INF2610

Chapitre 5 : Synchronisation

5.1 Traditionnellement, les sémaphores étaient implantés en mode noyau afin de pouvoir gérer les attentes et réveils lorsque les sémaphores sont inférieurs ou égaux à 0. Est-ce essentiel? Dans quels cas pourrait-on faire mieux? Comment?

Avec les systèmes multi-processeurs, les opérations de verrouillage utilisent les instructions atomiques qui sont disponibles autant en mode usager qu'en mode noyau. De plus, il est possible d'avoir des variables en mémoire partagées entre les processus et le noyau, par exemple un sémaphore qui doit être accédé de plusieurs processus ainsi que du noyau. Depuis quelques années, le système d'exploitation Linux tire parti de cela en implantant les sémaphores POSIX à l'aide des nouveaux *futex*, fast user mode mutex. L'appel en librairie C standard pour les sémaphores peut obtenir un mutex libre (> 0), ou relâcher un mutex sans queue d'attente (>= 0), directement en mode usager. Ceci a un impact significatif car c'est normalement le cas le plus courant. Autrement, il fait un appel système *futex* qui s'occupe de mettre le fil d'exécution en attente ou de réveiller les fils d'exécution en attente.

5.2 Est-ce que l'algorithme de l'attente active avec alternance fonctionne aussi sur un ordinateur multi-processeur à mémoire partagée?

Oui, le principe reste le même. Il faut toutefois faire attention à la cohérence mémoire, il est possible que des barrières mémoires soient requises. Ces barrières indiquent au compilateur de ne pas réordonnancer certaines opérations et s'assurent que les changements sur les variables partagées soient visibles sur tous les processeurs, même s'ils ont chacun leur mémoire cache. Le fait qu'il s'agit d'attente active, et potentiellement gaspille temps et énergie, demeure vrai sur multi-processeur.

5.3 Est-ce que l'algorithme de Peterson pour l'exclusion mutuelle peut fonctionner sur un système avec ordonnancement préemptif? Non préemptif?

Dans le cas non-préemptif, si un processus monopolise le CPU en attente, alors que celui après lequel il attend ne peut faire de progrès et relâcher le verrou, un interblocage s'ensuit.

5.4 Peut-on utiliser une instruction d'échange atomique (swap) plutôt qu'une instruction TSL (test and set lock) afin de réaliser une fonction permettant de gérer l'entrée dans une région critique?

Oui, si un verrou occupé prend la valeur 1, en faisant un échange avec la valeur 1 afin de tenter de prendre un verrou, deux résultats sont possibles : la valeur échangée est 0 et donc le verrou était libre et vient d'être obtenu, ou la valeur échangée est 1 (le verrou n'est pas libre) et il faut essayer à nouveau.

5.5 Comment peut-on programmer des sémaphores en utilisant seulement la capacité de l'ordinateur à désactiver les interruptions?

Sur un ordinateur mono-processeur, désactiver les interruptions permet de protéger une section critique. Il est donc possible d'incrémenter ou décrémenter le sémaphore et vérifier sa valeur pour savoir s'il faut mettre le processus en attente ou s'il faut réveiller un processus en attente. Cette technique ne fonctionne toutefois pas sur multi-processeur entre des processus qui roulent sur des processeurs différents.

5.6 Peut-on programmer des sémaphores à l'aide de mutex et de variables ordinaires?

Oui, ceci peut être réalisé de différentes manières. Une façon simple est d'avoir une variable pour le décompte et deux mutex, l'un V pour verrouiller la région critique de manipulation du compteur, et l'autre A pour servir de queue d'attente. Lors d'un décrément du sémaphore, V est pris et le compteur est décrémenté. Si le compteur est positif, V est relâché. Autrement, s'il est négatif, le processus est ajouté sur la liste des processus en attente, V est relâché et A est pris résultant en une attente. Lors d'un incrément du sémaphore, V est pris et le compteur incrémenté. S'il est supérieur à 0, V est simplement relâché. Autrement, s'il est négatif ou 0, un processus est retiré de la liste de ceux en attente, A est incrémenté réveillant un processus et finalement V est relâché.

5.7 Peut-il être pertinent d'utiliser une barrière si on n'a que deux processus?

Oui, il se peut que chaque processus ait besoin des résultats calculés en mémoire partagée par l'autre.

5.8 Est-ce que deux fils d'exécution dans un même processus peuvent se synchroniser à l'aide de sémaphore, s'il s'agit de fils gérés par le système? Gérés en mode usager?

En mode système, il n'y a aucun problème. En mode usager, il faudrait que l'appel système pour les sémaphores soit intercepté et géré en partie en mode usager. Autrement, le processus complet sera bloqué et l'autre fil ne pourra débloquer le sémaphore.

5.9 On vous propose de remplacer wait et signal pour les conditions par une forme plus générale permettant d'attendre sur une expression (e.g. wait until (x < 0 || x + y < n)). Cette primitive serait plus flexible. Est-ce que ce serait avantageux?

Non, ce genre d'attente serait coûteux à implémenter. A chaque modification d'une variable de l'expression, il faudrait réévaluer l'expression.

5.10 Considérez un système multicouche composé de trois couches P0, P1 et P2. Les couches sont des processus concurrents qui communiquent au moyen de deux tampons T0 et T1 de même taille N: P0 et P1 partagent le tampon T0 et P1 et P2 partagent le tampon T1. Chaque couche se charge d'un traitement particulier :

Le processus P0 se charge de lire du clavier des messages qu'il traite avant de les déposer dans le tampon T0. Le traitement d'un message par la couche P1 consiste à l'encrypter. Il est réalisé par la fonction Encrypter suivante :

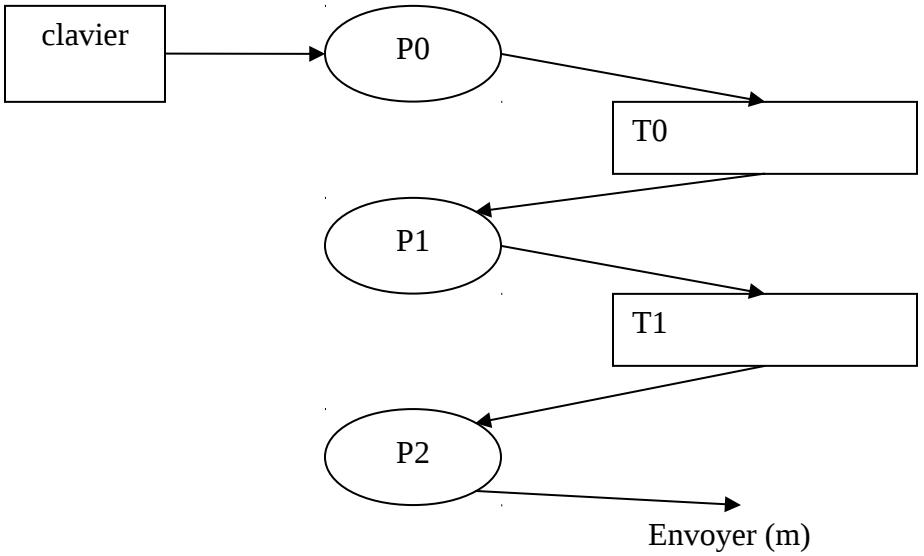
Message Encrypter (Message);

La fonction **Message Lire ();** permet de lire un message du clavier.

Le processus P1 se charge de transférer directement les messages du tampon T0 vers le tampon T1.

Le processus P2 récupère les messages du tampon T1 pour les envoyer à un destinataire. L'envoi d'un message est réalisé par la fonction Envoyer :

Envoyer (Message);



Expliquez comment les processus peuvent utiliser les sémaphores pour contrôler les accès aux tampons partagés (exclusion mutuelle, pas d'interblocage). Donnez les pseudocodes des trois processus.

5.11 Sémaphores

1. Expliquez ce qui peut arriver si la file d'attente d'un sémaphore est gérée selon la discipline LIFO (last in first out).
2. Expliquez un avantage de l'utilisation de moniteurs sur les sémaphores pour la synchronisation de processus.
3. Complétez, en ajoutant les sémaphores et les opérations P et V nécessaires, les codes du producteur et du consommateur suivants. Le producteur produit plusieurs ressources à la fois alors que le consommateur consomme une seule ressource à la fois.

```
char T[N]; // tableau de N caractères
Semaphore Plein =0, Vide=N, Mutex=1

Producteur {
int ip=0, M;
char ch[N];
  Repeter
  {
         M=Lire(ch,N);

        Deposer(ch, M, ip);


        ip = (ip + M) % N;

  }
}
```

```
Consommateur {
int ic=0;
char c;
  Repeter
  {

        c = Retirer( ic);
        ic = (ic+1) %N

        Traiter(c);
    }

}
```

La fonction « int Lire(char ch[], int N); » construit, dans ch, une chaîne de caractères de longueur comprise entre 1 et N inclusivement. Elle retourne la longueur de la chaîne.

La fonction « void Deposer(char ch[], int M, int ip); » insère, dans le tampon T, la chaîne de caractères ch. M est la longueur de la chaîne.

La fonction « char Retirer(int ic); » retire un caractère du tampon T. Elle retourne le caractère retiré.

La fonction « void Traiter(char c); » traite le caractère.

5.12 Trois processus concurrents P1, P2 et P3 exécutent chacun le programme suivant:

```
Pi ()  // i = 1,2,3
{       int n=0;
        while(true)
                printf("cycle %d de  %d", n++, i);
}
```

Synchronisez les cycles des processus à l'aide de sémaphores de manière à ce que :

- Chaque cycle de P1 s'exécute en concurrence avec un cycle de P2.
- Le processus P3 exécute un cycle, lorsque P1 et P2 terminent tous les deux l'exécution d'un cycle.
- Lorsque P3 termine un cycle, les processus P1 et P2 entament chacun un nouveau cycle et ainsi de suite…

5.13 La ville de Montréal veut restructurer la circulation sur son territoire. Le conseil municipal décide donc de resynchroniser les intersections pour réduire le trafic aux heures de pointe. La ville vous engage pour réaliser un modèle de synchronisation, à l'aide de sémaphores, d'une intersection typique de la ville. Il est important de noter que le virage à droite sur le feu rouge n'est pas légal sur l'île de Montréal. L'intersection choisie possède 3 voies nord-sud et 2 voies est-ouest.

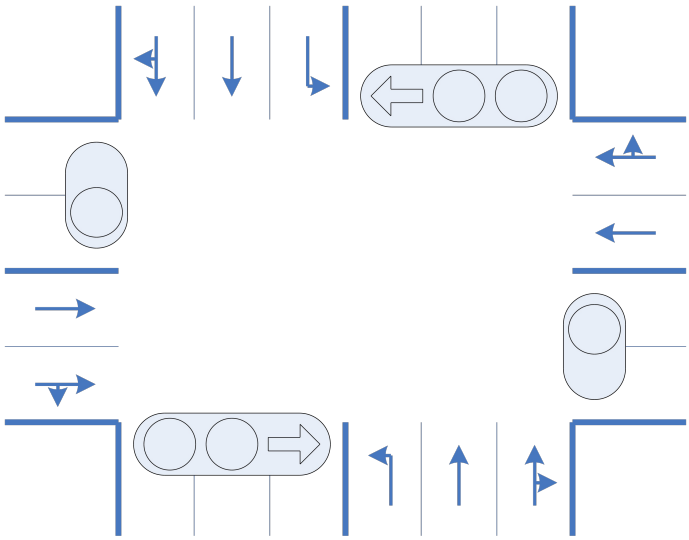
3 voies nord-sud :

Parmi les 3 voies nord-sud, la plus à droite permet de tourner à droite et d'aller tout droit. La centrale permet d'aller tout droit tandis que celle de gauche est une voie réservée pour le tournant à gauche. Les conducteurs engagés dans cette voie doivent attendre la flèche verte pour tourner, tandis que le feu vert dans les deux autres voies sont des feux pleins (feux ronds). Les voies opposées sont soumises aux mêmes règles.

2 voies est-ouest :

La voie la plus à droite de ces deux voies permet aux conducteurs de tourner à droite ou d'aller tout droit; par contre, la plus à gauche de ces voies permet seulement d'aller tout droit. Comme dans la majorité des intersections de la ville, le virage à gauche n'est pas permis dans ce sens de la circulation. Le feu qui permet de traverser l'intersection pour ces voies est un feu plein. Les voies opposées sont soumises aux mêmes règles.

Le schéma suivant illustre l'explication :



On suppose que l'intersection est toujours libre : dès qu'un conducteur s'engage, il sera apte à traverser entièrement l'intersection. Les feux du sens est-ouest ouest-est sont initialement verts. Les feux opposés sont toujours dans le même état : si la flèche est verte l'autre flèche l'est aussi. La séquence de passage au vert doit suivre l'ordre suivant : le feu est-ouest, puis le feux nord-sud, et la flèche pour finir. Seulement un des trois énumérés peut être vert à la fois.
Chaque voie possède sa file de voitures en attente du feu vert. Ces files seront modélisées par des listes de la STL. De plus, un processus léger sera en charge de synchroniser les feux à l'aide de sémaphores. L'attente active dans le modèle n'est pas acceptable.

Identifier les modèles de synchronisation classiques présents dans ce système. **Déterminez** le nombre de processus légers nécessaires, le nombre de sémaphores et de mutex avec leurs valeurs initiales. **Indiquez** le rôle de chacun.

5.14 Deux processus A et B communiquent au moyen d'un tampon T pouvant contenir qu'un seul message à la fois. Ce tampon est utilisé, de façon alternée, pour la communication dans les deux sens (attention un seul processus utilise à la fois ce tampon). Le processus A dépose un message dans le tampon puis attend la réponse de B avant de déposer à nouveau un autre message et ainsi de suite…. Lorsque B reçoit un message de A, il dépose sa réponse dans le tampon puis se met en attente d'un autre message de A et ainsi de suite… Synchronisez au moyen de sémaphores les processus A et B (pour répondre à la question, complétez le code suivant)

```
semaphore  ….. ;  /*0*/
char T[256] ;
void depot (char buf[] ) ;
void recuperer(char buf[] ) ;
```

Processus A

```
{  char mess[256], rep[256] ;

   while (1)
   {      /* 1 */
          lire (mess);
          depot (mess ) ;

          /* 2*/
          recuperer(rep) ;
         /*3*/
   }

   }
```

Processus B

```
{    char mess[256] , rep[256] ;

    while (1)
    {      /*  4 */

              recuperer( mess) ;
              reponse(mess,rep)
              /*5*/
                      depot(rep);
              /* 6*/
    }

}
```

Supposez maintenant qu'un troisième processus C veuille communiquer avec B en utilisant l'unique tampon T. Les processus A et C se comportent de la même manière. B peut donc recevoir un message de A ou C, la réponse doit être récupérée par le processus expéditeur du message. Synchroniser au moyen de sémaphores les processus A, B et C.

5.15 Synchronisez au moyen de sémaphores l'enchaînement des opérations de fabrication de stylos à bille. Chaque stylo est formé d'un corps, d'une cartouche, d'un bouchon arrière et d'un capuchon. Les opérations à effectuer sont les suivantes :

- remplissage de la cartouche avec l'encre (opération RC),

- assemblage du bouchon arrière et du corps (opération BO),
- assemblage de la cartouche avec le corps et le capuchon (opération AS),
- emballage (opération EM).

Chaque opération est effectuée par une machine spécialisée (mRC, mBO, mAS, mEM). Les stocks de pièces détachées et d'encre sont supposés disponibles quand la machine est disponible. Les opérations RC et BO se font en parallèle. L'opération AS doit être effectuée, après ces deux opérations, en prélevant directement les éléments sur les machines mRC et mBO. Le produit assemblé est déposé dans un stock en attente de l'opération EM. L'opération EM se fait donc après AS, à partir du stock. Le stock est supposé de taille N et de discipline FIFO.

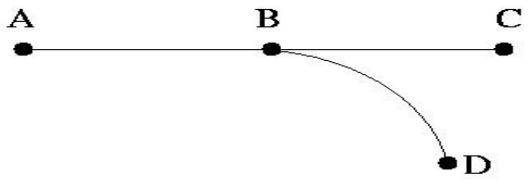
```
mRC( )
{  while (1)
   {
      RC( ) ;

   }
}
```

```
mBO( )
{   while (1)
   {

       BO( ) ;

   }
}
```

```
mAS( )
{    while (1)
   {
      AS( ) ;
      }
}
```

```
mEM( )
{     while (1)
          {
             EM() ;

          }
}
```

5.16 On dispose d'une carte électronique à base de microcontrôleurs pour contrôler un ensemble de robots. La carte est livrée avec un logiciel sous Linux, qui permet de créer son propre programme pour commander et coordonner un ensemble de robots, et de le charger ensuite dans la mémoire non volatile (sur la carte) par un port série. On vous sollicite pour écrire un pseudocode qui contrôle le déplacement de plusieurs robots sur les chemins suivants :



- Les robots peuvent partir de A vers C ou de D vers A.
- Pour éviter tout risque de collision, il faut s'assurer que chaque segment du chemin (segments AB, BC et DB) est utilisé par un robot au plus.

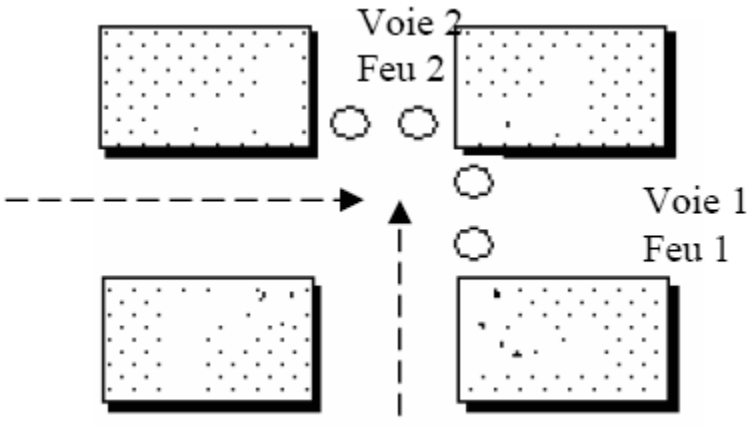
Pour répondre à cette question, complétez le pseudocode suivant afin que les règles ci-dessus soient respectées. Dans ce programme, chaque robot est commandé par un processus.

Est-ce que votre solution présente un problème de famine ? interblocage ? Justifiez.

```
/*0*/
void TraverserSegAB ( ) ; // Traverser le segment AB
void TraverserSegBC ( ) ; // Traverser le segment BC
void TraverserSegBD ( ) ; // Traverser le segment BD
```

Processus RobotAC	Processus RobotDA
{	{
/* 1 */ TraverserSegAB ( ) ; /* 2*/ TraverserSegBC( ) ; /*3*/	/* 4 */ TraverserSegBD( ) ; /*5*/ TraverserSegAB( ); /* 6*/
}	}

5.17 La circulation dans une intersection de deux voies à sens unique est réglée par des signaux lumineux (feu vert/rouge). On suppose que les voitures traversent l'intersection en **ligne droite** et que l'**intersection peut contenir au plus une voiture à la fois**.



On impose les conditions suivantes :

- toute voiture se présentant à l'intersection la franchit en un temps fini ;
- les feux de chaque voie passent alternativement du vert au rouge, chaque couleur étant maintenue pendant un temps fini (Duree_du_feu) ;
- les arrivées sur les deux voies sont réparties de façon quelconque.

Le fonctionnement de ce système peut être modélisé par un ensemble de processus parallèles:

- un processus `P` qui exécute la procédure `Changement` qui commande les feux;
- un processus est associé à chaque voiture; la traversée du carrefour par une voiture qui circule sur la voie i (i = 1, 2) correspond à l'exécution d'une procédure `Traverseei()` par le processus associé à la voiture.

Pour simuler un tel système, on vous demande, dans un premier temps, de compléter, en ajoutant les sémaphores et le code nécessaires, les procédures suivantes (attention : vous ne devez pas ajouter de variables d'autres types) :

```
Semaphore ……

Changement ()
{int Feu = 1;
 while(1)
 {sleep(Duree_du_feu);
   if  Feu == 1)
   { ……     }
   else
   { ……    }
 }
}
```

```
Traversee1()
{
  ……
  circuler();
  ……
}
```

```
Traversee2()
{
   ……
  circuler();
  ……
 }
```

// Feu = 1 si le feux de la voie 1 est vert, Feu =2 si le feux de la voie 2 est vert

//circuler simule la traversée de l'intersection

Dans une seconde étape, il vous est demandé de compléter le moniteur suivant, en ajoutant les variables de condition et le code nécessaires :

```
Moniteur Intersection
{   ……
    int Feu = 1, Nbw1=0 ; Nbw2=0;

    Changement ()
    {
     while(1)
     {sleep(Duree_du_feu);
       if  Feu == 1)
       { ……      }
        else
       { ……   }
     }
    }
  }

    Traversee1()
    {
      ……
      circuler();
      ……
            }

    Traversee2()
    {
       ……
      circuler();
      ……
            }
```

// Feu = 1 si le feux de la voie 1 est vert, Feu =2 si le feux de la voie 2 est vert

// Nbw1 est le nombre de voitures …… sur la voie 1

//Nbw2 est le nombre de voitures .…….sur la voie 2.

5.18 On souhaite implémenter au moyen de sémaphores les compteurs d'événements. Un compteur d'événements est un compteur associé à un événement. Sa valeur indique le nombre d'occurrences de l'événement associé. Généralement, ce compteur est partagé entre plusieurs processus et sert à les synchroniser. Dans ce sens, trois opérations atomiques sont définies pour un compteur d'événement E :

- Read(E) : retourne la valeur de E au processus appelant.
- Advance(E) : incrémente de 1 la valeur de E et débloque tous les processus en attente que E atteigne cette nouvelle valeur.
- Await(E, v) : bloque le processus appelant, si la valeur de E est strictement inférieure à v. Il n'y a pas de blocage du processus appelant si la valeur de E est déjà égale ou plus grande que v.

Complétez, au moyen de **sémaphores**, la structure et les procédures Await, Read et Advance suivantes (sous forme de pseudocode). **Indiquez** clairement les structures de données ajoutées dans la structure CompteurEvenement ainsi que les opérations associées.

```
struct CompteurEvenement
{   int val ;    /* 0 */     }
```

```
void Await (CompteurEvenement *E; int Valeur)
{         /*1*/      }

void Advance (CompteurEvenement *E)
{       /* 2 */  }

int Read (CompteurEvenement *E)
{         /* 3 * /  }

CompteurEvenement    CE =     /* 4*/ ;
```

5.19 On considère le problème du passage à niveau à voie unique. Pour simuler le fonctionnement de ce système, on se propose de le modéliser par un ensemble de processus parallèles :

- un processus Contrôleur qui se charge de commander la fermeture et l'ouverture des barrières. Les barrières doivent être fermées lorsqu'un train traverse le passage à niveau. Le contrôleur ouvre les barrières s'il n'y a aucun train en attente du passage à niveau.
- un processus Train est associé à chaque train. Il est créé de façon aléatoire pour simuler l'arrivée d'un train ainsi que la traversée du passage à niveau.

  Complétez, en ajoutant les **sémaphores** et le **code** nécessaires, les procédures suivantes exécutées par les processus Contrôleur et Train (attention : vous ne devez pas ajouter de variables d'autres types). Si besoin est, vous pouvez utiliser l'opération int PNB(semaphore s) qui est l'équivalent de sem_trywait(&s). **Expliquez** le rôle de chaque sémaphore (**indiquez** l'utilité de sa **queue** ainsi que sa **valeur initiale**).

  Attention : Vous ne devez pas ajouter de variables autres que les sémaphores.

```
Semaphore ..... /*0*/
Contrôleur ()
{
 while(1)
 { ..... /*1*/


FermerBarrieres() ;
  ..... /*2*/

  OuvrirBarrieres() ;
 }
}
```

```
Train()
{
 ..... /*3*/

  Traverser();
       ..... /*4*/
}
```

5.20 On dispose d'un mécanisme d'enregistrement à un ensemble de cours, tel que tout étudiant ne peut être inscrit qu'à au plus trois cours, et que chaque cours a un nombre limité de places. Un étudiant inscrit déjà à trois cours peut s'il le souhaite en abandonner un, pour en choisir un autre dans la limite des places disponibles. Si cet échange n'est pas possible, l'étudiant ne doit pas perdre les cours auxquels il est déjà inscrit.

Le bureau des affaires académiques souhaite donc mettre en place un système de permutation de cours, permettant à un étudiant de changer de cours. Il vous sollicite pour vérifier si l'implémentation que vous avez proposée il y a un an (avant se suivre le cours INF3600) est correcte :

```
void EchangeCours (Putilisateur utilisateur, PCours cours1, cours2) {
        cours1->verrouille ();  // verrouille l'accès à l'objet cours1
        cours1->desinscrit (utilisateur);
        if (cours2->estPlein == false) {
                cours2->verrouille ();   // verrouille l'accès à l'objet cours2
                cours2->inscrit (utilisateur);
                cours2->deverrouille ();  //déverrouille l'accès à l'objet cours2
        }
        cours1->deverrouille ();  //déverrouille l'accès à l'objet cours2
```

```
}
```

Vérifiez si l'implémentation est correcte : Si elle est correcte, **expliquez pourquoi**, en montrant comment est géré le cas où deux étudiants (ou plus) veulent accéder en même temps au système. Si elle est incorrecte**, listez et expliquez les problèmes**, et **proposez** une solution qui fonctionne.

5.21 On vous sollicite pour implémenter en utilisant les sémaphores un autre mécanisme de synchronisation appelé « Barrières ». Ce mécanisme est très utile pour synchroniser un groupe de processus composés chacun de plusieurs phases qui fonctionnent selon la règle suivante : Aucun processus ne peut entamer sa phase suivante tant que les autres n'ont pas fini leurs phases courantes. Pour réaliser une telle synchronisation, une barrière est placée à la fin de chaque phase. Lorsqu'un processus atteint une barrière, il est bloqué jusqu'à ce que tous les autres processus atteignent la barrière. Les processus peuvent alors exécuter leurs phases suivantes.

Par exemple, les trois processus A, B et C suivants consistent en deux phases « acquisition » et « traitement » de paramètres. Avec la barrière X, il ne peut y avoir un processus à l'étape acquisition de paramètres et un autre à l'étape de traitement. Cette barrière permet aux trois processus de se synchroniser avant d'entamer une autre phase.

```
Barrier_t     X (3) ;     // 3 est le nombre de processus utilisant la barrière X.
```

```
Processus  A
{   while (1)
    {      AcquisParam(1) ;
           X.Barriere( ) ;
           TrtParam(1) ;
           X.Barriere( ) ;
     }
}
```

```
Processus  B
{   while (1)
    {      AcquisParam(2) ;
           X.Barriere( ) ;
           TrtParam(2) ;
           X.Barriere( ) ;
     }
}
```

```
Processus  C
{   while (1)
    {      AcquisParam(3) ;
           X.Barriere( ) ;
           TrtParam(3) ;
           X.Barriere( ) ;
    }
}
```

Donnez une implémentation au moyen de sémaphores de ce mécanisme de synchronisation (sous forme de pseudo-code). Pour répondre à cette question, définissez une classe d'objet Barrier_t en précisant ses attributs et ses méthodes.

On veut maintenant utiliser les barrières pour synchronisez une chaîne de production de stylos à bille. Chaque stylo est formé d'un corps, d'une cartouche, d'un bouchon arrière et d'un capuchon. Les opérations à effectuer pour fabriquer un stylo sont les suivantes :

- remplissage de la cartouche avec de l'encre (opération RC( )),
- assemblage du bouchon arrière et du corps (opération BO( )),
- assemblage de la cartouche avec le corps et le capuchon (opération AS( )),
- emballage (opération EM( )).

Chaque opération est effectuée par une machine spécialisée commandée par un processus (mRC, mBO, mAS, mEM). Les stocks de pièces détachées et d'encre sont supposés disponibles quand la machine est disponible. Les opérations RC et BO se font en parallèle. Avant d'entamer l'opération AS, la machine mAS doit prélever les éléments sur les machines mRC et mBO (opération GP()). Le produit assemblé est ensuite prélevé par mEM pour l'emballage.

Donnez le pseudo-code de chaque processus. Indiquez clairement les barrières utilisées, leurs valeurs ainsi que leurs rôles.

```
Barrier_t       /*0*/

mRC( )

{  while (1)
   {
      /*1*/
      RC( ) ;
     /*2*/
   }
}

mBO( )
{   while (1)
   {
      /*3*/
       BO( ) ;
       /*4*/
    }
}
```

```
mAS( )
{   while (1)
  {
     /*5*/
     GP( ) ;
     /*6*/
      AS( ) ;
     /*7*/
   }
}

mEM()

{   while (1)
  {
     /*8*/
     GP( ) ;
     /*9*/
      EM( ) ;
      /*10*/
   }
}
```

Une mauvaise utilisation des barrières peut-elle mener vers des interblocages ? Justifiez ?

5.22 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:

- Semaphore mutex1 = 1 ;
- Semaphore mutex2 = 1 ;
- Code du 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) ;

Cette proposition est-elle correcte ? Sinon, indiquer parmi les 4 conditions requises pour réaliser une exclusion mutuelle correcte, celles qui ne sont pas satisfaites ? Proposer une solution correcte.

On veut effectuer en parallèle le produit de deux matrices A et B d'ordre n (nxn). Pour se faire, on crée m (m<n) processus légers (threads). Chaque processus léger se charge de calculer quelques lignes de la matrice résultat R :

Pour j = 0 à n-1 $R[i,j] = \sum_{k=0,n-1} A[i,k]*B[k,j]$;

Donner sous forme de commentaires (en utilisant les sémaphores et les opérations P et V), le code des processus légers : CalculLignes ( ). Préciser les sémaphores utilisés et les variables partagées.

5.23 Deux villes A et B sont reliés par une seule voie de chemin de fer. Les trains peuvent circuler dans le même sens de A vers B ou de B vers A. Mais, ils ne peuvent pas circuler dans les sens opposés. On considère deux classes de processus : les trains allant de A vers B (Train AversB) et les trains allant de B vers A (Train BversA). Ces processus se décrivent comme suit :

Train AversB :

- Demande d'accès à la voie par A ;
- Circulation sur la voie de A vers B; Sortie de la voie par B;

Train BversA :

- Demande d'accès à la voie par B ;
- Circulation sur la voie de B vers A;
- Sortie de la voie par A;

Parmi les modèles étudiés en classe (producteur/consommateur, lecteur/rédacteur, les philosophes), ce problème correspond à quel modèle ?

Ecrire sous forme de commentaires en utilisant les sémaphores, les opérations P et V, les codes de demandes d'accès et de sorties, de façon à ce que les processus respectent les règles de circulation sur la voie unique.

5.24 Considérons le problème producteur/consommateur, vu en classe. Adaptez la solution suivante :

Au cas de n producteurs, n consommateurs et un seul tampon de taille (Max, il peut contenir au plus Max messages). Les producteurs produisent des messages et les déposent dans le tampon. Chaque message déposé dans le tampon est récupéré (consommé) par un seul consommateur.

Au cas d'un seul producteur, n consommateurs et n tampons de même taille (Max). Chaque message produit par le producteur est déposé dans tous les tampons en commençant par le premier. Le consommateur i récupère (consomme) les messages déposés dans le tampon i.

```
Semaphore Mutex =1, Vide=Max, Plein=0 ;
Message tampon [Max] ;

Producteur ( ) {
        int ip =0 ;
        Message m ;
        Repeter
        {
                m = creermessage() ;
                P(Vide) ;
                P(Mutex) ;
                Tampon[ip]=m;
                V(Mutex) ;
                ip++ ;
                V(Plein) ;
        }tant que vrai ;
}

Consommateur( ){
        int ic =0 ;
        Message m ;
        Repeter
        {
                P(Plein) ;
                P(Mutex) ;
                m = Tampon[ic];
                V(Mutex) ;
                ic++ ;
                V(Vide) ;
        }tant que vrai ;
}
```