le papier est à déroulement continu ; on peut imprimer une ligne grâce à la procédure `ImprimeLigne( )` et passer à la page suivante grâce à `NouvellePage( )`.

1. Écrire un programme qui imprime un fichier donné.
2. Il y a maintenant *quatre* imprimantes nommées *imp1*, *imp2*, *imp3* et *imp4*. Même question (il faut trouver et réserver une imprimante !) Identifiez bien les *sections critiques*.
3. On désire toujours imprimer, mais depuis une disquette. Il y a un seul lecteur dans le système. Le système offre une méthode `MonterDisquette` qui ne doit être appelé que par un processus à la fois (sinon, l'appelant se fait tuer). Après, on a des méthodes usuelles d'ouverture/fermeture et de lecture d'une ligne dans le fichier ou à la console.

   Écrire un processus d'impression de fichier depuis la disquette. Bien entendu, l'utilisateur ne doit insérer sa disquette que si son processus à obtenu le lecteur, et non le processus du voisin !
4. Décrivez comment des interlocages peuvent arriver, si plusieurs processus utilisent simultanément imprimantes et lecteur de disquette. Comment garantir qu'il n'y aura pas interblocage ?

Exercice 5 – producteur/consommateur version classique

Une configuration très courante en informatique est quand deux processus communiquent : l'un **produit** des données en flux continu, et l'autre les **consomme**. Ils sont reliés par un tube (*pipe*).

Cela arrive explicitement par le shell (comme `sort truc | uniq`) ou implicitement (comme un processus écrivant dans un socket réseau).

Voilà une tentative d'implémentation par les processus utilisateur d'un tube. Bien sûr il est plus simple et plus efficace d'utiliser les tubes du système !

L'implémentation proposée est constituée de

- Une zone de mémoire partagée M (*buffer*) de T octets
- Un pointeur d'écriture (pointant dans cette zone)
- Un pointeur de lecture (pointant dans cette zone)

Le processus producteur, quand il a un octet à écrire, l'écrit à l'endroit pointé par le pointeur d'écriture, puis l'incrémente (modulo T). Symmétriquement, quand le processus consommateur peut lire un octet, il le lit à l'endroit pointé par le pointeur de lecture puis incrémente ce pointeur

```
processus producteur

repeter :

    calculer l'octet a ecrire

    ecrire l'octet en M[E]

    E = E +1 mod T

fin
```

```
processus consommateur

repeter :

    lire l'octet en M[L]

    L = L+1 mod T

    traiter l'octet lu

fin
```

1. Dans quel cas la lecture doit-elle etre différée ? En attente de quoi ?
2. Dans quel cas l'écriture doit-elle etre différée ? En attente de quoi ?
3. Indentifier les sections critiques dans les processus ci-dessus
4. Résoudre le problème en rajoutant quelques sémaphores

Exercice 6 – producteur/consommateur version multiple

Maintenant il y a une quantité indéfinie d'écrivains (mais toujours un seul processus lecteur). Les écrivains écrivent des *blocs* de données qui ne doivent pas être intercalés.

1. Donner des exemples où le cas arrive
2. Quelle(s) nouvelle(s) section(s) critique(s) apparaî(ssen)t ?
3. Résoudre le problème grâce à des sémaphores.

Maintenant supposons qu'il y a plusieurs lecteurs aussi. Deux cas se présentent :

1. N'importe quel lecteur peut récupérer n'importe quel bloc
2. Un bloc est adressé à un lecteur donné (identifié par un numéro)

Résoudre les deux problèmes. N'oubliez pas que quand plusieurs processus attendent un sémaphore, on ne peut pas choisir lequel sera réveillé ! Quelle hypothèse doit-on rajouter pour le deuxième cas ?

Exercice 7 – Sémaphores hygiéniques (exam 2007)

Les douches des garçons sont en rénovation à la cité universitaire. Les douches des filles serviront donc pour les deux sexes. La salle de douches comporte quatre cabines de douches. Le CROUS impose les conditions suivantes :

- Jamais un garçon et une fille ensemble dans les douches
- Pas plus de quatre personnes (du même sexe, donc) dans les douches
- L'attente se fait à la porte

Question 1

En modélisant garçons et filles commes des processus indépendants, et à l'aide de sémaphores, résolvez ce problème d'exclusion mutuelle (de la façon la plus simple possible !) en utilisant le canevas page suivante.

```
Processus garçon i          Processus fille j
{                           {
   // attendre                  // attendre


  se_doucher( );                se_doucher( );

  // sortir                     // sortir

}                           }
```

Question 2

Faisons le scénario suivant de déroulement d'une matinée :

- Toutes les 2 minutes, un étudiant arrive pour se doucher, alternativement une fille et un garçon (en commançant par une fille)
- Chaque douche dure 5 minutes [2]

Que se passe-t-il ?

2. Le modèle utilise l'hypothèse simplificatrice que les temps de douche sont les mêmes pour les deux sexes

Exercice 8 : sémaphores de rendez-vous [examen 2011]

Le problème du **rendez-vous** est le suivant. Il y a trois phase : inscription, rendez-vous, déblocage.

- En **phase d'inscription**, un processus exécute la fonction `int inscription(pid_t process_pid)` pour signaler qu'il souhaite participer. Cette phase commence au premier processus qui exécute `inscription()` et se termine quand la phase suivante commence.
- La **phase de rendez-vous** est déclenchée lorsqu'un premier processus (préalablement inscrit) exécute `void rendez_vous(pid_t process_pid)`. Ce processus est alors bloqué. Ensuite, chaque processus inscrit qui appelle `rendez_vous()` se bloque, sauf le dernier :
- Le **déblocage** a lieu quand le dernier processus inscrit exécute `rendez_vous()`. Tous les processus sont alors débloqués (la fonction `rendez_vous()` retourne chez tous).

On suppose que tout processus qui souhaite participer fait exactement ce code :

```
// attendre un certain temps
int ok = inscription(getpid()); // NB : getpid()  renvoie un identifiant unique
// attendre encore un certain temps
if(ok)
{
  printf("Je vais au rendez-vous...\n");
  rendez_vous(getpid()); // blocage éventuel
  printf("Le rendez-vous a lieu!!\n");
}
else
  printf("Je suis exclus du rendez-vous : inscription trop tardive...\n");
```

Vous avez à programmer les fonctions `inscription()` et `rendez_vous()` en pseudo-code. Bien entendu il faudra utiliser des sémaphores ! Notez que `inscription()` renvoie 1 si on est en phase d'inscription (elle inscrit alors le processus), et 0 sinon (ne fait rien). `rendez_vous()` ne fait rien d'autre que bloquer le processus en attendant que tous soient au rendez-vous (par définition cette fonction n'est appelée qu'en phase de rendez-vous).