6 - 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 le processus qui dépose l'information dans le tampon et Consommateur le processus qui retire 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.

1.1.

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

Exclusion Mutuelle : Deux producteurs (resp. consommateurs) ne doivent pas produire (resp. consommer) dans la même case. Pas d'accès simultané d'un producteur et d'un consommateur à la même case.

Ordre d'occurrence des opérations : Pas de consommation dans un tampon vide et pas de production dans un tampon plein.

On considère les trois tâches utilisateur suivantes :

```
main() {
   sprod = CS(v1);
   scons = CS(v2);
   CT(Production);
   CT(Consommation);
   ...
}
```

```
Production() {
   while (1) {
      Produire(message);
      P(sprod);
      Déposer(message);
      V(scons);
   }
}
```

```
Consommation() {
   while (1) {
      P(scons);
      Retirer(message);
      V(sprod);
      Consommer(message);
   }
}
```

La primitive `CT()` permet de créer une tâche. `main()` est le programme d'initialisation, `Production()` 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.

1.2.

Expliquer le rôle des sémaphores `sprod` et `scons`. Quelles doivent être les valeurs initiales de v1 et v2 ?

Le sémaphore sprod bloque un producteur lorsqu'il n'y a pas de cases libres, le sémaphore scons bloque un consommateur lorsqu'il n'y a pas de cases pleines. Le compteur de sprod doit donc être initialisé à n, celui de de scons à 0.

On s'intéresse maintenant au contenu des procédures `Déposer()` et `Retirer()` des tâches `Production()` et `Consommation()`. 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)
MESS *message;
{
  (a)  id = (id + 1) % n;
  (b)  T[id] = message;
}
```

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

1.3.

Montrer que deux producteurs (resp. consommateurs) peuvent produire (resp. consommer) dans la même case. Comment doit-on modifier les procédures `Déposer()` et `Retirer()` pour avoir une gestion correcte des cases du tampon ?

Si Déposer() (resp. Retirer()) est exécutée par un seul producteur (resp. consommateur), le fonctionnement est correct. Le problème vient de ce que les variables id et ir sont maintenant partagées, donc *critiques*. Considérons deux producteurs exécutant Déposer(). Avec la séquence d'exécution suivante, les deux producteurs écrivent dans la même case du tampon !

id = 0;
(a1) id =1;
(a2) id = 2;
(b2) T[2] = message;
(b1) T[2] = message;

Il faut introduire un sémaphore d'exclusion mutuelle entre les producteurs (MutexP) et entre les consommateurs (MutexC). Ainsi, deux producteurs ne pourront plus écrire dans la même case du tampon (idem pour les consommateurs).

```
MutexP.cpt = 1;
Déposer(message)
MESS* message; {
      P(MutexP);
      ++id %= n;
      T[id] = message;
      V(MutexP);
}
```

```
MutexC.cpt = 1;
Retirer(message)
MESS* message; {
      P(MutexC);
      ++ir %= n;
      message = T[ir];
      V(MutexC);
}
```

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.

Réécrivez simplement les procédures `Déposer()` et `Retirer()`. Montrez 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.

Un producteur ne bloque la primitive Déposer() que le temps de prendre le numéro de case où il va produire (I) (plutôt que de la bloquer pendant le temps nécessaire à la prise d'un numéro et le temps de remplissage de la case). Les procédures déposer et retirer deviennent :

Déposer(message)

```
MESS* message; {
      int I;
      P(MutexP);
      ++id %= n;
      I = id;
      V(MutexP);
      T[I] = message;
}
```

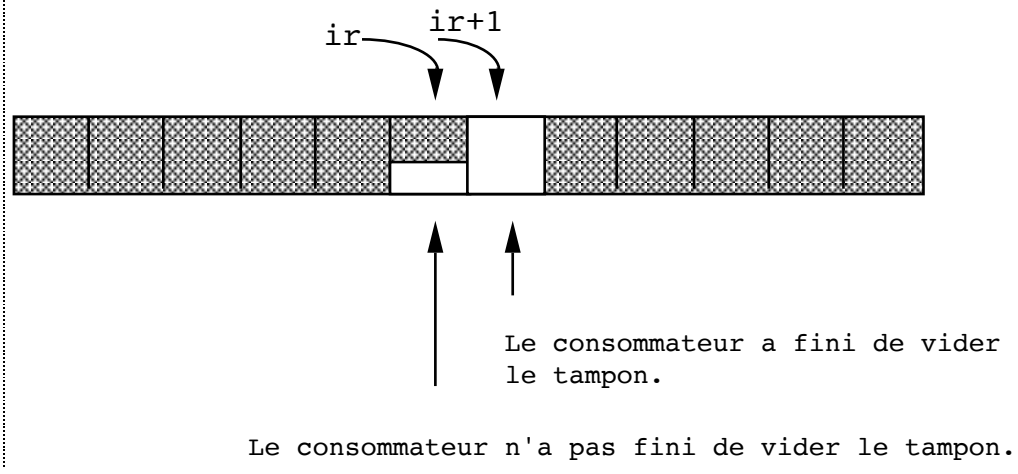
Retirer(message)

```
MESS* message; {
      int J; // Id. locale
      P(MutexC);
      ++ir %= n;
      J = ir;
      V(MutexC);
      message = T[J];
}
```

Note : id et ir sont des variables globales, alors que I et J sont des variables locales.

Petit Problème: Supposons que l'on dispose d'un producteur, deux consommateurs et de n cases dans le tampon et qu'elles sont pleines.

- Le producteur attend une case. La prochaine case qui lui sera allouée sera la première.
- Le premier consommateur arrive à l'instruction `message = T[J];` (Vidage de la case). Supposons que cela prenne un temps très long. Pendant ce temps, le second consommateur vide entièrement la seconde case et fait un V(sprod). Le producteur peut alors produire dans la case que le premier consommateur est toujours en train de vider…



Un producteur qui était bloqué sur P(sprod) va pouvoir écrire dans le tampon de la case "ir". ⇒Le premier consommateur va finir de vider une case où un producteur est en train d'écrire.

Solution: sprod (resp. scons) garantit qu'il n'y a pas plus de productions (resp. de consommations) en cours que de cases vides (resp. pleines). MutexP (resp. MutexC) garantit que deux producteurs (resp. consommateurs) ne s'attribuent pas le même numéro de case. Il reste à garantir qu'un producteur et un consommateur n'accèdent pas à la même case. Pour cela, on

utilise n sémaphores SC, initialisés à 1, qui indiquent s'il y a déjà un accès à la case. Le code devient alors :

```
    Déposer(message)                    Retirer(message)

MESS* message;          {               MESS* message; {
    int I;                              int J;     // Ide. local
    P(MutexP);                          P(MutexC);
    ++id %= n;                          ++ir %= n;
    I = id;                             J = ir;
    P(SC[I]);                           P(SC[J]);
    V(MutexP);                          V(MutexC);
    T[I] = message;                     message = T[J];
    V(SC[I]);                           V(SC[J]);
}                                   }
```

NB : il n'est pas possible d'inverser l'ordre de P(SC[]) et de V(Mutex*). Dans ce cas, si un processus était interrompu entre V(Mutex) et P(SC), un autre processus pourrait déposer un message dans la case suivante et envoyer un V(scons) au consommateur. Celui-ci accèderait librement à la première case qui n'a pas été remplie ni bloquée.

1.5.

Comment les performances pourraient-elles être encore améliorées ?

Le fait d'avoir dissocié l'acquisition d'un numéro de case et le remplissage (pour les consommateurs) ou le vidage (pour les producteurs) est un gain de temps évident lors de l'exécution : les producteurs pourront remplir leurs cases en parallèle, et les consommateurs pourront les vider en parallèle également. Mais du fait de la gestion circulaire du tampon, les producteurs (resp. consommateurs) peuvent se trouver bloqués alors qu'il y a des cases libres (resp. pleines) dans le tampon (voir le dessin de la question précédente).

Les performances pourraient donc être améliorées en redéfinissant l'acquisition d'une case libre (ou pleine), par exemple en gérant une liste d'indices de cases disponibles à la production ou à la consommation au lieu de parcourir circulairement les cases.

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.
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.

Un écrivain est bloqué dès qu'un autre processus (lecteur ou écrivain) accède à la ressource. Il faut donc un sémpahore Res, initialisé à 1, qui est décrémenté dès que la ressource est occupée. Ce sémaphore est manipulé à la fois par les lecteurs et les écrivains.

Un lecteur a besoin de savoir s'il est le premier à accéder à la ressource : dans ce cas, il doit bloquer toute tentative d'accès par un écrivain. Il doit aussi savoir, lorsqu'il termine son accès, s'il est le dernier pour rendre la ressource disponible. On a donc besoin d'un compteur nblec partagé entre tous les lecteurs, donnant le nombre d'accès en lecture. Ce compteur doit être protégé par un sémaphore d'exclusion mutuelle MutexL, initialisé à 1.

Un lecteur ne se bloque que si la ressource est accédée par un écrivain, ce qui peut être réalisé à l'aide du sémaphore Res.

2.2.

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

```
OuvreLecture()                  OuvreEcriture()
{                               {
      P(MutexL);                  P(Res);
      nblec =  nblec + 1;       }
      if (nblec == 1) {
      P(Res);
      }
      V(MutexL);
}

FermeLecture()                  FermeEcriture()
{                               {
      P(MutexL);                V(Res);
      nblec =  nblec - 1;       }
      if (nblec == 0) {
            V(Res);
      }
      V(MutexL)
}
```

Cette stratégie risque d'entrainer un problème de famine pour les processus écrivains : si des requêtes de lecture arrivent régulièrement, la ressource n'est jamais libérée et les écrivains ne peuvent jamais y accéder.

Pour assurer l'équité, on gère maintenant 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.

2.3.

Par rapport à la question 3.1, quel sémaphore supplémentaire est nécessaire pour réaliser cette stratégie ? Programmez cette synchronisation de processus.

On a besoin d'un sémaphore File, initialisé à 1, que tous les processus vont tester de manière à assurer l'accès dans l'ordre des requêtes.

```
OuvreLecture()                  OuvreEcriture()
{                               {
      P(File);                    P(File);
      P(MutexL);                  P(Res);
      nblec =  nblec + 1;       }
      if (nblec == 1) {
            P(Res);
      }
      V(MutexL);
      V(File);
}

FermeLecture()                  FermeEcriture()
{                               {
      P(MutexL);                  V(File);
      nblec =  nblec - 1;         V(Res);
      if (nblec == 0) {         }
            V(Res);
      }
      V(MutexL)
}
```

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.

L'inversion du P(File) et du P(MutexL) effectués par un lecteur peuvent mener à un interblocage. Tant que les lecteurs sont seuls à intervenir, il n'y a pas de problème. Par contre, s'il y a une lecture en cours et si un rédacteur vient effectuer P(File) puis se bloquer sur P(Res), le sémaphore File devient bloquant. Un nouveau lecteur qui se présente passe le P(MutexL) puis se bloque sur le P(File). Tous les sémaphores sont maintenant bloquants. Seule la primitive FermeEcriture() pourrait encore être exécutée, mais il n'y a pas d'écriture en cours. On arrive donc à un interblocage.

L'inversion des opérations P effectuées par un rédacteur supprime le caractère FIFO de la solution. De plus, lorsque le dernier lecteur débloque le rédacteur bloqué sur Res, si un nouveau lecteur se présente avant que le rédacteur n'ait effectué le P(File), ce lecteur va se bloquer sur le sémaphore Res sans libérer le sémaphore File. A nouveau, on arrive à un interblocage. Le problème est le même si on décide d'inverser les opérations P à la fois pour le lecteur et le rédacteur.