6-7 - SEMAPHORES : PROBLEMES CLASSIQUES

1. LE MODELE PRODUCTEUR-CONSOMMATEUR

Soit un tampon T constitué de n cases dont chacune est destinée à recevoir un message. On nomme Producteur un processus qui dépose de l'information dans le tampon et Consommateur un processus qui retire de l'information. Un tel type de communication est indirect et asynchrone.

- **Indirect** : car les producteurs et les consommateurs ne s'adressent pas l'un à l'autre.
- **Asynchrone** : car il n'est pas nécessaire que les producteurs et les consommateurs travaillent au même rythme.

On se place ici dans le cas où chaque message est lu par un unique consommateur.

1.1.

Quels sont les contraintes à respecter pour l'accès aux cases du tampon ?

On considère les trois tâches utilisateur suivantes :

```
main() {                Production() {          Consommation() {
   sprod = CS(v1);         while (1) {             while (1) {
   scons = CS(v2);            Produire(message);      P(scons);
   CT(Production);            P(sprod);               Retirer(message);
   CT(Consommation);          Déposer(message);       V(sprod);
   ...                        V(scons);               Consommer(message);
}                           }                       }
                         }                       }
```

`Production()` est le code exécuté par un producteur et `Consommation()` le code exécuté par un consommateur. La primitive `Produire()` crée un message et la primitive `Consommer()` traite un message reçu.

`main()` est le programme d'initialisation. Il crée ici un processus producteur et un processus consommateur.

1.2.

Dans quel cas faut-il bloquer l'exécution du producteur ? du consommateur ? En déduire les valeurs initiales de v1 et v2.

Système multi-producteurs / multi-consommateurs

On se place ici dans le cas où `main()` a lancé plusieurs producteurs et plusieurs consommateurs. Les tâches `Production()` et `Consommation()` ne sont pas modifiées, mais on s'intéresse maintenant au contenu des procédures `Déposer()` et `Retirer()`. Ces procédures manipulent le tampon avec les indices respectifs `id` et `ir`, initialisés à 0. Ces indices permettent une gestion circulaire du tampon.

`Déposer(message)` `Retirer(message)`

```
MESS *message;
{
  (a)  T[id] = message;
  (b)  id = (id + 1) % n;
}
```

```
MESS *message;
{
   (c)  message = T[ir];
   (d)  ir = (ir + 1) % n;
}
```

1.3.

Montrer que plusieurs producteurs (resp. consommateurs) peuvent déposer (resp. retirer) dans la même case. Comment doit-on modifier les procédures `Déposer()` et `Retirer()` pour que la gestion des cases du tampon reste correcte, même lorsqu'on a plusieurs producteurs et plusieurs consommateurs ?

On désire assurer une plus grande indépendance de fonctionnement des producteurs et des consommateurs. Un producteur en train de produire un message ne doit pas empêcher d'autres producteurs d'acquérir des cases vides pour produire. Pour cela il faut dissocier :

- l'acquisition d'une case vide, de son remplissage.
- l'acquisition d'une case pleine, de son vidage.

1.4.

Réécrire simplement les procédures `Déposer()` et `Retirer()`. Montrer que dans ce cas, il faut rajouter des sémaphores pour garantir qu'un producteur ne produira pas dans une case en train d'être consommée.

1.5.

Comment les performances pourraient-elles être encore améliorées ?

2. LE PROBLEME DES LECTEURS / ECRIVAINS

Soit un fichier pouvant être accédé en lecture ou en écriture. Pour garantir la cohérence des données de ce fichier, les processus qui souhaitent y accéder sont rangés dans deux classes, en fonction du type d'accès qu'ils demandent :

- les processus lecteurs ;
- les processus écrivains.

Le nombre de processus dans chaque classe n'est pas limité. On établit le protocole d'accès de la manière suivante :

- il ne peut y avoir simultanément un accès en lecture et un accès en écriture ;
- les écritures sont exclusives entre elles (au plus un accès en écriture à un instant donné).

Par contre, plusieurs lectures peuvent avoir lieu simultanément.

La structure des processus est la suivante :

```
            Lecteur
OuvreLecture() ;
   /* Accès en lecture */
FermeLecture() ;
```

```
            Ecrivain
OuvreEcriture() ;
   /* Accès en écriture */
FermeEcriture() ;
```

où la procédure OuvreLecture (resp. OuvreEcriture) contient les instructions que doit effectuer un lecteur (resp. un écrivain) avant de pouvoir accéder à la ressource, et la procédure FermeLecture (resp. FermeEcriture) contient les instructions qu'un lecteur (resp. un écrivain) exécute lorsqu'il relâche la ressource.

On considère dans un premier temps la politique d'accès suivante : si la ressource est disponible ou s'il y a déjà des requêtes de lecture en cours, une nouvelle requête de lecture est traitée immédiatement même s'il y a des demandes d'écriture en attente.

2.1.

Quelles sont les conditions de blocage pour un écrivain ? Pour un lecteur ? Donnez les sémaphores et les variables partagées nécessaires pour tester ces conditions, ainsi que leur initialisation.

2.2.

Programmez cette synchronisation de processus. Quel est l'inconvénient de cette stratégie d'accès ?

Pour assurer l'équité, on veut maintenant gérer les accès dans l'ordre FIFO : lors de leur arrivée, tous les processus sont rangés dans une même file. On extrait de cette file soit un unique écrivain, soit le premier lecteur d'un groupe. Dans le deuxième cas, on autorise les lecteurs suivants dans la file à passer, jusqu'à ce que le processus en tête de file soit un écrivain.

On fera dans cette question l'hypothèse que la file d'attente des sémaphores est FIFO (ce qui n'est pas forcément vérifié dans le cas général).

2.3.

Par rapport à la question 2.1, quel sémaphore supplémentaire est nécessaire pour réaliser cette stratégie ? Programmez cette synchronisation de processus.

2.4.

Montrez que, dans la solution de la question précédente, une mauvaise utilisation des sémaphores (inversion de deux opérations P) peut conduire à un interblocage.