Centre Universitaire de Mila

3 ème année licence LMD Informatique

Module : Systèmes d'exploitation 2

Bessouf Hakim

CHAPITRE 2:

Synchronisation des processus

- Problème de l’exclusion mutuelle
- Synchronisation
- Événements, Verrous
- Sémaphores
- Moniteurs
- Régions critiques.
- Expressions de chemins

Introduction

- Dans certain systèmes informatiques plusieurs processus (threads) peuvent partager des ressources communes (fichiers, mémoire ..etc) où chaque processus peut éventuellement lire et écrire. Ces processus s'exécutent d'une manière concurrente et peuvent accéder simultanément aux ressources.
- L'accès simultané au ressources peut engendrer des incohérences dans le système. L'objectif de la synchronisation des processus et justement d'éviter ces incohérences en utilisant différents mécanismes de synchronisation.

Introduction

Exemple:

- Dans un système de réservation de billets d'avions. Chaque agence qui veut réserver un billet d’avions va exécuter le programmes suivant:

  -Vérifier si il y a des places libres (Si Nbr_places< max)

  -Réserver une places (nbr_places ← nbr_places -1)

On suppose qu'il ne reste qu'une seule place à réserver. Si deux agences consultent le nombre de places en même temps , les deux agences vont chacune réserver un billet, on vas donc dépasser le nombre de places maximale dans l'avion. La variable nbr_places est donc une ressource critique qui doit être géré avec précautions.

Problème de l’exclusion mutuelle

- Il faut trouver des mécanismes pour interdire que plusieurs processus lisent ou écrivent simultanément des données partagés.
- **L'exclusion mutuelle** est une méthode qui permet de garantir qu'un seule processus à la fois peut lire ou écrire les données partagés.
- Les parties de programme ou on accède a des données ou des ressources partagées (**ressources critiques**) est appelé **section critique**.
- Si un processus entre dans sa section critique aucun autre processus ne peut entrer jusqu’à ce qu’il sort de sa section critique. ceci va permettre d'éviter les problèmes de la concurrence

Problème de l’exclusion mutuelle

- Chaque processus qui veut entrer dans sa **section critique** doit exécuter un **protocole d'entrée** qui permet de vérifier si la **section critique** n'est pas réservée. Si elle n'est pas libre il **se bloque** en attendant sa libération.
- Chaque processus qui termine l'exécution de sa **section critique** doit exécuter un **protocole de sortie** pour libérer la **section critique** et permettre au autres processus d'entrer dans leurs sections critiques

…
…
<Section non critique>
<Protocole d 'entrée a la section critique>
<**Section critique**>
<Protocole d e sortie de la section critique>
<Section non critique>
…
…

Problème de l’exclusion mutuelle

- La solution proposés pour la section critique doit respecter les conditions suivantes **[ref : « Systèmes d'exploitation » Andrew Tanenbaum]** :

1. **Exclusion mutuelle:** Deux processus ne doivent pas se trouver simultanément dans leurs SC.
2. **Aucune supposition:** Il ne faut pas faire des supposition sur la vitesse d'exécution des processus ou le nombre de processus.
3. **Pas d’interblocage:** Aucun processus qui se trouve à l'extérieur de sa SC ne peut bloquer les autres processus.
4. **Pas de famine:** On doit garantir qu'un processus qui demande d'entrée en SC doit pouvoir y enter dans un temps fini. La famine se produit lorsque la solution proposé n'est pas équitable.

Solutions pour l'exclusion mutuelle (section critique)

[ref : « Système d'exploitation mécanismes de base » ,Dr Belkhir Abdelkader, « Systèmes d'exploitation » Andrew tanenbaum]

Les solution pour l'exclusion mutuelle peuvent être classées en deux grandes catégories :

Les solutions basées sur l'attente active : dans ce cas les processus qui attendent d'entrer en SC reste en exécution et vérifies à chaque fois si la SC est libre ou non. Ces solution peuvent être logiciels ou matériels.

Les solutions basées sur l'attente passive : dans ce cas les processus qui attendent d'entrer en SC sont mis dans un état bloqué (le processeur est libéré) , ils seront réveillés lorsque la section critique deviens libre.

Ces solutions utilise des concepts évolués comme les **sémaphores**, les **région critiques**, et les **moniteurs**.

Solutions pour l'exclusion mutuelle (section critique)

- Les algorithmes qui suivent cherchent a réaliser l'exclusion mutuelle entre deux processus Pi et Pj.
- Les algorithmes sont donnés pour le processus Pi.

Solutions logiciels basées sur l'attente active

- «Vide=vrai»=>

pas de processus dans SC

- «Vide = faux» =>

∃ un processus dans SC

Algorithme 1 (variable verrou)

```
Répétez

Tantque (vide = faux) faire  /*ne rien faire*/  FinTantque;
        vide = faux;
        section critique
        vide = vrai;
        section non critique

Jusqu'à (faux) ;
```

Algorithme 1 (variable verrou)

Processus Pi

Répétez

➢ **Tant que** (vide = faux) faire
/*ne rien faire*/
FinTantque;
vide = faux;
section critique
vide = vrai;
section non critique

Jusqu'à (faux) ;

vide = **faux**

Processus Pj

Répétez

➢ **Tant que** (vide = faux) faire
/*ne rien faire*/
FinTantque;
vide = faux;
section critique
vide = vrai;
section non critique

Jusqu'à (faux) ;

Algorithme 1 (variable verrou)

Processus Pi

Répétez

➢ **Tant que** (vide = faux) faire
/*ne rien faire*/
FinTantque;
vide = faux;
section critique
vide = vrai;
section non critique

Jusqu'à (faux) ;

vide = **vrai**

Processus Pj

Répétez

➢ **Tant que** (vide = faux) faire
/*ne rien faire*/
FinTantque;
vide = faux;
section critique
vide = vrai;
section non critique

Jusqu'à (faux) ;

Algorithme 1 (variable verrou)

Processus Pi

Répétez

Tant que (vide = faux) faire

/*ne rien faire*/

FinTantque;

➢ vide = faux;

section critique

vide = vrai;

section non critique

Jusqu'à (faux) ;

vide = **vrai**

Processus Pj

Répétez

Tant que (vide = faux) faire

/*ne rien faire*/

FinTantque;

➢ vide = faux;

section critique

vide = vrai;

section non critique

Jusqu'à (faux) ;

Algorithme 1 (variable verrou)

Processus Pi

Répétez

Tant que (vide = faux) faire

/*ne rien faire*/

FinTantque;

vide = faux;

➢ section critique

vide = vrai;

section non critique

Jusqu'à (faux) ;

vide = **vrai**

Processus Pj

Répétez

Tant que (vide = faux) faire

/*ne rien faire*/

FinTantque;

vide = faux;

➢ section critique

vide = vrai;

section non critique

Jusqu'à (faux) ;

Solutions logiciels basées sur l'attente active

- Cette solution n'assure pas l'exclusion mutuelle, il se peut que deux processus lisent la variable vide en même temps et entre tous les deux en SC

Algorithme 1 (variable verrou)

```
Répétez

Tantque (vide = faux) faire  /*ne rien faire*/  FinTantque;
        vide = faux;
        section critique
        vide = vrai;
        section non critique

Jusqu'à (faux) ;
```

Solutions logiciels basées sur l'attente active

«turn = i» =>

processus Pi peut entrer dans SC

- «turn = j» =>
- processus Pj peut entrer dans SC

Algorithme 2 (alternance stricte)

```
Répétez
Tantque (turn ≠ i) faire /*ne rien faire*/ FinTantque;
        section critique
        turn = j;
        section non critique

Jusqu'à (faux) ;
```

Algorithme 2 (alternance stricte)

Processus Pi

Répétez

- **Tantque** (turn ≠ i) **faire**
  /*ne rien faire*/
  FinTantque;
  section critique
  turn = j;
  section non critique

Jusqu'à (faux) ;

turn= **j**

Processus Pj

Répétez

- **Tantque** (turn ≠ j) **faire**
  /*ne rien faire*/
  FinTantque;
  section critique
  turn = i;
  section non critique

Jusqu'à (faux) ;

Algorithme 2 (alternance stricte)

Processus Pi

Répétez

➢ **Tantque** (turn ≠ i) **faire**

/*ne rien faire*/

FinTantque;

section critique

turn = j;

section non critique

Jusqu'à (faux) ;

turn= **j**

Processus Pj

Répétez

Tantque (turn ≠ j) **faire**

/*ne rien faire*/

FinTantque;

➢ section critique

turn = i;

section non critique

Jusqu'à (faux) ;

Algorithme 2 (alternance stricte)

Processus Pi

Répétez

➢ **Tantque** (turn ≠ i) **faire**

/*ne rien faire*/

FinTantque;

section critique

turn = j;

section non critique

Jusqu'à (faux) ;

turn= **i**

Processus Pj

Répétez

Tantque (turn ≠ j) **faire**

/*ne rien faire*/

FinTantque;

section critique

➢ turn = i;

section non critique

Jusqu'à (faux) ;

Algorithme 2 (alternance stricte)

Processus Pi

Répétez

Tantque (turn ≠ i) **faire**

/*ne rien faire*/

FinTantque;

➢ section critique

turn = j;

section non critique

Jusqu'à (faux) ;

turn= **i**

Processus Pj

Répétez

Tantque (turn ≠ j) **faire**

/*ne rien faire*/

FinTantque;

section critique

turn = i;

➢ section non critique

Jusqu'à (faux) ;

Solutions logiciels basées sur l'attente active

Cette solution garantie l'exclusion mutuelle mais il se peut qu'un processus reste bloqué même si la SC n'est pas occupé (interblocage) puisque dans cette solution les processus sont obligés de s'exécuter en alternance stricte.

Algorithme 2 (alternance stricte)

```
Répétez
Tantque (turn ≠ i) faire /*ne rien faire*/ FinTantque;
        section critique
        turn = j;
        section non critique

Jusqu'à (faux) ;
```

Solutions logiciels basées sur l'attente active

- « flag » est un vecteur booléen dont les cases sont initialisées à faux.
- «flag[i] = vrai » =>

le processus Pi exécute la SC

- «flag[j] = vrai » =>

le processus Pj exécute la SC

Algorithme 3

```
Répétez
Tantque (flag[ j ]=vrai)  faire  /*ne rien faire*/ FinTantque;
        flag[ i ] = vrai ;
        section critique
        flag[ i ] = faux ;
        section non critique

Jusqu'à (faux) ;
```

Algorithme 3

Processus Pi

Répétez

➢ **Tantque** (flag[j]=vrai)
faire /*ne rien faire*/
FinTantque;
flag[i] = vrai ;
section critique
flag[i] = faux ;
section non critique

Jusqu'à (faux) ;

flag

i	**faux**
j	**faux**

Processus Pj

Répétez

➢ **Tantque** (flag[i]=vrai)
faire /*ne rien faire*/
FinTantque;
flag[j] = vrai ;
section critique
flag[j] = faux ;
section non critique

Jusqu'à (faux) ;

Algorithme 3

Processus Pi

Répétez

Tantque (flag[j]=vrai)
faire /*ne rien faire*/
FinTantque;
flag[i] = vrai ;
section critique
flag[i] = faux ;
section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

Processus Pj

Répétez

Tantque (flag[i]=vrai)
faire /*ne rien faire*/
FinTantque;
flag[j] = vrai ;
section critique
flag[j] = faux ;
section non critique

Jusqu'à (faux) ;

Algorithme 3

Processus Pi

Répétez

Tantque (flag[j]=vrai)
faire /*ne rien faire*/
FinTantque;
flag[i] = vrai ;
section critique
flag[i] = faux ;
section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

Processus Pj

Répétez

Tantque (flag[i]=vrai)
faire /*ne rien faire*/
FinTantque;
flag[j] = vrai ;
section critique
flag[j] = faux ;
section non critique

Jusqu'à (faux) ;

Solutions logiciels basées sur l'attente active

- Cette solution n'assure pas l'exclusion mutuelle, il se peut que deux processus lisent la variable flag en même temps et entrent tous les deux en SC

Algorithme 3

```
Répétez
Tantque (flag[ j ]=vrai)  faire  /*ne rien faire*/ FinTantque;
        flag[ i ] = vrai ;
        section critique
        flag[ i ] = faux ;
        section non critique

Jusqu'à (faux) ;
```

Solutions logiciels basées sur l'attente active

- «flag[i] = vrai » =>

le processus Pi désire exécuter SC,

- «flag[j] = vrai » =>
- le processus Pj désire exécuter SC

Algorithme 4

```
Répétez
        flag[ i ] = vrai ;
Tantque (flag[ j ] =vrai) faire /*ne rien faire*/ FinTantque;
        section critique
        flag[ i ] = faux ;
        section non critique

Jusqu'à (faux) ;
```

Algorithme 4

Processus Pi

Répétez

flag[i] = vrai ;

Tantque (flag[j] =vrai)

faire /*ne rien faire*/

FinTantque;

section critique

flag[i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**faux**
j	**faux**

Processus Pj

Répétez

flag[j] = vrai ;

Tantque (flag[i] =vrai)

faire /*ne rien faire*/

FinTantque;

section critique

flag[j] = faux ;

section non critique

Jusqu'à (faux) ;

Algorithme 4

Processus Pi

Répétez

➢ flag[i] = vrai ;
Tantque (flag[j] =vrai)
faire /*ne rien faire*/
FinTantque;
section critique
flag[i] = faux ;
section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

Processus Pj

Répétez

➢ flag[j] = vrai ;
Tantque (flag[i] =vrai)
faire /*ne rien faire*/
FinTantque;
section critique
flag[j] = faux ;
section non critique

Jusqu'à (faux) ;

Algorithme 4

Processus Pi

Répétez

flag[i] = vrai ;

➢ **Tantque** (flag[j] =vrai)

faire /*ne rien faire*/

FinTantque;

section critique

flag[i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

Processus Pj

Répétez

flag[j] = vrai ;

➢ **Tantque** (flag[i] =vrai)

faire /*ne rien faire*/

FinTantque;

section critique

flag[j] = faux ;

section non critique

Jusqu'à (faux) ;

Solutions logiciels basées sur l'attente active

- Cette solution garantie l'exclusion mutuelle mais peut faire en sorte qu'il n'y a aucun processus en SC (interblocage) et cela si les deux processus initialisent le flag à vrai en même temps.

Algorithme 4

```
Répétez
        flag[ i ] = vrai ;
Tantque (flag[ j ] =vrai) faire /*ne rien faire*/ FinTantque;
        section critique
        flag[ i ] = faux ;
        section non critique

Jusqu'à (faux) ;
```

Solutions logiciels basées sur l'attente active

- «flag[i] = vrai » =>

processus Pi désire exécuter SC,

- «flag[j] = vrai » =>

processus Pj désire exécuter SC,

Algorithme 5

```
Répétez

flag [ i ] = vrai ;

Tantque (flag[ j ] = vrai) faire
   flag[ i ] = faux ;
   Tantque (flag[ j ]=vrai) faire /*ne rien faire*/ FinTantque;
    flag[ i ] = vrai ;
FinTantque ;

        section critique
        flag[ i ] = faux ;
        section non critique

Jusqu'à (faux) ;
```

Algorithme 5

Processus Pi

Répétez

flag [i] = vrai ;

Tantque (flag[j] = vrai) faire

flag[i] = faux ;

Tantque (flag[j]=vrai) faire

/*ne rien faire*/

FinTantque;

flag[i] = vrai ;

FinTantque ;

section critique

flag[i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**faux**
j	**faux**

Processus Pj

Répétez

flag [j] = vrai ;

Tantque (flag[i] = vrai) faire

flag[j] = faux ;

Tantque (flag[i]=vrai) faire

/*ne rien faire*/

FinTantque;

flag[j] = vrai ;

FinTantque ;

section critique

flag[j] = faux ;

section non critique

Jusqu'à (faux) ;

Algorithme 5

Processus Pi

Répétez

- flag [i] = vrai ;

  Tantque (flag[j] = vrai) faire

    flag[i] = faux ;

    Tantque (flag[j]=vrai) faire

      /*ne rien faire*/

    FinTantque;

    flag[i] = vrai ;

  FinTantque ;

  section critique

  flag[i] = faux ;

  section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

Processus Pj

Répétez

- flag [j] = vrai ;

  Tantque (flag[i] = vrai) faire

    flag[j] = faux ;

    Tantque (flag[i]=vrai) faire

      /*ne rien faire*/

    FinTantque;

    flag[j] = vrai ;

  FinTantque ;

  section critique

  flag[j] = faux ;

  section non critique

Jusqu'à (faux) ;

Algorithme 5

Processus Pi

Répétez

flag [i] = vrai ;

➢ **Tantque** (flag[j] = vrai) faire

flag[i] = faux ;

Tantque (flag[j]=vrai) faire

/*ne rien faire*/

FinTantque;

flag[i] = vrai ;

FinTantque ;

section critique

flag[i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

Processus Pj

Répétez

flag [j] = vrai ;

➢ **Tantque** (flag[i] = vrai) faire

flag[j] = faux ;

Tantque (flag[i]=vrai) faire

/*ne rien faire*/

FinTantque;

flag[j] = vrai ;

FinTantque ;

section critique

flag[j] = faux ;

section non critique

Jusqu'à (faux) ;

Algorithme 5

Processus Pi

Répétez

flag [i] = vrai ;

Tantque (flag[j] = vrai) faire

flag[i] = faux ;

Tantque (flag[j]=vrai) faire

/*ne rien faire*/

FinTantque;

flag[i] = vrai ;

FinTantque ;

section critique

flag[i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**faux**
j	**faux**

Processus Pj

Répétez

flag [j] = vrai ;

Tantque (flag[i] = vrai) faire

flag[j] = faux ;

Tantque (flag[i]=vrai) faire

/*ne rien faire*/

FinTantque;

flag[j] = vrai ;

FinTantque ;

section critique

flag[j] = faux ;

section non critique

Jusqu'à (faux) ;

Algorithme 5

Processus Pi

Répétez

flag [i] = vrai ;

Tantque (flag[j] = vrai) faire

flag[i] = faux ;

➢ **Tantque** (flag[j]=vrai) faire

/*ne rien faire*/

FinTantque;

flag[i] = vrai ;

FinTantque ;

section critique

flag[i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**faux**
j	**faux**

Processus Pj

Répétez

flag [j] = vrai ;

Tantque (flag[i] = vrai) faire

flag[j] = faux ;

➢ **Tantque** (flag[i]=vrai) faire

/*ne rien faire*/

FinTantque;

flag[j] = vrai ;

FinTantque ;

section critique

flag[j] = faux ;

section non critique

Jusqu'à (faux) ;

Algorithme 5

Processus Pi

Répétez

flag [i] = vrai ;

Tantque (flag[j] = vrai) faire

flag[i] = faux ;

Tantque (flag[j]=vrai) faire

/*ne rien faire*/

FinTantque;

flag[i] = vrai ;

FinTantque ;

section critique

flag[i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

Processus Pj

Répétez

flag [j] = vrai ;

Tantque (flag[i] = vrai) faire

flag[j] = faux ;

Tantque (flag[i]=vrai) faire

/*ne rien faire*/

FinTantque;

flag[j] = vrai ;

FinTantque ;

section critique

flag[j] = faux ;

section non critique

Jusqu'à (faux) ;

Algorithme 5

Processus Pi

Répétez

flag [i] = vrai ;

➢ **Tantque** (flag[j] = vrai) faire

flag[i] = faux ;

Tantque (flag[j]=vrai) faire

/*ne rien faire*/

FinTantque;

flag[i] = vrai ;

FinTantque ;

section critique

flag[i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

Processus Pj

Répétez

flag [j] = vrai ;

➢ **Tantque** (flag[i] = vrai) faire

flag[j] = faux ;

Tantque (flag[i]=vrai) faire

/*ne rien faire*/

FinTantque;

flag[j] = vrai ;

FinTantque ;

section critique

flag[j] = faux ;

section non critique

Jusqu'à (faux) ;

Solutions logiciels basées sur l'attente active

- Cette solution peut faire en sorte qu'il n'y a aucun processus en SC (interblocage).

Algorithme 5

```
Répétez

flag [ i ] = vrai ;

Tantque (flag[ j ] = vrai) faire
   flag[ i ] = faux ;
   Tantque (flag[ j ]=vrai) faire /*ne rien faire*/ FinTantque;
    flag[ i ] = vrai ;
FinTantque ;

        section critique
        flag[ i ] = faux ;
        section non critique

Jusqu'à (faux) ;
```

Solutions logiciels basées sur l'attente active

Algorithme 6 (Solution de DEKKER 1965)

```
Répétez
  flag [ i ] = vrai ;
  Tant que (flag[ j ] = vrai) faire
    Si (turn =j) alors flag[ i ] = faux; FinSi
    Tant que (turn = j) faire /*ne rien faire*/ FinTantque;
    flag[ i ] = vrai ;
  FinTantque ;
  section critique
  turn = j ;
  flag [ i ] = faux ;
  section non critique
Jusqu'à (faux) ;
```

Algorithme 6 (Solution de DEKKER 1965)

Processus Pi

Répétez

flag [i] = vrai ;

Tantque (flag[j] = vrai) **faire**

Si (turn =j) **alors**

flag[i] = faux;

FinSi

Tantque (turn = j) **faire**

/*ne rien faire*/

FinTantque;

flag[i] = vrai ;

FinTantque ;

section critique

turn = j ;

flag [i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**faux**
j	**faux**

turn= **j**

Processus Pj

Répétez

flag [j] = vrai ;

Tantque (flag[i] = vrai) **faire**

Si (turn =i) **alors**

flag[j] = faux;

FinSi

Tantque (turn = i) **faire**

/*ne rien faire*/

FinTantque;

flag[j] = vrai ;

FinTantque ;

section critique

turn = i ;

flag [j] = faux ;

section non critique

Jusqu'à (faux) ;

Algorithme 6 (Solution de DEKKER 1965)

Processus Pi

Répétez

- flag [i] = vrai ;

Tantque (flag[j] = vrai) **faire**

Si (turn =j) **alors**

flag[i] = faux;

FinSi

Tantque (turn = j) **faire**

/*ne rien faire*/

FinTantque;

flag[i] = vrai ;

FinTantque ;

section critique

turn = j ;

flag [i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

turn= **j**

Processus Pj

Répétez

- flag [j] = vrai ;

Tantque (flag[i] = vrai) **faire**

Si (turn =i) **alors**

flag[j] = faux;

FinSi

Tantque (turn = i) **faire**

/*ne rien faire*/

FinTantque;

flag[j] = vrai ;

FinTantque ;

section critique

turn = i ;

flag [j] = faux ;

section non critique

Jusqu'à (faux) ;

Algorithme 6 (Solution de DEKKER 1965)

Processus Pi

Répétez

flag [i] = vrai ;

➢ **Tantque** (flag[j] = vrai) **faire**

Si (turn =j) **alors**

flag[i] = faux;

FinSi

Tantque (turn = j) **faire**

/*ne rien faire*/

FinTantque;

flag[i] = vrai ;

FinTantque ;

section critique

turn = j ;

flag [i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

turn= **j**

Processus Pj

Répétez

flag [j] = vrai ;

➢ **Tantque** (flag[i] = vrai) **faire**

Si (turn =i) **alors**

flag[j] = faux;

FinSi

Tantque (turn = i) **faire**

/*ne rien faire*/

FinTantque;

flag[j] = vrai ;

FinTantque ;

section critique

turn = i ;

flag [j] = faux ;

section non critique

Jusqu'à (faux) ;

Algorithme 6 (Solution de DEKKER 1965)

Processus Pi

Répétez

flag [i] = vrai ;

Tantque (flag[j] = vrai) **faire**

Si (turn =j) **alors**

➢ flag[i] = faux;

FinSi

Tantque (turn = j) **faire**

/*ne rien faire*/

FinTantque;

flag[i] = vrai ;

FinTantque ;

section critique

turn = j ;

flag [i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**faux**
j	**vrai**

turn= **j**

Processus Pj

Répétez

flag [j] = vrai ;

Tantque (flag[i] = vrai) **faire**

Si (turn =i) **alors**

flag[j] = faux;

FinSi

Tantque (turn = i) **faire**

/*ne rien faire*/

FinTantque;

➢ flag[j] = vrai ;

FinTantque ;

section critique

turn = i ;

flag [j] = faux ;

section non critique

Jusqu'à (faux) ;

Algorithme 6 (Solution de DEKKER 1965)

Processus Pi

Répétez

flag [i] = vrai ;

Tantque (flag[j] = vrai) **faire**

Si (turn =j) **alors**

flag[i] = faux;

FinSi

➢ **Tantque** (turn = j) **faire**

/*ne rien faire*/

FinTantque;

flag[i] = vrai ;

FinTantque ;

section critique

turn = j ;

flag [i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**faux**
j	**vrai**

turn= **j**

Processus Pj

Répétez

flag [j] = vrai ;

Tantque (flag[i] = vrai) **faire**

Si (turn =i) **alors**

flag[j] = faux;

FinSi

Tantque (turn = i) **faire**

/*ne rien faire*/

FinTantque;

flag[j] = vrai ;

FinTantque ;

➢ section critique

turn = i ;

flag [j] = faux ;

section non critique

Jusqu'à (faux) ;

Algorithme 6 (Solution de DEKKER 1965)

Processus Pi

Répétez

flag [i] = vrai ;

Tantque (flag[j] = vrai) **faire**

Si (turn =j) **alors**

flag[i] = faux;

FinSi

Tantque (turn = j) **faire**

/*ne rien faire*/

FinTantque;

➢ flag[i] = vrai ;

FinTantque ;

section critique

turn = j ;

flag [i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

turn= **i**

Processus Pj

Répétez

flag [j] = vrai ;

Tantque (flag[i] = vrai) **faire**

Si (turn =i) **alors**

flag[j] = faux;

FinSi

Tantque (turn = i) **faire**

/*ne rien faire*/

FinTantque;

flag[j] = vrai ;

FinTantque ;

section critique

➢ turn = i ;

flag [j] = faux ;

section non critique

Jusqu'à (faux) ;

Algorithme 6 (Solution de DEKKER 1965)

Processus Pi

Répétez

flag [i] = vrai ;

➢ **Tantque** (flag[j] = vrai) **faire**

Si (turn =j) **alors**

flag[i] = faux;

FinSi

Tantque (turn = j) **faire**

/*ne rien faire*/

FinTantque;

flag[i] = vrai ;

FinTantque ;

section critique

turn = j ;

flag [i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**faux**

turn= **i**

Processus Pj

Répétez

flag [j] = vrai ;

Tantque (flag[i] = vrai) **faire**

Si (turn =i) **alors**

flag[j] = faux;

FinSi

Tantque (turn = i) **faire**

/*ne rien faire*/

FinTantque;

flag[j] = vrai ;

FinTantque ;

section critique

turn = i ;

➢ flag [j] = faux ;

section non critique

Jusqu'à (faux) ;

Algorithme 6 (Solution de DEKKER 1965)

Processus Pi

Répétez

flag [i] = vrai ;

Tantque (flag[j] = vrai) **faire**

Si (turn =j) **alors**

flag[i] = faux;

FinSi

Tantque (turn = j) **faire**

/*ne rien faire*/

FinTantque;

flag[i] = vrai ;

FinTantque ;

➢ section critique

turn = j ;

flag [i] = faux ;

section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

turn= **i**

Processus Pj

Répétez

flag [j] = vrai ;

Tantque (flag[i] = vrai) **faire**

Si (turn =i) **alors**

flag[j] = faux;

FinSi

Tantque (turn = i) **faire**

/*ne rien faire*/

FinTantque;

flag[j] = vrai ;

FinTantque ;

section critique

turn = i ;

flag [j] = faux ;

➢ section non critique

Jusqu'à (faux) ;

Solutions logiciels basées sur l'attente active

- Cette solution assure l'exclusion mutuelle, et ne provoque pas d'interblocage des processus.
- Même si deux processus veulent entrer en section critique en même temps la variable «turn» va les dissocier.

Algorithme 6 (Solution de DEKKER 1965)

```
Répétez
  flag [ i ] = vrai ;
  Tant que (flag[ j ] = vrai) faire
    Si (turn =j) alors flag[ i ] = faux; FinSi
    Tant que (turn = j) faire /*ne rien faire*/ FinTantque;
    flag[ i ] = vrai ;
  FinTantque ;
  section critique
  turn = j ;
  flag [ i ] = faux ;
  section non critique
Jusqu'à (faux) ;
```

Solutions logiciels basées sur l'attente active

Algorithme 7 (Solution de PETERSON : 1981)

```
Répétez

 flag [ i ] = vrai ;
 turn = i ;
Tantque (turn = i) et (flag[ j ]=vrai) faire
    /*ne rien faire*/
FinTantque;

        section critique
 flag [ i ] = false;
        section non critique

Jusqu'à (faux) ;
```

Algorithme 6 (Solution de PETERSON : 1981)

Processus Pi

Répétez

flag [i] = vrai ;
turn = i ;
Tantque (turn = i) et
(flag[j]=vrai) faire
/*ne rien faire*/
FinTantque;
section critique
flag [i] = faux;
section non critique

Jusqu'à (faux) ;

flag

i	**faux**
j	**faux**

turn= **i**

Processus Pj

Répétez

flag [j] = vrai ;
turn = j ;
Tantque (turn = j) et
(flag[i]=vrai) faire
/*ne rien faire*/
FinTantque;
section critique
flag [j] = faux;
section non critique

Jusqu'à (faux) ;

Algorithme 6 (Solution de PETERSON : 1981)

Processus Pi

Répétez

- flag [i] = vrai ;
  turn = i ;
  Tantque (turn = i) et
  (flag[j]=vrai) faire
  /*ne rien faire*/
  FinTantque;
  section critique
  flag [i] = faux;
  section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

turn= **i**

Processus Pj

Répétez

- flag [j] = vrai ;
  turn = j ;
  Tantque (turn = j) et
  (flag[i]=vrai) faire
  /*ne rien faire*/
  FinTantque;
  section critique
  flag [j] = faux;
  section non critique

Jusqu'à (faux) ;

Algorithme 6 (Solution de PETERSON : 1981)

Processus Pi

Répétez

flag [i] = vrai ;

turn = i ;

Tantque (turn = i) et
(flag[j]=vrai) faire
/*ne rien faire*/

FinTantque;

section critique

flag [i] = faux;

section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

turn= **j**

Processus Pj

Répétez

flag [j] = vrai ;

turn = j ;

Tantque (turn = j) et
(flag[i]=vrai) faire
/*ne rien faire*/

FinTantque;

section critique

flag [j] = faux;

section non critique

Jusqu'à (faux) ;

Algorithme 6 (Solution de PETERSON : 1981)

Processus Pi

Répétez

flag [i] = vrai ;
turn = i ;
Tantque (turn = i) et
(flag[j]=vrai) faire
/*ne rien faire*/
FinTantque;
section critique
flag [i] = faux;
section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

turn= **j**

Processus Pj

Répétez

flag [j] = vrai ;
turn = j ;
Tantque (turn = j) et
(flag[i]=vrai) faire
/*ne rien faire*/
FinTantque;
section critique
flag [j] = faux;
section non critique

Jusqu'à (faux) ;

Algorithme 6 (Solution de PETERSON : 1981)

Processus Pi

Répétez

flag [i] = vrai ;
turn = i ;
Tantque (turn = i) et
(flag[j]=vrai) faire
/*ne rien faire*/
FinTantque;
section critique
flag [i] = faux;
section non critique

Jusqu'à (faux) ;

flag

i	**vrai**
j	**vrai**

turn= **j**

Processus Pj

Répétez

flag [j] = vrai ;
turn = j ;
Tantque (turn = j) et
(flag[i]=vrai) faire
/*ne rien faire*/
FinTantque;
section critique
flag [j] = faux;
section non critique

Jusqu'à (faux) ;

Algorithme 6 (Solution de PETERSON : 1981)

Processus Pi

Répétez

flag [i] = vrai ;
turn = i ;
Tantque (turn = i) et
(flag[j]=vrai) faire
/*ne rien faire*/
FinTantque;
section critique
flag [i] = faux;
section non critique

Jusqu'à (faux) ;

flag

i	**faux**
j	**vrai**

turn= **j**

Processus Pj

Répétez

flag [j] = vrai ;
turn = j ;
Tantque (turn = j) et
(flag[i]=vrai) faire
/*ne rien faire*/
FinTantque;
section critique
flag [j] = faux;
section non critique

Jusqu'à (faux) ;

Solutions logiciels basées sur l'attente active

- Cette solution assure l'exclusion mutuelle, et ne provoque pas d'interblocage.
- Même si deux processus arrivent en même temps c'est le premier qui a initialisé la valeur de turn qui entre en section critique, puisque turn prend une seule valeur a la fin du protocole d'entré en SC soit i soit j

Algorithme 7 (Solution de PETERSON : 1981)

```
Répétez

 flag [ i ] = vrai ;
 turn = i ;
Tantque (turn = i) et (flag[ j ]=vrai) faire
     /*ne rien faire*/
FinTantque;

          section critique
 flag [ i ] = false;
          section non critique

Jusqu'à (faux) ;
```

Solutions logiciels basées sur l'attente active

Algorithme 8 (algorithme de la boulangerie, Leslie Lamport, 1974)
Cette algorithme s'inspire de la façon dont on gère une boulangerie,

Chaque client qui veut acheter du pain va prendre un ticket contenant sont numéro (le numéro maximum + 1) puis attend son tour pour être servis.

Le tableau NUM_TICKET est initialisé a 0, et le tableau PRENDRE_TICKET est initialisé a faux. « i » est l'identifiant du processus qui veut entré en SC.

```
Répétez
PRENDRE_TICKET[i] = vrai ; /*je vais prendre un ticket*/
NUM_TICKET [i] = MAX + 1 ; /*Attribution d'un nouveau ticket*/
PRENDRE_TICKET[i] = faux ; /*j'ai pris un ticket*/
Pour j allant de 1 à N faire
  /* Attendre que les autres processus qui désire prendre un ticket ont terminé */
  Tant que (PRENDRE_TICKET[j] = vrai)
   /*ne rien faire*/
  Fin Tant que ;
  /*Attendre qui je sois prioritaire*/
  Tant que (NUM_TICKET [ j ] ≠ 0) et ((NUM_TICKET [j] <NUM_TICKET [i] ) ou
  (NUM_TICKET [j]=NUM_TICKET [i] et i<j )) faire
   /*ne rien faire*/
  Fin tant que ;
Fin pour ;
section critique
NUM_TICKET [i] = 0 ;
section non critique
Jusqu'à (faux) ;
```

Solutions matériels basées sur le masquage des interruptions

- Cette méthode peut être utilisé seulement dans le cas des machines avec un seul processeur. Elle consiste à masquer les interruptions avant d'entrer en SC puis de les démasquer après l'exécution de la SC.

Masquer_les_interruptions
section critique
Démasquer_les_interruptions
section non critique

- Problèmes:

- si un processus se bloque dans sa SC tous le système sera affecté.

- elle ne peut pas être utilisé dans les machine multiprocesseurs.

Solutions matériels basées sur des instruction indivisibles

L'instruction Test_and_set (TAS ou TSL)

C'est une instruction indivisible (elle s'exécute complètement et ne peut pas être interrompu) qui prend en paramètre un booléen. Elle fonctionne comme suit :

lock est initialisé à **faux** : aucun processus n'exécute la section critique. Pour réaliser l'exclusion mutuelle on utilise la variable lock comme suit :

```
Fonction Test_and_Set (var lock:booléen): booléen
b : booléen ;
Début
b ← lock ;
lock ← vrai ;
retourner(b);
Fin ;
```

```
Tantque (TAS(lock) = vrai) faire
/*ne rien faire*/
FinTantque
section critique
look = faux ;
section non critique
```

Solutions matériels basées sur des instruction indivisibles

Fonction Test_and_Set (var lock:booléen): booléen

b : booléen ;

Début

- b ← lock ;

  lock ← vrai ;

  retourner(b);

Fin ;

Processus Pi

- **Tantque** (TAS(lock) = vrai) **faire**

  /*ne rien faire*/

  FinTantque

  section critique

  lock = faux ;

  section non critique

TAS =

lock = faux

b = faux

Solutions matériels basées sur des instruction indivisibles

Fonction Test_and_Set (var lock:booléen): booléen

b : booléen ;

Début

b ← lock ;

- lock ← vrai ;

retourner(b);

Fin ;

Processus Pi

- **Tantque** (TAS(lock) = vrai) **faire**

/*ne rien faire*/

FinTantque

section critique

lock = faux ;

section non critique

TAS =

lock = vrai

b = faux

Solutions matériels basées sur des instruction indivisibles

Fonction Test_and_Set (var lock:booléen): booléen

b : booléen ;

Début

b ← lock ;

lock ← vrai ;

- retourner(b);

Fin ;

Processus Pi

Tantque (TAS(lock) = vrai) **faire**

/*ne rien faire*/

FinTantque

- section critique

lock = faux ;

section non critique

TAS = faux

lock = vrai

b = faux

Solutions matériels basées sur des instruction indivisibles

Fonction Test_and_Set (var lock:booléen): booléen

b : booléen ;

Début

- b ← lock ;

  lock ← vrai ;

  retourner(b);

Fin ;

Processus Pj

- **Tantque** (TAS(lock) = vrai) **faire**

  /*ne rien faire*/

  FinTantque

  section critique

  lock = faux ;

  section non critique

TAS =

lock = vrai

b = vrai

Solutions matériels basées sur des instruction indivisibles

Fonction Test_and_Set (var lock:booléen): booléen

b : booléen ;

Début

b ← lock ;

- lock ← vrai ;

retourner(b);

Fin ;

Processus Pj

- **Tantque** (TAS(lock) = vrai) **faire**

/*ne rien faire*/

FinTantque

section critique

lock = faux ;

section non critique

TAS =

lock = vrai

b = vrai

Solutions matériels basées sur des instruction indivisibles

Fonction Test_and_Set (var lock:booléen): booléen

b : booléen ;

Début

b ← lock ;

lock ← vrai ;

- retourner(b);

Fin ;

Processus Pj

- **Tantque** (TAS(lock) = vrai) **faire**

/*ne rien faire*/

FinTantque

section critique

lock = faux ;

section non critique

TAS = vrai

lock = vrai

b = vrai

Solutions matériels basées sur des instruction indivisibles

L'instruction SWAP:

Une instruction indivisible qui permute deux variable données en paramètre, elle s'exécute comme suit:

Pour effectuer l'exclusion mutuelle on utilise deux variables, Une variable globale lock initialisé a faux et une variable locale key pour chaque processus. Les deux variables sont utilisés comme suit :

```
Procédure SWAP (var a,b :booléen): booléen
c : booléen ;
Début
c ← a ;
b ← a ;
a ← b ;
Fin ;
```

```
key ← vrai ;
Répéter
  SWAP (lock,key) ;
Jusqu'à (key = faux) ;
section critique
lock = faux ;
section non critique
```

- Les solutions logiciels et matériels basé sur **l'attente active** vu précédemment, malgré elles fonctionnes, elle sont inefficaces, un processus ne pouvant pas entrer en SC va rester en exécution et utilise l'unité centrale inutilement, et ralenti aussi l'accès à la mémoire à cause des accès répétés aux variables de synchronisation.
- D'autre solutions plus efficaces au problème de l'exclusion mutuelle qui sont basées sur **l'attente passive** seront présentées dans les sections suivantes.

Solutions basées sur l'attente passive (verrou)

- Ces solution permettent de libérer le processeur quand les processus sont bloqués, parmi ces solutions on cite : **les verrous, les sémaphores, les régions critiques** et **les moniteurs**.

  - Les verrous:

- Des structures de données partagées par le système d'exploitation.
- Un verrou est représenté par une variable booléenne «libre» et une file d'attente des processus.
- Au début la variable libre est initialisé à vrai et la file d'attente des processus est vide.

```
Structure Verrou
        libre : booléen
        file : file d'attente des processus
FinStructure
```

Solutions basées sur l'attente passive (verrou)

- On peut accéder a un verrou seulement en utilisant les deux opérations indivisibles (deux primitives) « prendre » et « libérer » qui sont implémentés comme suit :

```
Prendre (v : verrou)
Début
   Si (v.libre = faux ) alors
        - Bloquer le processus  /*état processus ← bloqué*/
        - Ajouter le processus bloqué à la file d'attente du
          verrou
   Sinon
         v.libre ←  faux ;
   Finsi
Fin.
```

```
Libérer (v : verrou)
Début

   Si (v.file  =  vide )  alors

        v.libre ← vrai ;

   Sinon
        v.libre ← faux;
        -  Sortir un processus bloqué de la file d'attente du
        verrou,
        -  Débloquer le processus /*état processus ← prêt*/
   Finsi
Fin.
```

Solutions basées sur l'attente passive (verrou)

- Pour réaliser l'exclusion mutuel a l'aide d'un verrou on procède comme suit :

```
v :  verrou ;

    … section non critique
    …
Prendre (v);

    Section Critique

Libérer (v);
    …
    … section non critique
```

Solutions basées sur l'attente passive (sémaphore)

- Les sémaphores sont des outils qui permettent de mettre en œuvre des problème de synchronisation complexes. Un sémaphore est représenté par une variable entière « **value** » et une file d'attente des processus . On peut accéder à la variable entière **S.value** seulement en utilisant les deux opérations indivisibles (deux primitives) **Wait : P** et **Signal : V**

P (s :** sémaphore**) :
Début
s.value ← **s**.value - 1 ;
Si (**s** .value **<** 0) **alors**
- Bloquer le processus */*état processus ← bloqué*/*
- Ajouter le processus bloqué à la file d'attente du sémaphore

Finsi
Fin.

V (s :** sémaphore**) :
Début
s.value ← **s**.value + 1 ;
Si (**s**.value **≤** 0) **alors**
- Sortir un processus bloqué de la file d'attente du sémaphore
- Débloquer le processus */*état processus ← prêt*/*

Finsi
Fin.

Solutions basées sur l'attente passive (sémaphore)

- La valeur (**s.value**) d'un sémaphore est initialisé à 0 ou à une valeur positive.
- La file d'attente des processus est initialement vide.
- Pour mettre en œuvre des problèmes de synchronisation on peut utiliser plusieurs sémaphores à la fois.

Remarque importante :

- Pour implémenter les sémaphores et les verrous dans les machines monoprocesseurs ont peut utiliser le masquage des interruptions, par contre dans les machines multiprocesseurs le masquage des interruptions n'est pas approprié , des solutions matériels comme les instruction SWAP et TAS sont donc nécessaire

Exclusion mutuel par sémaphore

- Pour réaliser l'exclusion mutuel entre **n** processus on peut utiliser un sémaphore (appelé généralement **mutex**) initialisé à **1**
- Le programme d'un processus Pi est comme suit :

```
P (mutex) ;
        Section Critique
V (mutex) ;
        section non critique
```

Précédence des tâches par sémaphores

- Les sémaphores peuvent être utilisés pour synchroniser les systèmes de tâches.
- On considère un système de tâche composé de deux tache **T1** et **T2**. **T1** doit être exécuté avant **T2**. Ce système peut être synchronisé par un sémaphore **S** initialisé à **0**. Le système sera composé de deux processus (un processus pour chaque tâche). Les deux processus sont comme suit :

Processus T1

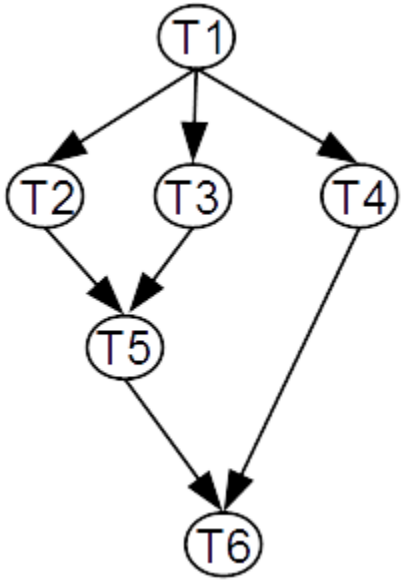
```
Début
        Exécuter T1 ;
        V(S) ;
Fin.
```

Processus T2

```
Début
        P(S) ;
        Exécuter T2 ;
Fin.
```

Précédence des tâches par sémaphores

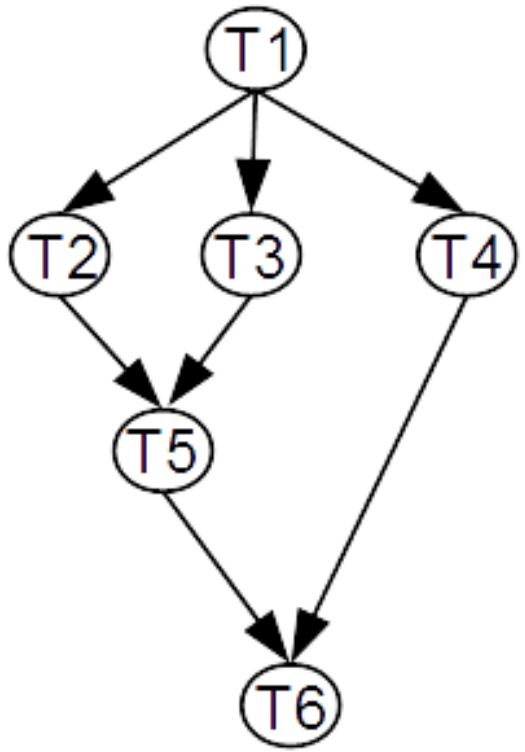
- **Exemple :** On veut réaliser le système de tâche représenté par le graphe de précédence suivant :
- Pour mettre en œuvre ce système on utilise 6 sémaphore S2,S2,S3,S4,S5,S6 tous initialisés à 0. Le système sera composé de 6 processus comme suit:



ProcessusT1	***ProcessusT2***
Début Exécuter T1 V(S1); V(S1); V(S1); **Fin ;**	**Début** P(S1) ; Exécuter T2 V(S2); **Fin ;**

ProcessusT3	***ProcessusT4***
Début P(S1) ; Exécuter T3 V(S3); **Fin ;**	**Début** P(S1) ; Exécuter T4 V(S4); **Fin ;**

ProcessusT5	***ProcessusT6***
Début P(S2) ; P(S3) ; Exécuter T5 V(S5); **Fin ;**	**Début** P(S4) ; P(S5) ; Exécuter T6 **Fin ;**



Coopération entre N processus(sémaphore)

- Les sémaphores peuvent être utilisé pour faire en sorte qu'un nombre précis de processus seulement coopèrent ensemble.

Exemple (parking)

- On suppose qu'on a **N** emplacements au parking,
- **N** conducteur seulement peuvent stationner leurs voitures.
- On peut régler ce problème en utilisant les sémaphores.

Coopération entre N processus(sémaphore)

- Chaque voiture est représenté par un processus
- On utilise une variable sémaphore appelé «parking» initialisée à **N**.

• Pour entrer dans le parking chaque processus exécute le protocole suivant :

Programme Voiture

```
….
….
P (parking) ;

        stationner la voiture
        ….
        sortir du stationnement

V (parking) ;
….
….
```

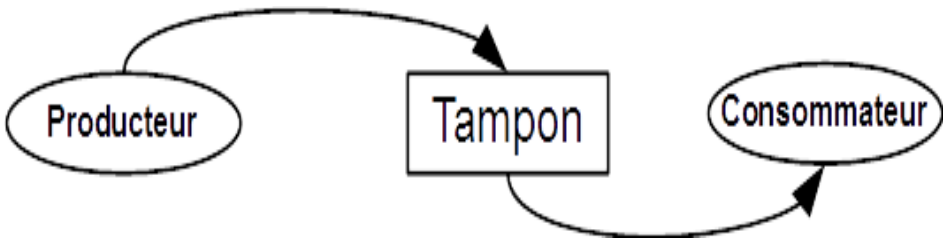
Rendez-vous entre deux processus (sémaphore)

- On peut établir un point de synchronisation (**rendezvous**) entre deux processus. Pour cela on utilise deux sémaphores, un pour chaque processus:
- On utilise deux variables sémaphore **S1** et **S2** initialisées à **0**.

Processus1		**Processus2**
….		….
V(S1) ;	←*Point de synchronisation* →	**V**(S2) ;
P(S2) ;		**P**(S1) ;
….		….

Modèle producteur/consommateur (sémaphore)

- Un processus Producteur crée à chaque fois un messages, le place dans une variable tampon, et un processus Consommateur récupère le message et le traite.

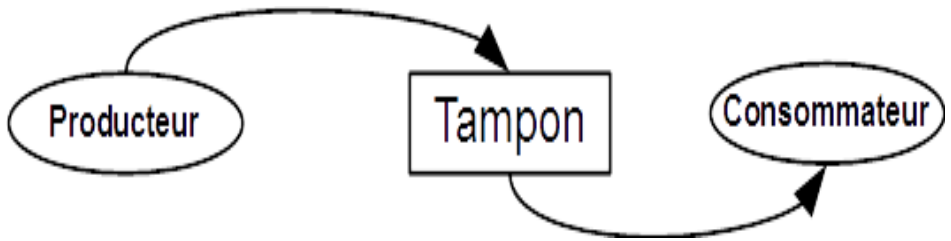


- **Première solution**: utilisation d'un sémaphore d'exclusion mutuel initialisé à **1**

Producteur	***Consommateur***
Répétez	**Répétez**
Produire un message **m ;**	**P**(mutex) ; Récupérer_message(m) ;
P(mutex) ; Déposer_message(m) ;	**V**(mutex) ;
V(mutex) ;	utiliser le message m
Jusqu'à(faux)	**Jusqu'à(faux)**

Modèle producteur/consommateur (sémaphore)

- Un processus Producteur crée à chaque fois un messages, le place dans une variable tampon, et un processus Consommateur récupère le message et le traite.



- **Deuxième solution:** utilisation de deux sémaphores, **s_vide** initialisé à **1** pour vérifier si le tampon est vide et **s_plein** initialisé à **0** pour vérifier si le tampon est plein.

Producteur	***Consommateur***
Répétez	**Répétez**
Produire un message **m** ;	**P**(s_plein) ;
P(s_vide)	Récupérer_message(m) ; **V**(s_vide)
Déposer_message(m) ; **V**(s_plein) ;	utiliser le message m
Jusqu'à(faux)	**Jusqu'à(faux)**

Modèle producteur/consommateur (sémaphore)

- **Cas tampon de N emplacement:**

Un sémaphore s_vide initialisé à N

Un sémaphore s_plein initialisé à 0

Une variable i initialisé a 0

Une variable j initialisé a 0

Producteur	***Consommateur***
Répétez Produire un message **m** ; **P**(s_vide) ; Tampon[i] ← m ; i ← (i +1) mod N ; **V**(s_plein) ; **Jusqu'à(faux)**	**Répétez** **P**(s_plein) ; m ← Tampon[j] ; j ← (j +1) mod N ; **V**(s_vide) ; utiliser le message m **Jusqu'à(faux)**

Modèle producteur/consommateur (sémaphore)

- **Cas M producteurs , K consommateurs et un tampon de N emplacement**
- mutex1 et mutex2 pour garantir qu'un seul producteur à la fois accède à la variable « i » et qu'un seul consommateur à la fois accède à la variable « j ».
- Un sémaphore **s_vide** initialisé à **N**
- Un sémaphore **s_plein** initialisé à **0**
- Un sémaphore **mutex1** initialisé à **1**
- Un sémaphore **mutex2** initialisé à **1**
- Une variable **i** initialisé à **0**
- Une variable **j** initialisé à **0**

Producteur	***Consommateur***
Répétez	**Répétez**
Produire un message **m** ;	**P**(s_plein) ;
P(s_vide) ;	**P**(mutex2) ;
P(mutex1) ;	m ← Tampon[j] ;
Tampon[i] ← m ;	j ← (j +1) mod N ;
i ← (i +1) mod N ;	**V**(mutex2) ;
V(mutex1) ;	**V**(s_vide) ;
V(s_plein) ;	utiliser le message m
Jusqu'à(faux)	**Jusqu'à(faux)**

Modèle du Lecteur / Rédacteur

- Dans ce modèle plusieurs processus veulent accéder à une ressource commune. Un exemple d'utilisation est l'accès à un fichier partagé. Plusieurs lecteurs peuvent accéder simultanément au fichier puisque la lecture ne modifie pas les données, mais un seul rédacteur à la fois peut accéder au fichier.
- Pour lire le fichier, un lecteur exécute la fonction **Lire_fichier()** ;
- Pour écrie dans le fichier, un rédacteur exécute la fonction **Ecrire_fichier()** ;
- On utilise une variable **nbr_lec** pour déterminer le nombre de lecteurs utilisant le fichier à un instant données et deux sémaphores : **mutex** pour garantir l'exclusion mutuel de la variable « **nbr_lec** » et **S_fichier** pour déterminer si le fichier est libre ou non.

Modèle du Lecteur / Rédacteur

Lecteur	***Rédacteur***
Répétez **P**(mutex) **Si**(nbr_lec = 0) **alors** **P**(S_fichier) ; **Finsi ;** nbr_lec ← nbr_lec +1 ; **V**(mutex) ; Lire_fichier() ; **P**(mutex) ; nbr_lec ← nbr_lec - 1 ; **Si**(nbr_lec = 0) **alors** **V**(S_fichier) ; **Finsi ;** **V**(mutex) ; **Jusqu'à(faux)**	**Répétez** **P**(S_fichier) ; Écrire_fichier() ; **V**(S_fichier) ; **Jusqu'à(faux)**

Les régions critiques

- Les programme basé sur les sémaphore sont très difficile à corriger, la détection des erreurs dans le programme est très difficile.
- Des programme qui sembles correctes donnes parfois des résultats imprévisibles qui sont inacceptable.
- Les régions critique sont un moyen de programmation qui facilite l'utilisation des variables partagés.
- Les variables partagés sont déclarés par un mot clé **shared** et la partie du code ou on utilise ces variables **(section critique)** est déclaré en utilisant le mot clé **region**

Les régions critiques

- Seulement un processus à la fois peut exécuter la partie **region**, l'exclusion mutuel dans une région est garantie.
- En utilisant une programmation structuré les erreurs de programmation peuvent être détectés par les compilateurs.
- Les région critiques sont réalisées en utilisant des sémaphores **mutex** d'exclusion mutuel.
- Les appels **P(mutex)** avant d'exécuter la **region** critique et **V(mutex)** après l'exécution de la **region** critique seront automatiquement générés par le compilateurs.
- L'inconvénient des régions critique est qu'il sont limités et ne permettent pas de réaliser des synchronisation complexe.

```
/* DECLARATION DES VARIABLES */

var  Data1 : shared   record
                      A : integer ;
                      B : integer ;
                      end ;

var  Data2 : shared   record
                      C : integer ;
                      D : integer ;
                      end ;


/* UTILISATION  DES VARIABLES */
region Data1  Do
              section critique
              end ;
 ....
 ....
 ....

region Data2  Do
              section critique
              end ;
 ....
 ....

region Data1  Do
              ....
              ....
              region  Data2  Do
                             ....
                             ....
                             end ;
              ....
              ....
              end ;
```

Les moniteurs

- les moniteurs sont une amélioration des régions critique. Un moniteur peut être vu comme une classe qui rassemble à la fois les données partagés et le code pour accéder à ces données (section critique).
- A un instant donnée un seul processus peut être actif dans le moniteur (un seul processus à la fois exécute les procédures du moniteur). L'exclusion mutuelle à l'intérieur du moniteur est donc garantie.

```
Moniteur  nom_moniteur

    /* Déclaration des variables du moniteurs */
    ……
    ……

    /* Déclaration des procédures du moniteurs */

    Procedure1 (paramètres)
    Début
    …..
    …..
    Fin ;

    Procedure2 (paramètres)
    Début
    …..
    …..
    Fin ;

    /* Initialisation des variables du moniteurs */
    Début
    …..
    …..
    Fin ;

Fin Moniteur ;
```

Les moniteurs

- En utilisant les moniteur un seul client à la fois peut mettre à jour le solde du compte. La cohérence des données est donc assuré.
- Le moniteur peut suspendre l'exécution d'un processus et le mettre dans une file d'attente si un autre processus exécute une procédure du moniteur.

```
Moniteur Compte_ccp
/* Déclaration des variables */
Solde : Réel ;
/* Déclaration des procédures */
Verser_argent (X : réel)
Début
Solde ← Solde + X :
Fin ;
Retirer_argent(X : réel ; var erreur: booléen)
Début
Si (Solde > X) alors
Solde ← Solde - X :
erreur ← faux ;
Sinon
erreur ← vrai ;
Fin_Si
Fin ;
/* Initialisation des variables */
Début
Solde ← 0 ;
Fin ;
Fin moniteur
---------------------------------
/* Un client qui veut verser de l'argent */
Début
Lire(montant) ;
Compte_ccp.Verser_argent (montant) ;
Fin ;
---------------------------------
/* Un client qui veut retirer de l'argent */
Début
Lire(montant) ;
Compte_ccp.Retirer_argent (montant, Erreur) ;
Fin ;
```

Les variables conditions(Moniteur)

- L'exclusion mutuelle dans le moniteur n'est pas suffisante, on a besoin d'un mécanisme qui permet de bloquer et débloquer les processus. Par exemple si on veut traiter le problème du **producteur/consommateur** on a besoin d'un mécanisme pour suspendre les producteurs si le tampon et plein et aussi pour suspendre les consommateurs si le tampon et vide.
- Dans les moniteurs on peut déclarer des variables spéciales appelés **variables conditions**. Ces variables n'ont pas de valeurs et ne sont pas initialisées, ils ont deux opération permises **wait** et **signal**.

Les variables conditions(Moniteur)

- L'opération **wait** suspend l'exécution du processus appelant et le met dans une file d'attente des processus suspendus.
- L'opération **signal** réveil un processus suspendu de la file d'attente.
- s'il n' y a aucun processus dans la file l'opération **signal** n'a aucun effet à l'inverse de la primitive **V (s)** des sémaphore qui enregistre l'appel en ajoutant **1** au sémaphore.

producteurs/consommateurs avec un tampon de N cases.

- **Remarque importante:**
- Une question se pose lorsque un processus **P** éveil un autre processus **Q** dans le moniteurs en exécutant une instruction signal, on aura donc deux processus dans le moniteurs (**P** et **Q**) mais un seul doit pouvoir s'exécuter.
- On a donc deux possibilité :
- 1) P attend que Q quitte le moniteur (ou se bloc dans une autre condition),
- 2) Q attend que P quitte le moniteur (ou se bloc dans une autre condition).

```
Moniteur Tampon
var Tab[N] : tableau de messages ;
compteur : entier
i , j : entiers
Vide , Plein : conditions
Déposer_message(m : message)
Début
Si ( compteur = N) alors Vide.wait()
Tab [ i ] ← m ;
i ← (i+1) mod N;
compteur ← compteur +1 ;
Si ( compteur = 1) alors Plein.signal() ;
Fin ;
Retirer_message (m : message)
Début
Si ( compteur = 0) alors Plein.wait()
m ← Tab [ j ];
j ← (j +1) mod N;
compteur ← compteur -1 ;
Si ( compteur = N-1) alors Vide.signal() ;
Fin ;
/* Partie initialisation */
Début
i ← 0 ; j← 0 ; compteur ← 0 ;
Fin ;
Fin Moniteur ;
---------/* Programme producteurs */---------
Répéter
Produire message m
Tampon.Déposer_message(m) ;
Jusqu'à (faux)
---------/* Programme consommateurs */----------
Répéter
Tampon.Retirer_message(m) ;
utiliser le message m
Jusqu'à (faux)
```

Les moniteurs

- La première alternative (Q attend P) semble raisonnable puisque P est déjà en exécution, mais dans ce cas la condition pour laquelle on a réveillé Q peut changer, la deuxième solution semble donc plus approprié.
- par la suite on va faire en sorte qu'un processus réveil un autre processus à la fin de son exécution ou lorsque il vas se bloquer dans une condition.

Les moniteurs

- Pour implémenter les moniteurs on utilise les sémaphores comme suit :

1) On utilise un sémaphore **mutex** initialisé à **1** qui permet de garantir l'exclusion mutuelle pour l'accès aux procédures du moniteur. On ajoute à chaque procédure du moniteur un **P(mutex)** au début et un **V(mutex)** à la fin,
2) Pour chaque variable condition **C** on associe un sémaphore **SC** initialisé à **0** pour suspendre les processus qui exécutent un **wait** d'une condition et libérer ces processus quant on exécute un **signal**, on lui associe aussi un entier **nbr_C** pour compter le nombre de processus bloqués dans la condition **C**.
3) Un autre sémaphore **SM (sémaphore moniteur)** initialisé à **0** pour suspendre les processus dans le moniteur après l'exécution d'un **signal**, Ainsi qu'un entier **nbr_SM** pour compter le nombre de processus bloqués dans le sémaphore **SM**.

Les moniteurs

- Une procédure du moniteur est implémenté comme suit:

```
Procedure_moniteur()
Debut
……
……
……
Fin ;
```

```
Procedure_moniteur()
Debut
P(mutex) ;
……
Si (nbr_SM>0) alors
V(SM)
Sinon
V(mutex) ;
Finsi ;
Fin ;
```

Les moniteurs

- Une condition est déclaré comme suit :

Condition C ;	Semaphore C=0 ; nbr_C=0 ;

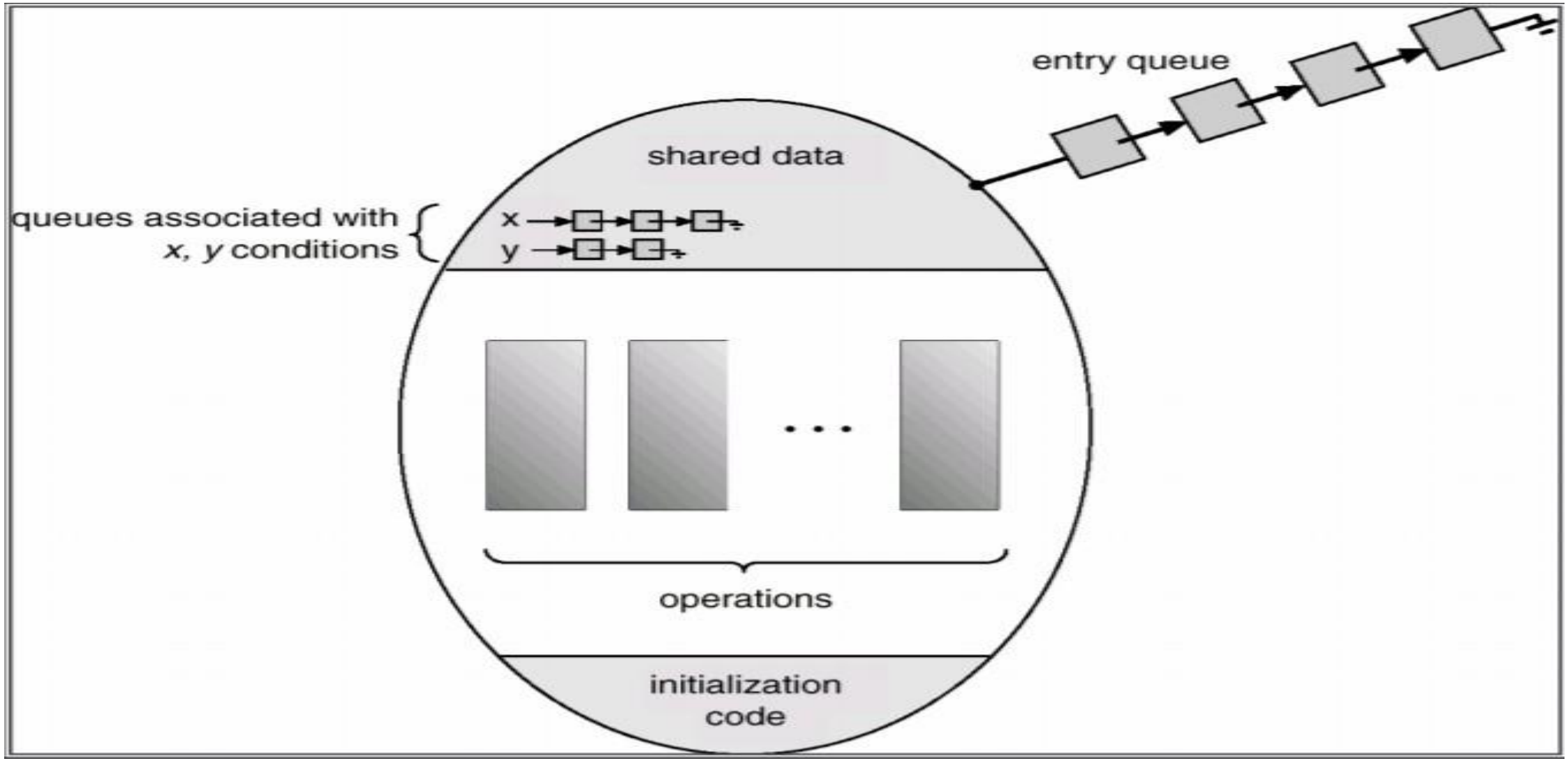
- Un wait est implémenté comme suit :

C.wait() ;	nbr_C = nbr_C+1;
	
	Si (nbr_SM>0) **alors**
	  V(SM) ;
	Sinon
	  V(mutex) ;
	Finsi
	
	P(SC) ;
	nbr_C = nbr_C-1;

Les moniteurs

- Un signal est implémenté comme suit :

C.signal() ;	**Si** (nbr_C>0) **alors**
	V(SC) ;
	nbr_SM = nbr_SM+1;
	P(SM) ;
	nbr_SM = nbr_SM-1;
	Finsi ;



La structure générale d'un moniteur. X et Y sont des variables condition (ref : OPERATING SYSTEMS PROCESS SYNCHRONIZATION , Jerry Breecher).

Les expressions de chemins

(Ref : « The specification of process synchronization by path expressions » R.H Campbell et A.N Heibermann university of Newcastle 1974)

- Les expressions de chemin permettent de spécifier tout les besoins de la synchronisation d'un ensemble d'opération d'une classe en un seul endroit. C'est au compilateur de synchroniser chaque opération selon la spécification d'un programmeur en déclarant des sémaphores et en ajoutant des P et des V au début et a la fins des opérations. Une expression de chemin s'écrit comme suit :

PATH expression END;

Les expressions de chemins

- Une expression contient les noms des opérations et les opérateurs de chemin. les opérateurs de chemin sont : « **,** **;** **{ }** ».
  Selection **,**
  Sequence **;**
  Exécution simultané **{ }**
  a , b : Veut dire que le système peut exécuter
  a , soit **b.**
  a ; b : veut dire que pour exécuter **b** ont doit d'abord exécuter **a** (séquence),
  {expression} : On peut avoir un nombre illimité d’exécution concurrentes de l'expression.

Les expressions de chemins

- **Exemple:**
- **PATH** write **END;** A un instant donné on peut exécuter qu'un seul *write* (exclusion mutuelle des *write*).
- **PATH** read, write **END;** A un instant donné on peut exécuter soit un *read*, soit un write.
- **PATH** deposer ; retirer **END;** Pour exécuter un *retirer* il faut d'abord exécuter un déposer.
- **PATH** p ; (q, r) ; s **END;** Les opérations de séquencement et de sélection sont combiné.
- **PATH** {read} **END;** On peut exécuter plusieurs operations *read* en concurrence.
- **PATH** {read}, write **END;** On peut exécuter plusieurs *read* en concurrence ou un seul *write.*

- **Le compilateur va ajouter les opérations P et V** des sémaphores **automatiquement** en suivant les étapes suivantes :
- **Etape 1:**

Le sémaphore **S** est initialisé à **1**

- **Etape 2:**

Le sémaphore **S** est initialisé à **0**

<expression>
deviens :
P(S) <expression> **V(S)**

<expression1> ; <expression2>
deviens :
<expression1> **V(S)** **P(S)** <expression1>

L <expression1> **,** <expression2> **R**
deviens :
L <expression1> **R** **L** <expression1> **R**

P(Si) {<expression>} **V(Sj)**
deviens :
PP(c , S , Si) {<expression>} **VV(c , S , Sj)**

PP(counter c , semaphore S , Si)
Début P(S) ; c=c+1 ; **Si** (c=1) **alors** P(Si) ; V(S) ; **Fin.**

Les expressions de chemins

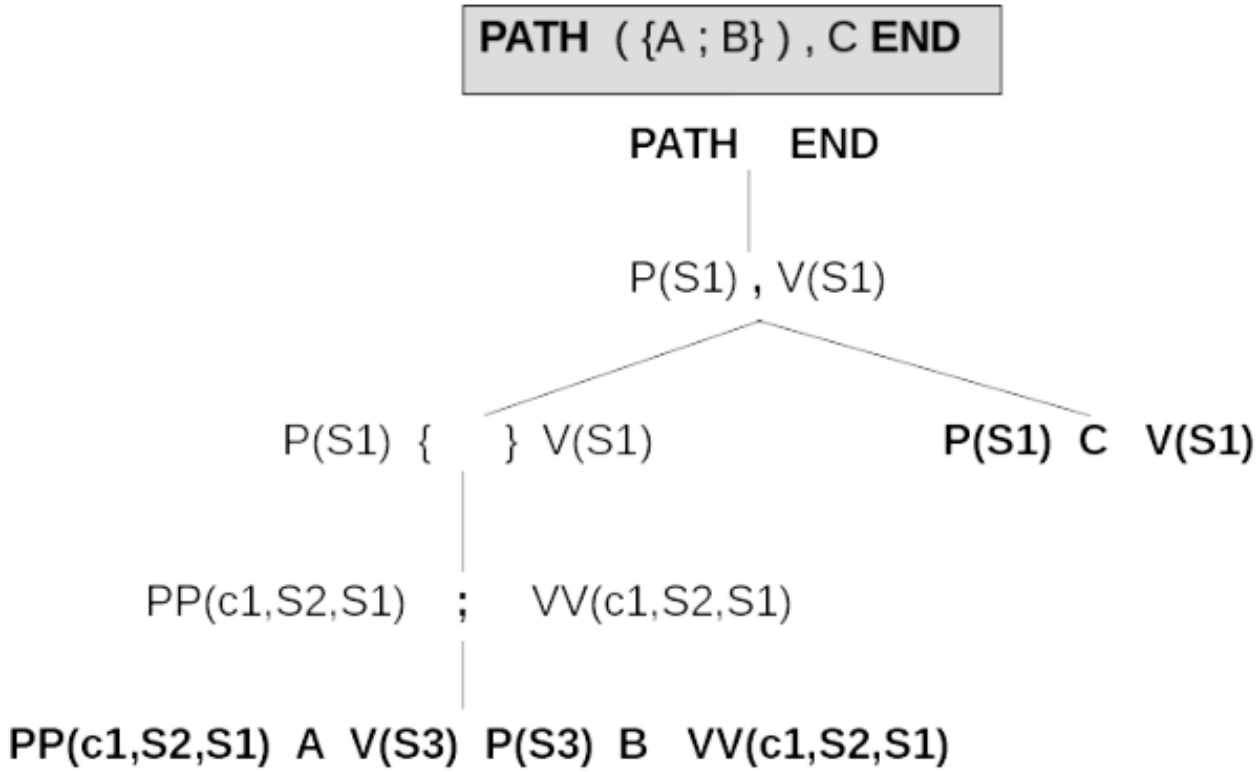
- Le sémaphore **S** est initialisé a **1**
- **Etape 3**
  Les opérations de synchronisation avant et après les tâches sont ajouté aux procédures des tâches

VV(counter c1 , semaphore S1 , Sj)

Début
P(S) ;
  c=c-1 ;
  Si (c=1) **alors** V(Sj) ;
V(S) ;
Fin.

Les expressions de chemins

- **Exemple :** On veut synchroniser le système pour exécuter les taches **A**,**B** et **C** selon l'expression de chemin suivante :



Les expressions de chemins

- Les procédures A, B et C sont transformé automatiquement par le compilateur comme suit:

- **Remarque :**
- Il existe d'autres formalismes d'expression de chemin qui permettent de fixer le nombre d'occurrences parallèle mais il ne serons pas abordé dans ce cours.

Procedure A :
Début
PP(c1,S2,S1) ;
Exécuter A ;
V(S3) ;
Fin.

Procedure B :
Début
P(S3) ;
Exécuter B ;
VV(c1,S2,S1) ;
Fin.

Procedure C :
Début
P(S1) ;
Exécuter C
V(S1) ;
Fin.

CHAPITRE 2

- **2.1 Introduction**
- **2.2 Problème de l’exclusion mutuelle**
- **2.3 Solutions pour l'exclusion mutuelle (section critique)**
- **2.3.1** Solutions logiciels basées sur l'attente active
- **2.3.2** Solutions matériels basées sur le masquage des interruptions
- **2.3.3** Solutions matériels basées sur des instruction indivisibles
- **2.3.4** Solutions basées sur l’attente passive
- **2.4 Les verrous**
- **2.5 Les sémaphores**
- **2.5.1** Exclusion mutuel par sémaphore
- **2.5.2** Précédence des tâches
- **2.5.3** Coopération entre N processus
- **2.5.4** Rendez-vous entre deux processus
- **2.5.5** Modèle du producteur / consommateur
- **2.5.6** Modèle du Lecteur / Rédacteur
- **2.6 Les régions critiques**
- **2.7 Les moniteurs**
- **2.7.1** Les variables condition
- **2.8 Les expressions de chemins**