Chapitre 2: Synchronisation des processus

1. Introduction

Pour analyser le fonctionnement d'un système d'exploitation (SE), on est amené à considérer l'ensemble des activités que gère ce système. Ces activités appelées aussi processus sont des entités de base évoluant dans ce système. Un processus peut être défini comme l'exécution d'un programme comportant des instructions et des données. C'est un élément dynamique créé à un instant donné et disparaît en général au bout d'un temps fini après avoir passé par différents états pendant sa durée de vie.

Les processus, multiples dans un SE, n'évoluent pas toujours de manière indépendante. Ils coopèrent pour réaliser des taches communes. Pour mettre en œuvre cette coopération, des mécanismes dits de synchronisation doivent être offerts par le SE. Ces mécanismes permettent d'ordonner l'exécution des processus qui coopèrent.

2. Partage de ressources et de données

L'exécution d'un processus nécessite un certain nombre de ressources. Ces ressources peuvent être logiques ou physiques. Les ressources physiques sont la mémoire, le processeur, les périphériques etc. Les ressources logiques peuvent être une variable, un fichier, un code etc. Les ressources peuvent être consommables ou réutilisables. Certaines ressources peuvent être utilisées en même temps (ou partagées) par plusieurs processus. Dans ce cas, elles sont dites partageables ou à accès multiples. D'autres ne peuvent être utilisées que par un seul processus à la fois. Dans ce dernier cas, il est nécessaire d'ordonner l'accès à ce type de ressources pour éviter des situations d'incohérences. Le mode d'allocation d'une ressource peut être implicite ou explicite. Ce dernier cas, une demande explicite peut être exprimée ou une synchronisation quant à l'accès est introduite.

3. Exclusion mutuelle

Quand il s'agit de partager une ressource à un seul point d'accès, les processus doivent l'utiliser de manière séquentielle selon un ordre défini par le mécanisme de synchronisation utilisé. La portion de code utilisant la ressource est appelée section critique (SC). La ressource est appelée ressource critique. Le mécanisme utilisé pour la synchronisation des différents processus quant aux accès à cette ressource est appelé mécanisme d'exclusion mutuelle (EM).

3.1 Problème de la section critique

Considérons un système composé d'un ensemble de processus $\{ P_0, P_1, ...P_{n-1}\}$.

P_0 P_1…… P_{n-1}

SC_0 SC_1 SC_{n-1}

Prenons l'exemple d'une variable commune. Les sections critiques manipulent cette variable. Quand un processus est en exécution dans sa section critique, aucun autre n'est permis à exécuter sa section critique à ce moment.

Il est donc nécessaire de concevoir un protocole permettent aux processus de s'exclure mutuellement quant à l'utilisation simultanée de la ressource critique.

La forme générale d'un processus qui utilise une ressource critique devient :

P_i
-
Protocole d'entrée
<Sc_i >
Protocole de sortie
-

3.2. Conditions de la section critique

Toute solution adoptée au problème de la section critique doit satisfaire les conditions suivantes :
1- Exclusion mutuelle : Si un processus P_i est dans sa SC, aucun autre ne doit être dans sa SC.
2- Progression : Si un processus opère en dehors de sa SC, il ne doit pas empêcher un autre d'entrer à sa SC dans le cas de besoin. En d'autres termes, il ne décidera pas lequel qui va opérer.
3- Blocage mutuel : Si deux processus désirent entrer en SC, au moins un des deux devra pouvoir y accéder.
4- Attente bornée : Un processus ayant demandé d'entrer en SC doit y accéder au bout d'un temps fini. Autrement dit, les processus permis d'entrer en SC doivent y accéder un nombre de fois limité si un processus a demandé sa SC.
5- Un processus qui est en SC doit sortir au bout d'un temps fini. Cette condition implique seulement le processus.

4. Mécanismes d'exclusion mutuelle

4.1. Hypothèses

- On suppose que le système contient *n* processus, $P_0, P_1, \ldots P_{n-1}$, qui évoluent de manière parallèle et asynchrone.
- Les vitesses d'exécution des processus sont quelconques. Cela suppose que la machine peut disposer de plusieurs processeurs.
- Les instructions de base du langage machine sont exécutées de manière atomique.

4.2. Solutions sans support matériel : Variables d'états

Ces solutions consistent à utiliser des variables communes partagées entre les processus selon des protocoles que nous allons présenter de manière progressive. Pour simplifier la présentation, on considère uniquement deux processus P_i et P_j.

Solution1 :

```
                    Var tour :=i ou j;
Processus Pi ;
Début
   Répéter
      Tant que (tour<>i) Faire rien_faire Fait ;
      < S.C i>
      tour :=j ;
      <Section Restante i>
   Jusqu'à Faux ;
Fin.
```

Interprétation : Quand le processus *i* désire d'entrer en S.C, il teste la valeur de *tour* si elle est égale à son identité. Dans le cas positif, il accède à sa SC et à la sortie, il affecte l'identité de son collègue à *tour*.

Déroulement :

- L'exclusion mutuelle est assurée car l'entrée en SC est conditionnée par la valeur de *tour* qui doit être identique à l'identité du processus. D'autre part, *tour* n'est modifiée qu'à la sortie de la SC.
- La solution engendre l'alternance quant à l'accès à la SC.

Cette solution indique seulement lequel des processus est en SC.

Solution2 :

La solution précédente ne garde pas des informations sur l'état d'un processus par rapport à la SC. La nouvelle solution remplace la variable *tour* par un tableau qui indique cet état.

```
        Var drapeau : Tableau [0 :1] de booléen :=faux ;
Processus Pi ;
Début
   Répéter
      drapeau[i] :=vraie ;
      Tant que drapeau[j] Faire rien_faire Fait ;
      < S.C i>
      drapeau[i] :=faux ;
      <Section Restante i>
   Jusqu'à Faux ;
Fin.
```

Interprétation : Quand un processus P_i désire entrer en S.C, il met *drapeau[i]* à *vrai*. Il teste si l'autre n'est pas intéressé par la SC. Si c'est le cas, il entre en S.C ; sinon, il boucle sur ce test tant que l'autre est en SC. A sa sortie de la SC, P_i remet son drapeau à *faux*.

Analyse :

- L'exclusion mutuelle est assurée car chaque processus *i* avant de tester *drapeau[j*], il prend la précaution de mettre *drapeau[i]* à *vrai* ; et de plus la mise à jour de *drapeau[i]* n'est réalisée qu'à la sortie de la SC.
- Le blocage mutuel n'est pas évité dans le cas où les deux *drapeaux* sont mis à *vrai* avant qu'aucun des deux processus ne teste la condition d'accès à la SC.

La solution dépend des vitesses relatives des deux processus.

Solution3 :

En combinant les avantages des deux solutions précédentes, l'algorithme suivant résout le problème de la SC [Peterson 81].

```
Var   drapeau : Tableau[0 :1] de booléen :=faux ;
      tour : entier :=<valeur arbitraire i ou j>
Processus Pi ;
Début
   Répéter
      drapeau[i] :=vraie ; tour :=j ;
      Tant que (drapeau[j]=vrai) et (tour=j) Faire rien_faire Fait ;
      < S.C i>
      Drapeau[i] :=faux ;
      <Section Restante i>
   Jusqu'à Faux ;
Fin.
```

Interprétation : Si un processus *Pi* désire entrer en SC, il met *drapeau[i]* à *vraie* et *tour* à *j*. Pour entrer en SC, il faut que *drapeau[j]= faux* (c.à.d l'autre processus n'est pas intéressé par la SC) ou *tour<>j* (c.à.d l'autre processus arrive ''après'' *Pi*). Si les deux processus arrivent en même temps, celui qui modifie *tour* en premier entre en SC. Donc, *tour* tronche au profit d'un des deux processus pour l'accès à la SC. Remarquons que quand un processus est loin de sa SC, son *drapeau* est à *faux,* ce qui permet à l'autre d'accéder à sa SC s'il le souhaite.

Analyse :

- L'exclusion mutuelle est assurée. Chaque processus *Pi* entre en SC si *drapeau[i]=faux* ou *tour=i*. D'autre part, si les deux processus sont en SC en même temps, *drapeau[i]=drapeau[j]=vraie*. Par conséquent, pour franchir la boucle il faut de tour soit égale à l'identité du processus, ce qui veut dire que l'autre processus l'a modifié en sa faveur. Ceci implique que les deux ne franchissent pas en même temps la boucle puisque *tour* prend une seule valeur *i* ou *j*. Remarquons que le processus qui modifie *tour* en premier, par exemple *Pi (i.e. tour=j),* va franchir la boucle après une éventuelle attente le temps que l'autre (i.e. *Pj*) modifie *tour* à l'identité du premier (i.e. *tour=i*). Ainsi, ce dernier processus se ''condamne'' en attendant au niveau de la boucle.
- La progression est assurée car si un processus Pi désire seul entrer en SC, *drapeau[j]=faux,* donc il accède sans contraintes.
- Le blocage mutuel est évité car si les deux désirent entrer en SC, *tour* tronche au profit d'un des deux processus pour l'accès à la SC (voir le déroulement pour la condition de l'EM).
- L'attente bornée est assurée car si un processus *Pi* est en attente de la SC sachant que *Pj* est en SC, alors *tour=j* et *drapeau[i]=drapeau[j]=vraie*. En sortant de la SC, *Pj* met *drapeau[j]=faux* et quelque soit sa vitesse c'est *Pi* qui va rentrer soit en constatant que *drapeau[j]=faux ;* sinon, si *Pj* est rapide et postule une autre fois, il met *tour=i*, ce qui donne la possibilité à *Pi* de franchir la boucle avec succès.

4.3. Solutions matérielles

Différentes solutions se basant sur le matériel sont développées pour résoudre le problème de la SC. Ces solutions rendent la tache de programmation aisée.

- Une classe de solutions se base sur le masquage des interruptions quand il s'agit d'utiliser la SC et le démarquage à la fin. Sur un multiprocesseur, même si le masquage est global, c'est à dire sur toutes les unités centrales, cela n'est pas suffisant car des processus déjà en exécutions sur différents processus peuvent utiliser simultanément la ressource.
  Sur un mono processeur, le masquage permet de résoudre le problème de la SC, cependant, certaines taches fonctionnant à l'aide des interruptions sont inhibées, par exemple l'horloge. D'autre part, si un processus autre n'est pas intéressé par cette ressource, son exécution est retardée inutilement.
- D'autres solutions utilisent des instructions fournies par la machine qui sont de nature indivisibles, telles que Test And Set et Swap.

- L'instruction *Test-And-Set*

```
Fonction Test-And-Set (var x : booléen) : booléen ;
   Début
      Test-And-Set :=x ;
      x :=vraie ;
   Fin ;
```

La variable *x* est verrouillée pendant l'exécution de cette fonction même s'il s'agit d'un multiprocesseur car le bus de données subit ce même verrouillage.

Cette fonction est utilisée come suit pour résoudre le problème de l'exclusion mutuelle. La solution obtenue est valable pour un nombre indéterminé de processus.

Var lock :=booléen :=faux ;
Processus Pi ;
-
Répéter
Tantque Test-And-Set(lock) Faire rien Fait ;
<SCi>
lock :=faux ;
<Section Restante i>
Jusqu'à Faux ;
-

Analyse :

Un premier processus, qui arrive, exécute la boucle en initialisant *Test-And-Set* à *faux* (qui est la valeur de *lock*) puis met *lock* à *vrai* et entre en SC. Tous les autres processus qui arrivent plus tard séquentiellement se bloquent en attendant la sortie du premier processus qui remet *lock* à *faux* pour permettre à un seul processus en attente d'y accéder et ainsi de suite. Notons que si deux processus ou plus essayent d'exécuter cette fonction en même temps, cette simultanéité se traduit automatiquement en une séquentialité.
De l'étude précédente, on constate facilement que la progression est assurée et que le blocage mutuel est évité. L'attente bornée n'est pas assurée : Si le processus qui est en SC sort et arrive à chaque fois à récupérer la valeur *faux* de *lock* avant celui ou tous ceux en attente, il accédera infiniment en SC en bloquant les autres. On admet que pratiquement cette situation ne peut pas se produire.

- L'instruction Swap : Le rôle de l'instruction *Swap* est d'échanger les contenus de deux variables comme suit :

Primitive Swap (var X, Y : booléen) ;
Var Z : booléen ;
Début Z :=X ; X :=Y : Y :=Z Fin ;

L'utilisation de cette instruction dans le cadre d'un protocole d'exclusion mutuelle est comme suit :

Var Lock : booléen :=faux ;

Processus Pi ;
Var key_i : booléen ;
-
Répéter
Répéter
key_i :=vrai ;
Swap(Key_i, Lock) ;
Jusqu'à(Key_i=faux) ;
<SC_i>
Lock :=faux ;
<Autres instructions>
Jusqu'à Faux
Fin.

4.4- Les sémaphores

Les solutions précédentes, selon le cas, possèdent trois inconvénients suivants :

- Elles ne peuvent pas être généralisées à un nombre indéterminé de processus,
- Elles ne s'adaptent pas à un problème général de synchronisation,
- Elles posent le problème d'attente active tant qu'un processus est en SC, ce qui consomme inutilement du temps machine.

Pour résoudre ce dernier problème, il faut introduire un mécanisme qui permet de bloquer un processus si la condition d'entrée en section critique n'est pas remplie et de le réveiller pour y accéder le plus tôt que la condition est satisfaite.

4.4.1- Définition [Dijkstra 65]

Un sémaphore *S* est une variable entière associée d'une file d'attente *f(S)*. La variable peut prendre des valeurs *positives*, *nulles* ou *négatives*. Cependant, sa valeur initiale est toujours *>=0*. La politique de gestion de la file est quelconque, mais en général, elle est FIFO. Le sémaphore est géré par les trois primitives suivantes :

```
Primitive Init;          Permet de mettre une valeur initiale >=0 dans S (avant le
   Début                 lancement des processus qui l'utilisent)
      S :=n ;
      F(S) := Φ
   Fin ;

Primitive P(S);
   Début
      S :=S-1 ;
      Si (S<0) Alors
         Etat(pi) :=Bloqué ; /* pi est le processus qui exécute P(S) */
         <Mettre le processus pi dans la file f(S) >;
         [Appeler le Scheduler]
      Fsi ;
   Fin ;

Primitive V(S);
   Début
               S :=S+1 ;
      Si (S<=0) Alors
         <Choisir un processus de la file f(S) : soit pj >;
         Etat[j] :=prêt ;
         [Appeler le Scheduler]
      Fsi
   Fin ;
```

Deux primitives *bloquer (p)* et *réveiller (p)* du système d'exploitation permettent de gérer l'attente passive des processus.

Chaque processus exécutant *P(S)* décrémente la valeur de *S* de *1*. Ensuite, il teste la valeur de *S*. Si celle-ci est *négative*, il est mis en file d'attente associée à *S* et le Scheduler est appelé à choisir un autre processus selon sa politique de gestion du processeur.
Remarquons que si la valeur de *S* est *négative*, sa valeur absolue est le nombre de processus bloqués dans la file *f(S)*. Et si la valeur est *positive*, elle indique le nombre de processus qui peuvent franchir la primitive *P(S)* sans être bloqués.

L'exécution de *V(S)* implique l'incrémentation de la valeur de *S* de *1*. Puis, si la valeur est toujours <=*0* (cela veut dire que la file n'est pas vide), un processus est mis à l'état prêt. Le scheduler est ensuite invoqué pour choisir un autre processus pour exécution : Il peut être le processus en cours, le processus extrait de *f(S)* ou un autre processus existant dans la file des processus prêts.

Notons que le scheduler n'influe pas sur le mécanisme de synchronisation : Une solution de synchronisation utilisant les sémaphores fonctionne pour toute technique de gestion du processeur. Ainsi, son évolution et son analyse sont faites en faisant abstraction de la manière dont le processeur est géré.

Enfin, les primitives *P* et *V* sur un même sémaphore s'exécutent de manière atomique (voir section 4.4.5) pour préserver la cohérence de la variable *S*.

4.4.2- Sémaphores binaires

Un sémaphore binaire est un sémaphore dont la valeur initiale est *1*. Il est utilisé en général dans l'exécution mutuelle. La forme générale d'un processus qui résout ce problème est :

```
Processus Pi ;
   -
   Début
      -
      P(S);
      <SCi>
      V(S) ;
      -
   Fin.
```

Un processus qui exécute en premier *P(S)* met *S* à *0* et accède à la SC. Le deuxième qui arrive et exécute *P(S)* met *S* à *–1* et se bloque dans f(S). Tous les autres qui arrivent par la suite, sachant que le premier est toujours dans la SC, se bloquent de même et dans l'ordre de leurs arrivées. $/S/$ indique le nombre de processus bloqués. Quand le Premier processus sort de la SC, il incrémente *S* et réveille le deuxième et ainsi de suite. Quand le dernier en attente sort de la SC, il met *S* à *1* (état de repos). Notons que la concurrence, quant à l'exécution de *P(S*), se traduit automatiquement en une séquentialité. Les quatre conditions de la SC sont vérifiées.

4.4.3- Sémaphores privés

La valeur initiale du sémaphore est à *zéro*. On dit que *S* est privé à un processus (ou à un groupe de processus), s'il est le seul (ou les seuls) à pouvoir exécuter *P(S) et seulement P(S)*, et les autres processus ne peuvent exécuter que *V(S)*. Ce type de sémaphore est utilisé quand un processus désire se bloquer volontairement.

Exemple 1 : Deux processus coopèrent pour un travail en parties. *P1* doit réaliser la première partie avant que *P2* n'entame la seconde, la solution sera alors comme suit :

Sémaphore S:=0 ;

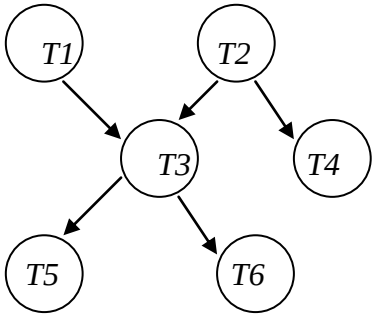
```
Processus P1;                    Processus P2 ;
      -                                -
   Réaliser le premier travail      P(S) ;
   V(S) ;                           Réaliser le deuxième travail
      -                                -
```

Exemple 2 : Représentation d'un graphe de précédences non proprement lié à l'aide des sémaphores privés et l'instruction *Parbegin Parend*. Le graphe ci-dessous est non proprement lié, son expression nécessite l'utilisation des sémaphores selon la solution ci-jointe.

Méthode à suivre :

- Supprimer le nombre minimum de précédences pour rendre le graphe proprement lié.
- Exprimer le graphe obtenu à l'aide de Parbegin Parend.
- Remettre une par une, les précédences supprimées
- A chaque précédence rétablie, ajouter V(S) juste à la fin de l'exécution de la tache d'où l'arc part et un P(S) juste avant la tache où l'arc arrive.

Attention, l'ajout de ''Début'' et ''Fin'' peuvent être nécessaires afin de respecter les précédences du graphe original.




```
Programme Prg;
Début
   Parbegin
        Début T2 ; V(S) ; T4 Fin ;
        Début T1 ; P(S) ; T3 ;
              Parbegin T5 ; T6 Parend ;
        Fin ;
   Parend ;
Fin ;
```

Remarquons que différentes solutions sont possibles. On peut supprimer par exemple la précédence T1 vers T3 ou celle de T2 vers T4.

4.4.4- Sémaphores compteurs

Dans ce cas, la valeur initiale du sémaphore est n *>1*. Il peut être utilisé quand il s'agit d'accéder à une ressource à n points d'accès.

Le modèle d'un processus est :

```
Processus Pi ;
   -
   P(S) ;
   < Accès à la ressource partagée>
   V(S) ;
   -
```

Les *n* premiers processus franchissent *P(S)* sans se bloquer; le (*n+1*) ieme processus, si les *n* premiers sont en cours, se bloque car *S=-1*, de même que les autres qui arrivent par la suite. Quand un processus termine, il libère le premier bloqué. Ainsi, à tous moment au plus *n* processus partagent en même temps la ressource.

Application 1 : Accès à un parking possédant *n* places et avec une porte qui laisse passer une seule voiture à la fois, de même que pour la porte de sortie.

Sémaphore *S :=n ; mutex1, mutex2 :=1 ;*

-
P(S) ;

P(mutex1) ;
<Entrer$_i$ au parking>
V(mutex1) ;
<Stationner$_i$>
P(mutex2) ;
<Sortir$_i$>
V(mutex2) ;

V(S) ;
-

Quand une voiture désire accéder au parking, elle exécute *P(S)*. S'il y a de la place, elle s'engage et empreinte seule la porte d'entrée et stationne pour une durée désirée. Dans le cas contraire, elle attend au niveau de *S*. A la sortie, elle passe par la porte de sortie en exclusion mutuelle puis libère une éventuelle voiture en attente (si le parking était plein, *S<=0*) en exécutant *V(S)* sinon libère sa place (*S* représente le nombre de places libres). Ainsi, à tout moment, il y a au plus *n* parking voitures dans le parking.

Application 2 : Le barbier.

Le barbier a pour rôle évidemment de raser des clients. Pour cela, il dispose de deux salles : la salle d'attente de capacité N dans laquelle les clients attendent leurs tours et le barbier attend l'arrivée des clients ; la salle de rasage dans laquelle le barbier rase à chaque fois un client. Une grande porte coulissante sert en commun comme porte d'entrée à la salle d'attente et comme porte de passage entre la salle d'attente et la salle de rasage de telle sorte qu'une seule porte est utilisable à la fois. Initialement, le barbier est en attente sur une chaise dans la salle d'attente. Le premier client qui arrive passe directement à la salle de rasage avec le barbier qui est réveillé. Un client qui ne trouve pas le barbier en attente se met sur une chaise en attendant son tour. Une fois rasé, le client en cours sort par une porte de sortie et le barbier revient à la salle d'attente pour chercher un autre client s'il y a lieu sinon se met en attente.

Solution

Var Sémaphore Br :=0 ; Cl :=0 ; S :=N ; mutex:=1;

Processus Barbier ;	*Processus Client ;*
Début	*Début*
Répéter	*P(S) ;*
P(Br) ;	*P(mutex) ;*
V(Cl) ;	*<Entrer en salle d'attente>*
P(mutex) ;	*V(mutex) ;*
<Passer à la salle de rasage>	*V(Br) ;*
V(mutex) ;	*P(Cl) ;*
<Raser un client>	*P(mutex) ;*
P(mutex) ;	*< Passer à la salle de rasage>*
< Revenir à la salle d'attente>	*V(mutex) ;*
V(mutex) ;	*V(S) ;*
Jusqu'à Faux ;	*< Se fait raser>*
Fin.	*<Partir>*
	Fin.

4.4.5- Implémentation des sémaphores et difficultés

Il s'agit de rendre indivisible l'exécution des opérations *P()* et *V()* et d'interdire l'utilisation du sémaphore en dehors de ces deux primitives.

- Sur un monoprocesseur : L'indivisibilité peut être mise en œuvre en masquant les interruptions pendant leur exécution ou en rendant ces opérations comme primitives du système d'exploitation.

- Sur un multiprocesseur : On utilise l'attente active avec l'une des instructions indivisibles (*TAS* ou *SWAP*). L'attente est limitée par le temps d'exécution de la primitive jusqu'au blocage ou à la sortie sans blocage.

L'utilisation des sémaphores pose principalement deux problèmes : L'interblocage et la famine :

- **L'interblocage**

Exemple :

Sémaphore R1, R2 :=1 ;

Processus P1 ;	*Processus P2 ;*
-	-
P(R1) ;	*P(R2) ;*
P(R2) ;	*P(R1) ;*
-	-
V(R2) ;	*V(R1) ;*
V(R1) ;	*V(R2) ;*
-	-

Supposons l'ordre d'exécution suivant :

P1 : P(R1) Franchis avec R1 =0
P2 : P(R2) Franchis avec R 2=0
P1 : P(R2) Se bloque avec R2=-1
P2 : P(R1) Se bloque avec R1=-1
➔ Interblocage.

- **La famine :** Elle a lieu par exemple dans le cas où la file d'un sémaphore est gérée de manière LIFO ou suite à la mal utilisation des sémaphores.

4.4.6- Autres constructions de P et V

Les deux primitives peuvent être implémentées comme suit :

Initialement, S>=0 ;

```
Primitive P(S) ;
   Début
   Si (S>0) Alors S :=S-1
        Sinon
          Etat(pi) :=Bloqué ; /* pi est le processus qui exécute P(S) */
          <Mettre le processus pi dans la file f(S) >;
          [Appeler le Scheduler]
     Fsi ;
   Fin ;

Primitive V(S) ;
   Début
      Si (non vide f(S) ) Alors
          <Choisir un processus de la file f(S) : soit pj >;
          Etat[j] :=prêt ;
          [Appeler le Scheduler]
                Sinon S :=S+1
      Fsi
   Fin;
```

Un processus exécutant *P(S)* teste d'abord la valeur de *S*. Si elle est *nulle*, il ne décrémente pas *S* et il se met en attente. Remarquons que le sémaphore prend toujours des valeurs *>=0*. Quand *S=0*, le nombre de processus bloqués n'est connu qu'en consultant la file *f(S)*. Quand un processus exécute *V(S)*, il teste d'abord l'état de la file. Si elle n'est pas vide, il réveille un processus et *S* est toujours gardée à *0* ; sinon, le sémaphore est incrémenté (donc sa valeur devient *>0)*.

Les deux constructions des primitives *P* et *V* sont fonctionnellement les mêmes vis à vis de la synchronisation mais donnent des valeurs différentes au sémaphore dans le cas d'attente de processus. Dans la première construction, quand la valeur du sémaphore est *négative*, sa valeur absolue indique le nombre de processus bloqués. Dans la seconde, pour le même cas, la valeur reste *nulle* et le nombre de processus bloqués n'est connu qu'en consultant la file d'attente.

4.5- Autres mécanismes de synchronisation

La coopération des processus nécessite des mécanismes qui coordonnent leurs exécutions dans le temps selon la logique de la fonction commune à accomplir. Ces mécanismes doivent permettre, à la base, à un processus :

- de bloquer un autre ou de se bloquer lui même en attendant un signal d'un autre processus,
- d'activer un autre processus,

Dans ce dernier cas, le processus à activer peut déjà se trouver à l'état actif. Deux cas se présentent :

- Soit que le signal n'est pas mémorisé, et par conséquent, il est perdu si le processus ne l'attend pas.
- Soit qu'il est mémorisé et par conséquent le processus ne se bloquera pas lors de sa prochaine opération de blocage.

L'action sur un processus peut être :

- Direct : la désignation est faite à l'aide de son nom ou
- Indirect : l'identité du processus concerné n'est pas nécessaire.

4.5.1. Action directe

Deux primitives sont appliquées :

```
Bloquer(q) : Si non témoin(q)
               Alors état(q) :=bloqué
               Sinon témoin(q) :=faux
             Fsi ;

Réveiller (q) : Si état(q)=bloqué
                  Alors  état(q) :=prêt
                  Sinon témoin(q) :=vrai
                Fsi ;
```

Le blocage d'un processus est symbolisé par le changement de son état à *bloqué*. Le réveil du processus se fait en changeant son état à *prêt*. Si on désire mémoriser le signal d'activation alors que le processus est déjà actif, on utilise *témoin* qui indique ce fait par la valeur *vrai*. Dans ce cas, l'exécution prochaine de la primitive *bloquer* n'a alors aucun effet sur l'état du processus visé.

4.5.2. Action Indirecte

Les événements

Un événement est une abstraction d'une action qui se produit dans un système. Il peut être le résultat de l'exécution d'une instruction ou d'un groupe d'instructions reconnu comme tel [

]. Le concept d'événement peut être utilisé comme outil de synchronisation entre processus. Dans ce cas, il est représenté par un identificateur, et il est crée par une déclaration.

La synchronisation à l'aide des événements est mise en œuvre par les concepts d'*attente* et de *déclenchement* de l'événement. D'autre part, l'événement peut être mémorisé ou non mémorisé.

1- Evénements mémorisés : Il est représenté par une variable booléenne dont la valeur (*1* ou *0*) traduit le fait qu'il est mémorisé ou non. Un processus se bloque si et seulement si l'événement qu'il attend n'est pas mémorisé. Selon les systèmes d'exploitation, le déclenchement de l'événement débloque un ou tous les processus qui l'attendent et l'évènement est acquitté s'il y a au moins un processus qui l'attend.

Exemple1 : Expression d'un rendez-vous (RDV) de deux processus.

***Processus** P1 ;*	***Processus** P2 ;*
_	-
déclencher(e1) ;	*déclencher(e2) ;*
attendre(e2) ;	*attendre(e1) ;*
-	-

Exemple2 : Expression d'un RDV de Trois processus.

La solution suivante est déduite de celle appliquée à deux processus de manière intuitive.

***Processus** P1 ;*	***Processus** P2 ;*	***Processus** P3 ;*
_	-	-
déclencher(e1) ;	*déclencher(e2) ;*	*déclencher(e3) ;*
attendre(e2) ;	*attendre(e1) ;*	*attendre(e1) ;*
attendre(e3) ;	*attendre(e3) ;*	*attendre(e2) ;*
-	-	-

Cependant, cette solution n'est pas correcte. Supposons l'ordre suivant d'exécution des processus :

P1 : déclencher(e1) ;		
P1 : attendre(e2) ;		
	P2 : déclencher(e2) ;	
P1 : attendre(e3) ;		
	P2 : attendre(e1) ;	
	P2 : attendre(e3) ;	
		P3 : déclencher(e3) ;
P1 : -	*P2 : -*	
		P3 : attendre(e1) ;

Le processus 3 se bloquera indéfiniment sur *attendre (e1).* Le problème réside dans le fait qu'un processus arrivant au point de rendez-vous annonce son arrivée aux deux autres à travers le déclenchement d'un seul événement. Si aucun processus concerné par cet événement n'est bloqué actuellement, dès l'arrivée du premier pour l'attendre il le consommera et l'autre qui va l'attendre plus tard se bloquera indéfiniment. De ce fait,

l'annonce de l'arrivée au point de RDV doit être faite à chacun des processus concernés à l'aide d'un événement différent, d'où la solution suivante.

***Processus** P1 ;*	***Processus** P2 ;*	***Processus** P3 ;*
_	-	-
déclencher(e12) ;	*déclencher(e21) ;*	*déclencher(e31) ;*
déclencher(e13) ;	*déclencher(e23) ;*	*déclencher(e32) ;*
attendre(e21) ;	*attendre(e12) ;*	*attendre(e23) ;*
attendre(e31) ;	*attendre(e32) ;*	*attendre(e13) ;*
-	-	-

Application : Généraliser ce RDV à *n* processus.

2- Evénements non mémorisés *:* Un événement émis alors qu'aucun processus ne l'attend est perdu. Dans le cas où un ou plusieurs processus sont bloqués dans l'attente d'un événement, tous ces processus sont débloqués lors de son déclenchement. Ce type d'événement est utilisé dans la commande de procédés industriels pour traiter des informations qui deviennent très rapidement caduques. Cependant, son utilisation est délicate car il dépend des vitesses relatives d'exécution des processus.

Remarque : Le raisonnement sur les événements peut être développé en considérant qu'un processus attende sur une condition booléenne constituée de plusieurs événements.

Exemple : *attendre(e1 ou e2)* *attendre(e1et e2 ou e3)*

D'autres part, le concept d'événements peut être utilisé pour modéliser le comportement d'un programme et notamment les programmes distribués.