

CHAPITRE II : COMMUNICATION & SYNCHRONISATION

Dans ce chapitre nous présenterons la problématique de la communication inter-processus et des accès concurrentes.

2.0 PROBLEMATIQUE :

Lorsque plusieurs processus s'exécutent sur un ordinateur , monoprocesseur ou multiprocesseur avec mémoire commune , ils sont amenés à partager des variables communes soit volontairement s'ils coopèrent pour traiter un même problème, soit involontairement parce qu'ils sont obligés de se partager les ressources de l'ordinateur.

Malheureusement, le partage des variables sans précaution particulière peut conduire à des résultats imprévisibles.

Exemple : Le compte bancaire

Considérons un compte bancaire , dont le montant est mémorisé dans un emplacement mémoire A. Le programme qui consiste à ajouter 100 à ce compte pourrait être le suivant , où N est une variable locale du programme :

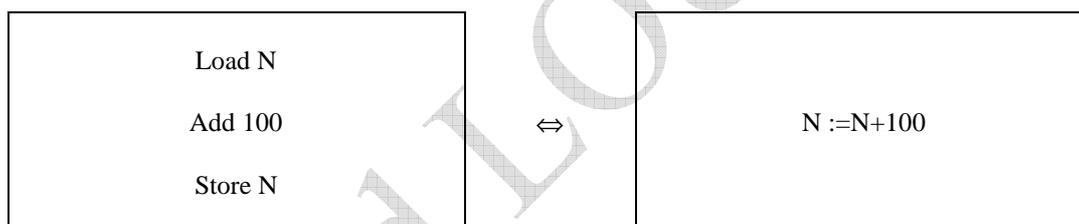
```
Algorithme Compte
Début
    Lire(N, A) ;
    N :=N+100 ;
    Ecrire(N, A)
Fin.
```

Si maintenant deux processus différents consistent en l'exécution de ce même programme, le processeur sera alloué à chacun d'eux dans un ordre quelconque. En particulier, on peut imaginer l'ordre suivant :

<i>Ordre d'exécution</i>	<i>Processus P1</i>	<i>Processus P2</i>
1	Lire(N, A)	
2		Lire(N, A) ;
3		N :=N+100 ;
4		Ecrire(N, A) ;
5	N :=N+100 ;	
6	Ecrire(N, A)	

N étant une variable locale du programme, cela implique qu'il en existe un exemplaire pour chacun des deux processus. Il s'ensuit que si la valeur initiale du compte est 1000, on constate qu'après ces exécutions, il est égal à 1100 au lieu de 1200. Les deux processus ne sont pas en fait totalement indépendants. Ils partagent la ressource commune qu'est la valeur du compte bancaire à l'adresse A sur disque. Cette ressource doit avoir un seul point d'accès ; c'est donc une ressource critique, et les processus sont en exclusion mutuelle sur cette ressource critique.

Si la variable N est commune aux deux processus, le problème n'est pas résolu pour autant. En effet, l'instruction $N := N + 100$ n'est pas une instruction « indivisible ». En fait, elle peut être décomposée en trois micro-instructions au niveau machine, comme le montre le code suivant :



Il est important de savoir que l'allocateur du processeur prend en compte les micro-instructions et non les instructions du langage évolué pour déterminer les moments où il peut remplacer le processus actif par un autre. On peut donc imaginer le scénario suivant d'allocation du processeur aux deux processus :

<i>Ordre d'exécution</i>	<i>Processus P1</i>	<i>Processus P2</i>
1	Load N	
2		Load N
3		Add 100
4		Store N
5	Add 100	
6	Store N	

Ainsi, on voit là aussi que le résultat obtenu à l'issue des deux processus est erroné.

On voit à travers cet exemple que l'exécution simultanée de programmes (processus) peut conduire à des résultats erronés si des précautions ne sont pas prises.

Concepts fondamentaux :

Section Critique

Une section critique (SC) est un ensemble d'instruction d'un programme qui peuvent engendrer des résultats imprévisibles lorsqu'elles sont exécutées simultanément par des processus différents.

D'une manière générale on peut dire qu'un ensemble d'instructions peut constituer une section critique (SC) s'il y a des variables partagées ; et non dans l'absolu. Autrement dit, l'existence de section critique implique l'utilisation de variable partagées , mais l'inverse n'est pas vrai.

De ce qui précède on peut facilement déduire que l'exécution simultanées de deux sections critiques appartenant à deux ensembles différents et ne partageant pas de variables ne peut poser aucun problème d'accès concurrent.

Ressource critique

On appelle ressource critique tout objet : variable, table, fichier, périphérique, ... qui peut faire l'objet d'un accès concurrent par plusieurs processus. Par exemple deux processus qui tentent d'envoyer chacun, simultanément, un fichier sur l'imprimante ; le périphérique imprimante devient ressource critique pour eux.

Exclusion mutuelle

Les problèmes posés par les accès concurrents décrits au §1.1 montrent que la solution consiste à exécuter les sections critiques en **exclusion mutuelle** (mutex). C'est à dire qu'une SC ne peut être entamée que si aucune autre SC du même ensemble n'est en exécution.

Ainsi, voici donc le principe général d'une solution garantissant que l'exécution simultanée de plusieurs processus ne conduirait pas à des résultats imprévisibles : 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.

Avant d'entrer en SC, le processus doit exécuter un protocole d'entrée. Le but de ce protocole est de vérifier justement si la SC n'est occupée par aucun autre processus.

A la sortie de la SC, le processus doit exécuter un protocole de sortie de la SC. Le but de ce protocole est d'avertir les autres processus en attente que la SC est devenue libre.

Le schéma suivant résume ce principe de fonctionnement :

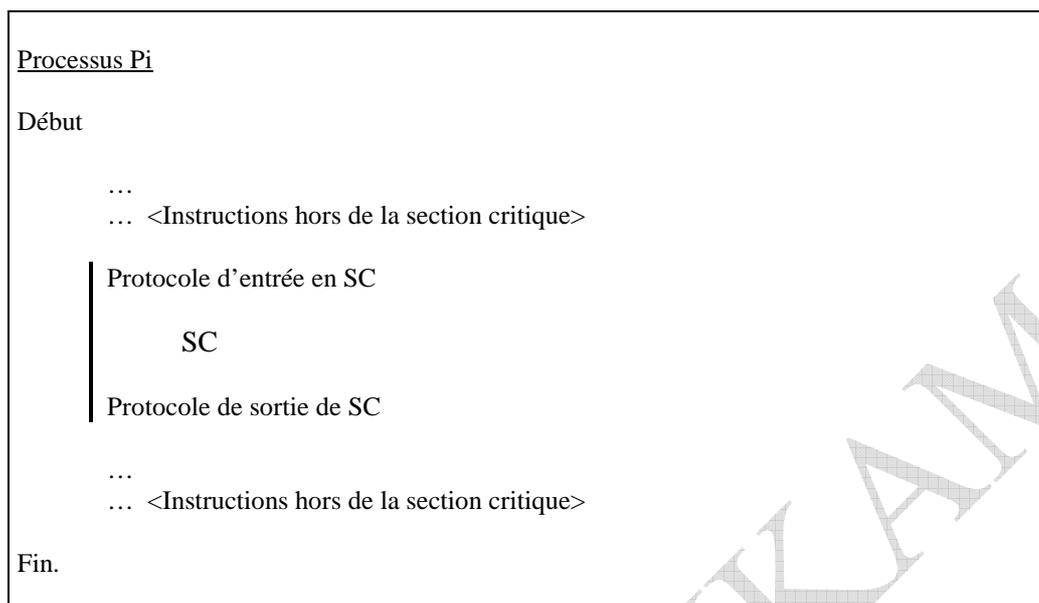


Figure 1.1 Structure d'un processus qui exécute une section critique.

Contraintes à respecter pour la réalisation d'une exclusion mutuelle :

Pour réaliser une exclusion mutuelle , nécessaire pour résoudre le problème des accès concurrents, on admet que certaines contraintes doivent être respectées :

Le déroulement : Le fait qu'un processus ne demande pas à entrer en section critique ne doit pas empêcher un autre processus d'y entrer. Cette contrainte exclut les méthodes fondées sur un tour de rôle strict.

L'attente finie : Si plusieurs processus sont en compétition pour entrer en SC, le choix de l'un d'eux ne doit pas être repoussé indéfiniment. Autrement dit, la solution proposée doit garantir que tout processus n'attend pas indéfiniment.

Tous les processus doivent être égaux vis à vis de l'entrée en SC.

2.1 SOLUTIONS LOGICIELLES POUR L'EXCLUSION MUTUELLE :

Soit un ensemble de N processus P1, P2, P3, ... PN-1 possédant chacun une SC dans laquelle le processus peut occuper une ressource critique. L'exécution de la SC par un processus doit se faire en exclusion mutuelle, c'est à dire qu'en entrant dans sa SC, aucun autre processus n'est autorisé à exécuter sa SC.

Nous présentons ci-après les plus importantes solutions logicielles qui ont été proposées pour réaliser l'exclusion mutuelle.

2.1.1 Solutions logicielles pour deux processus

Nous considérons dans cette section le problèmes de deux processus concurrents P0 et P1 dont la structure typique est :

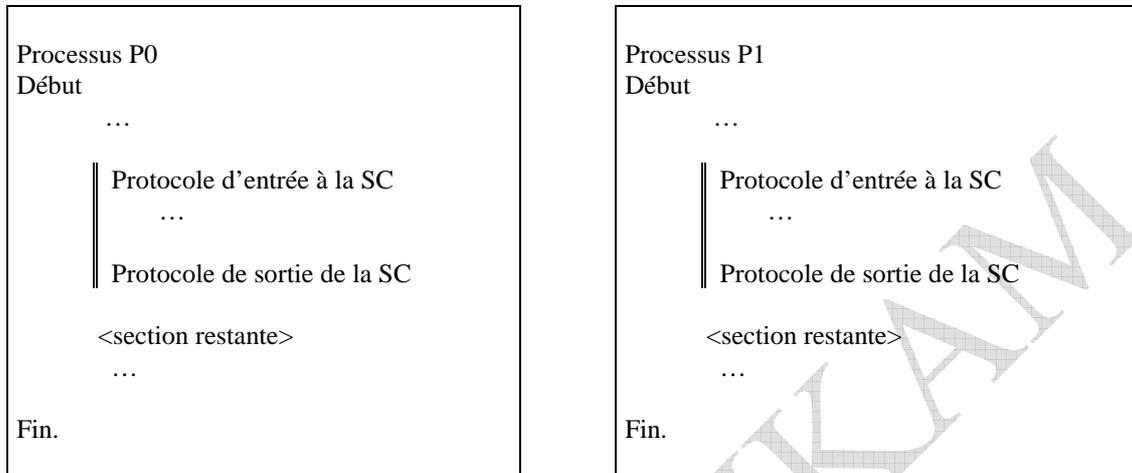


Figure 2.1 Cas de deux processus concurrents

La partie <section restante> désigne l'ensemble des instructions qui sont après la section critique.

Pour utiliser une notation pratique, si P_i désigne l'un des processus alors P_j désignera l'autre processus.

Pour ce problème précis de deux processus, plusieurs solutions ont été proposées. Nous les présentons ci-après.

Solution 1 : Cette solution consiste à laisser les deux processus partager une variable entière commune **turn**, initialisée à 0 (ou 1). Si la valeur de $turn=i$, alors on permet au processus P_i d'exécuter sa SC. Voici l'algorithme de cette solution :

```

Processus  $P_i$ 
Début

Tantque  $turn \neq i$ 
Faire
    Rien
Fait ;

|| SC

Turn := j ;

<Section Restante>
Fin.

```

Figure 2.2 Solution logicielle, basée sur le tour, pour le cas de deux processus concurrents

Commentaire : Cette solution garantit qu'un seul processus à la fois peut se trouver dans sa SC. Cependant elle exige une alternance stricte ; les processus entrent en SC à tour de rôle dans un ordre forcé. Autrement dit, cette solution ne vérifie pas le critère du déroulement (voir §1.2.4). Par exemple, si $turn=0$ et P1 est prêt à entrer dans sa SC, cela ne lui est pas possible même si P0 peut se trouver dans sa section restante.

Solution 2 :

Le problème de la solution 1 est qu'elle ne garde pas suffisamment d'informations sur l'état de chaque processus ; elle se rappelle seulement quel processus est autorisé à entrer en SC. Pour remédier à ce problème, nous pouvons remplacer la variable **Turn** par le tableau suivant :

Etat : Tableau[0..1] de Booléen.

Le tableau est initialisé à Faux. Si $Etat[i]$ est Vrai, cette valeur indique que le processus P_i est prêt pour entrer en SC.

L'algorithme de la solution est :

```

Processus Pi
Début
    Etat[i] :=Vrai ;
    Tantque Etat[j]
    Faire
        Rien
    Fait ;
    ||
    SC
    Etat[i] :=Faux ;
    <Section Restante>
Fin.

```

Figure 2.3 Solution logicielle, basée sur l'état, pour le cas de deux processus concurrents

Dans cette solution, le processus P_i fixe d'abord $Etat[i]$ à Vrai, signalant ainsi qu'il est prêt à entrer en SC. Ensuite, P_i vérifie l'état du processus P_j en consultant la valeur de $Etat[j]$. Le processus P_i attend alors jusqu'à ce que le processus P_j lui indique qu'il n'a plus besoin d'entrer dans sa SC (c'est à dire jusqu'à ce que la valeur de $Etat[j]$ devienne fausse). A ce moment là , le processus P_i entre dans sa SC. En sortant de sa SC, P_i remet son état à Faux, permettant à l'autre processus d'entrer éventuellement en SC.

Commentaire : Dans cette solution, le principe de l'exclusion mutuelle est respecté . Toutefois, un problème se pose si les deux processus mettent leur état à Vrai, puis vont , chacun, itérer sur la vérification de l'état de l'autre processus. On assiste ainsi à une boucle sans fin pour les deux

processus (aucun des deux processus ne peut entrer dans sa SC). Le critère de l'attente finie n'est donc pas vérifié.

Solution 3 Algorithme de Dekker : La solution 3, appelée algorithme de Dekker, combine les idées des solutions 1 et 2 précédentes : Les processus partagent 2 variables :

Etat : Tableau[0..1] de Logique ;
Turn : 0..1

Initialement Etat[0]=Etat[1] et la valeur de Turn est initialisée à 0 ou à 1.

L'algorithme de la solution est le suivant :



Figure 2.4 Solution de Dekker pour le cas de deux processus concurrents

Pour entrer dans la SC, le processus Pi met son état Etat[i] à Vrai et permet à l'autre processus d'entrer en SC en mettant la variable Turn à j. Si les deux processus tentent d'entrer en même temps en SC, Turn sera fixé à i et à j presque au même moment. Evidemment, une seule des ces affectation se produira l'autre sera perdue (la seconde affectation écrasera la première valeur). La valeur de Turn décidera quel processus entrera dans sa SC.

Commentaire : Cette solution garantit les trois critères :

- Exclusion mutuelle.
- Déroulement.
- Attente finie.

2.1.2 Solutions logicielles pour plusieurs processus

L'algorithme de Dekker précédent apporte une solution au problème de la SC pour deux processus seulement. L'algorithme qui suit est une généralisation du précédent, il porte le nom de l'algorithme du

Boulangier. A l'origine, cet algorithme a été conçu pour un environnement partagé, mais des variantes existent pour un système d'exploitation centralisé.

Cet algorithme s'inspire d'une technique de scheduling utilisée dans les anciennes boulangeries et magasins. En entrant dans la boulangerie, chaque client reçoit un numéro. Le client suivant à servir est celui qui aura le plus petit numéro.

L'algorithme du boulangier ne garantit pas que deux processus (clients) ne reçoivent pas le même numéro. Dans ce cas, le client à servir est celui qui sera le premier dans l'ordre alphabétique. C'est à dire que si P_i et P_j ont le même numéro et si $i > j$, alors on sert d'abord P_i avant P_j .

Les structures de données utilisées par l'algorithme sont :

Choosing : Tableau[0..N-1] de Logique ;
Number : Tableau[0..N-1] de Entier ;

Ces structures de données sont initialisées respectivement à Faux et à 0.

On utilisera la notation suivante , pour exprimer un ordre entre deux couples d'entiers : On notera

$$(a, b) < (c, d)$$

si si $a < c$ (ou $a = c$) et $(b < d)$.

L'algorithme du Boulangier est alors le suivant :

```

Processus Pi
Début
    Choosing :=Vrai ;
    Number[i] :=max(Number[0], Number[1], ... Number[N-1])+1 ;
    Choosing :=Faux ;

    Pour j :=0 jusqu'à N-1
    Faire
        Tantque (Choosing[j])
        Faire
            Rien ;
        Fait ;

        Tanque (Number[j] <> 0 et ((Number[j], j) < (Number[i], i))
        Faire
            Rien ;
        Fait
    Fait

    || SC

    Number[i] :=0 ;

    <Section Restante>
Fin.

```

Figure 2.5 L'algorithme du Boulangier.

Cet algorithme assure l'exclusion mutuelle. En effet, si P_i est dans sa SC, un processus P_k qui exécute la deuxième instruction tant que pour $j=i$, trouvera que :

$$(\text{Number}[i] < 0) \text{ et } ((\text{Number}[i], i) < \text{Number}[k], k)$$

Le processus P_k continuera donc à itérer dans l'instruction tant que jusqu'à ce que P_i abandonne sa SC.

2.2 SOLUTIONS MATERIELLES POUR L'EXCLUSION MUTUELLE :

On peut réaliser l'exclusion mutuelle en utilisant, lorsque cela est possible, des instructions matérielles mises à la disposition du programmeur pour réaliser certaines tâches de façon indivisible.

2.2.1 Masquage des interruptions :

Matériellement une exclusion mutuelle peut être réalisée si on peut interdire les interruptions pendant qu'une variable partagée est modifiée. De cette façon, on pourra être sûr que la séquence courante d'instructions s'exécute en ordre sans aucune réquisition. Malheureusement cette méthode n'est pas toujours efficace. En effet, le masquage et le démasquage des interruptions peut devenir très vite pénalisant pour le système d'exploitation.

Il existe une alternative plus intéressante à cette méthode : les instructions matérielles indivisibles. En effet, plusieurs constructeurs d'ordinateurs fournissent des instructions matérielles spéciales qui nous permettent soit de tester et modifier le contenu d'un mot-mémoire, soit d'échanger le contenu de deux mots d'une manière atomique (indivisible). Nous pouvons utiliser ces deux instructions pour résoudre le problème de l'exclusion mutuelle de manière relativement simple.

2.2.2 L'instruction Test-and-Set :

Cette instruction permet de tester et de modifier le contenu d'une variable d'une manière indivisible. Sa forme générale est :

```

Fonction Test_and_Set (var cible : logique) : Logique
Begin
    Test_and_Set := cible ;
    Cible := Vrai ;
End ;

```

Figure 2.6 L'instruction Test_and_Set.

Rappelons le caractère indivisible de cette instruction. C'est à dire qu'elle est considérée comme une unité non interruptible. Ainsi si deux instructions Test_and_Set sont exécutées simultanément, elles seront exécutées séquentiellement dans un ordre quelconque.

Si la machine supporte l'instruction `Test_and_Set`, alors nous pouvons réaliser l'exclusion mutuelle en déclarant une variable booléenne `Lock` initialisée à faux, comme le montre la figure suivante :

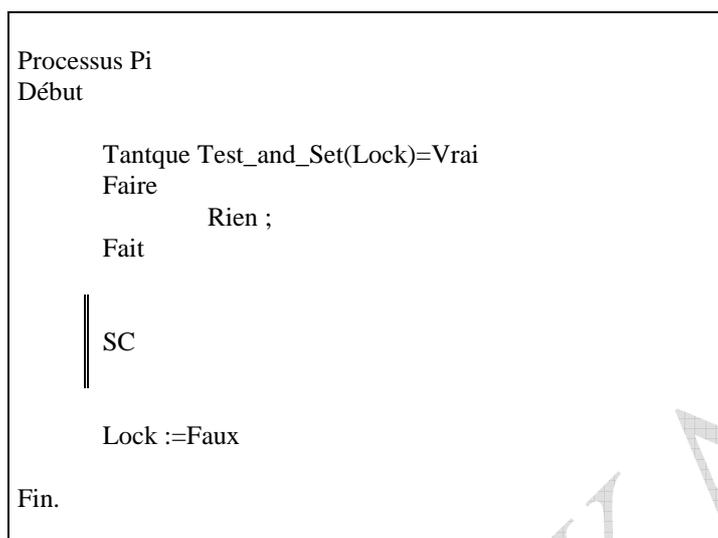


Figure 2.7 Réalisation de l'exclusion mutuelle avec l'instruction `Test_and_Set`.

2.2.3 L'instruction `Swap` :

Cette instruction permet d'échanger le contenu de deux mots de manière atomique. Sa forme générale est la suivante :

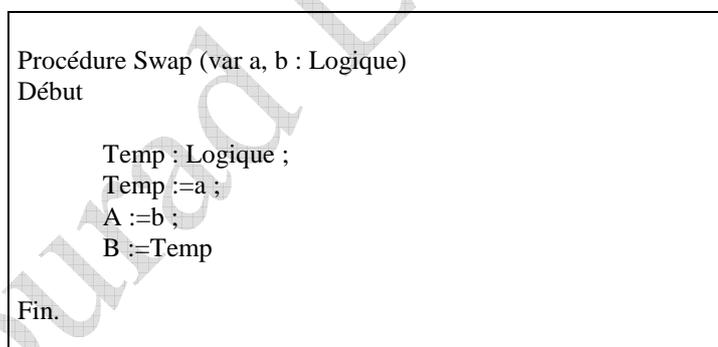


Figure 2.8 L'instruction `Swap`.

On peut réaliser l'exclusion mutuelle en utilisant l'instruction `Swap`, en déclarant une variable logique `Lock` initialisée à faux et une variable logique locale `Key` au niveau de chaque processus (voir figure suivante).

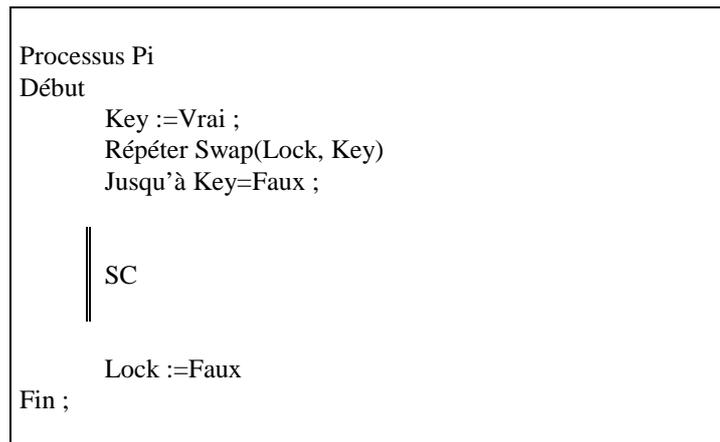


Figure 2.9 Réalisation de l'exclusion mutuelle avec l'instruction Swap.

2.3 LES SEMAPHORES :

Les solutions matérielles et logicielles présentées dans les sections précédentes sont difficiles à mettre en œuvre pour des problèmes de synchronisation complexes. Ainsi, plusieurs outils de synchronisation évolués ont été proposés pour traiter les multitudes de cas de synchronisation.

Dans cette section, nous définirons l'outil de synchronisation sans doute le plus connu qu'est le **sémaphore**. Nous aurons l'occasion de voir d'autres outils dans le chapitre 4.

Les sémaphores ont été introduits par Dijkstra, illustre informaticien hollandais, en 1965.

Définition :

Un sémaphore S est un ensemble de deux variables :

- Une valeur entière Value.
- Une file d'attente F de processus F

Quant un processus doit attendre un sémaphore, il est ajouté à la file de processus. Une opération Signal supprime un processus de la file des processus en attente et réveille ce processus.

Ainsi les opérations Wait et Signal peuvent être définies de cette façon :

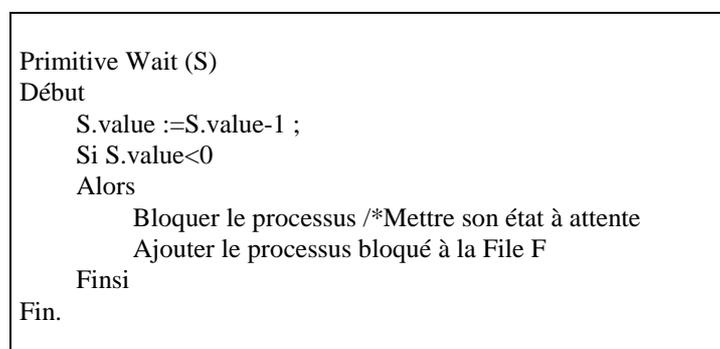


Figure 2.13 La primitive Wait avec file d'attente

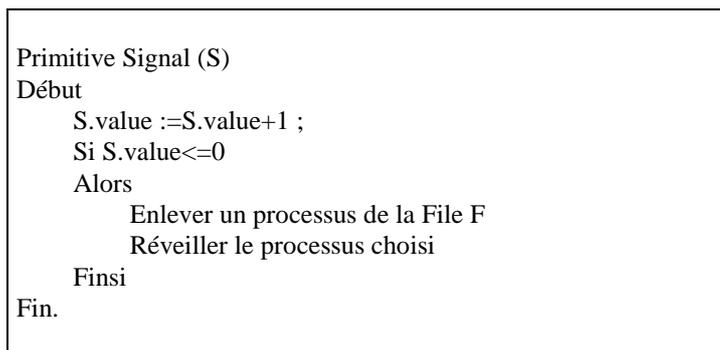


Figure 2.14 La primitive Signal avec file d'attente.

De ce qui précède, on peut facilement proposer un schéma de synchronisation de n processus voulant entrer simultanément en SC, en utilisant les deux opérations Wait et Signal. En effet, il suffit de faire partager les n processus un sémaphore mutex, initialisé à 1, appelé sémaphore d'exclusion mutuelle. Chaque processus P_i a la structure suivante :

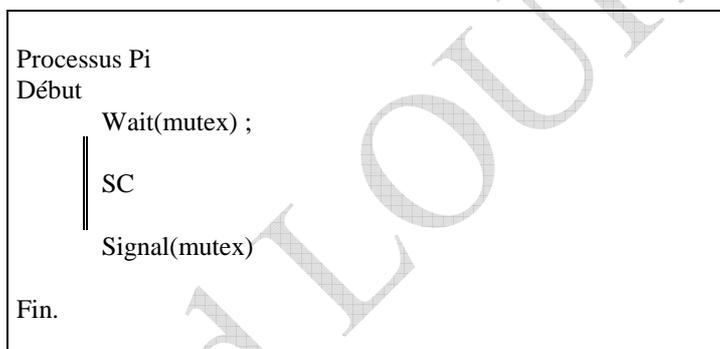
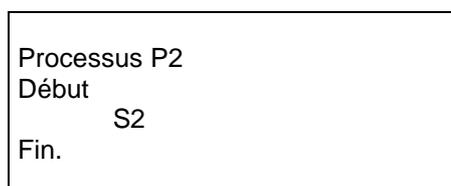
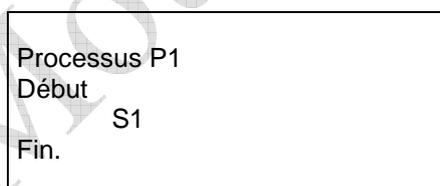
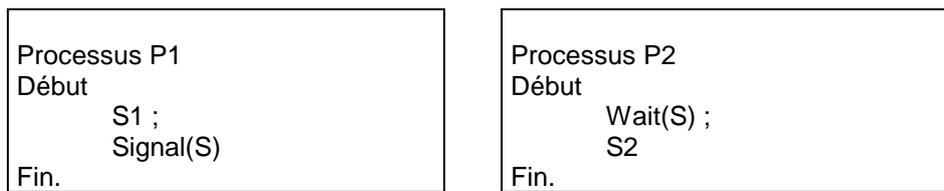


Figure 2.12 Utilisation des primitives Wait et Signal.

Pour voir davantage l'efficacité des sémaphores comme outil de synchronisation, considérons l'exemple suivant : Deux processus P1 et P2 exécutent respectivement deux instructions S1 et S2.



Si on souhaite que S2 ne doit s'exécuter qu'après l'exécution de S1, nous pouvons implémenter ce schéma en faisant partager P1 et P2 un sémaphore commun S, initialisé à 0 et en insérant les primitives Wait et Signal de cette façon :



Comme S est initialisé à 0, P2 exécutera S2 seulement une fois que P1 aura appelé Signal(S).

Remarque : Théoriquement, un sémaphore peut être initialisé à n'importe quelle valeur entière, mais généralement cette valeur est positive ou nulle.

2.4 QUELQUES PROBLEMES DE SYNCHRONISATION CLASSIQUES

2.4.1 Le problème du Producteur-Consommateur :

Le problème de Producteur-Consommateur est un problème de synchronisation classique représentant toute une classe de situations où un processus, appelé Producteur, délivre des messages (informations) à un processus Consommateur dans un tampon (par exemple, un programme qui envoie des données sur le spool de l'imprimante). La figure suivante résume bien ce type de problème.

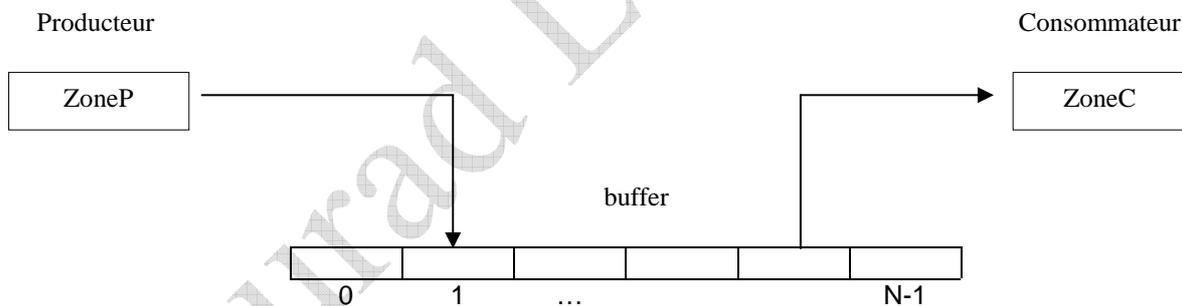
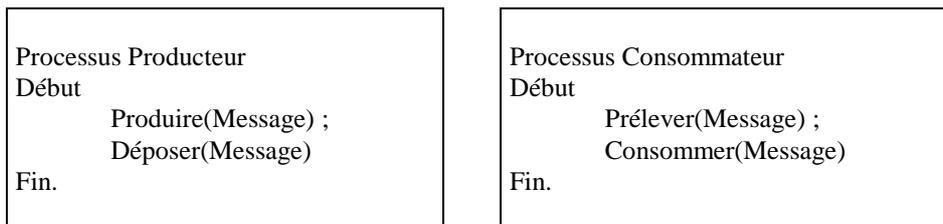


Figure 3.1 Schéma du problème de Producteur-Consommateur

Le Producteur produit un message dans la ZoneP, puis le dépose dans le buffer. Le Consommateur prélève un message du Buffer et le place dans la ZoneC où il peut le consommer.

Pour des raisons pratiques, on considérera que le buffer est de N cases, numérotées de 0 à N-1, et organisé de façon circulaire. Le Producteur dépose les messages par un bout du buffer alors que le consommateur les consomme au fur et à mesure par l'autre bout.

On peut écrire sommairement les codes de chacun des deux processus : Producteur et Consommateur.



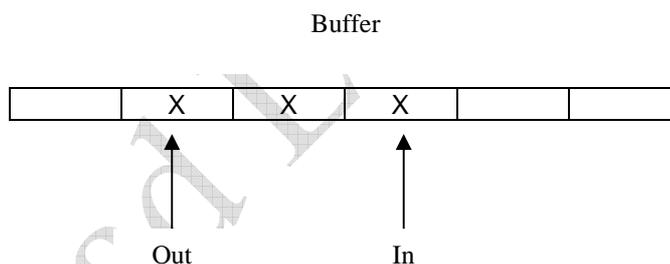
Le problème du Producteur-Consommateur consiste à trouver comment synchroniser les deux processus de sorte que :

- Le Producteur ne dépose une information que si le buffer n'est pas plein.
- Le Consommateur ne prélève une information que si le buffer n'est pas vide.
- Le Consommateur n'essaie pas de consommer une information qui est en train d'être produite par le Producteur.

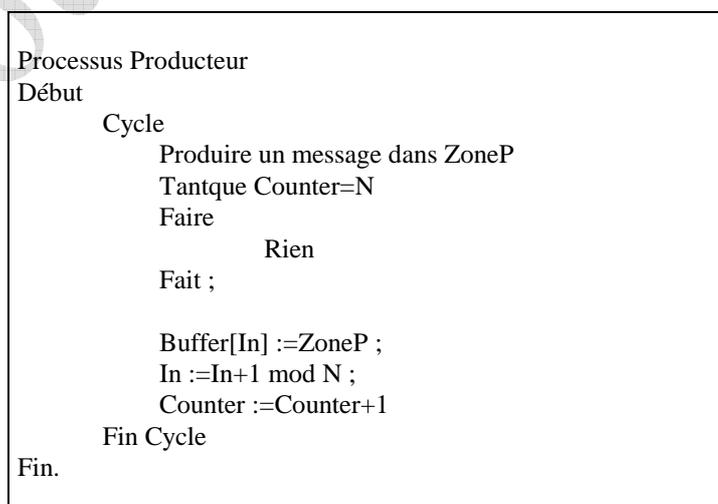
Solution 1 (fausse) :

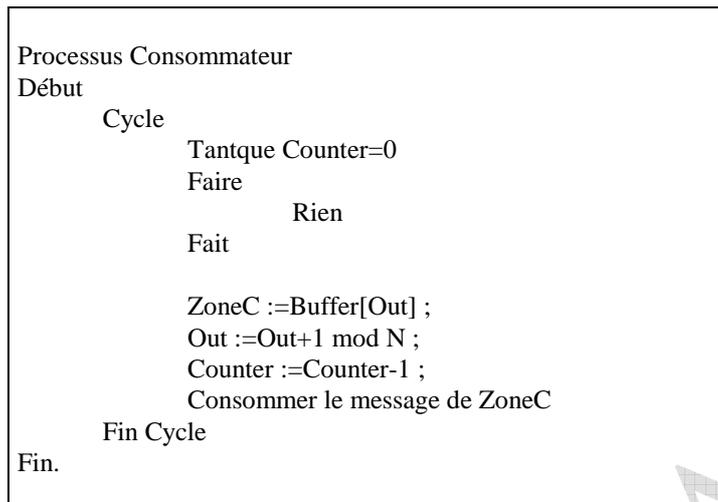
Dans une première approche du problème, on utilisera trois variables :

- Counter : Nombre d'éléments présents dans le buffer.
- In : Indice du dernier élément entré.
- Out : Indice de l'élément prêt à être consommé.



Le schéma de fonctionnement des deux processus, Producteur et Consommateur, peut être le suivant :





Cette solution est évidemment fautive, puisqu'elle ne garantit pas une exclusion mutuelle entre les deux processus pour le partage de la variable commune Counter. En effet, cette variable peut être modifiée par un processus, alors que l'autre est en train de la consulter ; ce qui peut provoquer des résultats incohérents.

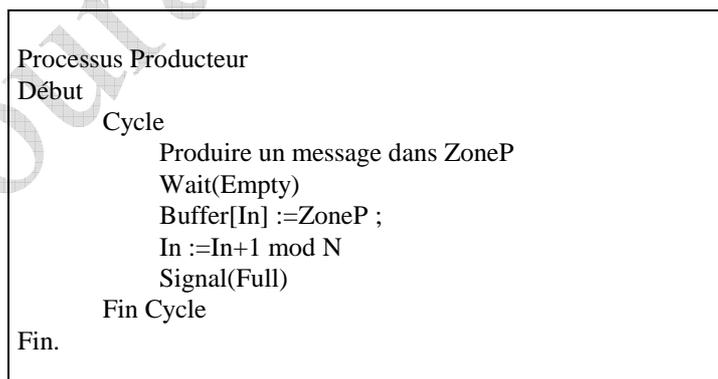
Solution 2 (correcte) :

Dans cette solution, on utilise deux sémaphores :

- Empty : compte le nombre de cases vides.
- Full : compte le nombre de cases pleines.

On initialise Empty à N et Full à 0.

Le schéma de synchronisation des deux processus est comme suit :



```

Processus Consommateur
Début
    Cycle
        Wait(Full)
        ZoneC :=Buffer[Out] ;
        Out :=Out+1 mod N ;
        Signal(Empty) ;
        Consommer le message de ZoneC
    Fin Cycle
Fin.

```

Cette solution assure correctement la synchronisation des processus Producteur et Consommateur. Si le Buffer est plein, le Producteur est bloqué jusqu'à ce qu'une case devienne disponible, à ce moment là il est réveillé par le processus Consommateur. Inversement, le Consommateur est bloqué tant que le Buffer est vide ; il est réveillé par le Producteur dès qu'un message a été déposé.

Généralisation à P producteurs et C consommateurs :

On s'intéresse maintenant au cas plus général, où nous avons P producteurs et C consommateurs. Un examen simple de la solution précédente fait apparaître clairement que dans ce cas les variables In et Out deviennent des ressources critiques pour les processus. En effet, plusieurs processus producteurs peuvent vouloir modifier simultanément la variable In, et plusieurs processus Consommateurs peuvent vouloir modifier la variable Out. Il y a lieu donc de les rendre accessibles uniquement en exclusion mutuelle.

On utilise pour ce problème 4 sémaphores :

- **Empty** : compte le nombre de case vides.
- **Full** : compte le nombre de cases pleines.
- **MutexProd** : sémaphore d'exclusion mutuelle pour protéger la variable In
- **MutexCons** : sémaphore d'exclusion mutuelle pour protéger la variable Out.

MutexProd et MutexCons sont initialisé à 1. Le schéma de synchronisation des processus Producteurs et Consommateurs est :

```

Processus Producteur
Début
    Cycle
        Produire un message dans ZoneP
        Wait(Empty)
        Wait(MutexProd) ;
        Buffer[In] :=ZoneP ;
        In :=In+1 mod N ;
        Signal(MutexProd) ;
        Signal(Full)
    Fin Cycle
Fin.

```

```

Processus Consommateur
Début
    Cycle
        Wait(Full)
        Wait(MutexCons) ;
        ZoneC :=Buffer[Out] ;
        Out :=Out+1 mod N ;
        Signal(MutexCons) ;
        Signal(Empty) ;
        Consommer le message de ZoneC
    Fin Cycle
Fin.

```

2.4.2 Le problème des Lecteurs-Rédacteurs :

Un fichier de données est partagé entre plusieurs processus concurrents. Certains de ces processus désirent lire le contenu du fichier, nous les appellerons : **Lecteurs**. Les autres processus veulent modifier le contenu du fichier, nous les appellerons : **Rédacteurs**.

Si plusieurs processus lecteurs accèdent au fichier simultanément, il ne produira aucun effet indésirable. Par contre, si un rédacteur et d'autres processus (lecteurs ou rédacteurs) accèdent simultanément au fichier, un résultat erroné peut se produire. En clair, à un moment donné on ne doit avoir que l'une des deux situations suivantes :

- Un seul rédacteur est en train de modifier le fichier
- Un ou plusieurs lecteurs sont en train de lire le contenu du fichier

De ce qui précède, on peut déduire que chaque processus Rédacteur doit avoir un accès exclusif au fichier. Pour réaliser simplement cette contrainte on utilisera un sémaphore d'exclusion mutuelle Wrt, initialisé à 1.

Le code d'un processus Rédacteur est donc :

```

Processus Redacteur
Début
    Wait(Wrt)

    /* Ecrire dans le fichier

    Signal(Wrt) ;
Fin.

```

Pour écrire le code d'un processus Lecteur, nous nous intéresserons au cas où on donnerait une certaine priorité aucun processus Lecteurs. C'est à dire qu'aucun Lecteur n'attend, à moins qu'un rédacteur n'ait déjà obtenu la permission pour utiliser le fichier.

Cette solution utilise les variables suivantes :

- Wrt : sémaphore d'exclusion mutuelle qui assure l'accès exclusif d'un rédacteur au fichier.

- Readcount : variable entière contenant le nombre de Lecteurs actuellement dans le fichier.
- Mutex : sémaphore d'exclusion mutuelle pour protéger l'accès à la variable commune ReadCount.

Ces variables sont initialisées à :

Wrt =1, ReadCount=0, Mutex=1

```

Processus Lecteur
Début
    Wait(Mutex) ;
    ReadCount :=ReadCount+1 ;
    Si ReadCount=1
        Alors
            Wait(Wrt)
        Finsi ;
    Signal(Mutex) ;

    /* Lire dans le fichier

    Wait(Mutex) ;
    ReadCount :=ReadCount-1 ;
    Si ReadCount=0
        Alors
            Signal(Wrt)
        Finsi
    Signal(Mutex)

Fin.

```

Le lecteur trouvera à la fin de ce chapitre un exercice proposant une réflexion sur les différentes variantes du problème de Lecteur-Rédacteur : priorité aux rédacteurs, priorité égale entre les lecteurs et les rédacteurs... etc.

2.4.3 Le problème des philosophes :

Cinq philosophes passent leur temps à penser et à manger. Les philosophes partagent une table circulaire commune entourée de cinq (5) chaises, chacune appartenant à l'un d'eux. Chaque philosophe dispose d'un bol de riz. La table est fournie avec 5 baguettes uniquement (voir figure).

Quand un philosophe pense, il n'interagit pas avec les autres philosophes. De temps en temps, un philosophe a faim et essaie de prendre deux baguettes pour manger : une baguette à sa droite et une autre à sa gauche.

Un philosophe peut prendre seulement une baguette à la fois. Evidemment, il ne peut pas prendre une baguette qui est dans la main d'un voisin. Quand un philosophe affamé possède les deux baguettes, il peut manger tant qu'il voudra. Quand il a fini de manger, il dépose ses deux baguettes et recommence à penser.

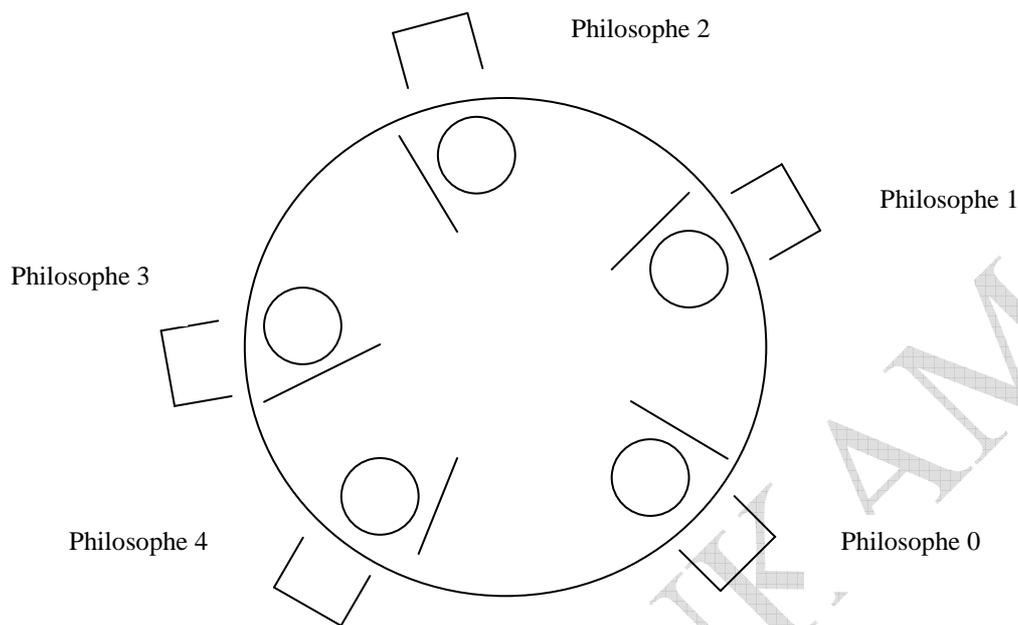


Figure 3.2 Schéma du problème des philosophes.

Le problème des philosophes est un problème classique de synchronisation. Il représente une grande classe de problèmes d'accès concurrents. C'est une représentation simple du besoin d'allouer plusieurs ressources à plusieurs processus tout en évitant le problème de l'interblocage ou de la famine.

Solution 1 (fausse) :

Une première approche du problème des philosophes consiste à modéliser soit l'état des baguettes (occupée/non occupée), soit l'état des philosophes (mange/ne mange pas).

En **modélisant l'état des baguettes**, on déclare un tableau Libre :

Libre : Tableau[0..4] de Logique

Où Libre[i]=Vrai si la baguette i est libre, Libre[i]=Faux si la baguette est occupée. Le tableau Libre est initialisé à Vrai.

Cette solution est donnée ci-après :

```

Processus Philosophe i
Début
  Cycle
    Penser ;
    Tantque Libre[i]=Faux ou Libre[i+1 mod N]=Faux
    Faire
      Attendre
    Fait ;

    Libre[i ]:=Faux ; Libre[i+1 mod N] :=Faux ;

    Mange ;

    Libre[i] :=Vrai ; Libre[i+1 mod N] :=Vrai
  Fin Cycle
Fin.

```

En **modélisant l'état des philosophes**, on déclare un tableau Mange :

Mange : Tableau[0..N] de Logique

Où, Mange[i]=vrai si le philosophe i mange, Mange[i]=Faux sinon. Le tableau Mange est initialisé à Faux.

Le schéma de cette solution est donné ci-après :

```

Processus Philosophe i
Début
  Cycle
    Penser ;
    Tantque (Mange[i-1 mod N]=Vrai) ou (Mange[i+1 mod N]=Vrai)
    Faire
      Attendre
    Fait ;

    Mange[i] :=Vrai ;
    Manger ;
    Mange[i] :=Faux ;
  Fin Cycle
Fin.

```

Cette solution n'est évidemment pas acceptable en raison de son inefficacité (attente active).

Solution 2 (avec des sémaphores) :

Une autre première approche du problème des philosophes consiste à utiliser des sémaphores. Pour cela, on utilise un tableau Chopstick :

Chopstick : Tableau[0..4] de sémaphore.

Tous les éléments de Chopstick sont initialisés à 1. Le code d'un processus Philosophe devient donc :

```

Processus Philosophe i
Début
    Cycle
        Penser ;
        Wait(Chopstick[i]) ;
        Wait(Chopstick[i+1 mod N]) ;
        Manger ;
        Signal(Chopstick[i]) ;
        Signal(Chopstick[i+1 mod N]) ;
    Fin Cycle
Fin.

```

Bien que cette solution garantisse que deux voisins quelconques ne mangent pas simultanément, elle doit néanmoins être rejetée car elle peut conduire à une situation d'interblocage. En effet, supposons que les cinq philosophes aient faim en même temps et que chacun saisisse sa baguette de gauche. Tous les éléments de Chopstick seront donc égaux à 0. Quand chaque philosophe essaiera de saisir sa baguette de droite, il sera retardé pour toujours.

Solution 3 (correcte) :

Une solution correcte du problème des philosophes utilise les variables suivantes :

- Un tableau Etat[0..4] dont chaque élément peut avoir l'une des valeurs : Pense, Faim, Mange.
- Un tableau de sémaphore S[0..4] dont chacun est initialisé à 1.
- Un sémaphore d'exclusion mutuelle Mutex, initialisé à 1.

Le code de cette solution est donné ci-après.

```

Processus Philosophe i
Début
    Cycle
        Penser ;
        Prendre_baguette(i) ;
        Manger ;
        Poser_Baguette(i)
    Fin Cycle
Fin.

```

Les différentes procédures appelées par cette solution, sont détaillées ci-après :

Prendre_Baguette(i) Début Wait(Mutex) ; Etat[i] :=Faim ; Test(i) ; Signal(Mutex) ; Wait(S[i]) Fin.	Poser_Baguette(i) Début Wait(Mutex) ; Etat[i] :=Pense ; Test(i+1 mod N) ; Test(i-1 mod N) ; Signal(Mutex) ; Fin.	Test(i) Début Si Etat[i]=Faim et Etat[i+1 mod N]<>Mange et Etat[i-1 mod N]<>Mange) Alors Etat[i] :=Mange ; Signal(S[i]) Finsi Fin.
--	--	---

2.5 LES OUTILS DE SYNCHRONISATION AVANCES

2.5.1 Les moniteurs :

Le type moniteur est une autre structure de synchronisation de haut niveau inventé par Hoare en 1973.

Présentation :

Un moniteur se caractérise par un ensemble d'opérateurs définis par le programmeur. La représentation d'un type moniteur consiste en des déclarations de variables dont les valeurs définissent l'état d'une instance de type, ainsi que du corps des procédures ou des fonctions qui implémentent les opérations sur le type.

```

Type <nom du moniteur> = Monitor
//Déclaration des variables

Procédure entry P1(...)
Begin
    ...
End ;

Procédure entry P2(...)
Begin
    ...
End ;

Procédure entry PN(...)
Begin
    ...
End ;

Begin
    ... //Code d'initialisation
End.

```

Figure 4.1 Structure d'un moniteur.

La structure d'un moniteur peut être décrite ainsi en utilisant la syntaxe de Pascal Concurrent (un langage variante du Pascal et dédié à la programmation concurrente).

La structure d'un moniteur a ceci de particulier : Il n'y a qu'un seul processus à la fois qui peut être actif dans le moniteur. Le programmeur n'est donc pas obligé de coder cette contrainte de synchronisation explicitement. Le schéma suivant décrit ce fonctionnement ;

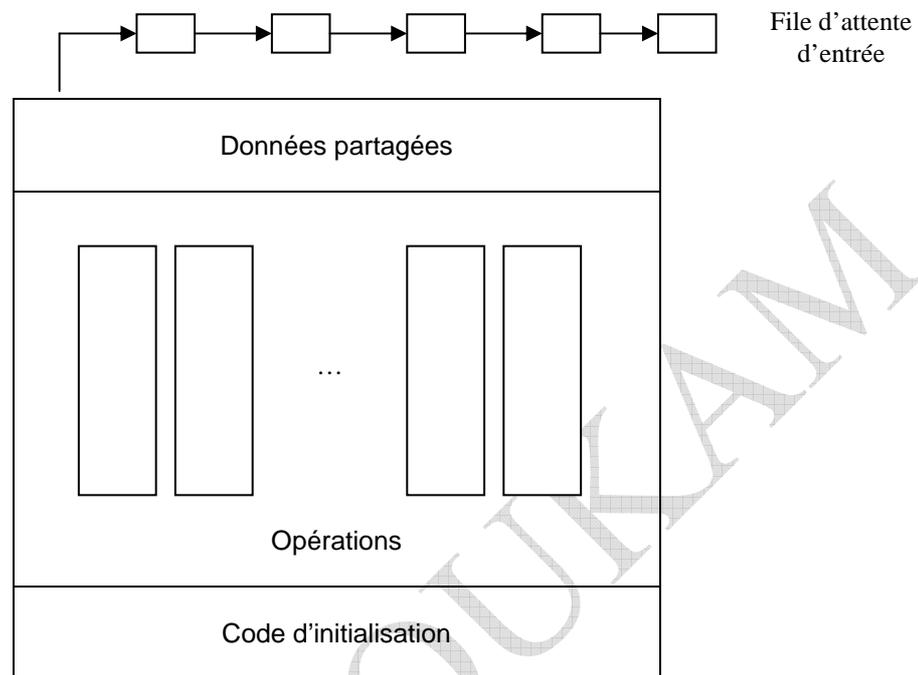


Figure 4.2 Vue schématique d'un moniteur.

Dans un moniteur, on peut déclarer des variables particulières : **les variables conditionnelles** qui permettent d'exprimer des contraintes de synchronisation. La déclaration de ces variables peut se faire ainsi par exemple :

```
Var x, y : condition ;
```

Les seules opérations qu'on peut faire sur une variable conditionnelle sont : **wait** et **signal**. Voici la sémantique de chacune des deux opérations :

- **x.wait** : cette opération provoque la suspension du processus qui l'a appelé. Il ne sera réactivé que par une opération x.signal lancée par un autre processus.
- **x.signal** : cette opération permet de reprendre l'exécution d'un processus suspendu. S'il n'existe aucun processus suspendu, cette opération n'a aucun effet.

Remarque :

Il existe une différence fondamentale entre l'opération wait applicable aux variables conditionnelles et celle applicable sur un sémaphore : La première est toujours bloquante : le processus qui l'exécute est automatiquement bloqué, alors que la seconde n'est bloquante qu'en fonction de la valeur du sémaphore.

On peut schématiser le fonctionnement d'un moniteur avec variables conditionnelles de cette façon :

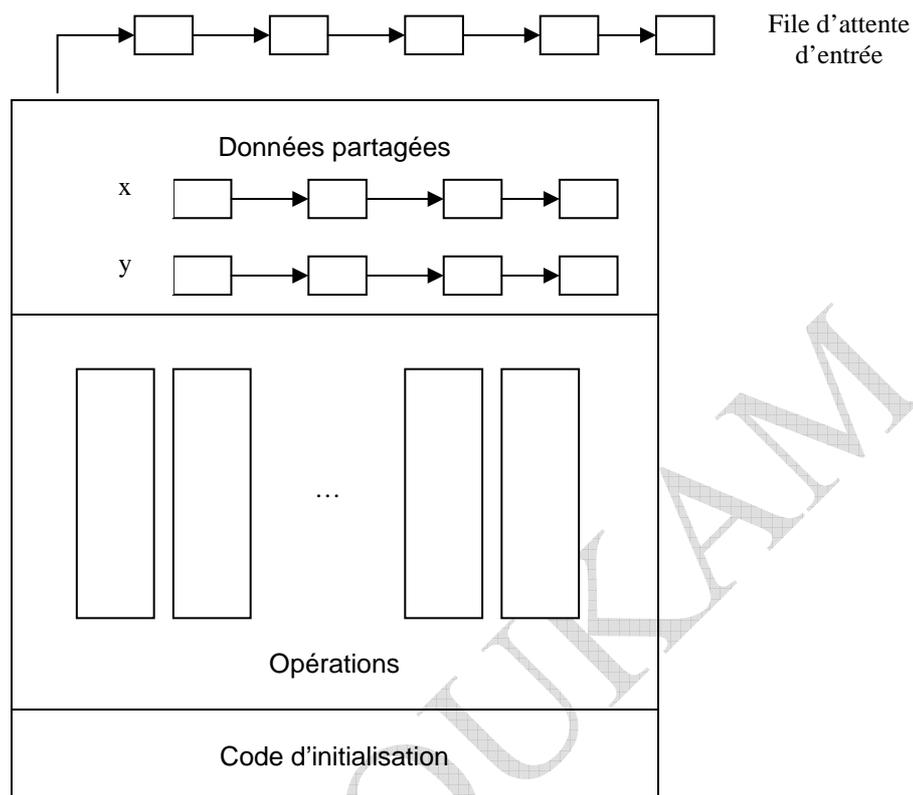


Figure 4.3 Moniteur avec variables conditionnelles.

Lorsque l'opération $x.signal$ est appelée par un processus P et il existe un processus suspendu Q associé à la condition x , deux cas sont à envisager puisqu'il ne doit y avoir qu'un seul processus au niveau du moniteur :

1. P attend jusqu'à ce que Q abandonne le moniteur ou attend une autre condition.
2. Q attend jusqu'à ce que P abandonne le moniteur ou attend une autre condition

Application au problème de la ressource unique :

Un moniteur contrôle l'allocation d'une ressource unique entre plusieurs processus concurrents.

```

Type Ressource_Allocation= Monitor

Var
  Busy : Boolean ;
  X : condition

Procedure entry Acquerir
Begin
  If busy then x.wait ;
  Busy :=True
End ;

Procedure entry Lliberer
Begin
  Busy :=False ;
  x.signal
End ;

Begin
  Busy :=False
End.

```

En utilisant une instance R du moniteur Ressource_Allocation, le code d'un processus P devient :

```

Processus
Début

  R.Acquerir

  //Accéder à la ressource

  R.Liberer

Fin.

```

Application au problème des philosophes :

Reprenons le problème des philosophes en utilisant les moniteurs. Pour cela déclarons les variables suivantes :

- Etat : Tableau[0..4] de (Faim, Pense, Mange)
- Self : Tableau[0..4] de Condition

La solution à proposer doit garantir les règles suivantes :

- Le philosophe i ne peut fixer la variable $etat[i]$ à mange que si ses deux voisins $(i-1 \text{ mod } 5)$ et $(i+1 \text{ mod } 5)$ ne sont pas en train de manger.
- Le philosophe i peut se retarder quand il a faim mais il est incapable d'obtenir les baguettes dont il a besoin.

La structure de moniteur utilisée pour la synchronisation des philosophes est :

```

Type Diner_Philosophe= Monitor

Var
  Etat : Array[0..4] of (Pense, Faim, Mange) ;
  Self : Array[0..4] of Condition ;

Procedure entry Prendre_Baguette(i : 0..4)
Begin
  Etat[i] :=Faim ;
  Test(i) ;
  If Etat[i]<>Mange
    Then Self[i].wait ;
End ;

Procedure entry Poser_Baguette(i : 0..4)
Begin
  Etat[i] :=Pense ;
  Test(i-1 mod 5) ;
  Test(i+1 mod 5)
End ;

Procedure entry Test(k : 0..4)
Begin
  If Etat[k-1 mod 5]<>mange and Etat[k]=Faim and Etat[k+1 mod 5]<>Mange
  Then Begin
    Etat[k] :=Mange ;
    Self[k] .signal
  End
End ;

Begin
  For i=0 to 4 do Etat[i] :=Pense
End.

```

En utilisant une instance dp du moniteur Diner_Philosophe, le code d'un processus philosophe devient :

```

Processus Philosophe i
Début
  Cycle
    Penser ;
    Dp.prendre_Baguette(i)
    Manger ;
    Dp.Poser_Baguette(i) ;
  Fincycle
Fin.

```

2.5.2 Les régions critiques :

Présentation : Une *région critique* est un autre outil de synchronisation de haut niveau. Dans une région critique, toute variable v de type T partagée par plusieurs processus doit être déclarée avec le mot réservé shared :

```
Var v : shared T ;
```

On peut accéder à la variable V uniquement dans une instruction *region* de la forme suivante :

```
region v when B do S ;
```

Cette expression signifie que, pendant que l'on exécute l'instruction S, aucun autre processus ne peut accéder à la variable v. L'expression B est une expression logique dont dépend l'accès à la variable v. Quand un processus essaye d'entrer dans la région critique, il évalue l'expression B ; si sa valeur est vraie il pourra exécuter l'instruction S, sinon il sera retardé jusqu'à ce que la valeur de B devienne vraie et qu'aucun autre processus n'est en train d'accéder à la variable v.

Application : Problème du Producteur-Consommateur avec buffer limité.

On utilisera une variable Buffer partagée de type Record :

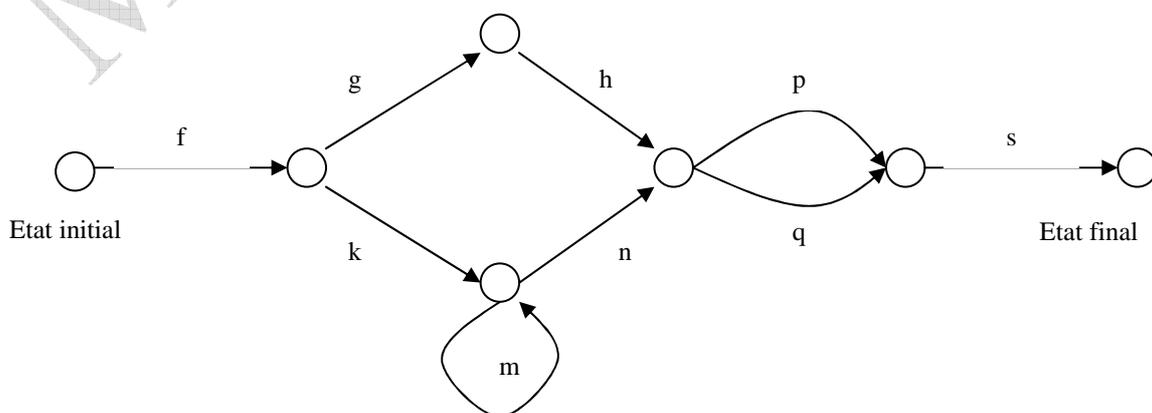
```
Var Buffer : Shared Record
    Tableau : Array[0..n-1] of Element ;
    Count, In, Out : Integer
End ;
```

Le producteur et le consommateur utilisent la région critique en exécutant respectivement les codes suivants :

```
Processus Producteur
Début
    While True
    Do Begin
        Produire un message dans ZoneP

        Region Buffer when Count<n
        Do Begin
            Buffer[In] :=ZoneP ;
            In :=In+1 mod N ;
            Counter :=Counter+1
        End ;
    End
End
Fin.
```

```
Processus Consommateur
Début
    While True
    Do Begin
        Region Buffer when count>0
        Do Begin
            ZoneC :=Buffer[Out] ;
            Out :=Out+1 mod N ;
            Counter :=Counter-1 ;
        End ;
        Consommer le message de ZoneC
    End
End
Fin.
```



Une expression de chemin pour laquelle il est possible de trouver un graphe tel que le même nom de procédure ne figure pas sur plus d'un arc issu du même nœud est appelé chemin simple.

Un chemin simple peut être mis en œuvre par génération d'opérations P et V sur sémaphores.

Application :

Soit un tampon pouvant contenir un message unique. Des processus garnissent ou vident ce tampon dans un ordre imprévisible.

```
class TamponUnique;
var mess : Message;
path déposer; retirer end
procédure déposer(m:Message);
begin
mess := m
end; {de déposer}
procédure retirer:Message;
begin
retirer := mess
end; {de retirer}
begin
mess := nil
end; {de TamponUnique}
```

2.6 LA COMMUNICATION INTER-PROCESSUS :

La communication interprocessus (interprocess communication, IPC) consiste à transférer des données entre les processus. Par exemple, un navigateur Internet peut demander une page à un serveur, qui envoie alors les données HTML.

La communication interprocessus peut se faire de différentes manières : mémoire partagée, mémoire mappée, tubes, files et socket.

- La mémoire partagée permet aux processus de communiquer simplement en lisant ou écrivant dans un emplacement mémoire prédéfini.
- La mémoire mappée est similaire à la mémoire partagée, excepté qu'elle est associée à un fichier.
- Les tubes permettent une communication séquentielle d'un processus à l'autre.
- Les files FIFO sont similaires aux tubes excepté que des processus sans lien peuvent communiquer car le tube reçoit un nom dans le système de fichiers.
- Les sockets permettent la communication entre des processus sans lien, pouvant se trouver sur des machines distinctes.

Ces types d'IPC diffèrent selon les critères suivants:

- Ils restreignent ou non la communication à des processus liés (processus ayant un ancêtre commun), à des processus partageant le même système de fichiers ou à tout ordinateur connecté à un réseau.
- Un processus communiquant n'est limité qu'à la lecture ou qu'à l'écriture de données.

- Le nombre de processus pouvant communiquer.
- Les processus qui communiquent sont-ils synchronisés par l'IPC ? par exemple, un processus lecteur s'interrompt-il jusqu'à ce qu'il y ait des données à lire ?.

2.6.1 MEMOIRE PARTAGEE

Une des méthodes de communication interprocessus les plus simples est d'utiliser la mémoire partagée. La mémoire partagée permet à deux processus ou plus d'accéder à la même zone mémoire comme s'ils avaient leurs pointeurs dirigés vers le même espace mémoire. Lorsqu'un processus modifie la mémoire, tous les autres processus voient la modification.

La mémoire partagée est la forme de communication interprocessus la plus rapide car tous les processus partagent la même mémoire. Elle évite également les copies de données inutiles.

Pour utiliser un segment de mémoire partagée, un processus doit allouer le segment. Puis, chaque processus désirant accéder au segment doit l'attacher. Après avoir fini d'utiliser le segment, chaque processus le détache. À un moment ou à un autre, un processus doit libérer le segment.

Sous Unix, un processus alloue un segment de mémoire partagée en utilisant `shmget` (« SHared Memory GET », obtention de mémoire partagée). Son premier paramètre est une clé entière qui indique le segment à créer. Le second paramètre indique le nombre d'octets du segment. Le troisième paramètre est un ensemble d'indicateurs binaires décrivant les options demandées : s'agit-il d'un nouveau segment ou un segment existant qu'il faut attacher, permissions de lecture/écriture, ... etc.

Inconvénient :

Les segments de mémoire partagée permettent une communication bidirectionnelle rapide entre n'importe quel nombre de processus. Chaque utilisateur peut à la fois lire et écrire. Cependant, le principe de l'exclusion mutuelle, nécessaire en cas d'accès concurrent, n'est pas garanti avec ce mode communication.

2.6.2 MEMOIRE MAPPEE

La mémoire mappée permet à différents processus de communiquer via un fichier partagé.

Pour mettre en correspondance un fichier ordinaire avec la mémoire d'un processus, on utilise l'appel `mmap` (« Memory MAPped », Mémoire mappée). Le premier paramètre est l'adresse de location du fichier. Le second paramètre est la longueur de l'espace de correspondance en octets. Le troisième paramètre définit la protection de l'intervalle d'adresses mis en correspondance.

2.6.3. TUBES

Un tube est un dispositif de communication qui permet une communication à sens unique. Les données écrites sur l'« extrémité d'écriture » du tube sont lues depuis l'« extrémité de lecture ». Les tubes sont des dispositifs séquentiels; les données sont toujours lues dans l'ordre où elles ont été écrites. Typiquement, un tube est utilisé pour la communication entre deux threads d'un même processus ou entre processus père et fils.

Dans un shell, le symbole `|` crée un tube. Par exemple, cette commande provoque la création par le shell de deux processus fils, l'un pour `ls` et l'autre pour `less`:

```
ls | less
```

Le shell crée également un tube connectant la sortie standard du processus ls avec l'entrée standard de less. Les noms des fichiers listés par ls sont envoyés à less dans le même ordre que s'ils étaient envoyés directement au terminal.

La capacité d'un tube est limitée. Si le processus rédacteur écrit plus vite que la vitesse à laquelle le processus lecteur consomme les données, et si le tube ne peut pas contenir de données supplémentaires, le processus rédacteur est bloqué jusqu'à ce qu'il y ait à nouveau de la place dans le tube. Si le lecteur essaie de lire mais qu'il n'y a plus de données disponibles, il est bloqué jusqu'à ce que ce ne soit plus le cas. Ainsi, le tube synchronise automatiquement les deux processus.

La création d'un tube se fait grâce à la fonction pipe. Elle admet comme paramètre un tableau de deux entiers. L'appel à pipe stocke le descripteur de fichier en lecture à l'indice zéro et le descripteur de fichier en écriture à l'indice un.

Exemple de création d'un tube :

```
int pipe_fds[2];
int fr;
int fw;

pipe (pipe_fds);
fr = pipe_fds[0];
fw = pipe_fds[1];
```

2.6.4 LES FILES FIFO

Une file premier entré, premier sorti (first-in, first-out, FIFO) est un tube qui dispose d'un nom dans le système de fichiers. Tout processus peut ouvrir ou fermer la file FIFO; les processus raccordés aux extrémités du tube n'ont pas à avoir de lien de parenté. Les FIFO sont également appelés canaux nommés.

On peut créer une FIFO via la commande mkfifo.

Pour créer une FIFO par programmation, utilisez la fonction mkfifo. Le premier argument est l'emplacement où créer la FIFO; le second paramètre spécifie les permissions du propriétaire du tube, de son groupe et des autres utilisateurs.

L'accès à une FIFO se fait de la même façon que pour un fichier ordinaire

2.6.5 LES SOCKETS

Un socket est un dispositif de communication bidirectionnel pouvant être utilisé pour communiquer avec un autre processus sur la même machine ou avec un processus s'exécutant sur d'autres machines. Les programmes Internet comme Telnet, rlogin, FTP, talk et le World Wide Web utilisent des sockets.

Pour créer un socket, il faut indiquer trois paramètres: le style de communication, l'espace de nommage et le protocole.

Un style de communication contrôle la façon dont le socket traite les données transmises et définit le nombre d'interlocuteurs. Lorsque des données sont envoyées via le socket, elles sont découpées en morceaux appelés paquets. Le style de communication détermine comment sont gérés ces paquets et comment ils sont envoyés de l'émetteur vers le destinataire.

- Le style *connexion* garantit la remise de tous les paquets dans leur ordre d'émission. Si des paquets sont perdus ou mélangés à cause de problèmes dans le réseau, le destinataire demande automatiquement leur retransmission à l'émetteur.
- Le style *datagramme* ne garantit pas la remise ou l'ordre d'arrivée des paquets. Des paquets peuvent être perdus ou mélangés à cause de problèmes dans le réseau.

L'espace de nommage d'un socket spécifie comment les adresses de socket sont écrites. Une adresse de socket identifie l'extrémité d'une connexion par socket. Par exemple, les adresses de socket dans « l'espace de nommage local » sont des noms de fichiers ordinaires. Dans « l'espace de nommage Internet », une adresse de socket est composée de l'adresse Internet (également appelée adresse IP) d'un hôte connecté au réseau et d'un numéro de port. Le numéro de port permet de faire la distinction entre plusieurs sockets sur le même hôte.

Un protocole spécifie comment les données sont transmises. Parmi ces protocoles, on peut citer TCP/IP; les deux protocoles principaux utilisés pour Internet, le protocole réseau AppleTalk; et le protocole de communication locale d'UNIX.