Module Système L3/S6 – TD-TP no 2
Exclusion mutuelle – Sémaphores

Vous pouvez récupérer du code ici :
`http ://www.lisyc.univ-brest.fr/pages_perso/rodin/FTP/Enseignements/L3/Systeme/`
Le fichier à récupérer est : `TD-TP2.tar.gz`
Pour décompresser l'archive faire : `tar zxvf TD-TP2.tar.gz` ou `gunzip TD-TP2.tar.gz` et `tar xvf TD-TP2.tar`

1 Généralités sur les sémaphores

1.1 Rappels sur les sémaphores

Un sémaphore est un mécanisme empêchant deux processus ou plus d'accéder simultanément à une ressource partagée.

Sur les voies ferrées, un sémaphore empêche deux trains d'entrer en collision sur un tronçon de voie commun. Sur les voies ferrées comme dans les ordinateurs, les sémaphores ne sont qu'indicatifs :
- Si un machiniste ne voit pas le signal ou ne s'y conforme pas, le sémaphore ne pourra éviter la collision.
- De même si un processus ne teste pas un sémaphore avant d'accéder à une ressource partagée, le chaos peut en résulter.

Un sémaphore binaire n'a que deux états :
- 0 verrouillé (ou occupé),
- 1 déverrouillé (ou libre).

Un sémaphore général peut avoir un très grand nombre d'états car il s'agit d'un compteur dont la valeur initiale peut être assimilée au nombre de ressources disponibles.

Par exemple, si le sémaphore compte le nombre d'emplacements libres dans un tampon et qu'il y en ait initialement 5 on doit créer un sémaphore général dont la valeur initiale sera 5.

Ce compteur :
- Décroît d'une unité quand il est acquis ("verrouillé").
- Croît d'une unité quand il est libéré ("déverrouillé").

Quand il vaut zéro, un processus tentant de l'acquérir doit attendre qu'un autre processus ait augmenté sa valeur car il ne peut jamais devenir négatif.

L'accès à un sémaphore se fait généralement par deux opérations :
P : pour l'acquisition (Proberen, tester).
V : pour la libération (Verhogen, incrémenter).

Un moyen mnémotechnique pour mémoriser le P et le V est le suivant :
P(uis-je) accéder à une ressource
V(as-y) la ressource est disponible.

1.2 Un mécanisme fourni par le noyau

La notion de sémaphore est implémentée dans la plupart des systèmes d'exploitation. Il s'agit d'un concept fondamental car il permet une solution à la plupart des problèmes d'exclusion.

Ce concept nécessite la mise en œuvre d'une variable (le sémaphore) et de deux opérations atomiques associées P et V.

Si nous devions écrire ces deux primitives nous aurions le résultat suivant :

Sémaphore S : Valeur $\geq$ 0 + File d'attente

Opérations :

- **P(S)** : Si Valeur = 0
  Alors Placer le processus en attente
  Sinon Valeur $\longleftarrow$ Valeur - 1
- **V(S)** : Si File d'attente $\neq \emptyset$
  Alors Débloquer **un** processus en attente
  Sinon Valeur $\longleftarrow$ Valeur + 1

2 Exercices sur les sémaphores : les rendez-vous

2.1 Rendez-vous à 2

Deux processus (P1 et P2) souhaitent établir un rendez-vous avant l'exécution de la fonction RDV1() pour l'un et RDV2() pour l'autre.

En utilisant les sémaphores, écrire la séquence de pseudo-code de P1 et P2 permettant d'établir ce rendez-vous.

Décrire le comportement des deux processus à partir d'un diagramme temporel.

2.2 Rendez-vous à 3

Effectuer un rendez-vous entre 3 processus P1, P2 et P3

2.3 Rendez-vous à N

Généraliser le rendez-vous précédent entre N processus avec un ensemble de sémaphores initialisés à 0 : sema [i] désigne le sémaphore i.

Ecrire le code du processus Pi permettant d'établir ce rendez-vous.

3 Exercice classique : les lecteurs/écrivains

Le but est ici de partager une ressource (ici un fichier `livre`) entre plusieurs processus, à l'aide de sémaphores et d'une mémoire partagée.

Deux types de processus peuvent exister : les processus lecteurs qui ne font que consulter le fichier `livre`, et les processus écrivains qui, eux, peuvent modifier le fichier `livre`.

On souhaite obtenir un maximum de parallélisme tout en conservant une information cohérente :
- Il peut y avoir plusieurs lecteurs en //
- Il ne peut y avoir plusieurs écrivains en //
- Il ne peut y avoir un lecteur et un écrivain en //

Soit le début de solution suivant :
- Un écrivain demande toujours l'autorisation d'utiliser la donnée `livre`.
- Un lecteur ne demande l'autorisation d'utiliser la ressource que s'il est le 1*er* lecteur.
- Le Sémaphore `MUTEX` protégeant l'accès au fichier `livre` est initialisé à 1.

Début du code des processus écrivains et des processus lecteurs :

```
Ecrivain()                          |      Lecteur()
{                                   |      {
 P(MUTEX)                           |       if pas-de-lecteur-en-cours {
                                    |        P(MUTEX)
 écrire dans le fichier livre       |       }
                                    |
 V(MUTEX)                           |       lire le fichier livre
}                                   |
                                    |       if plus-de-lecteur-en-cours {
                                    |        V(MUTEX)
                                    |       }
                                    |      }
```

3.1 Exercice : lecteurs/écrivains V1

A l'aide d'une variable `nbLecteurs` stockée dans un segment de mémoire partagée, écrire le code du processus lecteur.

Prévoir un sémaphore `MEM` (initialisé à 1) permettant de protéger l'accès à cette variable `nbLecteurs`.

3.2 Exercice : lecteurs/écrivains V2

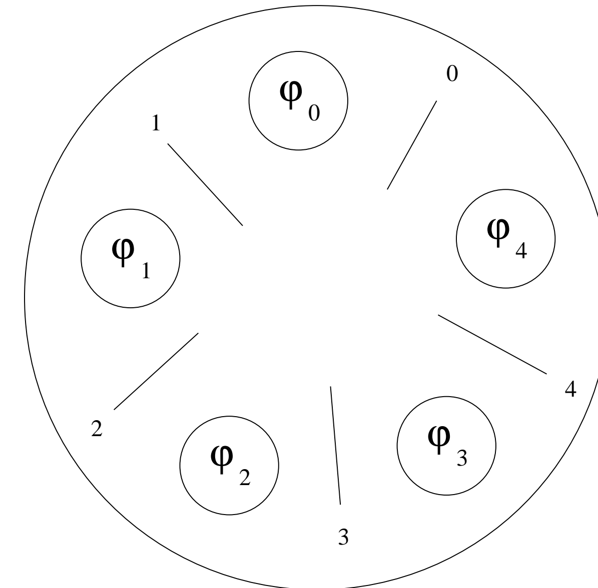
Dans l'exercice précédent, il n'y a pas de notion de priorité entre les lecteurs et les écrivains. On souhaite changer cela.

Répondez aux questions suivantes :
- Quel est le risque pour les écrivains dans la situation actuelle ?
- Comment garantir qu'un écrivain est prioritaire sur les lecteurs (c'est-à-dire que si un écrivain est bloqué, plus aucun lecteur ne peut accéder à la ressource `livre`) ?

4 Exercice classique : le problème des philosophes

4.1 Présentation

Il s'agit d'un problème de synchronisation très classique :
- Cinq philosophes sont assis autour d'une table.
- Chaque philosophe a devant lui une assiette de riz qu'il doit manger à l'aide de 2 baguettes.
- Une baguette sépare chaque assiette.



Un philosophe passe son temps à manger et à penser :
- Lorsqu'un philosophe a faim, il tente de s'emparer des 2 baguettes qui sont de part et d'autre de son assiette, l'une après l'autre (l'ordre n'importe pas).
- S'il obtient les deux baguettes, il mange pendant un certain temps, puis repose les 2 baguettes et se remet à penser.

Le problème qui se pose est d'écrire un programme qui permette à chaque philosophe de se livrer à ses activités (manger + penser) sans jamais être bloqué.

4.2 Exercice no 1 – Philosophes

La solution proposée ici est la plus évidente. Elle repose sur un ensemble de `N = 5` sémaphores correspondant aux 5 baguettes disponibles : `Sema baguette[N]={1,...,1}`

Soient les deux procédures suivantes :

- `P(i)` : attend avec P(`baguette[i]`) que la baguette `i` soit disponible **ET** la prend.
- `V(i)` : repose la baguette `i` **ET** la rend disponible avec V(`baguette[i]`).

Voici donc la solution proposée :

```
#define N 5                  // Nombre de philosophes

void philosophe(int i)
{
 while (1) {
  penser();                  // Le philosophe pense
  P(i);                      // Il prend sa baguette gauche
  P((i+1)%N);                // Il prend sa baguette droite
  manger();                  // Il peut enfin manger avec ses 2 baguettes
  V((i+1)%N);                // Il repose sa baguette droite
  V(i);                      // Il repose sa baguette gauche
 }
}
```

Que pensez-vous de cette solution ?

4.3 Exercice no 2 – Philosophes

Faire évoluer l'algorithme précédent de manière à ce qu'un philosophe après avoir pris la baguette gauche effectue d'abord un test sur la disponibilité de la baguette droite.

Si elle ne l'est pas, le philosophe repose sa baguette droite, attend un certain temps et recommence plus tard.

Il est possible d'utiliser la fonction `Ptest` qui permet d'effectuer une opération `P` sur un sémaphore uniquement si celui-ci est non bloquant.

`Ptest` retourne :

`0` : opération `P` réussie,
`-1` : opération `P` échouée, le sémaphore étant occupé.

Que pensez-vous de cette solution ?

4.4 Exercice no 3 – Philosophes

Etudions la solution suivante :

- ◇ Un tableau `etat_philosophe` mémorise l'activité courante de chaque philosophe. Un philosophe peut, au cours du temps, être dans les états suivants :
  - `PENSE` ... il peut être en train de penser
  - `FAIM` ... il peut avoir faim
  - `MANGE` ... il peut être en train de manger
- ◇ Il peut se mettre à manger si aucun de ses deux voisins ne mange. Les voisins du philosophe `i` sont définis par les macros :
  - `GAUCHE (i-1) modulo N`
  - `DROITE (i+1) modulo N`
- ◇ A chaque philosophe `i` est associé un sémaphore `sema_philosophe[i]`. Ces `N` sémaphores permettent de bloquer les philosophes qui ont faim si les baguettes dont ils ont besoin sont déjà utilisées.
- ◇ Le sémaphore `Mutex` permet de protéger l'accès à la ressource `etat_philosophe`

Voici donc la nouvelle solution proposée :

```
#define N 5                 // Nombre de philosophes
#define GAUCHE (i-1)%N      // Numéro du voisin de gauche
#define DROITE (i+1)%N      // Numéro du voisin de droite
#define PENSE 0             // Le philosophe pense
#define FAIM  1             // Le philosophe a faim. Il veut les baguettes
#define MANGE 2             // Le philosophe mange

int etat_philosophe[N];     // Mémorise l'état des philosophes
Sema Mutex=1;               // Pour exclusion mutuelle sur etat_philosophe
Sema sema_philosophe[N]={0,...,0}; // Un sémaphore par philosophe, init:0

void philosophe(int i)      // Code du philosophe numéro i
{
 while (1) {
  penser();                 // Le philosophe pense
  prendre_baguettes(i);     // Le philosophe prend 2 baguettes ou bloque
  manger();                 // Le philosophe mange
  poser_baguettes(i);       // Le philosophe pose les 2 baguettes
 }
}
```

Ecrire le code des 2 fonctions :

- `prendre_baguettes(int i);`
- `poser_baguettes(int i);`

Pour cela, vous pouvez écrire (et utiliser) une fonction `test_baguettes(int i);` qui teste si le philosophe numéro `i` peut manger (i.e, ses 2 voisins ne mangent pas) et qui, s'il peut manger, l'autorise à manger.

5 Exercices de TP

5.1 Le problème des lecteurs/écrivains

En ayant comme point de départ des exemples du cours sur les sémaphores IPC System V, implémenter la solution du problème des lecteurs/écrivains du TD.

A regarder plus particulièrement dans les exemples du cours :
- Utilisation de `ftok`, `semget`, `semctl`, `shmget`, `shmctl`, `shmat`, `shmdt`, ...
- Les fichiers `semOp.h` et `semOp.c` avec les 2 fonctions `P` et `V`

En C, les processus écrivains pourront modifier le fichier `livre` à l'aide de l'appel

```
system("gedit livre");
```

En C, les processus lecteurs pourront lire le fichier `livre` à l'aide de l'appel

```
system("less livre");
```

La commande shell `less` permet d'afficher un fichier page par page. Pour voir la page suivante appuyer sur l'`espace`. Pour quitter la commande appuyer sur `q`.

5.2 La fonction `Ptest`

Dans l'exercice de TD sur les philosophes, nous avons utilisé une fonction `Ptest` ayant le comportement suivant :

La fonction `Ptest` qui permet d'effectuer une opération `P` sur un sémaphore uniquement si celui-ci est non bloquant.

`Ptest` retourne :
`0` : opération `P` réussie,
`-1` : opération `P` échouée, le sémaphore étant occupé.

Ajouter cette fonction `Ptest` dans les fichiers `opSem.h` et `opSem.c` ...
... utilisation de `IPC_NOWAIT`

Tester cette fonction avec une simple exclusion mutuelle :

```
....
if (Ptest(sem_id,MUTEX)==0)
{
 system("less livre");
 V(sem_id,MUTEX);
}
....
```

5.3 Les rendez-vous à 2 et à 3

Programmer les rendez-vous à 2 et à 3 vus en TD.

6 Annexe : utilisation de l'outil "maison" `configure`

L'outil "maison" `configure`, écrit en Shell, permet de générer quasi automatiquement un fichier `makefile`.

Au début du fichier de commandes `configure` doivent se trouver un certain nombre de variables Shell :
- La variable `TARGET` permettant de préciser toutes les cibles (fichiers exécutables) que l'on souhaite obtenir.
- Pour chaque cible **`c`** précisée dans la variable `TARGET`, une variable `FILES`**`c`** doit être présente dans le fichier `configure`. Cette variable `FILES`**`c`** doit décrire la liste des fichiers C nécessaires pour pouvoir "construire" l'exécutable **`c`**.

Exemple de début de fichier `configure` :

```
#!/bin/sh

#----------------- Fichiers et librairies -----------

TARGET='creIPC desIPC lecteur ecrivain'

FILEScreIPC='creIPC.c'
FILESdesIPC='desIPC.c'
FILESlecteur='opSem.c lecteur.c'
FILESecrivain='opSem.c ecrivain.c'

#----------------- Options de compilation -----------

INCDIR='.'

#----------------- Options d'edition des liens -----------

LDFLAGS=' '

#----------------- Et d'autres choses en Shell -----------
.....
```

Remarque :
- Il n'est pas nécessaire d'exécuter à chaque fois la commande `configure` avant de faire `make`.
- L'exécution de la commande `configure` n'est nécessaire que lorsque l'on veut obtenir un nouveau fichier `makefile` à partir de nouvelles variables Shell contenues dans le fichier `configure`.
- Afin d'indiquer des librairies à utiliser lors de l'édition des lien, la variable `LDFLAGS` peut être précisée.
  On peut, par exemple, avoir : `LDFLAGS='-lm'`.
  Si deux (ou plus) librairies sont utiles, on peut les indiquer facilement.
  On peut, par exemple, avoir : `LDFLAGS='-lm -lpthread'`.