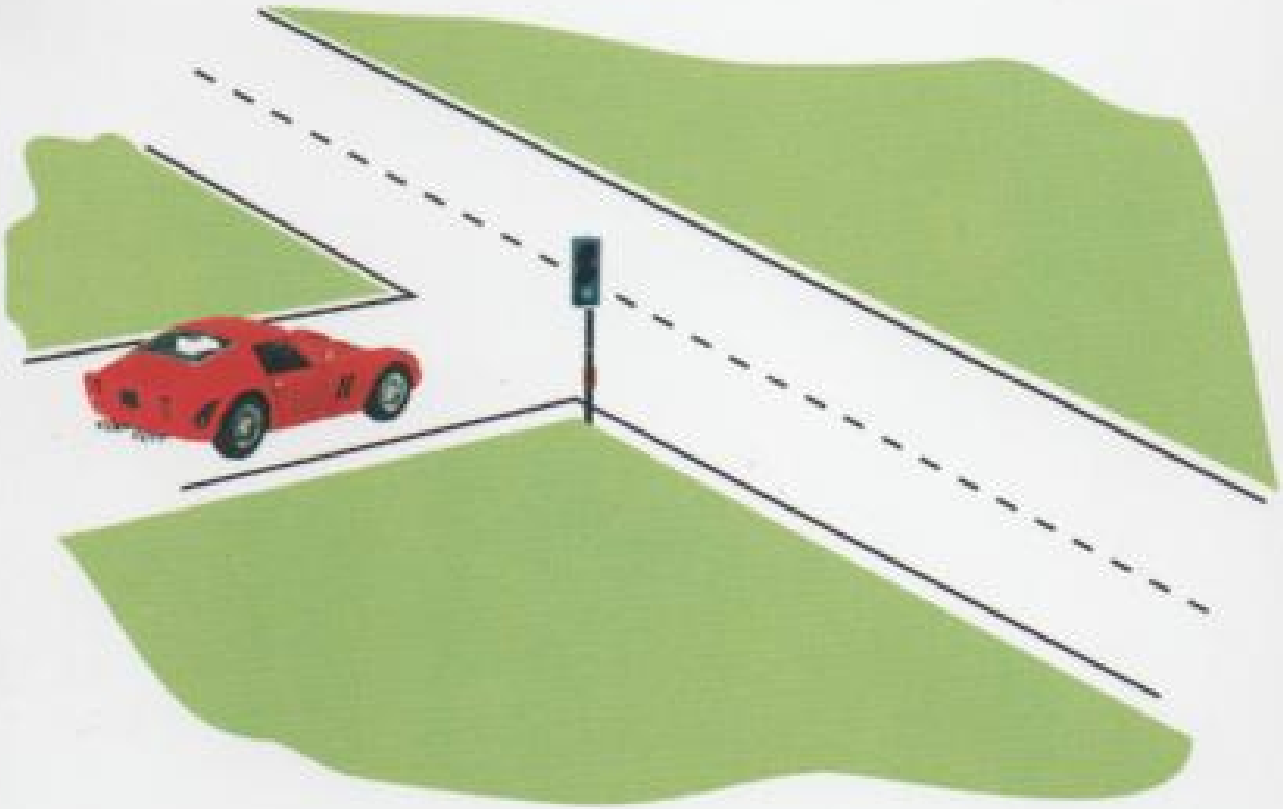
Synchronisation par sémaphores

Mohammed Saïd HABET



Résumés de cours • Exercices corrigés • Exercices supplémentaires

l'Abeille

SYNCHRONISATION PAR SEMAPHORES

rappels de cours , exercices

Par : HABET Mohammed-Said

Editions l'Abeille, 2006

Sommaire

Leçon Une :

Généralités

1. Définitions :

1.1 Processus

Un processus représente l'exécution d'un programme séquentiel : c'est une entité dynamique associée à la suite des actions réalisées par un programme séquentiel (qui ,lui, est une entité statique associée à la suite d'instructions qui le composent).
Cette distinction entre un programme et son exécution vient du fait que pour un système d'exploitation multiprogrammé le processeur ne peut travailler que pour un utilisateur/ programme à la fois. C'est dire que le système d'exploitation doit assurer l'allocation du processeur ; mais aussi d'autres ressources de la machine , aux différents programmes.
Les différents processus ne sont par tout à fait indépendants : tout en s'exécutant en parallèle ils doivent coopérer, se synchroniser . . . etc . Et cela est un rôle du système d'exploitation.

Remarque : *notion de processus du point de vue langages de programmation*

La notion de processus présentée précédemment est une notion relative aux systèmes d'exploitation. Il existe également la notion de processus du point de vue langages de programmation : dans quelques langages on peut décrire , syntaxiquement , l'activité parallèle. Pour cela il existe des structures de contrôle telle que : `ParBegin` (`débutpar`) / `ParEnd` (`finpar`). A titre d'exemple : on a ci-dessous un programme parallèle composé de n processus séquentiels Pi (i=1,..,n):

```
Débutpar
  Processus P1
  Début
    .
    .
    .
  Fin
  .
  .
  .
  Processus Pn
  Début
    .
    .
    .
  Fin
Finpar
```

Cette écriture signifie que les n processus Pi (i=1,..,n) entament leur exécution simultanément. La fin d'exécution du bloc `Débutpar/Finpar` a lieu lorsque tous les Pi (i=1,..,n) sont terminés.

1.2 Relations entre processus

Lorsque des processus s'exécutent , ils interagissent soit en coopérant (pour partager des informations ou accélérer un calcul par exemple) , soit en concourant pour acquérir des ressources quand celles-ci sont en quantité limitée.
Ainsi les relations entre processus peuvent être multiples, on en énumère quelques unes dans ce qui suit :

a) parallélisme :
c'est la simultanéité de l'exécution de plusieurs processus. Ce parallélisme peut être réel : dans le cas où les différents processus sont exécutés chacun sur un processeur différent. Mais dans le cas où les processus sont exécutés sur un même processeur (en temps partagé par exemple) on parle de pseudo-parallélisme (c'est-à-dire un parallélisme simulé).

b) communication :
Dans beaucoup d'applications , des processus peuvent avoir à communiquer entre eux , pour échanger des informations par exemple . Cette communication peut se faire soit par l'intermédiaire de variables ou objets partagés (mémoire commune) , soit par le biais de messages : envoi et réception de messages (canaux de communication).

c) synchronisation :
La synchronisation consiste à cadencer (contrôler) l'évolution des processus , et par suite l'occurrence des événements , en fonction de l'histoire passée de l'ensemble de ces processus.
Exemple : *synchronisation pour l'accès à une ressource critique*
Une ressource qui ne peut être utilisée que par un processus ,au plus, à la fois est appelée « ressource critique » . Lorsque plusieurs processus doivent utiliser une ressource critique (R.C) on doit les synchroniser en coordonnant les différentes demandes de chacun des processus de telle sorte qu'à tout moment un seul processus à la fois utilise la R.C. Cela est réalisé en faisant attendre les processus qui demandent à utiliser la R.C s'il y a déjà un processus qui l'utilise. Lorsque le processus qui utilise la R.C finit son utilisation , on alloue la R.C à un autre processus qui l'a demandée (et qui est en attente).
Remarque : On dit que les processus qui utilisent une R.C tel qu'énoncé précédement sont en « exclusion mutuelle » pour l'accés à la ressource en question. Et la partie du processus qui s'exécute en exclusion mutuelle est appelée « section critique ».
A titre d'exemple une R.C peut être une variable globale (variable commune à plusieurs processus : la mise à jour de cette variable doit se faire en exclusion mutuelle pour garantir la cohérence des données). Une R.C peut aussi être un fichier , un enregistrement de fichier , un périphérique . . . etc.

1.3 Propriétés

Lorsqu'on s'intéresse au bon fonctionnement des systèmes parallèles , il convient de déterminer les propriétés nécessaires qui doivent être vérifiées.

a) absence d'interblocage :
c'est une propriété fondamentale qui exprime qu'un ensemble de processus ne doit pas atteindre ,lors de son exécution, un état où chaque processus reste bloqué en attente d'une condition logique qui ne deviendra jamais vraie.
Exemple d'une situation d'interblocage : On considère deux processus P_1 et P_2 utilisant deux ressources critiques R_1 et R_2 comme suit :

```
Processus P1                    Processus P2
Début                           Début
  acquérir R1                     acquérir R2
  acquérir R2                     acquérir R1
  utiliser R1 et R2               utiliser R2 et R1
Fin                             Fin
```

La séquence d'exécution : acquérir R_1 (*processus* P_1) , acquérir R_2 (*processus* P_2) ; conduit à un interblocage : le processus P_1 en faisant acquérir R_2 sera bloqué (car R_2 est déjà détenue par P_2) , le processus P_2 en faisant acquérir R_1 sera bloqué (car R_1 est déjà détenue par P_1).
Remarque : L'absence d'interblocage est une propriété de sûreté (une propriété de sûreté est une propriété qui doit être vérifiée durant toute la durée du fonctionnement des processus du programme/système parallèle).

b) équité : absence de famine
Cette propriété est liée aux demandes d'accès à des ressources et/ou exécution de certaines sections de programmes : toute demande d'allocation de ressource doit finir par être satisfaite.
Remarque : L'équité est une propriété de vivacité (une propriété de vivacité spécifie des états qui devront être atteints par le système).

c) priorité :
Dans quelques cas on peut avoir à privilégier certains processus par rapport à d'autres : on dit que les processus qu'on a privilégié sont plus prioritaires que les autres.
A titre d'exemple : on peut classer les demandeurs d'une ressource critique en deux catégories où l'on donnera la priorité à une seule catégorie pour l'accès à la ressource (les processus de la catégorie la moins prioritaire n'auront l'accès à la ressource que lorsqu'il n'y a aucun processus de la catégorie le plus prioritaire qui attende).
Autre exemple : les processus urgents du système. Il se trouve que pendant l'exécution de progammes utilisateurs par le processeur , ceux-ci peuvent être interrompus momentanément pour que le processeur exécute des processus ou des tâches urgents et qui sont donc plus prioritaires.

1.4 Sémaphores

Un sémaphore est un outil de synchronisation . Il est matérialisé par une variable entière à laquelle est ajoutée une file d'attente de processus.
Si s est un sémaphore : il possède deux champs : val (valeur : entier) et un champ file (structure de file de BCP (Bloc de Contrôle de Processus)) :

Il n'est possible d'accéder à un sémaphore que par deux opérations notées P et V.
Les primitives P et V sont définies comme suit :

```
P(s) :  Début
          s.val := s.val - 1
          Si  s.val < 0  alors
            état(p) := bloqué        /* p est le processus */
            insérer p dans s.file    /* qui exécute P(s) */
          finsi
        Fin
V(s) :  Début
          s.val := s.val + 1
          Si  s.val ≤ 0  alors
            sortir un processus q de s.file
            état(q) := prêt
          finsi
        Fin
```

Remarque 1 :
Les primitives P et V sont exécutées de manière indivisible.

Remarque 2 :
La valeur initiale du champ val d'un sémaphore doit être un nombre non négatif.

Exemple : *Utilisation des sémaphores pour réaliser l'exclusion mutuelle*
Une ressource critique R est utilisée par un nombre quelconque de processus.

```
Processus Ayant_à_utiliser_R
début
  <section non critique>
  <utilisation de R>
  <section restante>
 fin
```

L'utilisation de R doit se faire en exclusion mutuelle pour tous les processus concernés.

Pour réaliser l'accès en exclusion mutuelle pour l'accès à R , on utilisera un sémaphore initialisé à 1 :

```
var s : sémaphore init 1;
     /* s est une variable globale de type sémaphore */
     /* dont la valeur est initialisée à 1 */
Processus Ayant_à_utiliser_R
début
  <section non critique>
  P(s)
  <utilisation de R>  /* section critique */
  V(s)
  <section restante>
 fin
```

2. Exercices :

Exercice 1 :

Soit s un sémaphore , on note :

`i(s)` : valeur initiale du sémaphore `s` (par définition `i(s)` ≥ `0`)

`np(s)` : nombre d'opérations `P(s)` exécutées sur `s`

`nv(s)` : " " `V(s)` " " `s`

`nf(s)` : nombre de processus qui ont franchi la primitive `P(s)`
(on a donc `nf(s)` ≤ `np(s)`)

Vérifier que

$$nf(s) = \min(np(s), i(s)+nv(s)) \quad (1)$$

Solution :

Initialement, `np(s) = nv(s) = 0` et `nf(s) = 0 = min(0,i(s))`

Lorsqu'on agit la première fois sur le sémaphore , par une primitive , on a :

- si P est éxécutée : `np(s)` devient égal à `1` ; (`nv(s)` reste = `0`)

  si `i(s)` ≥ `1` alors (* exécution de `P` sans blocage *)

  `nf(s) = 1 = min(1,i(s)) = min(np(s),i(s)+nv(s))`

  sinon (* `i(s) = 0` : exécution de `P` avec blocage *)

  `nf(s) = 0 = min(1,0) = min(np(s),i(s)+nv(s))`

  (finsi)

- si V est éxécutée : `nv(s)` devient égal à `1` ; mais `np(s) = 0` et ainsi
  `nf(s) = 0 = min(0,i(s)+nv(s)) = min(np(s),i(s)+nv(s))`

Remarque: `s.val = i(s)+nv(s)-np(s)`

(`s.val < 0`) ⇔ (∃ des procs bloqués en `s.file`)

Supposons maintenant que la formule (1) reste valide après l'exécution d'un certain nombre d'opérations sur s ; et voyons l'effet de l'exécution d'une primitive `P` ou `V` sur (1).

Effet de `P` :

Avant			Après	
(1)	s.val	effet sur np(s)	s.val	(1)
nf(s)=np(s)<i(s)+nv(s)	>0	np := np+1	≥0	nf(s)=np(s)≤i(s)+nv(s)
nf(s)=i(s)+nv(s)≤np(s)	≤0	np := np+1	<0	nf(s)=i(s)+nv(s)<np(s)

L'exécution de `P` laisse invariante la formule (1)

Effet de `V` :

Avant			Après	
(1)	s.val	effet sur nv(s)	s.val	(1)
nf(s)=np(s)≤i(s)+nv(s)	≥0	nv := nv+1	>0	nf(s)=np(s)<i(s)+nv(s)
nf(s)=i(s)+nv(s)<np(s)	<0	nv := nv+1	≤0	nf(s)=i(s)+nv(s)≤np(s)

L'exécution de `V` laisse invariante la formule (1)

conclusion: (1) est toujours valide

Exercice 2 :

On considère le programme parallèle suivant :

```
var n : entier init 0 ;  /* n entier initialisé à 0 */
Début
  Débutpar
    Processus P1
    Début
      n := n + 1
    Fin
    Processus P2
    Début
      n := n - 1
    Fin
  Finpar
Fin
```

1) Quel le résultat de son exécution dans les cas suivants :

a) on considère les opérations sur la variable n sont indivisibles

b) les opérations sur n ne sont pas indivisibles : elles peuvent être décomposées.

2) Dans le cas b) comment faire pour que la mise-à-jour de `n` se fasse en exclusion mutuelle.

Solution :

1)

a) Dans le cas de l'indivisibilité de l'accès à n , le résultat obtenu (après la fin de l'exécution des deux processus P1 et P2) est n = 0.

b) Dans le cas où l'on peut décomposer les opérations sur n le résultat n'est pas unique : suivant la séquence d'exécution considérée on peut obtenir n = 0 ou n = 1 ou n = -1.

En effet :

```
Processus P1
Début
  (1)  LOAD R1,n   /* charger le registre R1 par n */
  (2)  ADD R1,1    /* additionner 1 à R1 */
  (3)  STORE n,R1  /* ranger (R1) dans la variable n */
Fin
Processus P2
Début
  (1')  LOAD R2,n   /* charger le registre R2 par n */
  (2')  SUB R2,1    /* soustraire 1 de R2 */
  (3')  STORE n,R2  /* ranger (R2) dans la variable n */
Fin
```

Avec la séquence *(1), (2), (3), (1'), (2'), (3')* on obtient n = 0
avec la séquence *(1), (1'), (2'), (3'), (2), (3)* on obtient n = 1
avec la séquence *(1), (1'), (2), (3), (2'), (3')* on obtient n = -1

2) Pour que la mise-à-jour de `n` se fasse en exclusion mutuelle , il existe de nombreux moyens. Un de ces moyens est l'utilisation de sémaphore :

```
var n : entier init 0 ;
    s : sémaphore init 1 ;
Début
  Débutpar
    Processus P1
    Début
      P(s)
      LOAD R1,n
      ADD R1,1
      STORE n,R1
      V(s)
    Fin
```

```
    Processus P2
    Début
      P(s)
      LOAD R2,n
      SUB R2,1
      STORE n,R2
      V(s)
    Fin
  Finpar
Fin
```

De cette manière le seul résultat qu'on peut obtenir est n = 0 ; car on exclue les séquences d'exécutions pouvant aboutir à n = 1 ou n = -1 : dès qu'un processus fait `P(s)` , l'autre processus ne pourra plus accéder à n jusqu'à ce que celui qui a fait `P(s)` la première fois libère l'accès à n en faisant `V(s)`.

Remarque : Dans la pratique, les primitives `P` et `V` sont implémentées comme des appels système.

2. Exercices supplémentaires :

Exercice 1 :

L'algorithme d'un ensemble de processus est exprimé ainsi :

```
var S : sémaphore init N
Processus Pi (i≥0)
Début
  P(S)
  Action A
  V(S)
Fin
```

A l'aide de la formule `(1)` de l'exercice 1 précédent, montrer que l'on ne peut avoir , à tout moment, qu'au plus N processus en train d'exécuter leur `Action A`.

Exercice 2 :

Pour calculer la somme des éléments d'un tableau on utilise deux processus. L'un parcoure les éléments d'indice impair et l'autre les éléments d'indice pair comme suit :

```
Const M = … /* nombre d'éléments du tableau */
var T : Tableau[1..M] de entier ;
    Somme : entier init 0 ;
Processus P1
var i : entier  /* i est une variable locale à P1 */
Début
  i := 1
  Tant que (i ≤ M) faire
    Somme := Somme + T[i]
    i := i + 2
  FinTantque
Fin
```

```
Processus P2
var j : entier  /* j est une variable locale à P2 */
Début
  j := 2
  Tant que (j ≤ M) faire
    Somme := Somme + T[j]
    j := j + 2
  FinTantque
Fin
Début /* bloc principal */
  Débutpar
    P1
    P2
  Finpar
Fin
```

1) Montrer , à l'aide d'un exemple d'exécution , qu'à cause des mises à jour (m-à-j) simultanées de la variable `Somme` ; on peut obtenir un résultat erroné. (on considérera que les instructions d'ajout à la variable `Somme` ne sont pas indivisibles).
2) Résoudre le problème en utilisant un sémaphore d'exclusion mutuelle pour l'accès à la variable `Somme`.
3) Ecrire une autre solution où l'on utilisera deux autres variables globales `Somme1` et `Somme2` l'une pour calculer la somme des éléments d'indice impair (qui sera uniquement accédée en m-à-j par `P1`) ; l'autre pour calculer la somme des éléments d'indice pair (qui sera uniquement accédée en m-à-j par `P2`). Le résultat `Somme` sera calculé comme `Somme1 + Somme2` dans le bloc principal à l'issue de l'exécution de P1 et P2 :

```
Début
  Débutpar
    P1
    P2
  Finpar
  Somme := Somme1 + Somme2
Fin
```

(Dans ce cas il n'y a pas de synchronisation (pour l'accès aux variables) à réaliser car il n'y a pas de concurrence d'accès en m-à-j aux différentes variables globales).

Leçon Deux :

Classiques et autres

1. Présentation :

Dans les paragraphes suivants, il s'agira de présenter quelques problèmes classiques de synchronisation, ainsi que d'autres plus ou moins classiques.

1.1 Exclusion mutuelle

Considérons n processus $P_1, P_2, \ldots\ldots Pn$ s'exécutant en parallèle et utilisant une ressource critique R comme suit :

```
Processus Pi    /* i=1,n */
début
  <section non critique>
  <utilisation de R>
  <section restante>
fin
```

On doit garantir que l'utilisation de R se fasse en exclusion mutuelle : la section <utilisation de R> doit être une section critique (S.C).

Les conditions de l'exclusion mutuelle sont les suivantes :

a) à tout instant un processus au plus peut se trouver en section critique

b) un processus qui exécute du code hors section critique ne doit pas empêcher d'autres processus d'accéder en section critique

c) si plusieurs processus sont bloqués en attente d'entrer en section critique alors qu'aucun autre ne s'y trouve , l'un d'eux doit y accéder au bout d'un temps fini.

d) toute demande d'entrée en section critique doit être satisfaite.

Pour ce faire on peut utiliser un sémaphore `mutex` et exprimer les algorithmes des processus `Pi` ainsi :

```
var mutex : sémaphore init 1;
     /* mutex est une variable globale de type sémaphore */
     /* dont la valeur est initialisée à 1 */
Processus Pi    /* i=1,n */
début
  <section non critique>
  P(mutex)
  <utilisation de R>  /* section critique */
  V(mutex)
  <section restante>
fin
```

On peut aisément vérifier que les conditions de l'exclusion mutuelle sont respectées.

Soit `nsc = nf(mutex)-nv(mutex)`

condition a) :

Le nombre de processus en S.C est égal au nombre de processus qui ont franchi `P(mutex)` et qui n'ont pas encore exécuté `V(mutex)` :

on sait déjà que `nf(mutex) = min(np(mutex),1+nv(mutex))`

si `nf(mutex)=np(mutex)` :

`nf(mutex) ≤ 1+nv(mutex)` , d'où `nf(mutex)-nv(mutex) ≤ 1` ;

si `nf(mutex)=1+nv(mutex)` :

`nf(mutex) - nv(mutex) = 1` ;

dans les deux cas : `nf(mutex)-nv(mutex) ≤ 1` .

condition b) :
tels qu'exprimés les processus `Pi` n'agissent pas sur le sémaphore `mutex` en dehors du protocole d'entrée ou de sortie de la S.C.

condition c) :
il suffit de vérifier que s'il y a des processus qui attendent alors il y a un processus en S.C :
si au moins un processus attend alors `nf(mutex)<np(mutex)` et ainsi
`nsc = nf(mutex)-nv(mutex) = 1` processus en S.C

condition d) :
cette propriété est satisfaite si la file d'attente du sémaphore est gérée en F.I.F.O (First In First Out) ;
- si un processus bloqué est en tête de file ; dés que le processus qui est en S.C en sort en exécutant V , le processus bloqué en tête de file est débloqué pour accéder à la S.C
- sinon : un processus quelconque dans la file finira par arriver en tête de file (lorsque les processus bloqués avant lui auront accédé en S.C) ; et il sera réveillé à son tour dés que V(mutex) est exécutée.

1.2 Précédence des tâches

Dans quelques applications , notamment les applications temps réel , certaines tâches (processus) doivent avoir une relation de précédence : une tâche ne peut entamer son exécution que lorsque d'autres tâches ont terminé leurs exécution.

Exemple :

Considérons un système comportant deux tâches T_1 et T_2 et où T_1 doit précéder T_2

La synchronisation peut être réalisée simplement comme suit :

```
var s : sémaphore init 0;
     /* s est une variable globale de type sémaphore */
     /* dont la valeur est initialisée à 0 */
Processus T1              Processus T2
début                     début
  <Exécution>               P(s)
  V(s)                      <Exécution>
fin                       fin
```

1.3 Lecteurs / Rédacteurs

On considère un objet (un fichier par exemple) qui n'est accessible que par deux catégories d'opérations : les lectures et les écritures. Plusieurs lectures (consultations) peuvent avoir lieu simultanément ; par contre les écritures (mises à jour) doivent se faire en exclusion mutuelle.
On appellera « lecteur » un processus faisant des lectures et « rédacteur » un processus faisant des écritures.
Il s'agit donc de réaliser la synchronisation entre lecteurs et rédacteurs en respectant les contraintes suivantes :
- exclusion mutuelle entre lecteurs et rédacteurs : si un lecteur demande à lire et qu'il y a une écriture en cours , la demande est mise en attente. De même que si un rédacteur demande à écrire et qu'il y a au moins une lecture en cours , la demande est mise en attente.
- exclusion mutuelle entre rédacteurs : si un rédacteur demande à écrire et qu'il y a une écriture en cours , la demande est mise en attente.

L'attente d'un processus lecteur / rédacteur peut être assimilée au blocage du processus dans une file de sémaphore.
Pour satisfaire les contraintes ci-dessus , on peut procéder comme suit :

```
var s,mutex : sémaphore init 1,1 ;
    nl : entier init 0 ;
```

```
Processus Lecteur
Début
   .
   .
   .
   P(mutex)
   nl := nl + 1
   si nl=1 alors P(s) finsi
   V(mutex)
   Lecture
   P(mutex)
   nl := nl - 1
   si nl=0 alors V(s) finsi
   V(mutex)
   .
   .
   .
Fin
```

```
Processus Rédacteur
Début
   .
   .
   .
   P(s)
   Ecriture
   V(s)
   .
   .
   .
Fin
```

1.4 Producteurs / Consommateurs

On considère deux classes de processus :

- les producteurs : produisent des informations ,
- les consommateurs : consomment les informations produites par les producteurs.

Pour que les producteurs et consommateurs puissent s'exécuter en parallèle , ils partagent un tampon dans lequel seront stockées les informations (messages) produites et en attente d'être consommées.

La synchronisation peut s'exprimer comme suit :

```
var S1,S2,mutex : sémaphore init N,0,1 ; /* N : taille du tampon */
Processus Producteur
Début
  Cycle /* répeter  indéfiniment */
   <Produire un message>
   P(S1)
   P(mutex)
   <Déposer le message>
   v(mutex)
   V(S2)
  FinCycle
Fin
```

```
Processus Consommateur
Début
  Cycle
   P(S2)
   P(mutex)
   <Prelever un message>
   V(mutex)
   V(S1)
   <Consommer le message>
  FinCycle
Fin
```

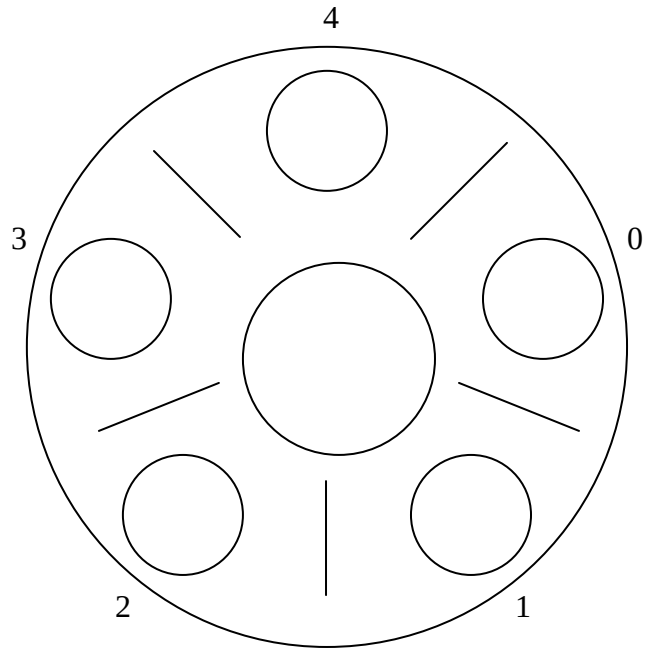
1.5 Philosophes

Cinq philosophes sont assis sur des chaises autour d'une table ronde pour philosopher et manger des spaghettis.Sur la table sont disposées cinq assiettes , cinq fourchettes et un plat de spaghettis qui est toujours plein.

Chaque philosophe passe son temps à penser puis manger. Pour manger il doit utiliser les deux fourchettes situées de par et d'autres de son assiette. Après avoir mangé, le philosophe repose les deux fourchettes sur la table et se remet à penser. Et ainsi de suite.

C'est-à-dire que le schéma d'un philosophe est :

```
Philosophe i (i=0,4)
Début
  Cycle
     Penser
     Manger   /* nécessite deux fourchettes */
  FinCycle
Fin
```



La solution à écrire doit éviter l'interblocage.

```
type e_ph = (pensif,attendant,mangeant) ;
var etat : tableau[0..4] de e_ph
                  init (pensif, pensif, pensif, pensif, pensif) ;
    S : tableau[0..4] de sémaphore init (0,0,0,0,0) ;
    mutex : sémaphore init 1 ;
Processus Philosophe i (i=0..4)
Début
  Cycle
    <Penser>
    P(mutex)
    Si (etat[Droite(i)]=mangeant) ou (etat[Gauche(i)]=mangeant)
     Alors
       etat[i] := attendant
       V(mutex)
       P(S[i])
     Sinon
       etat[i] := mangeant
       V(mutex)
    Finsi
    <Manger>
    P(mutex)
    etat[i] := pensif
    si (etat[Droite(i)]=attendant) et
           (etat[Droite(Droite(i))]<>mangeant) alors
       etat[Droite(i)] := mangeant
       V(S[Droite(i)])
    finsi
    Si (etat[Gauche(i)]=attendant) et
           (etat[Gauche(Gauche(i))]<>mangeant) alors
       etat[Gauche(i)] := mangeant
       V(S[Gauche(i)])
    Finsi
    V(mutex)
  FinCycle
Fin
```

```
fonction Droite(i : 0..4) : 0..4 ;
var d : entier
Début
  d := i - 1
  si d<0 alors d := 4 finsi
  Droite := d
Fin
```

```
fonction Gauche(i : 0..4) : 0..4 ;
Début
  Gauche := (i+1) mod 5
Fin
```

Remarque : la solution présentée présente le risque de famine. Il serait intéressant de trouver une solution équitable.

2. Exercices :

Exercice 1 :

Soit N processus P_i (i=1..N) et un processus P_s. Les processus P_i (i=1..N) remplissent une zone tampon pouvant contenir M messages, un seul à la fois étant autorisé à déposer son message.
Le processus P_i qui remplit la dernière case du tampon active le processus P_s qui fait alors l'impression de tous les messages déposés dans le tampon. Durant cette impression, les processus P_i (i=1..N) ne sont pas autorisés à accéder au tampon.
Ecrire les algorithmes des processus P_i (i=1..N) et P_s.

Solution :

Si on note : nm = nombre de messages contenus dans le tampon , le schéma des processus considérés peut s'exprimer comme suit :

```
Processus Pi ( i = 1..N )
Début
 <Fabriquer un message>
 <si le tampon est occupé
  alors attendre sinon
  bloquer l'accès au tampon>
 <Déposer le message>
 nm := nm + 1
 <si nm=M alors activer Ps
  sinon liberer l'accès au
  tampon>
Fin
```

```
Processus Ps
Début
 Cycle
  <attendre jusqu'à être
   reveillé par un des
   processus Pi>
  <Imprimer tous les messages>
  nm := 0
  <liberer l'accès au tampon>
 FinCycle
Fin
```

On peut assimiler l'attente d'un processus au blocage de celui-ci dans une file de sémaphore.
Soit Spriv un sémaphore privé au processus Ps qui y se bloquera en attente d'être réveillé ;
et mutex un sémaphore d'exclusion mutuelle pour l'accès au tampon.
On peut écrire les algorithmes des processus Pi et Ps comme suit :

```
var Spriv,mutex : sémaphore init 0,1 ;
    nm : entier init 0 ;
```

```
Processus Pi ( i = 1..N )
Début
 <Fabriquer un message>
 P(mutex)
 <Déposer le message>
 nm := nm + 1
 si nm=M alors V(Spriv)
         sinon V(mutex)
 finsi
Fin
```

```
Processus Ps
Début
 Cycle
  P(Spriv)
  <Imprimer tous les messages>
  nm := 0
  V(mutex)
 FinCycle
Fin
```

Exercice 2 :

On considère trois processus P1, P2 et P3. Le processus P1 produit des messages qu'il dépose dans un tampon T1. P2 prélève les messages contenus dans T1, les traite puis dépose les résultats dans un tampon T2. P3 prélève les messages contenus dans T2 et les consomme.

1) Ecrire les algorithmes de P1, P2 et P3 de façon à garantir le non-interblocage.

2) On considère maintenant que les P_i (i=1..3) travaillent sur le même tampon T (au lieu de T1 et T2). Réétudier la question 1).

Solution :

1)
```
Processus P1  /* producteur */
Début
α1 : <Produire un message>
     <Dépôt du message dans T1>
     Aller à α1
Fin
```

```
Processus P2  /* consommateur & */
Début            /* producteur */
α2 : <Prélever un message de T1>
     <Traiter le message>
     <Dépôt du résultat dans T2>
     Aller à α2
 Fin
```

```
Processus P3  /* consommateur */
Début
α3 : <Prélever un message de T2>
     <Traitement du message>
     Aller à α3
Fin
```

La synchronisation peut s'exprimer comme suit :

```
var S1,S2,S3,S4,mutex1,mutex2 : sémaphore init n1,0,n2,0,1,1 ;
                                    /* ni=taille de Ti, i=1,2 */
```

```
Processus P1
Début
α1 : <Produire un message>
     P(S1)
     P(mutex1)
     <Dépôt du message dans T1>
     V(mutex1)
     V(S2)
     Aller à α1
Fin
```

```
Processus P2
Début
α2 : P(S2)
     P(mutex1)
     <Prélever un message de T1>
     V(mutex1)
     V(S1)
     <Traiter le message>
     P(S3)
     P(mutex2)
     <Dépôt du résultat dans T2>
     V(mutex2)
     V(S4)
     Aller à α2
Fin
```

```
Processus P3
Début
α3 : P(S4)
     P(mutex2)
     <Prélever un message de T2>
     V(mutex2)
     V(S3)
     <Traitement du message>
     Aller à α3
Fin
```

2) Si les trois processus P1, P2 et P3 travaillent sur le même tampon T ; une situation d'interblocage peut se produire : si au début P1 remplit le tampon T ensuite P2 prélève un message de T pour le traiter. Avant que P2 ne termine le traitement , P1 dépose un autre message dans T (ce qui remplit T de nouveau). A ce moment aucun des trois processus ne peut plus progresser : P1 ne pourra plus déposer de nouveaux messages (T plein) ; P2 ne peut pas déposer le résultat (T plein) ; P3 ne peut pas prélever des messages de T (T ne contient aucun message destiné à P3).
   Une solution à ce problème serait de partager T en deux tampons T1 et T2 : T1 contiendra les messages destinés à P2 ; T2 contiendra les messages destinés à P3. Ce qui ramène à la solution de 1).

Exercice 3 :

On considère un ensemble de six tâches séquentielles {A, B, C, D, E ,F}.

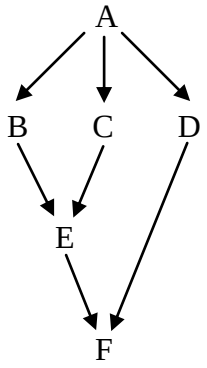
La tâche A doit précéder les tâches B, C, D. Les tâches B et C doivent précéder la tâche E.

Les tâches D et E doivent précéder la tâche F.

Réaliser la synchronisation de ces tâches en utilisant les sémaphores.

Solution :

On peut représenter le graphe de précédence des tâches {A,B,C,D,E,F.} comme suit :



La synchronisation des tâches en question peut s'exprimer comme suit :

```
var SA,SB,SC,SD,SE : sémaphore init 0,0,0,0,0 ;
```

```
Tâche A
Début
 Exécution
 V(SA) ;  V(SA) ;  V(SA)
Fin
```

```
Tâche B
Début
 P(SA)
 Exécution
 V(SB)
Fin
```

```
Tâche C
Début
 P(SA)
 Exécution
 V(SC)
Fin
```

```
Tâche D
Début
 P(SA)
 Exécution
 V(SD)
Fin
```

```
Tâche E
Début
 P(SB) ; P(SC)
 Exécution
 V(SE)
Fin
```

```
Tâche F
Début
 P(SE) ; P(SD)
 Exécution
Fin
```

Exercice 4 :

Soit P_0 et P_1 deux processus parallèles se partageant deux ressources R_1 et R_2. Les algorithmes de ces deux processus sont écrits comme suit :

```
var s1,s2 : sémaphore init 1,1 ;
Processus P0
Début
 a0: (1) P(s1)
     (2) utiliser R1
     (3) P(s2)
         utiliser R1 et R2
         V(s1)
         V(s2)
         Aller à a0
Fin
```

```
Processus P1
Début
 a1: (1') P(s2)
     (2') utiliser R2
     (3') P(s1)
          utiliser R1 et R2
          V(s2)
          V(s1)
          Aller à a1
Fin
```

1) à quelle situation anormale peut conduire l'exécution de ces deux processus ?

2) donner une solution à ce problème.

Solution :

1) Considérons la séquence d'exécution : (1) , (1') , (2) , (2') , (3) , (3')
après l'exécution de (1) , (1') $s_1.val$ devient égal à $s_2.val = 0$
lorsque P_0 exécutera (3) il se bloquera , et P_1 se bloquera lorsqu'il exécutera (3') : les deux processus sont bloqués sans aucun moyen d'être réveillés : interblocage.

2) Une solution à ce problème est de procéder comme suit :

```
var s1,s2 : sémaphore init 1,1 ;
Processus P0
Début
 a0: P(s1)
     utiliser R1
     V(s1)
     P(s1)
     P(s2)
     utiliser R1 et R2
     V(s2)
     V(s1)
     Aller à a0
Fin
```

```
Processus P1
Début
 a1: P(s2)
     utiliser R2
     V(s2)
     P(s1)
     P(s2)
     utiliser R1 et R2
     V(s2)
     V(s1)
     Aller à a1
Fin
```

Exercice 5 :

Un magasin peut accueillir un nombre limité de clients. Cette limite est représentée par le nombre N de chariots disponibles à l'entrée du magasin. Un client qui arrive attend s'il n'y a aucun chariot disponible. Lorsqu'un client acquiert un chariot il entre au magasin pour effectuer ses achats. Dès qu'il termine, il libère son chariot en sortant du magasin.
On peut assimiler les clients à des processus parallèles et les chariots à des ressources partagées.

1) Ecrire l'algorithme de chaque client.
2) On considère maintenant qu'il y a deux catégories de clients : les « abonnés » et les « non abonnés » . Il n'y a pas d'exclusion mutuelle entre abonnés et non abonnés , par contre les abonnés ont la priorité pour l'acquisition des chariots.
   Ecrire les algorithmes des processus « abonnés » et « non abonnés ».

Solution :

1) Il suffit d'utiliser un sémaphore initialisé à N

```
var S : sémaphore init N ;   /* S.val=N */
Processus Client
Début
  P(S)   /* prendre un chariot */
  Effectuer les achats
  V(S)   /* libérer le chariot */
Fin
```

2) Dans ce cas, on utilisera un sémaphore S_i (i=1,2) pour chaque classe de processus : un processus qui doit attendre sera mis dans S_1.file s'il est abonné et dans S_2.file sinon. Ainsi on pourra privilégier le réveil des processus bloqués dans la file de S_1 : lorsqu'un processus quelconque rend son chariot il vérifiera d'abord s'il y a au moins un processus abonné qui attend auquel cas il en réveillera un ; sinon il réveillera un processus non abonné éventuellement.
Pour cela, on utilisera également les entiers suivants :

uC : nombre d'utilisateurs ,en cours, de chariots = nombre de clients dans le magasin ;
na1 : nombre de processus abonnés qui attendent ;
na2 : nombre de processus non abonnés qui attendent.

Pour assurer l'accès en exclusion mutuelle à ces variables, il faudra utiliser un sémaphore d'exclusion mutuelle.
D'où les déclarations suivantes :

```
var S1,S2,mutex : sémaphore init 0,0,1 ;   /* S1.val=0; S2.val=0;
                                              mutex.val=1 */
    uC,na1,na2 : entier init 0,0,0 ;   /* uC=0; na1=0; na2=0 */
```

Les algorithmes des processus en question seront exprimés comme suit :

```
Processus Client_Abonné
Début
  P(mutex)
  si uC < N alors
    uC := uC + 1
    V(mutex)
  sinon
    na1 := n1 + 1
    V(mutex)
    P(S1)
  finsi
```

```
Processus Client_Non_Abonné
Début
  P(mutex)
  si uC < N alors
    uC := uC + 1
    V(mutex)
  sinon
    na2 := na2 + 1
    V(mutex)
    P(S2)
  finsi
```

```
   Effectuer les achats
   P(mutex)
   si na1 > 0 alors
     na1 := na1 - 1
     V(S1)
   sinon  /* na1=0 */
     si na2 > 0 alors
       na2 := na2 - 1
       V(S2)
     sinon  /* na1=na2=0 */
       uC := uC - 1
     finsi
   finsi
   V(mutex)
 Fin
```

```
   Effectuer les achats
   P(mutex)
   si na1>0 alors
     na1 := na1 - 1
     V(S1)
   sinon  /* na1=0 */
     si na2 > 0 alors
       na2 := na2 - 1
       V(S2)
     sinon  /* na1=na2=0 */
       uC := uC - 1
     finsi
   finsi
   V(mutex)
 Fin
```

Exercice 6 :

Une piscine peut accueillir N nageurs au plus. Ce nombre N est le nombre de paniers disponibles pour les habits des nageurs. A l'entrée comme à la sortie les nageurs entrent en compétition pour l'acquisition d'une cabine d'habillage/déshabillage, il y a C cabines (1 <= C << N).
Chaque nageur effectue les opérations :

```
Proc Nageur
Début
  <se déshabiller>
  <nager>
  <se rhabiller>
Fin
```

On peut assimiler ces nageurs à des processus concurrents; les cabines et les paniers étant des ressources partagées.

1) Ecrire l'algorithme des processus Nageur synchronisés par sémaphores.
2) On considère maintenant que les nageurs entrants sont prioritaires pour l'acquisition des cabines. Reécrire l'algorithme des processus Nageur.

Solution :

1)

```
var S1,S2 : sémaphore init N,C ;
Processus Nageur
Début
  P(S1)  /* prendre un panier */
  P(S2)  /* occuper une cabine */
  <se déshabiller>
  V(S2)  /* libérer la cabine */
  <nager>
  P(S2)  /* occuper une cabine */
  <se rhabiller>
  V(S2)  /* libérer la cabine */
  V(S1)  /* libérer le panier */
Fin
```

2) Une cabine libérée n'est allouée à un nageur sortant que s'il n'y a aucun nageur entrant qui attend. Par contre il n'y a pas d'exclusion mutuelle entre entrants et sortants.

```
var S1,Se,Ss,mutex : sémaphore init N,0,0,1 ;
    Ucb,ne,ns : entier init 0,0,0;
Processus Nageur
Début
  P(S1)
  Demander_cabine(ne,Se)
  <se déshabiller>
  Liberer_Cabine
  <nager>
  Demander_cabine(ns,Ss)
  <se rhabiller>
  Liberer_Cabine
  V(S1)
Fin
```

La procédure `Demander_Cabine` est écrite comme suit :

```
Procedure Demander_Cabine(var n : entier ; var s : sémaphore)
Début
  P(mutex)
  si Ucb<C alors
    Ucb := Ucb + 1
    V(mutex)
  sinon
    n := n + 1
    V(mutex)
    P(s)
  finsi
Fin
```

La procédure `Liberer_Cabine` est écrite comme suit :

```
Procedure Liberer_Cabine
Début
  P(mutex)
  si ne>0 alors
    ne := ne - 1
    V(Se)
  sinon
    si ns>0 alors
      ns := ns - 1
      V(Ss)
    sinon
      Ucb := Ucb -1
    finsi
  finsi
  V(mutex)
Fin
```

Exercice 7 :

Résoudre le problème des lecteurs rédacteurs dans les cas suivants :

1) priorité aux Rédacteurs ,
2) équité : toute demande de lecture / écriture est satisfaite.

Solution :

1) priorité aux rédacteurs : dès qu'un rédacteur fait sa demande d'écriture , toutes les nouvelles demandes de lecture sont mises en attente ; elles ne seront prises compte que lorsqu'il n'y a plus d'écritures en cours et/ou en attente.

```
var S,S2,S3,mutex,mutex2 : sémaphore init 1,1,1,1,1 ;
    nl : entier init 0 ;
Processus Lecteur
Début
  P(S3)
  P(S2)
  P(mutex)
  nl := nl + 1
  si nl=1 alors P(S) finsi
  V(mutex)
  V(S2)
  V(S3)
  Lecture
  P(mutex)
  nl := nl - 1
  si nl=0 alors V(S) finsi
  V(mutex)
Fin
```

```
Processus Rédacteur
Début
  P(mutex2)
  nra := nra + 1
  si nra=1 alors P(S2) finsi
  V(mutex2)
  P(S)
  Ecriture
  V(S)
  P(mutex2)
  nra := nra - 1
  si nra=0 alors V(S2) finsi
  V(mutex2)
Fin
```

2) Pour réaliser l'équité on reprendra la solution décrite dans le paragraphe **1.3** à laquelle on ajoutera un sémaphore d'attente commune Se.

Remarque : Il est nécessaire que les files des sémaphores utilisés soient gérées de manière équitable , par exemple une gestion F.I.F.O.

```
var S,Se,mutex : sémaphore init 1,1,1 ;
    nl : entier init 0 ;
Processus Lecteur
Début
  P(Se)
  P(mutex)
  nl := nl + 1
  si nl=1 alors P(S) finsi
  V(mutex)
  V(Se)
  Lecture
  P(mutex)
  nl := nl - 1
  si nl=0 alors V(S) finsi
  V(mutex)
Fin
```

```
Processus Rédacteur
Début
  P(Se)
  P(S)
  V(Se)
  Ecriture
  V(S)
Fin
```

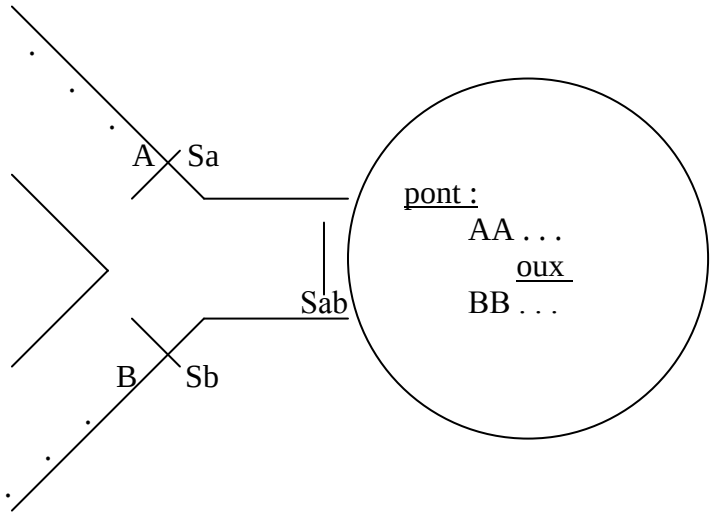
Exercice 8 :

Soit un pont ,à voie unique, traversé pas des véhicules en sens inverse (sens A et sens B).
A tout moment, le pont ne doit contenir que des véhicules allant dans un sens uniquement.
On assumera que le nombre de véhicules pouvant traverser le pont dans un sens est illimité.
On assimilera les véhicules à des processus parallèles synchronisés par sémaphores.
Ecrire les algorithmes de chaque classe de processus.

Solution :

1) Il y a exclusion mutuelle entre les deux classes de processus : à tout moment le pont ne doit contenir que des véhicules allant dans le sens A ou des véhicules allant dans le sens B.
Pour réaliser cette exclusion mutuelle on considérera que lorsqu'un véhicule de type A (*respectivement B*) arrive à l'entrée du pont trois situations peuvent se présenter :

- i l y a déjà des véhicules de type A (*resp^t B*) sur le pont , auquel cas le véhicule peut s'engager sur le pont ;
- il y a au moins un véhicule de type B (*resp^t A*) sur le pont , dans ce cas il doit attendre jusqu'à ce que tous les véhicules de type B (*resp^t A*) soient sortis du pont ;
- il n'y a aucun véhicule sur le pont, le véhicule s'engage sur le pont tout en assurant que s'il y a des véhicules de type B (*resp^t A*) qui se présenteront à l'entrée le pont, il s'arrêteront jusqu'à ce tous les véhicules A (*resp^t B*) sortent du pont.



Sab : barrière d'accès au pont
Sa : barrière faisant attendre les A(s) s'il y a au moins un B sur le pont
Sb : idem pour les B(s) et A.
oux : ou exclusif

Ce qui donne la solution suivante :

```
var Sa,Sb,Sab : sémaphore init 1,1,1 ;
    na,nb : entier init 0 ;
```

```
Processus A
Début
  P(Sa)
  na := na + 1
  si na=1 alors P(Sab) finsi
  V(Sa)
  Traverser le pont
  P(Sa)
  na := na - 1
  si na=0 alors V(Sab) finsi
  V(Sa)
Fin
```

```
Processus B
Début
  P(Sb)
  nb := nb + 1
  si nb=1 alors P(Sab) finsi
  V(Sb)
  Traverser le pont
  P(Sb)
  nb := nb - 1
  si nb=0 alors V(Sab) finsi
  V(Sb)
Fin
```

Exercice 9 :

Résoudre le problème du producteur/consommateur dans l'hypothèse que tampon est non borné (tampon de taille infinie) .

Solution :

Les algorithmes des producteurs / consommateurs s'exprimeront comme suit :

```
var S,mutex : sémaphore init 0,1 ;
Processus Producteur                    Processus Consommateur
Début                                   Début
  Cycle                                   Cycle
   <Produire un message>                   P(S)
   P(mutex)                                P(mutex)
   <Deposer le message>                    <Prelever un message>
   V(mutex)                                V(mutex)
   V(S)                                   <Consommer le message>
  FinCycle                                FinCycle
Fin                                     Fin
```

Exercice 10 :

N processus ,non cycliques, travaillent par point de rendez-vous :

Proc P_1 ...	Proc P_i ...	Proc P_N
Début	Début	Début
.	.	.
.	.	.
.	pt rdv	.
.	.	.
.	.	.
.	.	pt rdv
.	.	.
pt rdv	.	.
.	.	.
.	.	.
Fin	Fin	Fin

Programmer le rendez-vous de ces processus en utilisant les sémaphores.

Solution :

```
var att,mutex : sémaphore  init 0,1 ;
    na : entier  init 0 ;
Processus Pi  (1 ≤ i ≤ N)
Début
  .
  .
  P(mutex)
  na := na + 1
  si  na < N  alors
    V(mutex)
    P(att)
  sinon
    V(mutex)
  finsi
  V(att)  /* réveil des processus bloqués */
  .
  .
  .
Fin
```

Exercice 11 :

Une ressource partageable R peut être utilisée par au plus N processus simultanément. Un processus voulant utiliser R alors que celle-ci n'est pas disponible annule sa demande. Ecrire l'algorithme d'un tel processus.

Solution :

```
var mutex : sémaphore  init 1 ;
    nd : entier  init N ;
Processus Utilisant_R
Début
  P(mutex)
  si  nd > 0  alors
    nd := nd – 1
    V(mutex)
    Utiliser R
    P(mutex)
    nd := nd + 1
  finsi
  V(mutex)
Fin
```

Exercice 12 :

Un système d'interruption simple comporte deux bascules :

- de masquage m (interruption masquée si m=0, démasquée si m=1)
- de demande d'interruption t fonctionnant ainsi :

le signal d'interruption tente de faire t := t + 1 :

si l'interruption est démasquée , t passe immédiatement à 1

sinon, t passera à 1 au moment du démasquage ;

Réaliser ce système en utilisant les sémaphores.

Solution :

```
var m,t,mutex : sémaphore  init 1,0,1 ;
    masque : booléen  init faux ;
```

A l'arrivée du signal , le matériel fait

```
P(m)   /* demande de passage */
V(t)   /* déclenchement de la routine d'interruption */
V(m)   /* libération */
```

Pour masquer l'interruption on fait :

```
P(mutex)
si (masque = faux) alors
  masque := vrai
  V(mutex)
  P(m)
sinon
  V(mutex)
finsi
```

Pour démasquer l'interruption on fait :

```
P(mutex)
si (masque = vrai) alors
  masque := faux
  V(m)
finsi
V(mutex)
```

Pour traiter l'interruption on exécute le processus :

```
Processus Traitant_d'interruption
Début
  Cycle
    P(t)
    Routine d'interruption
  Fincycle
Fin
```

Exercice 13 :

On veut faire fonctionner deux processus P1 et P2 en coroutines.

Réaliser ce fonctionnement en utilisant les sémaphores.

Solution :

Il suffit d'utiliser , pour chacun des processus un sémaphore privé où il s'y bloquera lorsque l'autre processus s'exécute :

```
var S1,S2: sémaphore  init 0,0 ;
```

L'instruction `reprise(P2)` (exécutée par P1) sera réalisée par :

```
V(S2)
P(S1)
```

l'instruction `reprise(P1)` (exécutée par P2) sera réalisée par :

```
V(S1)
P(S2)
```

Exercice 14 :

On considère un ensemble de processus P*i* (*i*≥0) où chaque processus est identifié par un entier unique n*i* (n*i*>0).

Ces processus utilisent une ressource R avec la contrainte que le processus P*j* demandeur de R n'y accède que lorsque la somme des entiers ,associés aux processus en train d'utiliser R, est divisible par n*j*.

Réaliser la synchronisation de ces processus en utilisant les sémaphores.

Solution :

Une solution est de procéder comme suit :

```
var S : tableau[0.…] de sémaphore init (0,..,0) ;
    mutex : sémaphore init 1 ;
    bloqué : tableau[0.…] de booléen init (faux,..,faux) ;
    Somme,na : entier init 0,0 ;
Procedure Reveil
Début
  Tant que ∃ k tel que (bloqué[k] et ((Somme mod nk)=0)) faire
    Somme := Somme + nk
    na := na – 1
    bloqué[k] := faux
    V(S[k])
  FinTantque
Fin
Processus Pj (j≥0)
Début /* nj est l'entier associé à Pj */
  P(mutex)
  si ((Somme mod nj)=0) alors
    Somme := Somme + nj
    Reveil
    V(mutex)
  sinon
    na := na + 1
    bloqué[j] := vrai
    V(mutex)
    P(S[j])
  finsi
  Utiliser R
  P(mutex)
  Somme := Somme – nj
  Reveil
  V(mutex)
Fin
```

3. Exercices Supplémentaires :

Exercice 1 : *rivière*

Pour traverser une rivière, sont disposés N pavés espacés de telle sorte qu'une personne voulant traverser la rivière devra poser les pieds successivement sur tous ces pavés sans passer par dessus une personne qui se trouverait devant elle allant dans le même sens ou venant dans le sens opposé. On distingue deux types de personnes voulant traverser cette rivière : les gens de la rive gauche et les gens de la rive droite.

En assimilant ces personnes à des processus parallèles synchronisés par sémaphores , écrire les algorithmes des processus de chaque classe en garantissant le non interblocage.

Exercice 2 : *fumeurs*

On considère un système avec trois processus « fumeurs » et un processus « agent ».
Chaque fumeur roule une cigarette puis la fume et ce de façon continue. Pour fumer une cigarette trois ingrédients sont nécessaires : du tabac, du papier et des allumettes. L'un des fumeurs possède du tabac , l'autre du papier , et le troisième des allumettes. L'agent a une réserve infinie de ces trois ingrédients. L'agent met sur la table deux des trois ingrédients. Le fumeur auquel manque le(s) ingrédient(s) supplémentaires peut rouler et fumer sa cigarette puis le signale à l'agent. L'agent met alors deux autres ingrédients sur la table et le cycle se répète.
Ecrire les algorithmes des processus fumeurs et agent.

Exercice 3 : *coiffeur*

Un salon de coiffure est composé d'une salle d'attente contenant n chaises , et de la pièce de coiffure qui contient une chaise pour le coiffeur et une chaise pour le client. S'il n'y a aucun client , le coiffeur s'assied sur sa chaise et dort. Si un client entre dans le salon et trouve que toutes les chaises de la salle d'attente sont occupées , il s'en va. Si le coiffeur est occupé , après que le client constate qu'il y a des chaises libres , le client s'assied sur une chaise libre. Si le coiffeur est endormi, le client le réveille.
En assimilant les clients et le coiffeur à des processus parallèles synchronisés par sémaphores , écrire les algorithmes de ces processus.

Exercice 4 : *rond point*

On considère un rond-point comportant N voies ($N \geq 3$) numérotées de 0 à N-1. Tous les véhicules empruntant le rond-point tournent dans le même sens (sens croissant des indices des voies).
En assimilant les véhicules à des processus , et le rond-point à une ressource partagée écrire les algorithmes des processus synchronisés pas sémaphores ; en considérant que le nombre de véhicules pouvant se trouver dans le rond-point est illimité, mais doivent être issus d'une même voie (exclusion mutuelle entre les véhicules de voies différentes).

Exercice 5 :

On considère deux producteurs P1 et P2 qui produisent des messages et les déposent dans deux tampons T1 et T2 respectivement (T*i* pour P*i*, *i*=1,2). Deux processus consommateurs C1 et C2 consomment les messages : C1 ceux de T1 , C2 ceux de T2 ; avec la contrainte que lorsqu'un processus C*i* (*i*=1,2) consomme un message , il attendra que l'autre processus C*j* (*j*=3-*i*) ait consommé un message lui aussi pour continuer à consommer un autre message (Rendez-vous entre C1 et C2 après chaque consommation).
Synchroniser ces processus en utilisant les sémaphores.

Exercice 6 :

Une ressource R peut être utilisée par au plus N processus simultanément. Un processus voulant utiliser R doit attendre si R n'est pas disponible.
On considère qu'il y a m classes de processus Ci (i=1,..,m). Il n' y a pas d'exclusion mutuelle entre les différentes classes. Par contre, les processus de Ci (i<m) sont prioritaires sur ceux de Cj tel que j>i.
Ecrire les algorithmes des processus de différentes classes synchronisés par sémaphores.

Leçon Trois :

Plus formellement

1. Présentation : *Formalisme de Boksenbaum*

Soit $\{ S_k = \sum_{j=1}^{n} a_{kj} x_j + b_k \geq 0 \quad , \quad k=1,..,m \}$ un système d'inéquations décrivant les contraintes de synchronisation d'un problème , avec :

a_{kj} , b_k : constantes entières ; $b_k \geq 0$ pour k=1,..,m

x_j : variables entières

si on fait $x_j := x_j + 1$, cela implique $s_k := s_k + a_{kj}$

ce qui équivalent à :

faire $s_k := s_k + 1$ a_{kj} fois si $a_{kj}>0$ ou

faire $s_k := s_k - 1$ $|a_{kj}|$ fois si $a_{kj}<0$

on assimile l'inéquation s_k à un sémaphore s initialisé à b_k :

et lorsqu'on fait $x_j := x_j + 1$ cela implique :

faire a_{kj} fois V(s) si $a_{kj}>0$, ou

faire $|a_{kj}|$ fois P(s) si $a_{kj}<0$

Remarque :

Lorsqu'il faut faire une suite de $|a_{kj}|$ fois P(s) (avec $|a_{kj}| \geq 2$) il faut le faire en exclusion mutuelle ; sinon une situation d'interblocage peut se produire. Pour réaliser cette exclusion mutuelle il suffit d'utiliser un autre sémaphore, par exemple mutex :

```
P(mutex)
pour k := 1 à |akj| faire  /* k est une variable locale */
   P(s)
finpour
V(mutex)
```

Dans ce qui suit , sera présentée l'application du formalisme à quelques problèmes de synchronisation.

1.1 exclusion mutuelle

On considère un ensemble de processus p_i qui exécutent une action A en exclusion mutuelle.

```
Processus pi
Début
    .
    .
    x1 := x1 + 1
    Action A
    x2 := x2 + 1
    .
    .
Fin
```

x_1 : nombre d'actions A commencées depuis t_0

x_2 : nombre d'actions A terminées depuis t_0

(t_0 est un temps de référence, par exemple temps de début de premier processus concerné par l'action A)

contrainte : on doit avoir $x_1 - x_2 \leq 1$
ce qui est équivalent à avoir $x_2 - x_1 + 1 \geq 0$; on associera donc à cette inéquation un sémaphore s initialisé à 1.
Les processus en question s'exprimeront comme ainsi :

```
Processus pi
Début
    .
    .
    P(s)
    Action A
    V(s)
    .
    .
Fin
```

1.2 Lecteurs / Rédacteurs

Deux classes de processus se partagent en exclusion mutuelle une ressource : un fichier.
contraintes :

- un lecteur ne fait que de lectures sur le fichier , le nombre de lectures simultanées est non borné ,
- un rédacteur ne fait que des écritures sur un fichier , un seul rédacteur pouvant écrire à la fois ,
- exclusion mutuelle entre lecteurs et rédacteurs pour l'accès au fichier.

Si on note :
nl : nombre de lectures en cours ,
nr : nombre de rédactions en cours ;
on doit avoir :

$nr + signe(nl) \leq 1$; $signe(a) = \begin{cases} 0 \text{ si } a=0 \\ 1 \text{ si } a \geq 1 \end{cases}$

```
Processus Lecteur                    Processus Rédacteur
Début                                Début
   .                                    .
   .                                    .
   faire  nl := nl + 1                  faire nr := nr + 1
   Lire                                 Ecrire
   nl := nl - 1                         nr := nr - 1
   .                                    .
   .                                    .
Fin                                  Fin
```

La contrainte $1 - nr - signe(nl) \geq 0$ sera représentée par un sémaphore s initialisé à 1.
nl := nl + 1 : si nl passe de 0 à 1 on fait P(s)
nl := nl – 1 : si nl passe de 1 à 0 on fait V(s)
nr := nr + 1 : on fait P(s)
nr := nr – 1 : on fait V(s)
Et on utilisera , en plus du sémaphore s , une variable entière nl et par conséquent un sémaphore d'exclusion mutuelle pour la manipulation de celle-ci.

Ce qui donne la solution suivante :

```
Processus Lecteur                    Processus Rédacteur
Début                                Début
   .                                    .
   .                                    .
   P(mutex)                             P(s)
   nl := nl + 1                         Ecrire
   si nl=1 alors P(s) finsi             V(s)
   V(mutex)                             .
   Lire                                 .
   P(mutex)                          Fin
   nl := nl - 1
   si nl=0 alors V(s) finsi
   V(mutex)
   .
   .
Fin
```

1.3 Producteurs / Consommateurs

Un tampon de taille N est partagé par deux classes de processus : les producteurs et les consommateurs. Les producteurs produisent des messages qu'il déposent dans le tampon. Les consommateurs prélèvent les messages du tampon et les consomment.

Notons : x : nombre de messages déposés depuis t_0

y : nombre de messages prélevés depuis t_0

Les contraintes sont :

$y \leq x$

$x \leq y + N$

ce qui équivalent à :

$x - y \geq 0$: sémaphore S1 initialisé à 0

$y - x + N \geq 0$: sémaphore S2 initialisé à N

Les processus producteur et consommateur procèdent comme suit :

```
Processus Producteur                 Processus Consommateur
Début                                Début
   .                                    .
   .                                    .
 α: Fabriquer un message             β: faire y := y + 1
   faire x := x + 1                      prélever un message
   déposer le message                    Traiter le message
   Aller_à α                             Aller_à β
   .                                    .
   .                                    .
Fin                                  Fin
```

De l'incrémentation x := x + 1 on obtient : P(S2)
V(S1)

de y := y + 1 on obtient : P(S1)
V(S2)

D'où :

```
Processus Producteur                Processus Consommateur
Début                               Début
   .                                   .
   .                                   .
α: Fabriquer un message             β: P(S1)
   P(S2)                               prélever un message
   déposer le message                  V(S2)
   V(S1)                               Traiter le message
   Aller_à α                           Aller_à β
   .                                   .
   .                                   .
Fin                                 Fin
```

Remarque :

Pour l'accès au tampon, il faut ajouter un sémaphore d'exclusion mutuelle :

```
P(mutex)   /* mutex init 1 */
Accès au tampon  /*Dépôt ou prélèvement du message */
V(mutex)
```

2. Exercices :

Exercice 1 :

On considère une ressource R utilisée par N processus au plus simultanément.

1) Ecrire la (les) contrainte(s) (inéquation(s) régissant le problème).

2) En déduire la synchronisation des processus en question.

Solution :

1) En notant : x le nombre de processus en train d'utiliser R.

   La contrainte que l'on doit avoir est : $x \leq N$

2) L'inéquation de 1) est équivalente à $N - x \geq 0$ qui sera traduite par sémaphore S initialisé à N.

```
Processus Utilisant_R
Début
   .
   .
   x := x + 1  /* incrémenter le nombre d'utilisateurs de R */
   utiliser R
   x := x - 1  /* décrémenter le nombre d'utilisateurs de R */
   .
   .
Fin
```

`x := x + 1` engendre la décrémentation de $N - x$, sera traduite par P(S) ,

`x := x - 1` engendre l'incrémentation de $N - x$, sera traduite par V(S) ;

On aura donc :

```
Processus Utilisant_R
Début
   .
   .
   P(S)
   utiliser R
   V(S)
   .
   .
Fin
```

Exercice 2 :

On considère le problème des lecteurs/rédacteurs tel qu'énoncé au paragraphe 1.2 avec l'hypothèse que le nombre de lectures simultanées est borné.

1) Ecrire la (les) contrainte(s) (inéquation(s) régissant le problème).

2) En déduire les algorithmes des processus Lecteurs et Rédacteurs synchronisés par sémaphores.

Solution :

1) En notant : nl : nombre de lectures en cours ,
nr : nombre de rédactions en cours ;

on doit avoir :

$nr + signe(nl) \leq 1$
$nl \leq N$

Les deux inéquations ci-dessus peuvent être réécrites en l'inéquation : $N . nr + nl \leq N$ ou de manière équivalente $N - N . nr - nl \geq 0$: d'où l'utilisation d'un sémaphore S initialisé à N.

2) Si on fait nl := nl + 1 : cela se traduit par P(S)
nl := nl – 1 : par V(S)
nr := nr + 1 : par une suite de N fois P(S)
nr := nr – 1 : par une suite de N fois V(S)

Remarque :

Si plusieurs rédacteurs entament de faire la séquence des N fois P(S) ; il peut y avoir interblocage ; exemple : deux rédacteurs ,lors de leur exécution, font alternativement P(S) au bout de N/2 P(S) chacun , la valeur de S devient nulle et les deux rédacteurs se bloqueront sans aucun moyen d'être réveillés.

D'où la nécessité de faire de la séquence des N P(S) une section critique.

Et on obtient :

```
var S,mutex : sémaphore init N,1 ;
Processus Lecteur
Début
   .
   .
   P(S)
   Lecture
   V(S)
   .
   .
Fin
```

```
Processus Rédacteur
var k : entier ;
Début
   .
   .
   P(mutex)
   pour k := 1 à N faire
      P(S)
   finpour
   V(mutex)
   Ecriture
   pour k := 1 à N faire
      V(S)
   finpour
Fin
```

Remarque : Il aurait été plus simple de conserver la solution du paragraphe 1.2 précédent , en encadrant la section `Lire` des processus lecteurs comme suit

```
P(SN)
Lire
V(SN)
```

où `SN` est un sémaphore initialisé à N.

Exercice 3 :

On considère une ressource R qui peut être utilisée par deux classes de processus A et B. Le nombre d'utilisations simultanées est illimité. Par contre les deux classes A et B sont mutuellement exclusives. Et de plus, on considère que les processus de la classe B sont plus prioritaires que ceux de A pour l'accès à R.

1) Ecrire la (les) contrainte(s) (inéquation(s) régissant le problème).

2) En déduire les algorithmes des processus de A et B synchronisés par sémaphores.

Solution :

1) Pour représenter la priorité en question on introduit une antichambre (salle d'attente !) par laquelle les processus de A et B doivent passer ; mais en autorisant uniquement à un seul processus A de s'y trouver. Pour cela on introduit une barrière supplémentaire pour les processus de A.

En notant :

na : nombre de processus de la classe A en cours d'utilisation de la ressource R

nb : nombre de processus de la classe B en cours d'utilisation de la ressource R

nat : nombre de processus de la classe A dans l'antichambre

nbt : nombre de processus de la classe B dans l'antichambre

nar : nombre de processus de la classe A pouvant franchir la barrière en même temps

on a les inéquations :

$\text{signe}(na) + \text{signe}(nb) \leq 1$

$nat + \text{signe}(nbt) \leq 1$

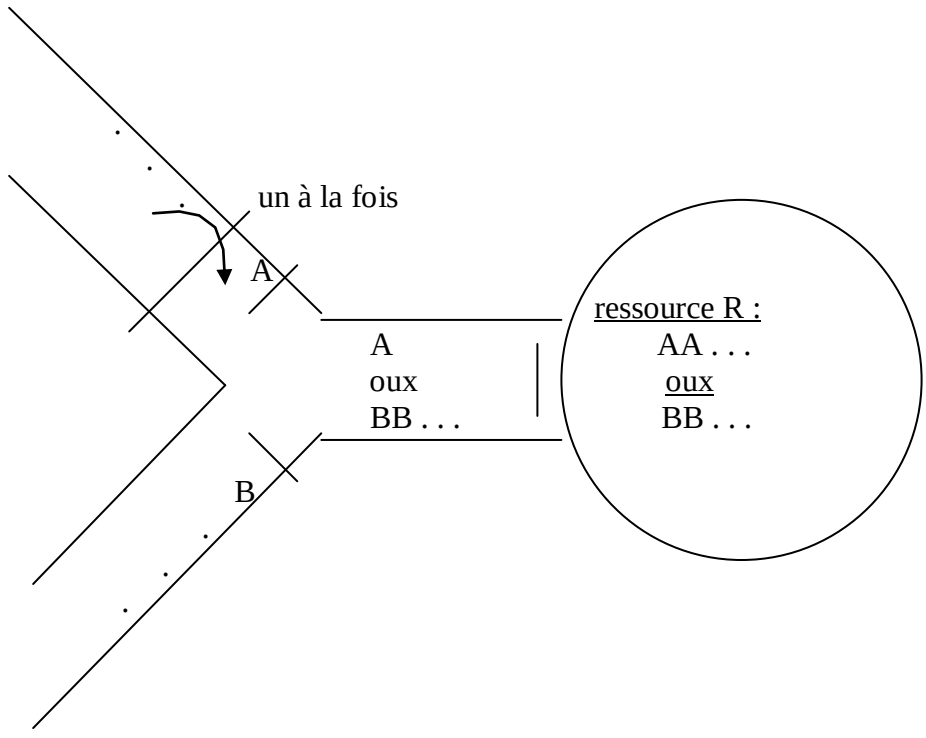
$nar \leq 1$

qui sont équivalentes respectivement à :

$1 - \text{signe}(na) - \text{signe}(nb) \geq 0$ $\Rightarrow$ sémaphore S1 initialisé à 1

$1 - nat - \text{signe}(nbt) \geq 0$ $\Rightarrow$ sémaphore S2 initialisé à 1

$1 - nar \geq 0$ $\Rightarrow$ sémaphore S3 initialisé à 1



2)

```
Processus A
Début
  faire nar := nar + 1
  faire nat := nat + 1
  faire na := na + 1
  nat := nat - 1
  nar := nar – 1
  Utiliser R
  na := na - 1
Fin
```

```
Processus B
Début
  faire nbt := nbt + 1
  faire nb := nb + 1
  Utiliser R
  nb := nb - 1
  nbt := nbt – 1
Fin
```

Les algorithmes des processus des deux classes sont comme suit

```
var S1,S2,S3,mutex1,mutex2,mutex3 : sémaphore init 1,1,1,1,1,1 ;
    na,nb,nbt : entier init 0,0,0 ;
```

```
Processus A
Début
  P(S3)
  P(S2)
  P(mutex1)
  na := na + 1
  si na=1 alors P(S1) finsi
  V(mutex1)
  V(S2)
  V(S3)
  Utiliser R
  P(mutex1)
  na := na - 1
  si na=0 alors V(S1) finsi
  V(mutex1)
Fin
```

```
Processus B
Début
  P(mutex2)
  nbt := nbt + 1
  si nbt=1 alors P(S2) finsi
  V(mutex2)
  P(mutex3)
  nb := nb + 1
  si nb=1 alors P(S1) finsi
  V(mutex3)
  Utiliser R
  P(mutex3)
  nb := nb – 1
  si nb=0 alors V(S1) finsi
  V(mutex3)
  P(mutex2)
  nbt := nbt – 1
  si nbt=0 alors V(S2) finsi
  V(mutex2)
Fin
```

Exercice 3 :

Cinq processus (trois producteurs P1 , P2 , P3 et deux consommateurs C1 , C2) se partagent un tampon de M cases en mémoire centrale. Les producteurs P1 et P2 fabriquent des messages qui seront consommés par C1 ; et le producteur P3 fabrique des messages qui seront consommés par C2.

A tout instant le tampon ne doit contenir que des messages destinés soit C1 seulement , soit à C2 seulement.

1) Ecrire la (les) contrainte(s) (inéquation(s) régissant le problème).

2) Ecrire les algorithmes des différents processus synchronisés par sémaphores.

Solution :

1) Notons

n_1 : nombre de messages produits par P1 et/ou P2 contenus dans le tampon ,

n_2 : nombre de messages produits par P3 contenus dans le tampon ;

x : nombre de processus pouvant accéder simultanément au tampon.

Les contraintes à respecter sont :

$0 \leq n_i \leq M$; i=1,2

$signe(n_1) + signe(n_2) \leq 1$

$0 \leq x \leq 1$

2) Les processus en question s'expriment ainsi :

```
Processus Pi (i=1,2)
Début
αi: Fabriquer un message
    faire n1 := n1 + 1
    faire x := x + 1
    deposer le message
    x := x - 1
    Aller_à αi
Fin
```

```
Processus C1
Début
β1: faire n1 := n1 - 1
    faire x := x + 1
    retirer un message
    x := x - 1
    Traiter le message
    Aller_à β1
Fin
```

```
Processus P3
Début
α3: Fabriquer un message
    faire n2 := n2 + 1
    faire x := x + 1
    deposer le message
    x := x - 1
    Aller_à α3
Fin
```

```
Processus C2
Début
β2: faire n2 := n2 - 1
    faire x := x + 1
    retirer un message
    x := x - 1
    Traiter le message
    Aller_à β2
Fin
```

Les inéquations de 1) peuvent être écrites comme suit :

$n_1 \geq 0$	$\Rightarrow$	sémaphore S1 initialisé à 0
$M - n_1 \geq 0$	$\Rightarrow$	sémaphore S2 initialisé à M
$n_2 \geq 0$	$\Rightarrow$	sémaphore S3 initialisé à 0
$M - n_2 \geq 0$	$\Rightarrow$	sémaphore S4 initialisé à M
$1 - signe(n_1) - signe(n_2) \geq 0$	$\Rightarrow$	sémaphore S5 initialisé à 1
$1 - x \geq 0$	$\Rightarrow$	sémaphore mutex initialisé à 1

L'incrémentation `n1 := n1 + 1` engendre :

V(S1)
P(S2)
si n1 passe de 0 à 1 alors P(S5)

La décrémentation `n1 := n1 - 1` engendre :

P(S1)
V(S2)
si n1 passe de 1 à 0 alors P(S5)

Il en est de même pour n2 (remplacer S1 par S3 et S2 par S4).

Ce qui nous conduit à la solution suivante :

```
var S1,S2,S3,S4,S5,mutex,mutex1,mutex2 :
   sémaphore init 0,M,0,M,1,1,1,1;
```

```
Processus Pi (i=1,2)
Début
αi: Fabriquer un message
    P(mutex1)
    n1 := n1 + 1
    si n1=1 alors P(S5) finsi
    V(mutex1)
    P(S2)
    P(mutex)
    deposer le message
    V(mutex)
    V(S1)
    Aller à αi
Fin
```

```
Processus C1
Début
β1: P(S1)
    P(mutex)
    retirer un message
    V(mutex)
    V(S2)
    P(mutex1)
    n1 := n1 - 1
    si n1=0 alors V(S5) finsi
    V(mutex1)
    Traiter le message
    Aller à β1
Fin
```

```
Processus P3
Début
α3: Fabriquer un message
    P(mutex2)
    n2 := n2 + 1
    si n2=1 alors P(S5) finsi
    V(mutex2)
    P(S4)
    P(mutex)
    deposer le message
    V(mutex)
    V(S3)
    Aller à α3
Fin
```

```
Processus C2
Début
β2: P(S3)
    P(mutex)
    retirer un message
    V(mutex)
    V(S4)
    P(mutex2)
    n2 := n2 - 1
    si n2=0 alors V(S5) finsi
    V(mutex2)
    Traiter le message
    Aller à β2
Fin
```

Exercice 4 :

On considère trois classes de processus A=(ai) , B=(bi) , C=(ci) dans lesquelles le nombre de processus de chaque classe pouvant s'exécuter simultanément est respectivement 1 , 2 et 3. Ces trois classes sont en exclusion mutuelle.

1) Ecrire la (les) contrainte(s) (inéquation(s) régissant le problème).

2) Ecrire les algorithmes des processus des différentes classes synchronisés par sémaphores.

Solution :

1) Soit n_1 : nombre de processus de la classe A en train de s'exécuter
n_2 : nombre de processus de la classe B en train de s'exécuter
n_3 : nombre de processus de la classe C en train de s'exécuter

Les contraintes que l'on doit avoir sont :

$n_i \leq i$; i=1,2,3

$signe(n_1) + signe(n_2) + signe(n_3) \leq 1$

équivalentes à :

$i - n_i \geq 0$; i=1,2,3

$1 - signe(n_1) - signe(n_2) - signe(n_3) \geq 0$

2) Les algorithmes des processus de A , B et C sont les suivants :

```
var mutex1,mutex2,mutex3,Sc,S1,S2,S3 : semaphore init 1,1,1,1,1,2,3;
    n1,n2,n3 : entier init 0,0,0;
```

```
Processus A
Début
  P(mutex1)
  n1 := n1 + 1
  si n1=1 alors
    P(Sc)
  finsi
  V(mutex1)
  P(S1)
  Execution
  V(S1)
  P(mutex1)
  n1 := n1 - 1
  si n1=0 alors
    V(Sc)
  finsi
  V(mutex1)
Fin
```

```
Processus B
Début
  P(mutex2)
  n2 := n2 + 1
  si n2=1 alors
    P(Sc)
  finsi
  V(mutex2)
  P(S2)
  Execution
  V(S2)
  P(mutex2)
  n2 := n2 - 1
  si n2=0 alors
    V(Sc)
  finsi
  V(mutex2)
Fin
```

```
Processus C
Début
  P(mutex3)
  n3 := n3 + 1
  si n3=1 alors
    P(Sc)
  finsi
  V(mutex3)
  P(S3)
  Execution
  V(S3)
  P(mutex3)
  n3 := n3 - 1
  si n3=0 alors
    V(Sc)
  finsi
  V(mutex3)
Fin
```

Exercice 5 :

On considère le problème des cinq philosophes (paragraphe 1.5 de la leçon 2) , où l'on ajoute la condition supplémentaire qu'à tout moment quatre philosophes au plus sont assis autour de la table ; cela pour éviter le risque d'interblocage dû au nombre et à la disposition des fourchettes autour de la table.

Ainsi le schéma d'un philosophe est :

```
Philosophe i (i=0,4)
Début
  Cycle
     S'asseoir
     Penser
     Prendre les fourchettes (la droite  puis la gauche)
     Manger
     Reposer les fourchettes
     Se lever
  FinCycle
Fin
```

1) Ecrire la (les) contrainte(s) (inéquation(s) régissant le problème).

2) Ecrire les algorithmes des processus philosophes synchronisés par sémaphores.

Solution :

1) Soit n : nombre de philosophes assis autour de la table

n_i : - - - détenteurs de la fourchette i (i=0,..,4)

Les contraintes que l'on doit avoir sont :

$n \leq 4$

$n_i \leq 1$; i=0,..,4

desquelles on peut obtenir :

$4 - n \geq 0$

$1 - n_i \geq 0$; i=0,..,4

2)

```
var accès : sémaphore init 4 ;
    S : tableau[0..4] de sémaphore init 1 ::0..4 ;
Processus Philosophe i (i=0..4)
Début
  Cycle
    P(accès)  /* 4 philosophes au plus sont assis autour de la table */
    <Penser>
    P(S[i])  /* prendre la fourchette i */
    P(S[(i+1) mod 5])  /* prendre la fourchette (i+1) mod 5 */
    <Manger>
    V(S[i])  /* reposer la fourchette i */
    V(S[(i+1) mod 5])  /* reposer la fourchette (i+1) mod 5 */
    V(accès)
  FinCycle
Fin
```

3. Exercices Supplémentaires :

Exercice 1 :

Un parking pour automobiles comporte N places. On considère que les véhicules (entrants ou sortants) sont des processus.

1) a) Ecrire la (les) contrainte(s) (inéquation(s) régissant le problème).
   b) Ecrire les algorithmes des processus automobiles synchronisés par sémaphores.
2) On considère maintenant que l'entrée au parking , ou la sortie de celui-ci se fait par une longue passerelle pouvant contenir plusieurs véhicules , mais allant dans une seule voie (entrée ou sortie). Lorsque des véhicules désirent entrer au parking alors que d'autres véhicules désirent en sortir on donnera la priorité à ceux qui sortent.
   a) Ecrire la (les) contrainte(s) (inéquation(s) régissant le problème).
   b) Ecrire les algorithmes des processus automobiles synchronisés par sémaphores.

Exercice 2 :

On considère une ressource R utilisées par trois classes C1,C2 et C3 en exclusion mutuelle (entre les classes). Le nombre de processus de chaque classe utilisant la ressource à un instant donné est illimité.

1) On considère que la classe C1 est prioritaire sur les deux autres .
   a) Ecrire la (les) contrainte(s) (inéquation(s) régissant le problème).
   b) Ecrire les algorithmes des processus des classes C1 , C2 et C3 synchronisés par sémaphores.
2) Mêmes questions que précédemment en considérant qu'il n'y a pas de priorité entre les classes , et il y a équité (toute demande d'utilisation de la ressource R doit être satisfaite).

Exercice 3 :

On considère le problème du pont (exercice 8 de la leçon 2) ; où l'on ajoute les hypothèses suivantes :

- nombre de véhicules allant dans le sens A $\leq$ N
- nombre de véhicules allant dans le sens B $\leq$ M
- priorité des véhicules allant dans le sens B

1) Ecrire la (les) contrainte(s) (inéquation(s) régissant le problème).
2) Ecrire les algorithmes des processus des classes A et B synchronisés par sémaphores.

Exercice 4 :

On considère le problème du lecteurs/rédacteurs (paragraphe 1.3 de la leçon 2) , où l'on décide de donner la priorité aux lecteurs : une lecture n'est pas retardée par une écriture en attente mais seulement par une écriture en cours.

1) Ecrire la (les) contrainte(s) (inéquation(s) régissant le problème).
2) Ecrire les algorithmes des processus lecteurs et rédacteurs synchronisés par sémaphores.

Leçon Quatre :

Variantes

1. Présentation :

Il existe de nombreuses variantes des sémaphores (dits simples ou sémaphores de Dijkstra). Dans ce qui suit on présentera les sémaphores de Vantilborgh, de Patil et les sémaphores binaires.

1.1 Sémaphores de Vantilborgh

Le franchissement de l'opération P (dans les sémaphores simples) par un processus p peut être vu comme l'acquisition d'un ticket par celui-ci. L'exécution de V par p est ,ainsi, la restitution d'un ticket par le processus p.

Vantilborgh a défini une variante des sémaphores qui permet à un processus de préciser le nombre de tickets qu'il désire obtenir ou qu'il restitue.

Les primitives P et V s'exécutent comme suit :

```
P(n,s) : Début
           Si  n ≤ s.val  alors
             s.val := s.val – n
           sinon
             état(p) := bloqué /* p est le processus qui exécute P(n,s) */
             rang(p) := n
             insérer p dans s.file
           finsi
         Fin
V(n,s) : Début
           s.val := s.val + n
           Tant que (∃ q ∈ s.file) et (rang(q) ≤ s.val) faire
             s.val := s.val – rang(q)
             sortir le processus q de s.file
             état(q) := prêt
           finTantque
         Fin
```

Exemple : *Lecteurs / Rédacteurs*

On considère le problème des lecteurs/rédacteurs (paragraphe 1.3 de la leçon 2) auquel on ajoute la condition que le nombre de lectures simultanées est limité à N.

En notant : nl : nombre de lectures en cours ,

nr : nombre de rédactions en cours ;

on doit avoir :

$N \cdot nr + nl \leq N$

ou de manière équivalente $N - N \cdot nr - nl \geq 0$: d'où l'utilisation d'un sémaphore S initialisé à N.

Si on fait nl := nl + 1 : cela se traduit par P(1,S)
nl := nl – 1 : par V(1,S)
Si on fait nr := nr + 1 : cela se traduit par P(N,S)
nr := nr – 1 : par V(N,S)

Et on aura donc :

```
var S : sémaphore init N;
Processus Lecteur                    Processus Rédacteur
Début                                Début
  .                                    .
  .                                    .
  P(1,S)                               P(N,S)
  Lecture                              Ecriture
  V(1,S)                               V(N,S)
  .                                    .
  .                                    .
Fin                                  Fin
```

Remarque :

En conservant l'algorithme des lecteurs tel quel et en récrivant l'algorithme des rédacteurs ainsi :

```
Processus Rédacteur
Début
  .
  .
  P(N,S)
  Ecriture
  V(1,S)
  V(N-1,S)
  .
  .
Fin
```

on donne la priorité aux lecteurs : si un rédacteur est en train d'écrire et qu'il y a des lecteurs et des rédacteurs en attente ; alors lorsqu'il termine d'écrire il fera d'abord V(1,S) , ce qui libère un lecteur, puis V(N-1,S) , ce qui peut libérer N-1 lecteurs.

1.2 Sémaphores de Patil

La primitive P multiple a été définie ,par Patil, comme suit :

```
P(S1,...,Sk) :
Début
  Si (∃ j tel que Sj.val≤0) alors
    Se bloquer jusqu'à ce que (∀ j Sj.val>0)
  finsi
  Pour j := 1 à k Faire
    Sj.val := Sj.val - 1
  FinPour
Fin
```

Remarque 1 :
La primitive V(s) conserve sa définition habituelle , à savoir l'incrémentation de s.val (ce qui peut ,éventuellement, réveiller un processus bloqué à cause d'un P sur s).

Remarque 2 :
L'importance des sémaphores de Patil réside dans le fait que dans quelques applications un processus peut avoir besoin d'exécuter successivement plusieurs primitives P (simples) sur des sémaphores différents : c'est le cas par exemple d'un processus qui a besoin de plusieurs ressources.
En procédant par la primitive P de Patil on peut éviter ,dans certains cas, l'interblocage.
Soit , par exemple, deux processus P1 et P2 s'exécutant comme suit :

```
var s1,s2 : sémaphore init 1,1 ;
Processus P1                    Processus P2
Début                           Début
  P(s1)                           P(s2)
  P(s2)                           P(s1)
  utiliser R1 et R2               utiliser R2 et R1
  V(s2)                           V(s2)
  V(s1)                           V(s1)
Fin                             Fin
```

On constate que l'on peut tomber dans une situation d'interblocage (considérer la séquence P(s_1) *(proc. P_1)* P(s_2) *(proc. P_2)* P(s_2) *(proc. P_1)* P(s_1) *(proc. P_2)*).
En utilisant la primitive P de Patil on peut éviter cet interblocage. On écrit donc :

```
Processus P1                    Processus P2
Début                           Début
  P(s1,s2)                        P(s2,s1)
  utiliser R1 et R2               utiliser R2 et R1
  V(s1)                           V(s2)
  V(s2)                           V(s1)
Fin                             Fin
```

Exemple : *Fumeurs*

On considère le problème des fumeurs de cigarettes (tel qu'énoncé dans l'exercice supplémentaire 2 de la leçon 2).
En associant à chaque ingrédient un sémaphore , plus un sémaphore privé pour le processus agent, on obtient la solution suivante :

```
var Tabac,Papier,Allumettes,S : sémaphore init 0,0,0,1 ;

Processus Fumeur_0              Processus Fumeur_1
Début                           Début
α0: P(papier,Allumettes)        α1: P(Tabac,Allumettes)
    <Rouler et Fumer                <Rouler et Fumer
     une cigarette>                  une cigarette>
    V(S)                            V(S)
    Aller à α0                      Aller à α1
Fin                             Fin
```

```
Processus Fumeur_2
Début
α2: P(Tabac,papier)
    <Rouler et Fumer
     une cigarette>
    V(S)
    Aller à α2
Fin
```

```
Processus Agent
var i : entier
Début
α: P(S)
   i := Random(0,2)
   /* nombre aléatoire ∈ 0..2 */
   selon i faire
     0 : début
           <mettre sur la table du papier et des allumettes>
           V(papier)
           V(Allumettes)
         fin
     1 : début
           <mettre sur la table du tabac et des allumettes>
           V(Tabac)
           V(Allumettes)
         fin
     2 : début
           <mettre sur la table du tabac et du papier>
           V(Tabac)
           V(papier)
         fin
   finselon
   Aller à α
Fin
```

1.3 Sémaphores binaires : *les verrous*

Un sémaphore binaire est un sémaphore (simple) dont le champ `val` (valeur) ne peut prendre que les valeurs 0 ou 1.
Un concept équivalent est le verrou dont le champ `val` est de type `booléen` ; c'est-à-dire qu'il ne peut prendre que les valeurs faux ou vrai.
Si v est un verrou , les primitives ENQ et DEQ manipulant ce verrou s'exécutent , de manière indivisible , comme suit :

```
ENQ(v) : Début  /* p est le processus qui exécute ENQ(v) */
           Si v.val=vrai alors
              v.val := faux
            Sinon
              < Insérer p dans v.file >
              état(p) := bloqué
           FinSi
         Fin
```

```
DEQ(v) : Début
           Si (v.file non vide) alors
              < extraire un processus q de v.file >
              état(q) := prêt
            Sinon
              v.val := vrai
           FinSi
         Fin
```

Remarque :
Les primitives ENQ et DEQ sont aussi dénommées Lock et Unlock respectivement.

Exemple : *ressource critique*
L'exclusion mutuelle pour l'utilisation d'une ressource critique peut s'exprimer comme suit :

```
var mutex : verrou init vrai;
     /* mutex est une variable globale de type verrou */
     /* dont la valeur est initialisée à vrai */
Processus Pi   /* i=1,n */
début
  <section non critique>
  ENQ(mutex)
  <utilisation de R>  /* section critique */
  DEQ(mutex)
  <section restante>
fin
```

Remarque :
L'utilisation d'ENQ et DEQ à la place de P et V ne donne pas toujours le même résultat , par exemple :

```
var s : sémaphore init 0 ;
Processus p
Début
  V(s)
  V(s)
  P(s)
  P(s)
  Action A
Fin
```

le processus p arrive à exécuter son Action A.

En faisant le remplacement :

```
var s : verrou init faux ;
Processus p
Début
  DEQ(s)
  DEQ(s)
  ENQ(s)
  ENQ(s)
  Action A
Fin
```

le processus p n'arrive pas à exécuter son Action A : il est bloqué en exécutant le second ENQ(s).

2. Exercices :

Exercice 1 :

On considère un ensemble de (n+1) processus P_1 ,....., P_n, P_{n+1} où les P_i ,i=1..n précédent P_{n+1} dans l'exécution : P_{n+1} ne peut s'exécuter effectivement que lorsque tous les P_i ,i=1..n se sont achevés.

Réaliser la synchronisation de ces processus en utilisant les sémaphores de Vantilborgh.

Solution :

On peut exprimer la synchronisation des processus P_i ,i=1..n+1 comme suit :

```
var s : sémaphore init 0 ;
Processus Pi (i=1..n)
Début
  Execution
  V(1,s)
Fin
```

```
Processus Pn+1
Début
  P(n,s)
  Execution
Fin
```

Exercice 2 :

Programmer le problème des lecteurs / rédacteurs à l'aide des sémaphores de Vantilborgh avec les contraintes suivantes :

- exclusion mutuelle entre lecteurs et rédacteurs ,
- exclusion mutuelle entre rédacteurs ,
- nombre de lectures simultanées limité à N ($N \geq 1$) ,
- priorité aux rédacteurs.

Solution :

On procède comme dans l'exercice 2 de la leçon 3.

On obtient les contraintes suivantes :

$N - n_l - N \,.\, n_r \geq 0$ $\Rightarrow$ sémaphore S1 initialisé à N

$N - N \,.\, n_{lt} - n_{rt} \geq 0$ $\Rightarrow$ sémaphore S2 initialisé à N

$1 - n_{lr} \geq 0$ $\Rightarrow$ sémaphore S3 initialisé à 1

Les lecteurs et rédacteurs font :

```
Processus Lecteur
Début
  faire nlr := nlr + 1
  faire nlt := nlt + 1
  faire nl := nlr + 1
  nlt := nlt - 1
  nlr := nlr - 1
  Lecture
  nlr := nlr - 1
Fin
```

```
Processus Rédacteur
Début
  faire nrt := nrt + 1
  faire nr := nr + 1
  Ecriture
  nr := nr - 1
  nrt := nrt - 1
Fin
```

Les algorithmes des processus des deux classes sont donc comme suit :

```
var S1,S2,S3 : sémaphore init N,N,1 ;
Processus Lecteur
Début
  P(1,S3)
  P(N,S2)
  P(1,S1)
  V(N,S2)
  V(1,S3)
  Lecture
```

```
Processus Redacteur
Début
  P(1,S2)
  P(N,S1)
  Ecriture
  V(N,S1)
  V(1,S2)
Fin
```

```
  V(1,S1)
Fin
```

Exercice 3 :

On considère le problème du producteur/consommateur , où le tampon est de longueur N (N cases).

Les producteurs produisent des messages de taille variable (pouvant varier de 1 à N cases).

Les consommateurs consomment les messages indifféremment de leur taille.

Ecrire les algorithmes des processus producteur et consommateur synchronisés par sémaphores de Vantilborgh.

Solution :

Les processus producteurs/consommateurs exécutent :

```
Processus Producteur
Début
α : <Produire un message
     de taille i>  /* 1 ≤ i ≤ N */
    <Dépôt du message
     dans le tampon>
    Aller à α
Fin
```

```
Processus Consommateur
Début
β : <Retirer du tampon un
     message de taille j>
    <Traiter le message
     (de taille j)>
    Aller à β
Fin
```

Pour assurer la synchronisation on procédera comme suit :

```
var  S1, mutex : sémaphore  init N,1;
     S2 : tableau[1..N] de sémaphore  init (0,..,0);
     /* S1.val  représente le nombre de cases vides du tampon */
     /* S2[k]  sémaphore pour les consommateurs de messages de taille k */
Processus Producteur
Début
α : <Produire un message
     de taille i>  /* 1 ≤ i ≤ N */
    P(i,S1)
    P(1,mutex)
    <Dépôt du message
     dans le tampon>
    V(1,mutex)
    V(1,S2[i])
    Aller à α
Fin
```

```
Processus Consommateur
Début
β : P(1,S2[j])
    P(1,mutex)
    <Retirer du tampon un
     message de taille j>
    V(1,mutex)
    V(j,S1)
    <Traiter le message
     (de taille j)>
    Aller à β
Fin
```

Exercice 4 :

Résoudre le problème des cinq philosophes (paragraphe 1.5 de la leçon 2) en utilisant les sémaphores de Patil.

Solution :

On peut associer à chaque fourchette un sémaphore d'exclusion mutuelle :

```
var Fourchette : tableau[0..4] de sémaphore init (1,1,1,1,1) ;
Processus Philosophe i (i=0..4)
```

```
Début
  Cycle
    <Penser>
    P(Fourchette[i],Fourchette[(i+1) mod 5])
    <Manger>
    V(Fourchette[i])
    V(Fourchette[(i+1) mod 5])
  FinCycle
Fin
```

Exercice 5 :

Résoudre l'exercice 3 de la leçon 2 en utilisant les sémaphores de Patil.

Solution :

La synchronisation peut être réalisée comme suit :

```
var SA, SB, SC, SD, SE : sémaphore init 0,0,0,0,0 ;
Tâche A                    Tâche B                  Tâche C
Début                      Début                    Début
  Exécution                  P(SA)                    P(SA)
  V(SA) ; V(SA) ; V(SA)      Exécution                Exécution
Fin                          V(SB)                    V(SC)
                           Fin                      Fin

Tâche D                    Tâche E                  Tâche F
Début                      Début                    Début
 P(SA)                       P(SB,SC)                 P(SE,SD)
 Exécution                   Exécution                Exécution
 V(SD)                       V(SE)                  Fin
Fin                        Fin
```

Exercice 6 :

Résoudre le problème des producteurs/consommateurs (paragraphe 1.4 de la leçon 2) en utilisant les verrous.

Solution :

Soit N le nombre de messages que peut contenir le tampon.

On peut utiliser une variable entière `nm` pour compter le nombre messages contenus dans le tampon. On bloquera l'accès au tampon pour les producteurs lorsque `nm` devient supérieur ou égal à N. Et on bloquera l'accès au tampon pour les consommateurs lorsque `nm` devient nul.

```
var S1,S2,m,mutex : vérou init vrai,faux,vrai,vrai ;
    nm : entier init 0 ;
```

```
Processus Prod
Début
α : <Produire un message>
    ENQ(S1)
    ENQ(m)
    nm := nm + 1
    Si nm<N alors DEQ(S1) finsi
    DEQ(m)
    ENQ(mutex)
    <Dépôt du message>
    DEQ(mutex)
    DEQ(S2)
    Aller à α
Fin
```

```
Processus Cons
Début
β : ENQ(S2)
    ENQ(m)
    nm := nm – 1
    Si nm>0 alors DEQ(S2) finsi
    DEQ(m)
    ENQ(mutex)
    <Prélever un message>
    DEQ(mutex)
    DEQ(S1)
    <Traitement du message>
    Aller à β
Fin
```

Exercice 7 :

Implémenter les verrous à l'aide des sémaphores de Dijkstra.

Solution :

Pour implémenter un verrou v on lui associera les variables suivantes :

```
var S,mutex : sémaphore init 0,1 ;
    n : entier init 0 ;
    b : boolean init valeur_initiale(v) ;
```

les primitives ENQ et DEQ peuvent s'exprimer comme suit :

```
ENQ(v) :
  Début
    P(mutex)
    Si b alors
      b := faux
      V(mutex)
    Sinon
      n := n + 1
      V(mutex)
      P(S)
    FinSi
  Fin

DEQ(v) :
  Début
    P(mutex)
    Si n>0 alors
      n := n – 1
      V(S)
    Sinon
      b := vrai
    FinSi
    V(mutex)
  Fin
```

3. Exercices Supplémentaires :

Exercice 1 :

Soit deux classes de processus L et R qui s'exécutent comme suit :

```
var Sa,Sb : sémaphore init n,n;   /* n >= 1 */
Processus i de L                  Processus j de R
Début                             Début
  P(n,Sb)                           P(1,Sb)
  P(1,Sa)                           P(n,Sa)
  V(n-1,Sb)                          Action A
  V(1,Sb)                           V(n,Sa)
   Action A                         V(1,Sb)
  V(1,Sa)                         Fin
Fin
```

1) Discuter les propriétés :
   - d'absence d'interblocage ,
   - d'absence de famine (équité),
   - de priorité ;

   pour les processus de L et R.
2) Même question si l'on remplace , dans l'algorithme des L , les deux instructions V(n-1,Sb) et V(1,Sb) par V(n,Sb) uniquement.

Exercice 2 :

Implémenter les primitives de Vantilborgh en utilisant les sémaphores de Dijkstra.

Exercice 3 :

Résoudre l'exercice 3 (sur la précédence de tâches) de la leçon 2 en utilisant les sémaphores de Patil.

Exercice 4 :

Résoudre le problème des lecteurs / rédacteurs (1.3 de la leçon 2) en utilisant les verrous (sémaphores binaires).

Exercice 5 :

Implémenter les sémaphores (simples) à l'aide des verrous.

Exercice 6 :

Lorsque la file d'attente d'un sémaphore est gérée en F.I.F.O on parle de définition forte des sémaphores. Lorsque la file est gérée comme un ensemble (choix aléatoire du processus à débloquer) on parle de définition faible des sémaphores.
Trouver une solution , qui soit équitable, au problème de l'exclusion mutuelle en utilisant la définition faible des sémaphores.

Exercice 7 :

On définit une primitive P' :

P'(s,p) : s : sémaphore , p : paramètre = 0 . . 1

P' fonctionne de la même manière que le P simple ; sauf que lorsque le processus qui l'exécute se bloque, il est inséré en tête de s.file lorsque p = 0 , et en queue de s.file lorsque p = 1.

On considère une ressource R qui peut être utilisée par au plus N processus simultanément.
Un processus voulant utiliser R attend si celle-ci n'est pas disponible.

1) Ecrire l'algorithme d'un processus ayant à utiliser R en utilisant les primitives P' et V.
2) On considère maintenant qu'il y a deux classes C1 et C2 de processus ayant à utiliser R.
   Il n'y a pas d'exclusion mutuelle entre C1 et C2, par contre les processus de C1 sont prioritaires sur ceux de C2 pour l'accès à R.
   Ecrire les algorithmes des processus de C1 et C2 en utilisant les primitives P' et V.

N.B. : Dans 1) et 2) on suppose que la primitive V réveille toujours le processus qui se trouve en tête de file, lorsque celle-ci n'est pas vide.

Bibliographie

André F. , Herman D. , Verjus J.-P.
« Synchronisation de programmes parallèles »
Dunod , 1984

Beauquier J. , Bérard B.
« Systèmes d'exploitation , concepts et algorithmes »
Ediscience international , 1993

Ben-Ari M.
« Processus concurrents » *(traduction française)*
Masson , 1986

Bouzefrane S.
« Les systèmes d'exploitation »
Dunod , 2003

Crocus
« Systèmes d'exploitation des ordinateurs »
Dunod , 1975

Kaiser C.
« Cours Systèmes informatiques B » (polycopié)
Conservatoire National des Arts et Métiers - Paris , 2000

Krakowiak S.
« Principes des systèmes d'exploitation »
Dunod , 1987

Lucas M.
« Parallélisme » (support de cours)
Ecole Nationale Supérieure de Mécanique de Nantes , 1989

Padiou G. , Sayah A.
« Techniques de synchronisation pour les applications parallèles »
Cepadues , 1990

Peterson J.L. , Silberschatz A.
« Operating system concepts »
Addison Wesley , 1985

Raynal M.
« Algorithmique du parallélisme – le problème de l'exclusion mutuelle »
Dunod , 1984

Schiper A.
« Programmation concurrente »
Presses polytechniques romandes , 1986

Tanenbaum A.S.
« Operating systems , design and implementation »
Prentice-Hall , 1987

Thorin M.
« Parallélisme : génie logiciel temps réel »
Dunod , 1990