Systèmes d'exploitation –Recueil d'exercices

TD-Synchronisation et Communication des Processus

Partie I : Sémaphores

Exercice I.1. Exclusion mutuelle (DS 11/2010)

Soient trois processus concurrents P1, P2 et P3 qui partagent les variables n et out. Pour contrôler les accès aux variables partagées, un programmeur propose les codes suivants :

Int out=0, n=0 ;

Semaphore mutex1=1, mutex2=1;

Code processus P1

```
P(mutex1);
  P(mutex2);
    out=out+1;
    n=n-1;
  V(mutex2);
V(mutex1);
```

Code du processus P2

```
P(mutex2);
   out=out-1;
V(mutex2);
```

Code du processus P3

```
P(mutex1);
   n=n+1;
V(mutex1);
```

1) Cette proposition est-elle correcte ? Sinon indiquez au moins une condition de section critique qui n'est pas satisfaite?

2) Proposer une solution correcte.

Exercice I.2. Partage d'imprimantes (DS 11/2010)

Un ensemble de N (N > 4) processus partagent 3 imprimantes LP0, LP1, LP2. Pour éviter de mélanger les lignes sorties de chaque processus, le processus Pi doit réserver l'imprimante avant de l'utiliser. Pour connaître l'état de chaque imprimante il y a une variable globale *int LP[3]*. Les codes des deux fonctions : *prendre()* et *liberer()*sont comme suit:

```
int prendre()
 { int i=0;
   while(i<3)
   { if(LP[i]==0){
       LP[i]=getpid();
       return i;
     }
     ++i;
   }
   return -1;
 }
```

```
int liberer(int i)
   {
     LP[i]=0;
   }
```

En utilisant 2 sémaphores complétez prendre() et liberer() pour synchroniser entre les N processus.

Exercice I.3. Producteur/consommateur –Diffusion atomique (examen 06/2010)

Le modèle de producteur/consommateur est un exemple classique de synchronisation de 2 processus, l'un produit des informations qu'il dépose dans un ensemble borné de cases, l'autre qui les retire une à une pour les consommer.

Si l'on considère chaque case comme une ressource, le but est de synchroniser les 2 processus, de sorte à ne pas écraser de l'information dans une case, ou ne pas retirer d'une case de l'information inexistante. De plus, on souhaite que les informations produites soient consommées dans le même ordre. Pour se faire, l'ensemble de cases est géré de façon circulaire.

Il y a plusieurs manières pour implanter les contraintes de synchronisation.

Etendre la solution classique initiale, donnée en cours (voir annexe), avec des sémaphores de sorte à permettre d'avoir un nombre fixe, soit n, de consommateurs (et toujours un seul producteur). En d'autres termes, on veut qu'une information soit consommée par les n consommateurs. Cette situation se rencontre dans un système quand on veut, par exemple, avertir tous les n utilisateurs à la fois.

Une information occupe donc une case tant que tous les consommateurs ne l'ont pas lue. Autrement dit, la consommation d'une information n'entraine plus systématiquement la libération de la case qu'elle occupe. Par ailleurs, un consommateur peut continuer à consommer de l'information issue d'autres cases, même si les autres consommateurs sont en retard par rapport à lui.

Exercice I.4. Synchronisation avec des sémaphores (DS 11/2007)

On dispose de 3 processus P1, P2 et P3 qui sont lancés au même instant. Le but est de contrôler l'ordonnancement des actions des processus P1, P2 et P3. Pour cela on dispose d'un langage de spécification comportant les opérations de concaténation séquentielle '.', de répétitions '*', de choix exclusif '+' et de mise en parallèle '//'.

Les processus P1, P2 et P3 exécutent le code suivant :

```
Processus Px ()  /* les codes sont différents pour P1, P2, et P3 */

{  while (true) {prologuex ;      Ax ;      epiloguex ;}

/* les actions Ax ne sont pas forcément atomique */
```

Donnez pour la spécification suivante :

((A1.A2) + (A2.A1)) soit par exemple : A1.A2 . A1.A2 . A2.A1 etc.*

les sémaphores correspondants (à initialiser) et les codes des *prologuex*, *epiloguex* pour chaque processus.

Exercice I.5. (DS 11/2007)

Dans un lavomatique, on cherche une solution pour permettre de répartir les machines à laver équitablement entre les clients. Considérez le programme suivant. Pour obtenir une machine, chaque client doit utiliser la fonction *allouer()*. Après usage de la machine, il doit utiliser *liberer().*

Conditions initiales

```
#define NMACHINES 5

Semaphore nlibre = 5 ;
int dispo[NMACHINES] =(1,1,1,1,1) ;
```

Fonctions d'un Client

```
Allouer()
 { Int i ;

  P(nlibre) ;
  for (i=0; i < NMACHINES; i++)
        if (dispo[i] != 0) {
          dispo[i] = 0;
          return i; }
  }

liberer(int machine) {
    dispo(machine) = 1 ;
    V(nlibre) ; }
```

1) Ce programme présente une condition de compétition, laquelle ? pourquoi?

2) Comment corriger cette solution ?

Exercice I.6. (DS 11/2006)

Un pont supporte une charge maximale de 15 tonnes. Ce pont est traversé par des camions dont le poids est de 15 tonnes ainsi que par des voitures dont le poids est de 5 tonnes. On vous demande de gérer l'accès au pont de sorte que :

- La charge maximale du pont soit respectée.
- La priorité doit être donnée aux voitures: lorsqu'une voiture et un camion demandent l'accès au pont, la voiture doit être choisie en priorité, sous réserve, bien sûr, que la capacité maximale du pont soit respectée.

1) Dire à quel type de problèmes classiques appartient ce problème ?

2) Proposez un schéma de synchronisation des processus camion et voiture en utilisant les sémaphores**.**

Exercice I.7. (DS 11/2005)

Soient les 3 processus suivants :

Processus P1	***Processus P2***	***Processus P3***
Début	*Début*	*Début*
Cycle	*Cycle*	*Cycle*
A1	*A2*	*A3*
Fin Cycle	*Fin Cycle*	*Fin Cycle*
Fin	*Fin*	*Fin*

Proposez un schéma de synchronisation de ces trois processus en utilisant des sémaphores dans chacun des cas suivants :

1) Les actions *Ai* ne doivent jamais être simultanées.

2) Les actions *Ai* ne doivent jamais être simultanées et doivent se dérouler toujours dans l'ordre *A1A2A3A1A2A3*...

3) Les actions Ai ne doivent jamais être simultanées et doivent se dérouler toujours dans l'ordre *A1*(*A2* ou *A3*)*A1*(*A2* ou *A3*)...

N. B. *Déclarez clairement vos sémaphores et bien précisez leurs valeurs initiales.*

Exercice I.8. (DS 11/2005)

Un stade d'athlétisme peut recevoir les athlètes de trois (3) clubs A, B et C qui viennent s'y entraîner. Pour organiser les entraînements, on impose la règle suivante :

A un instant donné, le stade peut recevoir un *nombre quelconque* d'athlètes mais de *deux clubs au maximum*. Par exemple, 5 athlètes du club B et 3 athlètes du Club C peuvent s'entraîner en même temps, mais si un athlète du club A veut accéder au stade, il doit attendre jusqu'à ce que tous les athlètes aient quitté le stade, soit du club B soit du club C.

1) On vous demande de proposer un schéma de synchronisation des processus: Processus A, Processus B et Processus C correspondant respectivement à des athlètes des clubs A, B et C, et ce en utilisant des sémaphores. Déclarez clairement vos variables et précisez leurs initialisations.
2) La solution proposée dans 1) présente-t-elle un risque de famine ? justifier votre réponse.

Partie II : Moniteurs

Exercice II.1: (exam. 06/2010)

1) On se propose une mauvaise solution au problème de l'exclusion mutuelle:

```
Monitor SC {
        Boolean token;
        Condition sc;
   public void EntreeSC()
   { if (token)
        token = False;
     else  wait(sc); }
   public void SortieSC ()
   { token = True;
     signal(sc); }
{ token = True;}
}
```

Code des processus

```
While (True) {
    SNC
    SC.EntreeSC() ;
        SC
    SC.SortieSC() ;
    SNC ;
  }
```

Une erreur classique en synchronisation de processus s'est glissée dans la solution proposée. Donnez brièvement :

a) l'erreur.
b) La condition au problème de l'exclusion mutuelle qui n'est pas satisfaite.
c) Une correction.

Exercice II.2. Moniteur (Exam. 01/2006)

Un système dispose d'une ressource unique à accès exclusif utilisable à travers une procédure *occuper_ressource*. Pour gérer cette ressource, on propose la structure de moniteur SC suivante et le schéma d'un processus P utilisant ce moniteur :

```
Monitor SC{
public void occuper()
   { Occuper_ressource; }
{ /* initialisation vide du moniteur}
```

```
Code du processus P {
      SNC
      SC.occuper();
      SNC ;
```

| } | }

1) Pensez vous que cette façon de faire garantit l'exclusion mutuelle? Justifiez.
2) Critiquez la solution proposée.
3) Proposez une autre solution plus adéquate.

Exercice II.3 Problème des Producteurs/Consommateurs (DS 11/2006)

On se propose le problème classique de synchronisation de type producteur/consommateur avec tampon borné (vu en cours), la procédure publique *retirer* dans le moniteur *tampon* est donnée comme suit :

```
Monitor tampon {
……
Condition vide, plein ;

public void retirer (objet x)
{
        if (tampon_vide)
             wait(vide);
        Retirer_objet(x) ;
     Signal(plein) ;
}
…..

} //fin monitor
```

1) Cette procédure fonctionne-t-elle correctement si plusieurs processus consommateurs l'exécutent de manière concurrente ? si oui prouvez le sinon corrigez la procédure.
2) Supposons maintenant l'existence de deux catégories de consommateurs : les prioritaires et les non prioritaires. En considérant qu'une fonction prédéfinie is_prioritary() renvoie cette propriété pour le processus courant, complétez le code de la procédure *retirer* donnée ci-dessous, de manière à favoriser systématiquement la progression des processus prioritaires.

```
Public void retirer (objet x){
    If (is_priority()) {
      ……..
      While (tampon_vide)
           ……..
      Nb_prio--; }
    Else {
           ……..
           {  Wait(vide);
              If (……..)
                   ……..
```

}

}

Retirer_objet(x) ;

Signal(plein) ; }

3) Proposez une solution avec moniteur pour le problème de producteurs/consommateurs avec tampon infini.

Exercice II.4. Synchronisation par moniteurs (DS –11/2010)

Un stade d'athlétisme peut recevoir les athlètes de trois (3) clubs A, B et C qui viennent s'y entraîner. Pour organiser les entraînements, on impose la règle suivante :

A un instant donné, le stade peut recevoir un *nombre limité (soit N par club)* d'athlètes mais de *deux clubs au maximum*. Par exemple, 5 athlètes du club B et 3 athlètes du Club C peuvent s'entraîner en même temps (ici N=5), mais si un athlète du club A veut accéder au stade, il doit attendre jusqu'à ce que tous les athlètes aient quitté le stade, soit du club B soit du club C.

On vous demande de mettre en œuvre la synchronisation des trois types de processus, correspondant à des athlètes des clubs A, B et C, et ce en utilisant les moniteurs et en évitant la famine.

Partie III. IPC Multithread

Exercice III.1. Readers/Writers Multithread (DS 11/2013)

En s'inspirant du problème des lecteurs rédacteurs vu en cours, on souhaite disposer de verrous similaires aux « Mutex », mais permettant d'établir facilement une synchronisation de type « lecteurs/rédacteurs » sans tenir compte de la priorité au sein des applications. L'idée est donc de fournir un type *rwlock_t* et les quatre primitives associées (*rwl_readlock()*, *rwl_readunlock()*, *rwl_writelock()*, *rwl_writeunlock()*) qui permettent à un processus lecteur (resp. rédacteur) d'encadrer la zone de code critique où il accèdera aux données partagées en lecture (resp. écriture).

Rappelons que la lecture est inclusive (possibilité d'avoir plusieurs lecteurs en meme temps) et que la rédaction est exclusive (un seul rédacteur à la fois et jamais de lecteurs)

1) Ecrire la structure *rwlock_t* qui devrait contenir le nombre de lecteurs actifs (*nbLec*), le nombre des rédacteurs actifs (*nbRed*) ainsi que les variables nécessaires à une synchronisation par variables conditionnelles.

2) Ecrire le code des quatre primitives associées à la gestion des verrous en lecture-écriture.

```
/* code à écrire */
typedef ... rwlock_t ;

void rwl_readlock(rwlock_t *l) ;
void rwl_readunlock(rwlock_t *l) ;
void rwl_writelock(rwlock_t *l) ;
void rwl_writeunlock(rwlock_t *l) ;
```

Partie IV. —Les pipes Unix (TP) : Tri parallèle

Ce premier exercice à pour but de gérer plusieurs tubes de communication entre un processus et ses deux clones. Le processus principal à pour rôle de générer aléatoirement des entiers qui seront envoyés à deux processus fils qui devront trier ces nombres dans l'ordre croissant pour le premier et décroissant pour le second.

Pour accélérer la communication, le processus générateur devra créer des paquets de nombres aléatoires à envoyer aux processus fils. La taille de ce paquet dépend de l'argument *-s* passé en paramètre au programme. Le nombre de paquets dépend de l'argument *-n*.

Exemple d'exécution :

 # ./tri -n 512 -s 8000 output1.txt output2.txt

 # ./tri -n 16384 -s 4096

Le processus principal va générer 512 paquets de 8000 entiers. Ces paquets sont envoyés aux deux processus de tris. Ceux-ci, écriront dans les fichiers output1.txt et output2.txt ou sur la sortie standard si ceux-ci ne sont pas spécifiés.

Les tris peuvent être effectués grâce à la fonction *qsort* définie dans l'entête standard *stdlib.h*.

*void qsort(void *base , size_t nmemb , size_t size, int (*compar)(const void *, const void *));*

Vous pouvez utiliser, dans cet exercice, la fonction *getopt* pour analyser les options en ligne de commande. Cette fonction est définie dans l'entête *unistd.h*.