CHAPITRE 4 :

LA COMMUNICATION ET LA SYNCHRONISATION INTERPROCESSUS

Objectifs spécifiques

- Comprendre la problématique de concurrence interprocessus
- Connaître les primitives de communication interprocessus
- Connaître la notion de section critique et de l'exclusion mutuelle
- Connaître quelques solutions pour la mise en œuvre de l'exclusion mutuelle (attente active et attente passive)

Eléments de contenu

I. Introduction à la concurrence interprocessus

II. Communication interprocessus

III. Synchronisation interprocessus

1. Section critique et exclusion mutuelle

2. Solution mettant en œuvre l'exclusion mutuelle

Volume Horaire :

Cours : 6 heures

TD : 3 heures

4.1 Introduction

Lorsqu'on observe un écran, il apparaît clairement que l'ordinateur mène plusieurs activités en parallèle : une horloge, par exemple, affiche l'heure pendant qu'on utilise un traitement de texte. Mieux encore on apprécie de pouvoir imprimer un document tout en continuant à écrire un autre texte. Ces activités sont gérées, en apparence, simultanément alors que la machine ne dispose que d'un processeur unique. Ces processus échangent des informations: ainsi le traitement de texte déclenche régulièrement une copie sur disque qui sauvegarde le texte écrit. Ce logiciel consulte donc l'horloge afin de déclencher ce mécanisme au moment voulu.

Ce chapitre est consacré aux notions de communication et synchronisation entre processus.

4.2 Communication interprocessus

Les processus ont besoin de communiquer, d'échanger des informations de façon plus élaborée et plus structurée que par le biais d'interruptions.
Les fichiers constituent le mécanisme le plus fréquemment utilisé pour le partage d'information : les informations écrites dans un fichier par un processus peuvent être lues par un autre processus. Le volume d'information pouvant être partagé est uniquement limité par la capacité de taille de fichier du système.
Les emplacements de mémoire partagés sont une autre solution. Il existe également des mécanismes d'échange de la mémoire entre processus.
Un mécanisme de signalisation peut également être utilisé pour communiquer des informations. Il en existe deux types : la signalisation directe et la signalisation indirecte, par le biais d'une boîte aux lettres.

Prenons par exemple un système doté de deux primitives de signalisation.

- Send-signal(pid, signalId) : envoie un numéro de signal, signalId au processus ayant le numéro d'identification pid.
- React-signal(function, signalId) : configure le processus pour qu'à la réception d'un numéro de signal signalId, celui-ci réponde en passant à la fonction function.

En associant une signification à différents numéros de signaux, les processus peuvent se communiquer des informations entre eux.

Nombreux systèmes d'exploitation prennent en charge un système de messagerie pour l'envoi de messages entre processus. Voici certains facteurs qui entrent en compte lors de la conception d'un tel système :

- **A quel genre d'objets est envoyé un message ?** Il peut s'agir d'un processus, d'une boîte aux lettres ou d'un tampon de communication ; l'objet peut être identifié par un nom ou un numéro d'identification.
- **Comment les processus sont-ils associés aux objets du message ?** Le système d'exploitation peut exiger que toute communication soit précédée d'une primitive qui établit une connexion avec l'objet de destination. Tout processus peut être autorisé à communiquer avec l'objet de destination, mais il est possible de mettre en places certaines restrictions. Si l'objet de destination est une boîte aux lettres ou un tampon de communication, les objets peuvent être crées automatiquement, mais il peut exister un nombre fixe d'objets prédéfinis ou encore y avoir une primitive pour créer l'objet.
- **Combien de processus peuvent partager un même objet de messagerie ?** Le nombre de processus en mesure de lire et d'écrire dans l'objet du message peut faire l'objet de diverses restrictions. Ainsi, un certain nombre de processus peuvent être autorisés à écrire dans l'objet, avec un seul pouvant lire les messages depuis l'objet.

- **L'objet de messagerie est-il bidirectionnel ou unidirectionnel ?** Un processus peut être limité à un accès en lecture ou à un accès en écriture, mais pas aux deux. L'accès bidirectionnel peut être autorisé, permettant aux processus de lire et d'écrire à la fois dans un objet de message.
- **A combien d'objets de messagerie un processus peut accéder ?** Le système d'exploitation peut imposer des limites à l'échelle du système ou restreindre le nombre d'objets pouvant être utilisé pour la communication entre deux processus.
- **Les messages sont-ils de taille fixe ou variable ?** S'ils sont de taille variable, le système d'exploitation peut imposer une taille maximale.
- **Combien de messages peuvent être stockés dans un objet de message ?** La mise en mémoire tampon des messages peut être inexistante, bornée, non bornée ou basée sur un mécanisme d'accusé de réception. Le premier cas nécessite que l'opération d'envoi de bloque le processus d'envoi jusqu'à la réception du message par le récepteur. Ce type d'opération est appelé rendez-vous. La mise en tampon bornée ne bloque l'opération d'envoi que si la mémoire tampon est pleine. Avec la mise en tampon non bornée, l'émetteur n'est jamais bloqué lorsqu'il essai d'envoyer un message. La mise en tampon reposant sur un mécanisme d'accusé de réception. Tous deux sont libres de poursuivre leur exécution une fois la réponse du récepteur émise.
- **Les messages sont-ils envoyés par valeur ou par référence ?** Les messages envoyés par référence nécessitent moins de surcharge, mais ils peuvent être modifiés après leur envoi. Pour ce qui est des messages envoyés entre processus, le passage des messages par référence enfreint le principe selon lequel chaque processus doit recevoir un espace d'adressage indépendant.
- **Que se passe t-il si un processus se termine alors qu'un autre processus est bloqué, dans l'attente d'une action du processus qui vient de s'achever ?** Généralement la primitive à l'origine du blocage renvoie un statut d'erreur en indiquant un échec ou bien le processus bloqué se termine également.

Examinons les trois mises en œuvre d'un système de messagerie.

Communication directe entre les processus :

- Int send_process(pid, message) : envoie message au processus avec le numéro d'identification pid. Le message est copié dans la file de message du récepteur. Tout processus, quel qu'il soit, peut envoyer un message à tout autre processus. Cette primitive ne se bloque jamais, car un nombre illimité de messages peut résider dans la file du récepteur. Les messages sont des chaînes de n'importe quelle longueur. Cette primitive échoue si le processus de destination n'existe pas.
- Int receive_process(pid, message) : reçoit le prochain message du processus dont le numéro d'identification est pid et place ce message dans message. Les messages sont reçus dans l'ordre où ils sont envoyés. Si pid contient la valeur spéciale de 0, il reçoit un message de n'importe quel processus (en partant du principe qu'aucun processus ne peut avoir le numéro d'identification 0).

La valeur de retour de cette primitive est le numéro d'identification de processus du processus émetteur. Cette primitive échoue si un processus émetteur est spécifié n'existe pas.

Communication indirecte entre les processus par le biais d'une boîte de messages :

D'après le second schéma, le passage du message est réalisé indirectement par le biais d'une boîte aux lettres.

- int create_mailBox(mbx) : crée une boîte aux lettres appelée mbx. Cette primitive échoue si la boîte aux lettres existe déjà.
- int delete_mailBox(mbx) : supprime une boîte aux lettres appelée mbx. Cette primitive échoue si le processus qui émet la primitive n'est pas propriétaire de la boîte aux lettres ou si la boîte aux lettres n'existe pas.
- int send_mailBox(mbx, message) : envoie message dans boîte aux lettres mbx : le message est copié dans la boîte aux lettres. Tout processus peut envoyer un message à n'importe quelle boîte aux lettres. L'émetteur est bloqué jusqu'à réception du message. Les messages sont des chaînes de n'importe quelle longueur. Cette primitive échoue si la boîte aux lettres n'existe pas.
- int receive_mailBox(mbx, message) : reçoit le message en provenance de la boîte aux lettres mbx et place ce message dans message. Les messages sont reçus dans l'ordre de leur envoi. S'il n'y a aucun message dans la file, reste bloqué jusqu'à ce qu'un message soit envoyé. Cette primitive échoue si le processus qui émet la primitive n'est pas le propriétaire de la boîte aux lettres ou si cette dernière n'existe pas.

Communication indirecte entre les processus par le biais d'un canal de communication :

Le dernier exemple repose sur un canal de communication mis en tampon appelé tube (pipe). A la différence des autres exemples pour lesquels la destination du message était un numéro d'identification ou un nom pouvant être spécifié par tout processus quel qu'il soit, les primitives d'envoi et de réception utilisent un descripteur de canal de communication associé uniquement au processus ayant créé le tube et aux processus fils de ce dernier.

- Void create_pipe(pdr,pdw) : crée un tube, stocke la valeur d'un descripteur de lecture dans pdr et celle d'un descripteur d'écriture dans pdw. Cette primitive ne peut connaître d'échec.
- int close(pd) : pour les processus qui émettent cette primitive, ferme le descripteur pd qui peut être un descripteur de lecture ou d'écriture. Cette primitive échoue si le descripteur n'est pas ouvert.

- Int send_pipe(pdw, byte) : écrit byte dans le tube associé au descripteur de lecture pdw. Si le tube contient déjà 4096 octets, cette primitive se bloque jusqu'à ce qu'une opération de lecture fasse de la place pour d'autres octets dans le tube. Cette primitive échoue si le descripteur n'est pas ouvert ou si ce tube n'est ouvert à aucun procesus pour la lecture.
- Int receive_pipe(pdr,byte) : lit un octet byte depuis le tube associé au descripteur de lecture pdr. Les octets sont lus dans l'ordre dans lequel ils ont été écrits dans le tube. Si le tube est vide et qu'aucun processus n'a de tube ouvert pour l'écriture, renvoie une valeur spéciale indiquant que le tube est vide. Sinon, le processus récepteur doit se bloquer si le tube est vide. Cette primitive échoue si le descripteur n'est pas ouvert.

Les processus coopérants sont confrontés à deux grands problèmes : la famine et l'inter-blocage (deadlock). Tout comme algorithmes d'ordonnancement peuvent retarder indéfiniment l'exécution d'un processus, la nature exacte de la coopération entre les processus peut également conduire à la famine. Observons par exemple la mise en œuvre lettres et n'ayant pas pris en charge les requêtes de lecture dans l'ordre du premier entré, premier servi (FIFO). Si un processus émettait un flux constant de requêtes de lecture et si toutes ses requêtes étaient satisfaites en premier, il pourrait arriver que la requête d'un autre processus ne soit jamais satisfaite.

Les inter-blocages surviennent lorsque deux ou plusieurs processus se bloquent dans l'attente d'un événement, sous le contrôle de l'autre processus bloqué ; prenons l'exemple d'une application dans laquelle le processus A envoie des messages au processus B par l'intermédiaire d'une boîte aux lettres et que le processus B envoie des messages au processus A par le biais d'une seconde boîte aux lettres. Envisageons ce qui se passerait si les 2 processus essayaient de lire un message entrant avant d'envoyer un message. Tous les deux resteraient bloqués, dans l'attente d'un envoie de l'autre processus, condition qui ne peut jamais être remplie.

Outre la famine et l'inter-blocage, les processus qui partagent des données peuvent être confrontés aux problèmes d'incohérence des données. Prenons l'exemple de tout processus qui accomplissent chacun une tâche, puis qui essaient chacun d'incrémenter une variable partagée qui compte le nombre de tâches, puis qui essaient chacun d'incrémenter une variable partagée qui compte le nombre de tâches réalisées par le processus coopérant. Si la valeur du compteur est 10, celui-ci atteint 12 après l'incrémentation par les 2 processus. Toutefois, si les 2 processus accèdent à la valeur du compte en même temps, tous deux vont incrémenter 10 et par conséquent, stocker 11 dans la variable compteur. Le code ne fonctionne correctement que si aucun processus n'a accès à la variable partagée pendant qu'un autre processus tente d'incrémenter la variable.

4.3 Synchronisation Interprocessus

4.3.1 Les sections critiques

Une section critique (SC) est une séquence d'instructions manipulant des données partagées par plusieurs processus et qui peut produire des résultats imprévisibles lorsqu'elle est exécutées simultanément par de différents processus, on se donne comme exemple la lecture ou l'écriture de données partagées ceci est appelé accès concurrentiel. Une SC ne peut être commencée que si aucune autre SC du même ensemble n'est en cours d'exécution. Avant d'exécuter une SC, un processus doit s'assurer qu'aucun autre processus n'est en train d'exécuter une SC du même ensemble. Dans le cas contraire, il ne devra pas progresser, tant que l'autre processus n'aura pas terminé sa SC.

C'est en travaillant sur la section critique que l'on doit éviter les problèmes dans les mises en œuvre de partage de ressources (fichiers, mémoires, etc..) entre les processus. Il faut s'assurer que plusieurs processus ne puissent accéder à une section critique en même temps. Mais dans ce cas on ralentira et on limitera le travail de deux processus parallèles partageant une ressource.

Dans ce cas quatre conditions sont nécessaires :

1- deux processus ne peuvent être en même temps en section critique

2-Aucune hypothèse ne doit être faite sur les vitesses relatives des processus et sur le nombre de processeurs.

3- Aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres processus.

4- Aucun processus ne doit attendre trop longtemps avant d'entrer en section critique.

Pour résoudre ce problème de contrôle d'occupation de la zone critique, par un et un seul processus à la fois, il existe l'exclusion mutuelle.

4.3.2 L'exclusion mutuelle

L'exclusion mutuelle est une méthode qui permet de s'assurer que si un processus utilise une variable ou une ressource partagée, les autres processus seront exclus de la même activité.

Pour assurer l'exécution des sections critiques des processus en exclusion mutuelle il est nécessaire de trouver une solution qui permet d'éviter les conditions de concurrence.

4.4 Les solutions de mise en œuvre de l'exclusion mutuelle

4.4.1 Solution monoprocesseur : désactivation des interruptions

L'idée principale de cette solution est le masquage d'interruptions les commutations de processus qui pourraient violer l'exclusion Mutuelle des SC sont empêchées : chaque processus entrant dans la section critique désactive toutes les interruptions et les réactive à sa sortie de la section critique. Seule l'interruption générée par la fin du quantum de temps nous intéresse, il ne faut pas qu'un processus attende une interruption de priorité inférieure à celle générée par la fin du quantum de temps à l'intérieur de SC. Cette solution n'est pas applicable au sein d'un environnement multiprocesseur.

Inconvénients :

- Les interruptions restent masquées pendant toute la SC, d'où risque de perte d'interruptions ou de retard de traitement.
- Une SC avec while(1) bloque tout le système
- Les systèmes ne permettent pas à tout le monde de masquer n'importe comment les interruptions.

4.4.2 Variables de verrou

Un mécanisme proposé pour permettre de résoudre l'exclusion mutuelle d'accès à une ressource est le mécanisme de *verrou* (en anglais *lock*). Un verrou est un objet système représentée par une variable unique et partagée dont la valeur initiale est de 0. Cette solution consiste à résoudre le problème de partage de variables par l'utilisation du verrou. Un processus tentant à entrer en section critique doit tout d'abord tester la valeur du verrou si elle est égal à 0, le processus lui affecte 1 puis entre en SC ; sinon le processus doit attendre que le verrou passe de nouveau à 0. En sortant de la SC un processus doit libérer le verrou et ce par sa remise à 0. Si un ou plusieurs processus étaient en attente de ce verrou, un seul de ces processus est réactivé c'est lui qui reçoit le verrou et le met à 1.

1 : variable ou ressource partagée occupée : le verrou est non disponible => interdiction de l'entrée en SC.

0 : variable ou ressource partagée libre : le verrou est disponible et le processus peut l'acquérir =>autorisation de l'entrée en SC.

4.4.3 Alternance Stricte

L'allocation des ressources par alternance est une proposition au problème qui est simple et que nous proposons ici pour bien comprendre l'idée de l'attente active. Elle n'est pas applicable aux cas généraux, puisqu'elle suppose que les processus accédant aux ressources s'exécutent à la même vitesse, et font des accès en alternance. Elle repose sur un compteur *tour* qui indique l'identifiant de processus qui a le droit d'utiliser la ressource et qui est partagée entre les deux processus. Lorsqu'un processus a terminé sa section critique, il incrémente tour, modulo le nombre de processus, pour que sa valeur pointe sur le processus suivant. Lorsqu'un processus est en attente de l'allocation de la ressource, il exécute une boucle infinie. C'est le fait d'exécuter des commandes inutiles (*i.e.* la boucle infinie) qui forme la caractéristique principale de l'attente active; un processus en attente occupe le processeur avec des opérations inutiles.

(a) : ce code est exécute par le processus 0, il entre en SC si tour=0.

→ Le processus 0 teste la valeur de la variable tour

→ Si tour est différente de 0, il est mis en attente et il n'est pas autorisé à entrer en SC. Au cours de cette attente il fait des tests de façon continue sur la valeur de tour

→ Lorsque tour passe à 0, le processus 0 entre en SC

→ Après avoir fini avec sa SC il met tour à 1 en faveur du processus 1 pour que ce dernier entre en SC.

```
while (TRUE) {
 while (tour != 0) /* attente */
 section_critique();
 tour = 1;
 section_noncritique();
}
```

(b) ce code est exécute par le processus 1 afin de vérifier s'il possède l'autorisation d'entrer en SC.

```
while (TRUE) {
 while (tour != 1) /* attente */
 section_critique();
 tour = 0;
 section_noncritique();
}
```

En dehors de l'hypothèse de l'alternance la solution présentée ne marche pas. Supposons, par exemple, que le processus (a) est un processus très rapide, et qu'il vient de terminer sa section critique. (b) est très lent, est actuellement en section non critique, et n'aura pas besoin d'entrer en section critique avant très longtemps. (a) termine rapidement sa section non critique, et veut à nouveau entrer en section critique. Il ne pourra pas, puisque la variable tour n'est pas positionnée correctement. Ces tests sont très consommateurs en termes de temps processeur.

4.4.4 Solution de Peterson

Peterson offre un algorithme élégant et simple à l'exclusion mutuelle. Solution symétrique pour N processus. L'inter-blocage est évité grâce à l'utilisation d'une variable partagée *Tour* qui est utilisée de manière absolue et non relative;

```
#define N    2        /* nombre de processus */
 /* Les variables partagées par tous les processus */
int tour;              /* à qui le tour */
int interesse[N];      /* initialisé à FALSE */
void entrer_region(int process)  /* numéro du processus appelant */
 /* ici process vaut 0 ou 1 */
{  int autre;       /* numéro de l'autre processus */
```

```
 autre = 1-process;
 interesse[process] = TRUE;  /* indiquer son intérêt */
 tour = process;  /* positionner le drapeau d'accès */
 while(tour == process && interesse[autre] == TRUE);
}

void quitter_region(int process)  /* numéro du processus appelant */
 /* ici process vaut 0 ou 1 */
{
 interesse[process] = FALSE; /* indiquer la sortie de
 la section critique */
}
```

Dans cette solution, chaque processus doit, avant d'utiliser des variables partagées, appeler entrer_region() en lui fournissant son identifiant de processus en paramètre. Cet appel ne retournera la main que lorsqu'il n'y a plus de risque. Dès que le processus n'a plus besoin de la ressource, il doit appeler quitter_region() pour indiquer qu'il sort de sa section critique et que les autres processus peuvent accéder à la ressource partagée. L'idée principale de l'algorithme est d'utiliser deux variables différentes, l'une indiquant que l'on s'apprête à utiliser une ressource commune (interesse[ ]), et l'autre (tour) qui, comme dans le cas de l'alternance, indique le processus qui utilisera la ressource au prochain tour. Lorsqu'un processus entre en section critique, il notifie les autres de ses intentions en positionnant correctement son entrée dans le tableau interesse[ ], puis il s'approprie le prochain tour. Tant que lui, est le prochain utilisateur (tour == process), mais l'autre processus a également l'intention d'utiliser (ou utilise déjà) la ressource, le processus courant attend que l'autre libère la ressource. En fin de section critique, en appelant quitter_region(), le processus sortant signale aux autres qu'il n'utilise plus la ressource.

Il est à noter que toute la partie traitée dans ce cours concernant les exclusions mutuelles, repose fondamentalement sur le respect des règles d'utilisation de tous les processus voulant accéder aux ressources partagées. Ici, par exemple, on utilise un tableau partagé (interesse[ ]) dans lequel chaque processus indique son intention d'utiliser la ressource partagée. La règle implicite est, bien-sûr, que chaque processus n'écrit qu'à l'indice du tableau qui lui est dédié, et non pas aux autres endroits.

4.4.5 L'attente passive

Les solutions déjà énoncées possèdent toutes l'inconvénient de faire appel à l'attente active : un processus souhaitant entrer en section critique vérifie s'il en a le droit sinon il entre en boucle d'attende de l'autorisation de l'entrée en SC.

L'attente passive demande une intervention de la part du système (qui doit par conséquent fournir les appels au noyau nécessaire), et évite une surcharge inutile du processeur. Il évite également le problème

d'*inversion de priorité*, puisque l'idée de base consiste à basculer le processus en attente de l'accès à une ressource partagée en mode *bloqué*, jusqu'à ce que la ressource se libère, ce qui la fait basculer en mode *prêt*. Il existe une quantité importante de solutions pour implanter des méthodes d'attente passive.

Pour remédier à ce problème, certaines primitives de communication inter-processus permettent de bloquer les processus au lieu de consommer du temps processeur lorsqu'ils n'obtiennent pas l'autorisation d'entrer dans leurs sections critiques. L'une des simples fonctionne avec la paire **Sleep et Wakeup** .Sleep et Wakeup prennent un paramètre qui indique le processus à bloquer ou réveiller.

Problème du producteur_consommateur

Pour illustrer l'emploi de ces primitives, étudions le problème du producteur – consommateur (tampon délimité ou bounded buffer). Deux processus partagent un tampon commun de taille fixe. L'un d'eux le producteur, place des informations dans le tampon et l'autre le consommateur les récupère. Les problèmes se produisent lorsque le producteur souhaite placer un nouvel élément dans le tampon alors que ce dernier est déjà plein. La solution pour le producteur est d'entrer en sommeil, pour être réveillé lorsque le consommateur aura supprimé un ou plusieurs éléments du tampon. De la même façon, si le consommateur souhaite récupérer un élément dans le tampon, alors que celui ci est vide, il entre en sommeil jusqu'a ce que le producteur ait placé quelque chose dans le tampon, opération qui va le réveiller.

```
#define N 100 /*le nombre max d'éléments*/
int count = 0 ; /*nbre d'éléments dans le tapon*/
void producteur(void)
{
int élément ;
while (true)
{
élément = produire-élément(); /* génère l'élément suivant */
if (count = = N) sleep(); /* Si tampon plein entre en sommeil */
insérer_élément (élément) ; /* place l'élément dans le tampon */
count = count +1 ; /* incrémente le décompte des éléments*/
if (count = = 1) wakeup (consommateur) ; /* Si le tampon était vide il réveille le
consommateur */
}
}
void consommateur (void)
{
int élément ;
while(true)
```

```
{
if (count = = 0) sleep (); /* si tampon vide entre en sommeil*/
élément = extraire_élément () ; /* prélève un élément dans le tampon*/
count = count-1 ; /* décrémente le décompte des élts dans le tampon*/
If (count = = N-1) wakeup (producteur) ; /* le tampon était vide */
}
}
```

4.4.6 Les sémaphores

Le contrôle de la synchronisation par l'utilisation d'un type de données abstrait appelé sémaphore a été proposé par Dijkstra en 1965. Les sémaphore sont très facilement mis en œuvre dans les systèmes d'exploitation et constituent une solution générale pour contrôler l'accès aux sections critiques.

Un sémaphore est une variable entière non négative à partir de laquelle sont définies deux opérations élémentaires : P et V (d'après les termes néerlandais proberen : tester et verhogen : incrémenter).

```
Struct semaphore
{
Int count ;
ProcessQueue queue ;
} ;
Void P(semaphore s){
        if(s.count>0){
                s.count=s.count-1;
        else
                s.queue.Insert();  //Bloque ce processus
}
Void V(semaphore s){
                s.count=s.count+1 ;
        if(!s.queue.empty())
                s.queue.Remove() ;//libère un processus
                                  //bloqué sur 's', s'il existe.
}
```

Il se présente comme un distributeur de jetons, mais le nombre de jetons est fixe et non renouvelable: les processus doivent restituer leur jeton après utilisation. S'il y a un seul jeton en circulation, on retrouve le verrou.

Les opérations sémaphores sont parfois également connues sous le nom de Up (haut) et Down(bas) ou Wait et Signal.

- *P(s)* ou wait correspondant à sleep permet à un processus d'obtenir un jeton, s'il y en a de disponibles. Si aucun n'est disponible, le processus est bloqué. (P(S) : si S=0 alors mettre le processus en attente, sinon S ← S-1)

```
Wait(S)

    If(S=0) then sleep() ;

    S-- ;
```

- V(s) ou signal correspondant à wakeup permet à un processus de restituer un jeton. Si des processus étaient en attente de jeton, l'un d'entre eux est réactivé et le reçoit. (V(S) : S ← S+1 ; réveiller un (ou plus) processus en attente)

```
Signal(S)
    Wakeup() ;
    S++ ;
```

Une fois instanciée, la valeur compteur d'un sémaphore peut recevoir toute valeur non négative. Avec une valeur initiale de 1, un sémaphore assure mutuellement un accès exclusif si P est exécuté avant d'entrer dans une section critique et si V l'est à la sortie. En initialisant la valeur initiale du sémaphore sur n, P et V peuvent être utilisés pour autoriser jusqu'à n processus dans leurs sections critiques.
Un sémaphore binaire est un sémaphore dont le compteur ne peut admettre que les valeurs 1 ou 0.

```
Void P(semaphore s){
    if(s.count==1)
        s.count=0 ;
    else
        s.queue.Insert() ; // Bloque ce processus
}
void V(semaphore s){
    if(s.queue.empty())
        s.count=1 ;
    else
            s.queue.Remove() ; ;//libère un processus
                        //bloqué sur 's', s'il existe.
}
```

Pour distinguer plus clairement les types de sémaphores, précisons que ceux peuvent prendre des valeurs non négatives peuvent être appelés *sémaphores généraux* ou *sémaphores compteur*.

On arrive à l'exclusion mutuelle en initialisant une variable sémaphore à un, en exécutant une opération P avant d'entrer dans la section critique et en exécutant une opération V après avoir quitté la section critique.

À la création d'un sémaphore, il faut décider du nombre de jetons dont il dispose. On voit que si une ressource est à un seul point d'accès (critique), le sémaphore doit avoir initialement 1 jeton, qui sera attribué successivement à chacun des processus demandeurs. Si une ressource est à n points d'accès, c'est-à-dire, peut être utilisée par au plus n processus à la fois, il suffit d'un sémaphore initialisé avec n jetons. Dans les deux cas, un processus qui cherche à utiliser la ressource, demande d'abord un jeton, puis utilise la ressource lorsqu'il a obtenu le jeton et enfin rend le jeton lorsqu'il n'a plus besoin de la ressource.

4.4.7 Solution avec échanges de messages

Certains ont estimé que les sémaphores sont de trop bas niveau. Ils ont proposé un mode de communication inter-processus qui repose sur deux primitives qui sont des appels système :

- send (destination , &message)
- receive (source , &message), où source peut prendre la valeur générale ANY

Généralement, pour éviter les problèmes dans les réseaux, le récepteur acquitte le message reçu. L'émetteur envoie à nouveau son message s'il ne reçoit pas d'acquittement. Le récepteur, s'il reçoit deux messages identiques, ne tient pas compte du second et en tire la conclusion que l'acquittement s'est perdu.

Dans le contexte d'un mode client-serveur, le message reçu contient le nom et les paramètres d'une procédure à lancer. Le processus appelant se bloque jusqu'à la fin de l'exécution de la procédure et le message en retour contient la liste des résultats de la procédure. On parle d'appel de procédure à distance.

On peut proposer une solution au problème producteur-consommateur par échange de messages avec les hypothèses suivantes :

- les messages ont tous la même taille
- les messages envoyés et pas encore reçus sont stockés par le SE dans une mémoire tampon
- le nombre maximal de messages est N
- chaque fois que le producteur veut délivrer un objet au consommateur, il prend un message vide, le remplit et l'envoie. Ainsi, le nombre total de messages dans le système reste constant dans le temps
- si le producteur travaille plus vite que le consommateur, tous les messages seront pleins et le producteur se bloquera dans l'attente d'un message vide ; si le consommateur travaille plus vite que le producteur, tous les messages seront vides et le consommateur se bloquera en attendant que le producteur en remplisse un.

producteur

```
Faire toujours
        produire_objet (&objet)    /* produire un nouvel objet              */
        receive (consommateur , &m)   /* attendre un message vide           */
        faire_message (&m , objet)   /* construire un message à envoyer     */
        send (consommateur , &m)    /* envoyer le message                   */
Fait
```

consommateur

```
pour (i = 0 ; i < N ; i++) send (producteur , &m)     /* envoyer N messages vides   */
Faire toujours
        receive (producteur , &m)                     /* attendre un message        */
        retirer_objet (&m , &objet)                   /* retirer l'objet du message */
        utiliser_objet (objet)
        send (producteur , &m)                        /* renvoyer une réponse vide  */
Fait
```

On peut également imaginer une solution de type boîte aux lettres de capacité N messages, avec un producteur se bloquant si la boîte est pleine et un consommateur se bloquant si la boîte est vide.