GI-SE Série TD2: Synchronisation

Exercice 1 :

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

Semaphore mutex1.Value = 1;
Semaphore mutex2.Value = 1;

Code du processus P1 :

```
mutex1.wait();
mutex2.wait();
out=out+1;
n=n-1;
mutex2.signal();
mutex1.signal();
```

Code du processus P2 :

```
mutex2.wait();
mutex1.wait();
out=out-1;
n=n+1;
mutex1.signal();
mutex2.signal();
```

1) Cette proposition est-elle correcte ? Si non, pourquoi ?
2) Proposer une solution correcte.

Exercice 1

Semaphore mutex1.Value = 1;
Semaphore mutex2.Value = 1;

Code du processus P1 :

```
mutex1.wait();
mutex2.wait();
out=out+1;
n=n-1;
mutex2.signal();
mutex1.signal();
```

Code du processus P2 :

```
mutex2.wait();
mutex1.wait();
out=out-1;
n=n+1;
mutex1.signal();
mutex2.signal();
```

1) Interblocage.
2) Code des processus P1 et P2 :

Code du processus P1 :

```
mutex1.wait() ;
n=n-1 ;
mutex1.signal() ;
mutex2.wait();
Out = out +1 ;
mutex2.signal();
```

```
Code du processus P2 :
mutex2.wait();
out=out-1;
mutex2.signal();
mutex1.wait();
n=n+1;
mutex1.signal();
```

- **Exercice 2 :**

- On souhaite proposer une généralisation du problème du producteur et consommateur à n producteur et m consommateur.
- Le rôle de **chaque producteur P** est de produire un item puis de le déposer dans une case libre du buffer. Pour cela, il faut disposer d'une case libre.
- Le rôle de **chaque consommateur C** est de supprimer un item du buffer puis de le consommer. Pour cela, il faut avoir des items dans le buffer.
- Quand un processus (qu'il soit **P ou C**) accède au buffer (c'est la section critique), il faut empêcher les autres d'y accéder (exclusion mutuelle).
- Utilisez pour votre solution trois sémaphores:
  - `empty` : vérifie que le buffer contient des cases vides
  - `full` : vérifie que le buffer contient des items.
  - `mtx` : exclusion mutuelle pour l'accès au buffer.

Exercice 2

- La généralisation du problème du P/C à plusieurs processus en utilisant la méthode étudiée en cours est très complexe (i.e. counter). Une généralisation simple repose sur **les sémaphores**.
- Rappelons que:
  - Le rôle de P est de produire un item puis de le déposer dans une case libre du buffer. Pour cela, il faut disposer d'une **case libre**.
  - Le rôle de C est de supprimer un item du buffer puis de le consommer. Pour cela, il faut avoir **des items dans le buffer**.
  - Quand un processus (qu'il soit P ou C) accède au buffer (c'est la section critique), il faut **empêcher les autres d'y accéder** (exclusion mutuelle).

```
do{ //Producteur P
/*produire un item dans  next_prod*/
…
/*Ajouter next_prod dans buffer*/
…
}while(true);
```

```
do{ //Consommateur C
…
/*supprimer item du buffer vers next_cons*/
…
/*Consommer item dans next_cons*/
}while(true);
```

Exercice 2

```
wait(semaphore *S){
 S.value--;
 if (S.value < 0){
 add_Proc(S.list);
 block();}}
```

```
signal(semaphore *S){
 S.value++;
 if (S.value <= 0){
Proc=remove_Proc(S.list);
 wakeup(Proc);}}
```

Comportement de chaque producteur P:

- Pour que P vérifie qu'un emplacement est vide, on introduit un sémaphore **empty** initialisé par la taille du buffer et dont le rôle est de compter les cases vides. P doit donc appeler **`wait(&empty)`** avant de déposer l'item dans buffer.
- Avant de déposer dans le buffer, il faut que P vérifie qu'aucun autre processus n'accède au buffer. On introduit donc un sémaphore **mtx** initialisé à 1 dont le rôle est de protéger l'accès au buffer (section critique). P doit donc appeler **`wait(&mtx)`** avant d'accéder au buffer.
- P peut déposer l'item dans buffer et libérer l'accès au buffer par **`signal(&mtx)`**.

```
do{ //Producteur P
/*produire un item dans  next_prod*/
wait(&empty);
wait(&mtx);
/*Ajouter next_prod dans buffer*/
signal(&mtx)

}while(true);
```

```
do{ //Consommateur C


/*supprimer item du buffer vers next_cons*/


/*Consommer item dans next_cons*/
}while(true);
```

Exercice 2

```
wait(semaphore *S){
 S.value--;
 if (S.value < 0){
 add_Proc(S.list);
 block();}}
```

```
signal(semaphore *S){
 S.value++;
 if (S.value <= 0){
Proc=remove_Proc(S.list);
 wakeup(Proc);}}
```

Comportement de chaque consommateur C:

- Pour que C vérifie que le buffer contient des items, on introduit un sémaphore **full** initialisé à 0 et dont le rôle est de compter les cases pleines. C doit donc appeler `wait(&full)` avant de supprimer l'item du buffer.
- Comme pour P, C doit vérifier qu'aucun autre processus n'accède au buffer avant d'y supprimer l'item. Il appelle `wait(&mtx)`.
- C peut supprimer l'item du buffer et libérer l'accès au buffer par `signal(&mtx)`

```
do{ //Producteur P
/*produire un item dans  next_prod*/
wait(&empty);
wait(&mtx);
/*Ajouter next_prod dans buffer*/
signal(&mtx)

}while(true);
```

```
do{ //Consommateur C
wait(&full);
wait(&mtx);
/*supprimer item du buffer vers next_cons*/
signal(&mtx).

/*Consommer item dans next_cons*/
}while(true);
```

Exercice 2

```
wait(semaphore *S){
 S.value--;
 if (S.value < 0){
 add_Proc(S.list);
 block();}}
```

```
signal(semaphore *S){
 S.value++;
 if (S.value <= 0){
Proc=remove_Proc(S.list);
 wakeup(Proc);}}
```

Libération des sémaphores empty et full:

- Lorsque P dépose un item dans le buffer, il faut incrémenter le nombre d'items disponibles dans le buffer et en informer les consommateurs. Ce nombre est géré par le sémaphore **full**. Il appelle donc `signal(&full)`
- Lorsque C supprime un item du buffer, il faut incrémenter le nombre de cases disponibles dans le buffer et en informer les producteurs. Ce nombre est géré par le sémaphore **empty**. Il appelle donc `signal(&empty)`

```
do{ //Producteur P
/*produire un item dans  next_prod*/
wait(&empty);
wait(&mtx);
/*Ajouter next_prod dans buffer*/
signal(&mtx)
signal(&full)
}while(true);
```

```
do{ //Consommateur C
wait(&full);
wait(&mtx);
/*supprimer item du buffer vers next_cons*/
signal(&mtx).
signal(&empty)
/*Consommer item dans next_cons*/
}while(true);
```

Exercice 3

- **Le problème:**
  - Le salon de coiffure dispose de **`n`** chaises représentant le nombre de clients pouvant rester en attente. Le sémaphore **clients** compte le nombre de clients en attente. Il est initialisé à 0
  - Le client dans le fauteuil est servi par le coiffeur. Le sémaphore **coiffeur** indique l'état du coiffeur s'il est libre (1) ou occupé (0). Il est initialisé à 1.
  - Deux clients ne peuvent pas être servi en même temps par le coiffeur. Le sémaphore **mutex** gère l'exclusion mutuelle pour l'accès au fauteuil du coiffeur . Il est initialisé à 1.
  - La variable **`attente`** est une copie du sémaphore **clients** pour que chaque nouveau client puisse comparer le nombre de clients en attente au nombre de chaises **`n`** et savoir s'il reste ou part. Elle est initialisée à 0 (comme le sémaphore **clients**).

```
do{ //Coiffeur

/*effectuer une coupe*/



}while(true);
```

```
//un Client


/*obtenir une coupe*/
```

Exercice 3

```
wait(semaphore *S){
 S.value--;
 if (S.value < 0){
 add_Proc(S.list);
 block();}}
```

```
signal(semaphore *S){
 S.value++;
 if (S.value <= 0){
Proc=remove_Proc(S.list);
 wakeup(Proc);}}
```

Comportement coiffeur:

- Dormir si aucun client . Le coiffeur appelle **`wait(&clients)`** avant d'effectuer une coupe.
- Lorsque le coiffeur invite un client à son fauteuil (section critique), il faut vérifier qu'un seul client puisse y accéder (décrémenter la variable **`attente`** de façon atomique). Pour cela, il faut la protéger par le sémaphore **`mutex`** pour que deux clients ne soit pas traités en même temps par le coiffeur.
- Le coiffeur appelle **`wait(&mutex)`** avant d'inviter le client au fauteuil (décrémenter **`attente`**), puis appelle **`signal&mutex)`**.
- Le coiffeur effectue la coupe puis appelle **`signal(&coiffeur)`** pour indiquer qu'il est libre.

```
do{ //Coiffeur
wait(&clients);
wait(&mutex);
attente - -;
signal(&mutex)
/*effectuer une
coupe*/
signal(&coiffeur)
```

```
//un Client


/*obtenir une coupe*/
```

Exercice 3

```
wait(semaphore *S){
 S.value--;
 if (S.value < 0){
 add_Proc(S.list);
 block();}}
```

```
signal(semaphore *S){
 S.value++;
 if (S.value <= 0){
Proc=remove_Proc(S.list);
 wakeup(Proc);}}
```

Comportement client:

- Lorsque le client entre dans le salon, il vérifie s'il y a des places libres (compare **attente** au nombre chaises **n**). Evidement, il doit vérifier qu'**attente** ne change pas en cours de comparaison, il appelle **wait(&mutex)**
  - S'il ne trouve pas de place, il n'attend pas et quitte le salon en appelant **signal(&mutex)** .
  - S'il y a une place:
    - il prend une chaise (incrémenter **attente**), et appelle **signal(&clients)** pour augmenter le nombre clients en attente et réveiller le coiffeur si nécessaire.
    - Il appelle **signal(&mutex)** pour indiquer qu'il ne manipule plus la variable **attente** .
    - Il appelle **wait(&coiffeur)** pour attendre d'obtenir une coupe si ce dernier est occupé.

```
do{ //Coiffeur
wait(&clients);
wait(&mutex);
attente - -;
signal(&mutex)
/*effectuer une
coupe*/
signal(&coiffeur)
}while(true);
```

```
//Client
wait(&mutex);
If (attente < n){
attente ++;
signal(&clients);
signal(&mutex);
wait(&coiffeur);
/*obtenir une coupe*/
}
else signal(&mutex);
```