1. Processus séquentiels

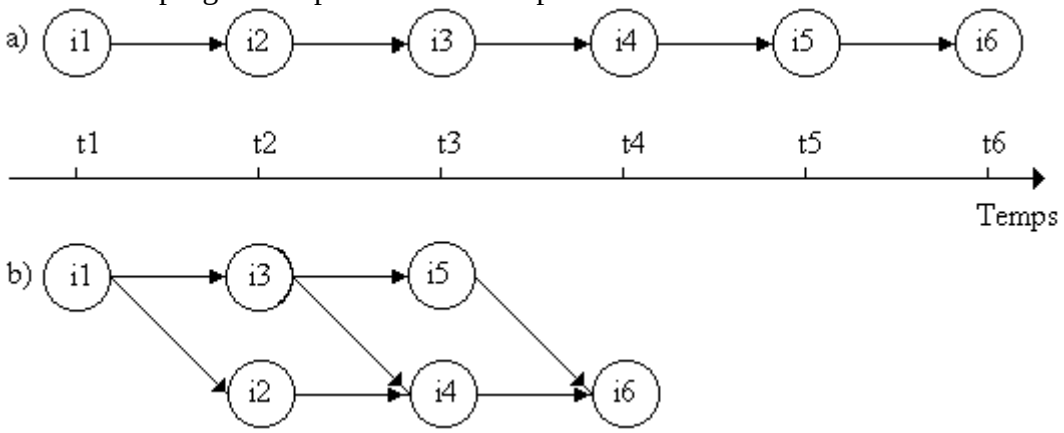
Les exécutions des instructions d'un programme doivent respecter un certain ordre de précédence. On peut représenter l'exécution d'un programme à l'aide d'un graphe de précédence.

- **Exemple**

Soit un programme qui lit 2 variables a et b à partir d'un même fichier, calcule la somme de ces 2 variables et imprime ces 2 variables et leur somme sur la même imprimante.

i1. lire a
i2. écrire a
i3. lire b
i4. écrire b
i5. calculer c = a + b
i6. écrire c

L'exécution de ce programme peut être faite de plusieurs manières :



Exécution a): i1 < i2 < i3 < i4 < i5 < i6
Exécution b) : i1 < (i2 || i3) < (i5 || i4) < i6 (< : avant; || : en parallèle)

2. Définition d'un processus séquentiel

Dans l'exécution (a) les instructions ont été exécutées dans un ordre séquentiel strict; on appellera cette exécution séquentielle un *processus séquentiel* (ou processus).
L'exécution (b) ne correspond pas à un processus séquentiel car certaines instructions telles que i2 et i3 peuvent se dérouler en parallèle (ou simultanément); on peut décomposer cette exécution en 2 processus séquentiels concurrents:

P1 : i1 < i3 < i5 ;
P2 : i2 < i4 < i6.

- **Différence entre programme et processus**

De manière informelle, un processus est l'exécution séquentielle d'un programme. On peut dire qu'un processus est une représentation abstraite de l'exécution d'un programme.
Un processus est une entité dynamique représentant l'exécution séquentielle d'un programme sur le processeur; l'état d'un processus évolue au cours de l'exécution de son programme. Alors qu'un programme est une entité statique qui peut donner naissance à un ou plusieurs processus.

3. Relations entre processus

3.1.Processus parallèles

On appelle processus parallèles des processus séquentiels dont les exécutions chevauchent dans le temps.

- **Exemple**

Soient deux processus P1 et P2 dont les exécutions de leur programme sont représentées par les schémas suivants:

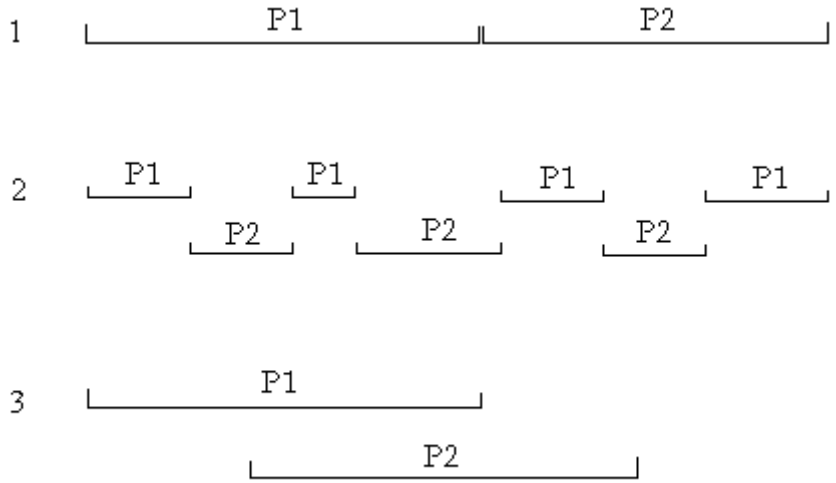


Schéma1: Exécution séquentielle du programme du processus P1 ensuite exécution séquentielle du programme du processus P2 sur un seul Processeur.

Schéma2: Exécution séquentielle d'une séquence d'instructions de P1 ensuite exécution séquentielle d'une séquence d'instructions de P sur un seul Processeur.

Schéma3: Exécutions parallèles des programmes des processus P1 et P2, ce schéma n'est possible que si l'on dispose de 2 processeurs physiques (U.C).

Pour comparer ces 3 schémas, on introduire la notion de *niveau d'observation* qui peut être:

- le programme
- la procédure (séquence d'instructions)
- l'instruction

Le schéma1 représente une exécution séquentielle de 2 processus quel que soit le niveau d'observation. Les événements qui nous intéressent sont : début(P1), fin(P1),début(P2) et fin(P2) avec fin(P1)< début(P2).
Le *schéma2* et le *schéma3* sont des exécutions parallèles de 2 processus séquentiels :

Si le niveau d'observation est le programme ou la séquence d'instructions alors le *schéma2* et le *schéma3* sont équivalents.

Si le niveau d'observation est l'instruction alors, le *schéma2* est différent du *schéma3*.

Dans le *schéma2* les instructions s'exécutent de manière séquentielle sur un seul processeur (U.C), on parle alors de psœudo-parallèlisme.

Dans le *schéma3* les instructions des processus P1 et P2 s'exécutent sur 2 processeurs (U.C) distincts : c'est du parallélisme réel.

- **Remarque**

La distinction entre les différents schémas est liée au niveau d'observation.
Classes de processus parallèles
Selon la nature des relations qui existent entre les processus parallèles on distingue deux classes de processus parallèles.

3.2.Les processus indépendants

Ce sont des processus parallèles qui n'ont aucune interaction. Les contextes de ces processus sont disjoints. Ces processus ne se connaissent pas; la seule relation entre ces processus est une *relation de compétition*. Il y a *relation de compétition* lorsque plusieurs processus tentent d'utiliser simultanément un objet non partageable ou à accès exclusif (un seul processus peut l'utiliser à la fois).

Les processus indépendants entrent en conflit pour l'utilisation d'objet qu'ils doivent quitter sans le modifier, car l'objet doit être le même pour tous. Ces objets de conflit constituent ce que l'on appelle *ressources* dans les systèmes (Processeur, périphériques, ...). La relation de compétition peut être éliminée si l'on disposait d'un nombre suffisant de ressources pour satisfaire les besoins des processus. Or les besoins des processus ne sont pas connus et on ne peut pas créer dynamiquement des ressources physiques.

Le résultat de chacun des processus ne peut pas être influencé par les autres processus. Par contre, leur comportement peut être affecté. Si une ressource en un exemplaire unique (Processeur ou imprimante) est l'objet d'une compétition entre deux processus alors l'un d'eux devra attendre la fin de l'utilisation de l'autre avant d'utiliser à son tour la ressource : Un processus a alors été ralenti par un autre.

Si deux processus p1 et p2 sont indépendants les exécutions suivantes donnent les mêmes résultats:
p1 // p2
p1; p2
p2; p1

"//" Exécution parallèle de p1 et p2 ";" Exécution séquentielle.

3.3.Les processus concurrents (coopérants)

Ce sont des processus qui coopèrent pour la réalisation d'un travail commun (tâche commune). Ces processus se partagent et/ou s'échangent des informations; Ils se connaissent de manière directe ou indirecte. La coopération entre ces processus nécessite une

synchronisation de leurs actions (opérations). Les processus concurrents ont 2 types de relations :

- Relation de compétition pour l'usage de ressource commune.
- Relation de coopération.

Coopération par partage : les processus se connaissent indirectement.

- **Exemple**

Des processus se partageant une base de données de comptes Bancaires (Processus virement, Processus retrait, processus consultation,…).

Coopération par communication : les processus se connaissent directement et s'échangent des messages (envoyer, recevoir).

4. L'exclusion Mutuelle

Un processus coopératif peut affecter ou être affecté par les autres processus en exécution dans le système. Les processus coopératifs peuvent :

- partager un espace d'adresses logique (code et données)
- partager seulement les données au moyen de fichiers.

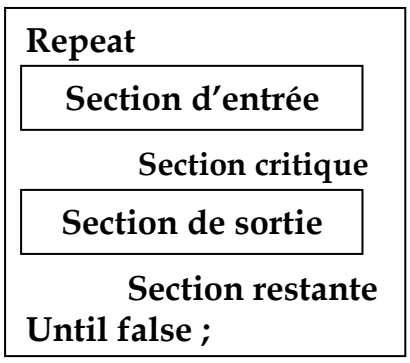
Les accès *concurrents* à des données partagées peuvent produire des *incohérences* de données.

4.1.Section critique

Soit n processus $\{P_0, P_1, \ldots, P_{n-1}\}$
Chaque processus possède un segment de code partagé ou **section critique**, dans laquelle le processus peut modifier des variables communes, actualiser une table ou écrire un fichier.

Convention : Lorsqu'un processus exécute sa section critique, aucun autre processus n'est autorisé à exécuter sa section critique. On dit que l'exécution des sections critiques est *mutuellement exclusive* dans le temps.

Chaque processus doit demander la permission d'entrer dans sa section critique. Pour cela, il utilise une section de code dite **section d'entrée**. Lorsqu'il termine sa section critique, il exécute une **section de sortie** pour libérer les autres processus. Le reste du code constitue la **section restante**. Voici la structure générale d'un processus typique P_i :



4.2.Ressource critique

Une ressource critique est une ressource ne pouvant être utilisée que par un seul processus à la fois (ressource à un seul point d'accès, ressource à accès exclusif ou ressource non partageable). *Une ressource critique doit être utilisée en exclusion mutuelle.*

- **Exemples**: processeur, imprimante, fichier, variable …

4.3.Réalisation de l'exclusion mutuelle

- **Hypothèses de travail (Dijkstra)**

Les vitesses relatives des processus sont quelconques et inconnues. On suppose que tout processus sort de sa section au bout d'un temps fini. L'ordre d'accès à la ressource est quelconque (ordre d'exécution de la section critique). La solution doit avoir les *propriétés* suivantes:

a) A tout instant *un processus au plus* peut se trouver en section critique (par définition).

b) Si plusieurs processus sont bloqués en attente de la ressource critique alors qu'aucun processus ne se trouve en section critique l'un d'eux doit pouvoir y entrer au bout d'un temps fini (éviter les blocages indéfinis).

c) Si un processus est bloqué en dehors d'une section critique, ce blocage ne doit pas empêcher l'entrée d'un autre processus en section critique.

d) La solution doit être la même pour tous les processus, c'est à dire aucun processus ne doit jouer le rôle de privilégié.

4.3.1. Solutions logicielles: Utilisation de variables communes

Les algorithmes d'exclusion mutuelle utilisent *l'attente active* et des *variables partagées* (ou variables globales) ; ils algorithmes doivent satisfaire les trois premières propriétés de Dijkstra (exclusion mutuelle (a), éviter l'interblocage (b) et éviter la famine (c)). Ces algorithmes doivent d'assurer l'équité (d) autant que possible et d'éviter l'attente inutile (mauvaise utilisation du processeur). On suppose que l'affectation et le test sont des instructions (ou opérations) indivisibles.

- **Solution pour deux processus avec une variable partagée**

Cette solution consiste à utiliser une variable partagée *tour* indiquant le numéro du processus qui a accès à la section critique; tour peut prendre la valeur 0 ou 1 (0 : processus 0; 1 : processus 1) et est initialisée à n'importe laquelle des deux valeurs. Lorsque *tour = i* le processus *i* peut entrer en section critique.

tour : entier; Tour =0 ou 1;

Processus 0	**Processus1**
Début *Répéter* *tantque tour # 0 faire rien finfaire* *« section critique »* *tour = 1;*	*Début* *Répéter* *tantque tour # 1 faire rien finfaire* *« section critique »*

jusqu'à faux; *fin;*	*tour = 0;* *jusqu'à faux;* *fin;*

Cette solution garantit qu'un seul processus à la fois peut se trouver en section critique. Cependant, elle ne respecte pas la contrainte (c) : Les processus ne peuvent entrer en section critique qu'à tour de rôle(alternance stricte).

- **Solution pour deux processus avec deux variables partagées**

Cette solution consiste à utiliser un tableau partagé v de booléen(initialisé à faux). Ainsi *v[i] = vrai* indique que le processus i est dans sa section critique ou demande à y entrer .

v : tableau [0..1] de booléen;
v[0] = faux; v[1] = faux;

Processus 0	**Processus1**
Début *Répéter* *V[0] = vrai;* *tantque v[1]=vrai faire rien finfaire* *« section critique »* *v[0] = faux;* *jusqu'à faux;* *fin;*	*Début* *Répéter* *v[1] = vrai;* *tantque v[0]=vrai faire rien finfaire* *« section critique »* *[1] = faux;* *jusqu'à faux;* *fin;*

Cette solution garantit qu'un seul processus, à la fois, peut se trouver dans sa section critique. Néanmoins, elle ne respecte pas la contrainte (b) : Si les deux processus s'engagent simultanément en affectant la valeur vraie à la variable v[i]; aucun ne pourra entrer dans la section critique (chacun croyant que l'autre est engagé dans la section critique).

- **Solution de Dekker pour deux processus avec 3 variables partagées**

L'algorithme de Dekker résout le problème d'exclusion mutuelle entre deux processus P0 et P1 en utilisant trois variables communes.

Algorithme de Dekker

```
var     v: tableau[0..1] de booléen;       tour: 0..1;
/* initialisation des variables */
v[0]:= faux;    v[1]:= faux;      tour:=0;        /*  ou tour
:= 1 */
Le protocole d'un processus_i  [i=0,1] est alors:
Processus_i
     Répéter
          j:=1-i;
          v[i] := vrai;
          tantque v[j] faire
                    si  tour = j alors
                    v[i] := faux;
                    tantque tour = j faire   rien  finfaire;
                    v[i] := vrai
```

```
                    finsi;
               finfaire;
          < section critique >
          tour := j;
          v[i] := faux;
           Le reste du programme
     Jusqu'à faux;
fin du processus_i
```

- **Solution de Dijkstra pour n processus**

L'algorithme de Dijksta résout le problème d'exclusion mutuelle entre n processus P0,… Pn-1 en utilisant deux tableaux communs de n booléens chacun et une variable commune.

```
var b,c: tableau[0..n] de booléen;
          tour: 0..n;
/* initialisation des variables */
pour i=0 à n faire
     b[i]:= faux;
     c[i]:= faux;
finfaire
tour:=n;        /*  tour := n; le processus n  n'existe pas*/
Le protocole d'un processus_i  [i=0,n-1] est alors:
Processus_i
     entier j;
     répéter
          b[i] := vrai; /* le processus i demande à faire tour
= i */
          répéter
               tantque tour # i faire
                    c[i]= faux;
                    si b[tour]= faux alors  tour =i finsi;
               finfaire;
               c[i]= vrai;
               j=0;
               tantque  (j<n) et ( (j=i) ou c[j]= faux)   faire
                    j=j+1;
               finfaire;
          jusqu'à j=n;
          < section critique >
          b[i] := faux;   c[i] := faux;
          tour := n;
           Le reste du programme
     jusqu'à faux;
fin du processus_i
```

- **Autre solution pour plusieurs processus**

Pour synchroniser plusieurs processus, nous utilisons un algorithme connu sous le nom d'**algorithme du boulanger** : en rentrant dans un magasin, chaque client reçoit un numéro. Le client suivant est celui qui a le plus petit numéro. Si deux processus Pi et Pj ont le même numéro, on sert le processus Pi tel que i<j. Nous utilisons les variables suivantes :

```
var choosing : array [0..n-1] of boolean ; --- initialisé à
false ---
number: array[0..n-1] of integer ; --- initialisé à 0 ---
```

Conventions
(a,b) < (c,d) si a<c ou si a=c, b<d
max(a0,a1,..,an-1)=k | k≥ai ∀ i∈[0..n-1]

Voici la structure d'un processus Pi :

```
Repeat
choosing :=true ;
number[i] :=max(number[0],..,number[n-1])+1 ;
choosing :=false ;
for j :=0 to n-1
     do begin
     while choosing[j] do no-op ;
     while number[j]≠0 and (number[j],j)<(number[i],i) do no-
op ;
end;

<Section critique>

Number[i] :=0 ;

<Section restante>
Until false ;
```

- **Solution de Peterson pour deux processus avec 3 variables partagées**

Algorithme de Peterson(2)

```
var       v: tableau[0..1] de booléen;          tour: 0..1;
/* initialisation des variables */
     v[0]:= faux;        v[1]:= faux;
Le protocole d'un processus_i  [i = 0,1   et j = i+1 mod 2] est
le suivant:

Processus_i
     répéter
          v[i] := vrai;
          tour :=j;
          attendre ( v[j] = faux )  ou (tour = i )
          /* tantque  v[j]  et (tour = j) faire  rien finfaire
*/
          < section critique>
          v[i] := faux;
          le reste du programme
     jusqu'à faux;
fin du processus_i
```

- **Solution de Peterson pour n processus**

Algorithme de Peterson(n)

```
/* déclaration des variables communes */
var     v: tableau[0..n-1] de -1..n-2;
          tour: tableau[0..n-2] de 0..n-1;
/* initialisation des variables */
Pour i  de 0 à  n-1 faire
    v[i] := -1;
finfaire;
pour i  de 0 à  n-2 faire
    tour[i]:= 0;
finfaire;
```

Le protocole d'un $processus_i$ [i = 0 .. n-1] est alors :

```
Processusi
     var   j: 0..n-2;   k:0..n-1;
     répéter
          pour j de 0 à n-2 faire
               v[i] :=j;
               tour[j] :=i;
               attendre (∀k #i : v[k] < j)  ou (tour[j] # i)
          finfaire
          < section critique>
          v[i] := -1;
          le reste du programme
     jusqu'à faux;
fin de l'algorithme
```

4.3.2. Solutions matérielles

4.3.2.1.Machine monoprocesseur: Les interruptions

Sur une machine monoprocesseur l'exclusion mutuelle peut être assurée en masquant les interruptions; Cela évite au processus actif de perdre le processeur, pendant l'exécution de sa section critique, en fin de quantum (interruption horloge) ou lors d'un réveil d'un processus plus prioritaire que lui.

1- Entrée en section critique	*:*	*Masquer les interruptions* *< Section critique >*
2- Sortie de la section critique	*:*	*Démasquer les interruptions*

Cette solution doit être utilisée avec précaution car elle peut affecter le temps de réponse du système. Elle est, généralement, utilisée pour implémenter des primitives très courtes du noyau (noyau du système) telles que les primitives P et V que nous verrons dans la section suivante.

La section critique doit être la plus courte possible car certaines interruptions de même niveau peuvent être déclenchées plusieurs fois successivement pendant l'exécution de la section

critique, et elles ne sont mémorisées qu'une seule fois. Par exemple une interruption horloge non traitée en temps voulu peut retarder l'horloge du système.

L'utilisation de cette solution doit être réservée à des utilisateurs ("avertis") travaillant en mode maître (ou système) afin d'éviter le blocage(volontaire ou involontaire) du système(l'oubli du démasquage des interruptions en fin de la section critique ou entrée/sortie dans une section critique,…).

4.3.2.2.Machine multiprocesseur : Les instructions spéciales

Sur une machine multiprocesseur le masquage des interruptions est insuffisant pour assurer l'exclusion mutuelle entre des processus s'exécutant sur des processeurs différents. Certains processeurs offrent des instructions spéciales permettant de lire, et modifier un mot ou cellule mémoire de manière indivisible (ou atomique). L'instruction la plus connue est TAS(m).

- **Test And Set** ou **TAS(m)**

Cette instruction existe sur certaines machines (Motorola 68000) et fonctionne comme suit:

```
TAS(m):
     bloquer l'accès à la cellule mémoire m;
     lire le contenu de m;
          si m = 0 alors
                m := 1;
                compteur_ordinal := compteu_ ordinal + 2;
          sinon
                  compteur_ordinal := compteur_ordinal + 1;
          finsi;
          Libérer l'accès à la cellule mémoire m;
fin_de_TAS,
```

La consultation et la modification de *m* sont réalisées par une seule instruction indivisible.

Emploi de TAS:

Soit *v* la variable commune protégeant la ressource critique.
Initialisation v=0;

1. Entrée en section critique	*Etiq:*	*TAS(v);* *Aller à Etiq;* *< Section Critique >*
2. Sortie de la section critique		*v := 0;*

- **Instruction LOCK XCHG du 80x86 :**

Le préfixe LOCK permet de rendre l'instruction (XCHG) qui suit indivisible.

```
Initialisation    mov v,0
boucle: MOV   AL,1              ; AL =1
        LOCK   XCHG   AL, v   ; échange du contenu du registre AL avec celui de v
                               ; v=1 à la fin de l'exécution de l'instruction et
```

```
                    ; AL = ancienne valeur de v.
CMP    AL, 0        ; tester section critique libre : si v était égale à 0;
JNE    boucle       ; si section critique occupée aller à boucle.
Section critique
MOV    v, 0         ; libérer la section critique
```

- **Solution pour une machine multiprocesseur en utilisant les interruptions et l'instruction spéciale TAS**

Sur une machine multiprocesseur, la solution consiste à masquer les interruptions pour assurer l'exclusion mutuelle entre les processus s'exécutant sur un processeur donné et utiliser une instruction spéciale telle que TAS pour résoudre les conflits entre processeurs: donc on aura deux étapes:

a. Permettre à un seul processeur d'accéder à la section critique,
b. Permettre à un seul processus s'exécutant sur ce processeur d'exécuter sa section critique.

On doit d'abord masquer les interruptions et ensuite demander l'entrée en section critique à l'aide de l'instruction TAS.

```
Début
   masquer les interruptions;
   boucle : TAS(m);                /* attente active */
                Aller à boucle;
                    <Section Critique>;
               m=0; /* libérer de la  section critique */
   démasquer les interruptions;
   Le reste du programme
Fin;
```

5. Les sémaphores de Dijkstra

5.1.Définition

Un sémaphore S est une variable entière, à laquelle (sauf pour l'initialisation), on accède seulement à travers deux opérations standards *atomiques* (sans interruption):

wait, notée **P** (en hollandais proberen, tester)

```
wait(S): while S≤0 do no-op ;
         S :=S-1 ;
```

signal, notée **V** (en hollandais verhogen, incrémenter).

```
signal(S): S :=S+1 ;
```

Lorsqu'un processus modifie la valeur du sémaphore, aucun autre processus ne peut le modifier.

Utilisation : Traiter le problème de la section critique pour plusieurs processus.
Les n processus partagent un sémaphore **mutex** initialisé à 1.

Processus P_i

Repeat

wait(mutex) ;

Section critique

signal(mutex) ;

Section restante

Until false ;

5.2.Implémentation

Les solutions que nous avons abordées requièrent une **attente active**. Pendant qu'un processus se trouve dans sa section critique, tout autre processus essayant d'entrer dans sa section critique doit itérer continuellement dans le code d'entrée. L'attente active gaspille des cycles d'UC.

Le type de sémaphore que nous avons défini est dit **spinlock** (le processus tourne pendant qu'il est bloqué jusqu'à son prochain scheduling, donc sans commutation de contexte).
Pour éliminer le problème de l'attente active, il faut modifier la définition des opérations **wait** et **signal**. Quand un processus exécute l'opération **wait** et trouve que la valeur du sémaphore est négative, il doit attendre. Au lieu d'une attente active, le processus se bloque lui-même en invoquant l'opération **block**. Le processus sera alors placé dans une file d'attente associée au sémaphore et son état sera *commuté* à l'état d'attente. Le scheduleur de l'UC sélectionnera un autre processus afin de l'exécuter.

Le processus bloqué, en attente d'un sémaphore S, sera redémarré lorsqu'un processus exécute l'opération **signal** sur le même sémaphore. Le processus bloqué sera redémarré par une opération **wakeup**. Cette opération change l'état du processus de « en attente » à « prêt ».
Les opérations **block** et **wakeup** sont fournies par le système d'exploitation comme des *appels système* de base.

- **Nouvelles définitions :**

Nous définissons un sémaphore comme un enregistrement :

```
type semaphore = record
     value :integer ;
     L : list of process ;
     end ;
```

Ensuite les opérations **wait** et **signal** seront définies comme suit :

```
wait(S) : S.value :=S.value-1 ;
     if S.value<0
          then begin
               ajouter ce processus à S.L ;
               block ;
          end ;

signal(S) : S.value :=S.value+1 ;
     if S.value ≤ 0
          then begin
               enlever un processus P de S.L ;
               wakeup(P) ;
          end ;
```

5.3.Propriété des sémaphores

Un sémaphore ne peut être initialisé à une valeur inférieure à 0(zéro), mais sa valeur peut devenir négative après un certain nombre d'opérations P. La valeur d'un sémaphore donne le nombre de processus pouvant exécuter simultanément leur section critique. C'est à dire, si une ressource est partageable avec k points d'accès, la valeur initiale du sémaphore protégeant cette ressource est égale à k.

- **Exemple**

Si un fichier est partagé, en lecture, entre n processus; la valeur initiale du sémaphore protégeant ce fichier est alors égale n.

A tout instant la valeur e(s) d'un sémaphore s est donnée par la relation suivante:

e0(s) : valeur initiale du sémaphore s,

$$\begin{cases} e(s) = e0(s) - nP + nV \\ e0(s): valeur\ initiale\ du\ sémaphore\ s \end{cases}$$

Où

nP: nombre de d'exécution de P sur le sémaphore s;
nV: nombre de d'exécution de V sur le sémaphore s;

5.4.Implantation des primitives P et V

En plus des primitives P et V, on doit ajouter une primitive qui permet la définition et l'initialisation des sémaphores. Ces primitives doivent être indivisibles: Un seul processus peut exécuter une primitive P ou V sur un sémaphore donné (P et V exécutées en exclusion mutuelle):

- *Sur une machine monoprocesseur*, on doit exécuter ces primitives en masquant les interruptions.
- *Sur une machine multiprocesseur*, deux étapes sont nécessaires:

1. Garantir qu'un seul processeur (U.C.) peut exécuter une primitive: utilisation de l'instruction TAS ou d'une instruction équivalente.
2. Masquer les interruptions avant de demander l'entrée dans la section critique protégée par l'instruction TAS (c'est à dire masquer les Interruptions avant d'exécuter l'instruction TAS).

Remarque: il y a attente active pendant la durée de l'exécution de la primitive.

Les primitives, les variables d'état et les files d'attente doivent être protégées (utilisation des primitives par appel au superviseur).

Initialisation de la variable commune m à 0 : m=0;	
Primitive P(s)	**Primitive V(s)**
Début *1) masquer les interruptions;* *2) Boucle : TAS(m); /* attente active */* *Aller à boucle;* *Programme de P(s)* *3) m=0; /* libérer la section critique */* *4) Démasquer les interruptions;* *Fin de P(s);*	*Début* *1) masquer les interruptions;* *2) boucle : TAS(m); /* attente active */* *Aller à boucle;* *Programme de V(s)* *3) m=0; /*libérer la section critique*/* *4) démasquer les interruptions;* *Fin de V(s);*