LA GESTION DES FICHIERS ET LA SYNCHRONISATION DANS LES SYSTEMES CENTRALISES

PLAN

CHAPITRE II : SYNCHRONISATION DES PROCESSUS

1 Introduction

Pour analyser le fonctionnement d'un système d'exploitation, on est amené à considérer l'ensemble des activités que gère ce système. Ces activités appelées aussi processus sont des entités de base évoluant dans ce système.

Un **processus** peut être défini comme l'exécution d'un programme comportant des instructions et des données : c'est un élément dynamique créé à un instant donné et qui disparaît en général au bout d'un temps fini, après être passé par différents états au cours de sa durée de vie dans le système.

Les processus, multiples dans un système d'exploitation, n'évoluent pas toujours de manière indépendante. Ils **coopèrent** pour réaliser des tâches communes. Ils partagent donc des données et plus généralement des ressources.

Pour mettre en œuvre cette coopération, **des mécanismes dits de synchronisation** doivent être offerts par le système d'exploitation. Ces mécanismes permettent d'ordonner l'exécution des processus qui coopèrent entre eux.

2 Partage des ressources et des données

L'exécution d'un processus nécessite un certain nombre de ressources. Ces ressources peuvent être **logiques** ou **physiques**. Les ressources physiques sont la mémoire, le processeur, les périphériques etc. Les ressources logiques peuvent être une variable, un fichier, un code, etc.

Certaines ressources peuvent être utilisées en même temps (ou partagées) par plusieurs processus. Dans ce cas, elles sont dites **partageables** ou **à accès multiple**. Dans le cas contraire, elles sont dites **à un seul point d'accès** ou **à accès exclusif**. Dans ce dernier cas, il est nécessaire d'ordonner l'accès à ce type de ressources pour éviter des **situations incohérentes**.

2.1 Exemples de situations incohérentes

A. Problème de ressource matérielle partagée

Soient deux programmes P1 et P2 désirant imprimer sur la même imprimante (mode caractère). La fonction **Ecrire** est supposée être une **opération atomique** d'impression d'un caractère.

P1 **P2**

```
var tampon t1='XYZT' ;   var tampon t2='ABCD' ;
pour i :=1 à 4 faire     pour i :=1 à 4 faire
      Ecrire(t1[i]) ;          Ecrire(t2[i]) ;
Fait                     Fait
```

En l'absence de synchronisation explicite, on peut voir imprimer n'importe quelle séquence de caractère respectant l'ordre d'exécution de chaque programme ('XYZTABCD' 'XAYBZCTD', 'XYZABCDT', …etc.). Mais, on devrait voir imprimée la séquence 'abcdABCD'.

B. Problèmes des variables partagées

Le partage de variables sans précaution particulière peut conduire à des résultats imprévisibles.

Exemple 1 : Soit le programme de réservation de places (avion, théâtre, etc.) suivant :

```
Programme_reservation
      If PLACE_DISPO > 0
            Then PLACE_DISPO := PLACE8DISPO – 1 ;
                  Répondre (‘Place réservée’) ;
            Else  Répondre (‘Plus de place’) ;
      Endif
```

Traduit en langage d’assemblage, ce programme devient :

```
(B1)  charger     R1    PLACE_DISPO
(B2)  sauterinst  R1    Etiq  // Saut à Etiq si R1 = 0%
(B3)  Soust       R1    1
(B4)  charger     R1    PLACE_DISPO
(B5)  Repondre    (‘Place réservée’)
(B6)  sauter            Fin
(B7)  Etiq: Repondre    (‘Plus de place’)
(B8)  Fin:  Stop
```

Plusieurs demandes de réservation simultanées émanant d’agences de réservation différentes engendreront l’exécution de plusieurs processus composés de cette même séquence d’instructions avec PLACE_DISPO comme **variable commune**.

Un scénario possible de l’exécution de deux processus P1 et P2 est le suivant :

PLACE_DISPO := 1 ;

P1 s’exécute jusqu’à B3, les valeurs correspondantes de R1 seront : R1= 1 **(B1)**

R1 ≠ 0 **(B2)**

R1 = 0 **(B3)**

Puis, P2 s'exécute jusqu'à la fin (PLACE_DISPO est toujours égale à 1), il réserve une place et se termine.

Ensuite, P1 continue son exécution à partir de B4 : PLACE_DISPO = `0` **(B4)**

P1 réserve une place et se termine **(B5)**

Il y a alors incohérence car une même place a été réservée deux fois.

Ce phénomène n'est pas limité aux processus qui exécutent le même programme.

Exemple 2 : Incrémentation et décrémentation d'un compteur.

Soit deux processus P1 et P2 qui respectivement incrémente et décrémente une variable x.

P1 :	**P2 :**
`1 move Ax, x`	`1' move Ax, x`
`2 inc Ax`	`2' dec Ax`
`3 move x,Ax`	`3' move x,Ax`

Si x = 1, l'entrelacement de l'exécution de P1 et P2 donne en résultat une valeur de x égale à 0, 1 ou 2.

3 Les règles de Bernstein

Pour autoriser deux tâches à s'exécuter en parallèle sans risques d'incohérence, Bernstein a mis en place un ensemble de règles qui doivent être respectées avant de lancer l'exécution des taches. Pour cela, il a définit deux ensembles R et W ; ou R définit l'ensemble de lecture (Read), c'est l'ensemble des variables consultées par la tache . Et W est l'ensemble d'écriture (Write) qui définit l'ensemble des variables modifiées par la Pour que deux taches T1 et T2 s'exécutent en // sans risque d'incohérence, il faudrait vérifier les trois égalités suivantes :

$$R(T1) \cap W(T2) = \emptyset$$

$$W(T1) \cap R(T2) = \emptyset$$

$$W(T1) \cap W(T2) = \emptyset$$

Application :

T1 : read(x) ; T2 : read(y) ; T3 : Z1= x+y ; T4 : Z2= x*y ; T5 : v= z1+z2 ; T6 : write (v)

4 Exclusion mutuelle

Quand il s'agit de partager une ressource à un seul point d'accès, les processus doivent l'utiliser de manière séquentielle selon un ordre défini par le mécanisme de synchronisation utilisé. La portion de code utilisant la ressource est appelée **section critique**. La ressource est appelée **ressource critique** et le mécanisme utilisé pour la synchronisation, mécanisme d'**exclusion mutuelle**.

4.1 Problème de la section critique

Considérons un système comprenant *n* processus $\{P_0, P_1, \ldots, P_{n-1}\}$

$\mathbf{P_0}$	$\mathbf{P_1}$	…	$\mathbf{P_{n-1}}$	
…	…		…	
$<SC_0>$	$<SC_1>$		$<SC_{n-1}>$	$\Rightarrow$ un seul processus qui exécute $<SC_i>$
…	…		…	

Quand un processus est en exécution dans sa section critique, aucun autre n'est autorisé à exécuter sa section critique. Il est donc nécessaire de concevoir un protocole permettant aux processus de s'exclure mutuellement quant à l'utilisation de la ressource critique.

La forme générale d'un processus devient alors :

Pi

Répéter

…

<Protocole d'entrée>

<Section critique>

<Protocole de sortie>

…

Jusqu'à faux

4.2 Conditions de la section critique

Une solution adaptée au problème de la section critique (sc) doit satisfaire les conditions suivantes :

1. **Exclusion mutuelle :** Si un processus Pi est dans sa section critique, aucun autre ne doit être dans la sienne (au plus un processus doit être autorisé à y accéder : 0 ou 1)

2. **Absence de blocage mutuel :** Les processus ne doivent pas s'empêcher mutuellement d'entrer en sc au point où la ressource reste libre et aucun ne pourra rentrer en se bloquant indéfiniment..

3. **Progression ;** Si un processus opère en dehors de sa section critique, il ne doit pas empêcher un autre d'entrer dans sa section critique.

4. **Attente bornée :** un processus ayant demandé d'entrer en section critique doit y accéder au bout d'un temps fini. On doit éviter le problème de famine.

5. Mécanisme d'exclusion mutuelle

5.1 Hypothèses

On suppose que :

- le système est composé de *n* processus P_0, P_1, …, P_n qui s'exécutent de manière parallèle,
- les vitesses d'exécution des processus sont quelconques et inconnues,
- les instructions de base du langage machine sont exécutées de manière atomique.
- tout processus qui entre dans sa section critique doit forcement sortir au bout d'un temps fini.

5.2 Solutions sans support matériel (les variables d'état)

Ces solutions consistent à utiliser des **variables communes** partagées entre les processus. Pour simplifier la présentation de ces solutions, on considérera uniquement deux processus Pi et Pj.

- **Solution 1**

```
Tour := i ;
Pi :
      Répéter
            Tant que (tour = j) faire <rien> fait ;
```

```
                <SCi>
            tour := j
      Jusqu'à faux
```

L'analyse :

- **Solution 2**

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

```
drapeau : Tableau[0,1] de booléen := Faux ;
Pi :
      Répéter
            drapeau[i] := vrai ;
            Tant que drapeau[j] faire <rien> fait ;
                  <SCi>
            drapeau[i] := faux ;
            <Section restante>
      Jusqu'à faux
```

L'analyse :

- **Solution 3 (Algorithme de Peterson, 81)**

  En combinant les avantages des deux solutions 1 et 2, l'algorithme suivant résout le problème de la section critique.

```
drapeau : Tableau[0,1] de booléen := Faux ;
tour := i ou j ;
Pi :
      Répéter
            drapeau[i] := v ;
            tour := j ;
            Tant que (drapeau[j] ∧ tour=j) faire <rien> fait ;
                  <SCi>
            drapeau[i] := f ;
            <Section restante>
      Jusqu'à faux
```

L'analyse

5.3 Solutions matérielles

Différentes solutions se basant sur le matériel ont été développées pour résoudre le problème de la section critique. Ces solutions rendent la tâche de programmation aisée.

- **Masquage des interruptions**

  Avant d'entrer dans une section critique, le processus masque les interruptions. Il les restaure à la sortie de la section critique. Il ne peut être alors suspendu durant l'exécution de la section critique.

  Problèmes

  - **Solution dangereuse ;** certaines tâches du système fonctionnant à l'aide des interruptions seront donc inhibées (**Ex.** horloge). Il n'est pas très judicieux de données aux processus utilisateur le pouvoir de désactiver les interruptions.

  - **Elle n'assure pas l'exclusion mutuelle en multiprocesseur;** si le système n'est pas monoprocesseur (le masquage des interruptions concerne uniquement le processeur qui a demandé l'interdiction). Les autres processus exécutés par un autre processeur pourront donc accéder aux objets partagés.

  En revanche, cette technique est parfois utilisée par le système d'exploitation.

- **Instructions spéciales**

  Ce sont des instructions fournies par la machine et qui sont de nature **indivisibles**. Lorsqu'un processeur exécute ces instructions, il verrouille le bus de données pour empêcher les autres processeurs d'accéder à la mémoire pendant la durée de l'exécution.

  Instruction Test-and-Set

  De nombreux ordinateurs disposent d'une instruction élémentaire (**Test−and−Set, TAS**) exécutée par le **matériel**, qui permet de lire et d'écrire le contenu d'un mot de la mémoire de manière indivisible.

```
Function Test-and-Set(var x : boolean) : boolean ;
begin
      Test-and-Set := x ;
      x := true ;
End
```

Programmation de la section critique à l'aide de TAS

```
var Lock : booleen := false ;
Pi
      Répéter
            Tant que Test-and-Set(Lock) faire <rien> fait ;
               <SCi> ;
            Lock := false ;
            <Reste du processus>
      jusqu'à faux
```

Cette solution fonctionne pour un nombre indéterminé de processus.

Le premier processus qui arrive exécute la boucle en initialisant Test-and-Set à faux (c'est-à-dire, la valeur initiale de Lock) et entre en section critique.

Tous les autres processus qui arrivent après se bloquent car Lock est maintenant à vrai et par conséquent la fonction Test-and-Set retournera la valeur vrai tant que le premier processus est en section critique. Lorsque ce dernier quitte la section critique, il remet Lock à faux et donc permettra à un des processus en attente d'entrer à son tour en section critique et ainsi de suite.

NB : Notons que si deux processus tentent d'exécuter cette instruction simultanément ; ceci se traduira par une exécution séquentielle.

Problèmes

- Attente active (**Busy Waiting**) = consommation du temps CPU.
- Un processus en attente active peut rester en permanence dans la boucle (**Attente non bornée**).

Remarques

- Cette instruction TAS permet de synchroniser des activités en environnement multiprocesseur.
- Les constructeurs ont introduit différents types d'instructions offrant les mêmes fonctions que TAS. Par exemple, l'**Intel pentium** dispose d'une instruction appelée **XCHG** qui échange de façon indivisible les contenus d'un registre et d'un emplacement de mémoire. Sur les processeurs **SPARC**, qui équipent les stations Sun, l'instruction de TAS s'appelle **ldstub** (**load store unsigned byte**).

Chapitre III Les Sémaphores

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

- Elles ne peuvent être généralisées à un nombre **illimité** de processus,
- Elles sont valable pour la synchronisation à un seul point d'acces
- Elles ne s'adaptent pas à un problème général de synchronisation,
- Elles posent le problème d'**attente active** tant qu'un processus est en section critique, ce qui induit une consommation inutile du temps CPU.

Pour résoudre principalement ce dernier problème, il faut introduire un mécanisme qui permet de **BLOQUER** un processus si la condition d'entrée en section critique n'est pas vérifier et de le **REVEILLER** pour y accéder aussitôt que la condition soit satisfaite.

A. Définition [Dijkstra 1965]

Un sémaphore **S** est une variable entière associée à une file d'attente **f(S)**. La variable S peut prendre des valeurs positives, nulles ou négatives. Cependant, sa valeur initiale est toujours **≥ 0**. La politique de gestion de la file d'attente f(S) est quelconque, mais elle est en général de type **FIFO**.

Un sémaphore **S** est manipulé **EXCLUSIVEMENT** par l'intermédiaire des trois primitives suivantes : INIT, P, V

```
INIT
Début
   S := n ;
   f(s) := ∅ ;
Fin
P (S)  /* P pour Proberen */
Début
  Si S = 0 Alors
         Début
               Etat(Pi) := Bloqué ; /*Pi le processus qui exécute P(S)*/
               Mettre le processus Pi dans f(S) ;
               Appel du Scheduler pour choisir un autre processus ;
         Fin
   Sinon S-- ;
   Fsi ;
Fin
```

```
V (S) /* V pour Verhogen */
Début
   Si F(s) est non vide
     Alors
         Début
               Faire sortir un processus de f(S);
               /*Soit Q ce processus*/
```

```
                Etat(Q) := Prêt ; /* Appel du Scheduler */
          Fin
    Sinon  S++
    Fsi ; Fin
```

Remarques

- L'exécution de P et V est **atomique**.
- On peut avoir une autre implémentation de P et V

```
 P(S)
Début
   S := S – 1 ;
   Si S < 0 Alors
          Début
                Etat(Pi) := Bloqué ; /*Pi le processus qui exécute P(S)*/
                Mettre le processus Pi dans f(S) ;
                Appel du Scheduler pour choisir un autre processus ;
          Fin
   Fsi ;
Fin
```

```
V (S)
Début
   S := S + 1 ;
   Si S ≤ 0 Alors
          Début
                Faire sortir un processus de f(S);
                /*Soit Q ce processus*/
                Etat(Q) := Prêt ; /* Appel du Scheduler */
          Fin
   Fsi ;
Fin
```

Remarques

- Si la **valeur de S** est **négative**, sa valeur absolue, **|S|**, est le **nombre de processus bloqués** dans la file f(S).
- Si la **valeur de S** est **positive**, elle indique le **nombre de processus** qui peuvent **franchir** la primitive P(S) **sans se bloquer**.

Il existe trois mode de fonctionnement de sémaphores selon leurs initialisations

A. Sémaphore binaire

Un sémaphore binaire est un sémaphore dont la **valeur initiale** est **égale à 1**. Il est utilisé en général dans l'exclusion mutuelle.

La forme générale d'un programme est :

```
Pi
      début
            ...
            P(S)
            <SCi>
            V(S)
      Fin
```

B. Sémaphore privé

La **valeur initiale** d'un sémaphore privé est **égale à 0**. On dit que S est privé à un processus s'il est le seul à pouvoir exécuter P(S), les autres processus ne peuvent exécuter que V(S).

Ce type de sémaphore est utilisé par un processus quand il désire se bloquer volontairement et un autre qui doit le reveiller.

Utilisation des sémaphores privés

1. Soit deux processus P1 et P2 coopérant entre eux pour réaliser un travail constitué de deux parties. Le processus P1 doit terminer la première partie avant que le processus P2 n'entame la deuxième.

```
                        S : sémaphore := 0 ;

P1                                  P2

   <Réaliser la première partie>      P(S)

   V(S)                               <Réaliser la deuxième partie>
```

2. Ils peuvent être utilisés pour réaliser un rendez vous entre processus

3. Ils peuvent être utilisés pour exprimer tout graphe par l'outil **parbegin/parend**

C. Sémaphore compteur (Counting Semaphores)

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

Le modèle d'un processus est alors :

```
P(S) ;

< Utilisation de la ressource > ;

V(S) ;
```

Les n premiers processus franchissent P(S) sans se bloquer.

Le (n+1)ième processus, si les n premiers processus sont en cours d'utilisation de ressource, sera bloqué.

De même que les autres processus qui arrivent après.

Quand un processus termine d'utiliser la ressource (il exécute V(S)), il libère le premier processus bloqué. Ainsi, à tout moment **au plus** n processus accèdent en même temps à la ressource partagée.

Exemple

Accès à un parking possédant n places et ayant une porte d'entrée et une porte de sortie qui laissent passer une seule voiture à la fois.

```
mutex1 ; mutex2 : sémaphore := 1 ;

S : sémaphore := n ;

Pi

   P(S) ;      % y-a-t-il une place libre ?%

   P(mutex1) ;

   <Entrée> ;  %Accès d'une seule voiture à la fois%

   V(mutex1) ;

   <Se parquer> ;

   P(mutex2) ;

   <Sortie> ;  %Sortie d'une seule voiture à la fois%

   V(mutex2) ;

   V(S) ;      %libération d'une place%
```

- **Problèmes de la mise en œuvre**

L'utilisation des sémaphores pose principalement un problème appelé **interblocage**

- **Interblocage (Deadlock)**

Considérons l'exécution suivante de deux processus P1 et P2 utilisant deux ressources non partageables R1 et R2.

R1, R2 : sémaphore := 1 ;	
P1	**P2**
P(R1) :	P(R2) ;
P(R2) ;	P(R1) ;
<accès à R1 et R2> ;	<Accès à R2 et R1> ;
V(R2) ;	V(R1) ;
V(R1)	V(R2)

Considérant l'ordre d'exécution suivant des deux processus P1 et P2 :

P1 exécute P(R1) ↔ R1 = 0

P2 exécute P(R2) ↔ R2 = 0

P1 exécute P(R2) ↔ R2 = -1 ⇒ P1 se bloque

P2 exécute P(R1) ↔ R1 = -1 ⇒ P2 se bloque

Aucun des deux processus P1 et P2 ne pourra continuer son exécution. On dira que P1 et P2 sont **interbloqués**.

Application : Gestion d'une bibliothèque

5.5 Autres mécanismes de synchronisation

La coopération des processus nécessite des mécanismes qui coordonnent leur exécution dans le **temps**, selon la logique de la fonction commune à accomplir.

Ces mécanismes doivent permettre à un processus :

- De bloquer un autre processus ou de se bloquer en attendant un signal d'un autre processus.
- D'activer un autre processus. Deux cas peuvent se présenter :
  - 1er cas : soit le signal est mémorisé. Dans ce cas, le processus ne se bloquera pas lors de sa prochaine opération de blocage.
  - 2ème cas : soit le signal n'est pas mémorisé. Il est alors perdu si le processus ne l'attend pas.

A. Les événements

Un **événement** est une **abstraction d'une action** qui se produit dans un système.

Il peut être le résultat de l'exécution d'une instruction ou d'un ensemble d'instructions reconnu comme tel.

Un événement est désigné par un **identificateur** et il est créé par une **déclaration**.

La synchronisation à l'aide des événements est mise en œuvre par l'**attente** ou le **déclenchement** de l'événement.

L'événement peut être **mémorisé** ou **non mémorisé**.

B. Evénements mémorisés

Un processus se bloque ssi l'événement qu'il attend n'est pas mémorisé.

Selon les systèmes d'exploitation, un ou plusieurs processus sont débloqués lors du déclenchement d'un événement.

Exemple (expression d'un RDV de deux processus)

Processus P1	Processus P2
...	...
Déclencher(e1) ;	Déclencher(e2) ;
Attendre(e2) ;	Attendre(e1) ;
...	...

C. Evénements non mémorisés

Un événement déclenché alors qu'aucun processus ne l'attend est **perdu**.

Dans le cas où un ou plusieurs processus sont bloqués dans l'attente d'un événement, tous les processus sont débloqués lors de son déclenchement.

Ce type d'événements est utilisé principalement pour le contrôle de procédés industriels (Application temps réel).

Remarque

Le raisonnement sur les événements peut être développé en considérant qu'un processus attend sur une condition booléenne constituée de plusieurs événements (**Ex.** Attendre(e1 ou e2), Attendre((e1 et e2) ou e3)).

CHAPITRE IV : COMMUNICATION INTER-PROCESSUS

1 Introduction

Dans un système d'exploitation, en plus de leur compétition pour l'acquisition de ressources, les processus peuvent **COOPERER** pour réaliser des tâches communes. Cette coopération nécessite un échange d'informations entre ces processus (**Communication Inter-Processus)**.

Pour réaliser cette communication, il est nécessaire d'utiliser des outils de synchronisation permettant de coordonner les processus dans leur communication.

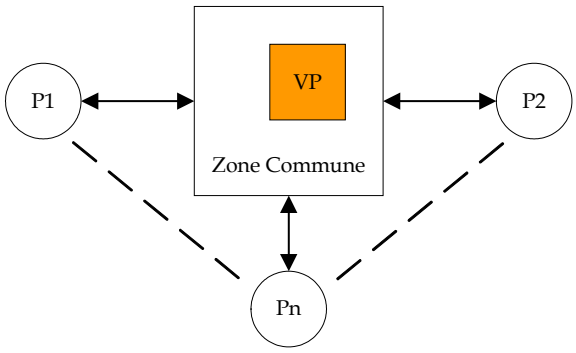
2 Types de communication

Les processus peuvent communiquer entre eux des deux manières suivantes :

- Par **partage de variable** et ceci à travers une zone mémoire commune. La synchronisation dans ce cas est à la charge de l'utilisateur.
- Par **échange de messages** où la communication est gérée par le système d'exploitation à travers deux primitives ; à savoir : **Send(Message)** et **Receive(Message)**.

3 Communication par partage de variable

La communication se fait à travers une zone mémoire commune. Chaque processus peut lire ou écrire dans cette zone. Plusieurs processus peuvent réaliser cette opération en concurrence d'où la nécessité d'utiliser des outils de synchronisation.

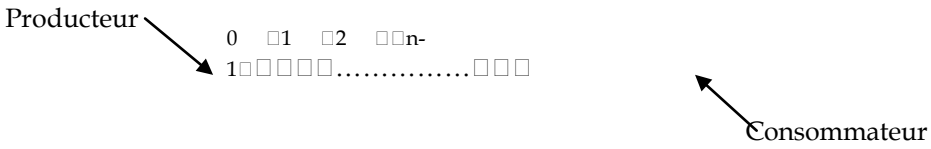


a. Modèle du Producteur/Consommateur

On distingue deux types de processus :

- Les processus qui produisent de l'information, dits **producteurs**.
- Les processus qui consomment cette information, dits **consommateurs**.

La communication se fait à travers un **tampon** de n cases.



Les processus utilisent deux primitives :

- Déposer(article).
- Prélever(article).

Soit **nb** le nombre d'articles contenus dans le tampon à un moment donné.

- Le producteur ne peut déposer que s'il y a de la place, i.e. nb < n.
- Le consommateur ne peut prélever que s'il y a au moins une case pleine, i.e. nb ≠ 0.
- Le consommateur doit prélever un article une seule fois.
- L'exclusion mutuelle doit être assurée au niveau de la même case.

A. Exemples du modèle producteur / consommateur

- Le processus clavier produit des caractères qui sont consommés par le processus d'affichage à l'écran.
- Le pilote de l'imprimante produit des lignes de caractères, consommées par l'imprimante.
- Un compilateur produit des lignes de code consommées par l'assembleur.

B. Modèle de 1 producteur / 1 consommateur

1ère solution

Processus Producteur	**Processus Consommateur**
Var art : Tart	**Var** art : Tart
Début	**Début**
Répéter	**Répéter**
Produire(art) ;	**Tq** (nb = 0) **Faire** <Rien>
Tq (nb = n) **Faire** <Rien>	**Fait** ;
Fait ;	Prélever(art) ;
Déposer(art) ;	nb := nb – 1 ;
nb := nb + 1 ;	Consommer(art) ;
<Suite>	<Suite>
Jusqu'à Faux ;	**Jusqu'à** Faux;
Fin.	**Fin.**

Problèmes

Si la condition de dépôt où de prélèvement n'est pas satisfaite, le processus attend de manière active ce qui induit une consommation inutile du temps CPU.

Exclusion mutuelle sur la variable partagée nb entre le producteur et le consommateur.

2ème solution

Dans cette solution, si la condition de dépôt ou de prélèvement n'est pas satisfaite, le processus se bloque. On utilise dans ce cas les primitives : Attendre() (**Sleep**), Réveiller() (**Wakeup**), tester l'état d'un processus.

Attendre() est un appel système qui provoque le blocage de l'appelant. Celui-ci est suspendu jusqu'à ce qu'un autre processus le réveille.

L'appel **Réveiller()** prend un paramètre, désignant le processus à réveiller.

Processus Producteur	**Processus Consommateur**
Var art : Tart	**Var** art : Tart
Début	**Début**
Répéter	**Répéter**
Produire(art) ;	**Si** (nb = 0) **Alors** Attendre() **Fsi** ;
Si (nb = n) **Alors** Attendre() **Fsi** ;	Prélever(art) ;
Déposer(art) ;	nb := nb – 1 ;
nb := nb + 1 ;	**Si** Attente(Producteur) **Alors** Réveiller(Producteur) **Fsi** ;
Si Attente(Consommateur) **Alors** Réveiller(Consommateur) **Fsi** ;	Consommer(art) ;
<Suite>	<Suite>
Jusqu'à Faux ;	**Jusqu'à** Faux;
Fin.	**Fin.**

Problème

Exclusion mutuelle sur la variable partagée nb entre le producteur et le consommateur.

3ème solution (Utilisation de sémaphores)

Var np, nv : entier ; np := 0; /* Nombre de cases pleines */ nv := n; /* Nombre de cases vides */	
Processus Producteur	**Processus Consommateur**
Var art : Tart	**Var** art : Tart
Début	**Début**
Répéter	**Répéter**
Produire(art) ;	np := np – 1 ;
nv := nv – 1 ;	**Si** (np = -1) **Faire** Attendre()

```
     Si (nv = -1) Alors Attendre()
     Fsi ;
     Déposer(art) ;
     np := np + 1 ;
     Si     np     =     0     Alors
     Réveiller(Consommateur) Fsi ;
   Jusqu'à Faux ;
Fin.
```

```
     Fait ;
     Prélever(art) ;
     nv := nv + 1 ;
     Si     nv     =     0     Alors
     Réveiller(Producteur) Fsi ;
     Consommer(art) ;
   Jusqu'à Faux;
Fin.
```

```
Var np, nv : Sémaphore ;
Init(np, 0); /* Nombre de cases pleines */
Init(nv, n); /* Nombre de cases vides */
```

```
Processus Producteur
Var art : Tart
Début
   Répéter
     Produire(art) ;
     P(nv) ;
     Déposer(art) ;
     V(np) ;
   Jusqu'à Faux ;
Fin.
```

```
Processus Consommateur
Var art : Tart
Début
   Répéter
     P(np) ;
     Prélever(art) ;
     V(nv) ;
     Consommer(art) ;
   Jusqu'à Faux;
Fin.
```

Propriétés

1. Si la consommation suit l'ordre de la production, le producteur et le consommateur n'opèrent jamais sur la même case.

2. Le producteur et le consommateur ne se bloquent jamais en même temps dû à la politique de gestion du tampon ; **gestion circulaire**.

   On utilise deux pointeurs :

   - t qui pointe la 1ère case pleine.
   - q qui pointe la 1ère case vide.

```
Déposer
Début
     T[q] := art ;
     q := (q+1) mod n ;
Fin.
```

```
Pvélever
Début
     art := T[t] ;
     t := (t+1) mod n ;
Fin.
```

C. Modèle de plusieurs producteurs / plusieurs consommateurs

```
Var
np, nv : Sémaphore ;
```

```
mutexp, mutexv : Sémaphore ;
Init(np, 0); /* Nombre de cases pleines */
Init(nv, n); /* Nombre de cases vides */
Init(mutexp, 1) ; /*Assurer l'accès en EM au tampon entre producteurs */
Init(mutexc, 1) ; /*Assurer l'accès en EM au tampon entre consommateurs*/
```

Processus Producteur	Processus Consommateur
Var art : Tart	**Var** art : Tart
Début	**Début**
Répéter	**Répéter**
Produire(art) ;	**P(np) ;**
P(nv) ;	**P(mutexc) ;**
P(mutexp) ;	Prélever(art) ;
Déposer(art) ;	**V(mutexc) ;**
V(mutexp) ;	**V(nv) ;**
V(np) ;	Consommer(art) ;
Jusqu'à Faux ;	**Jusqu'à** Faux;
Fin.	**Fin.**

Il est nécessaire de rendre l'accès au tampon en exclusion mutuelle dans chaque famille de processus.

b. Modèle des Lecteurs / Rédacteurs (Courtois et al. 1971)

Il s'agit de gérer l'accès concurrent à une ressource commune (**Ex.** Fichier) de plusieurs processus.

Deux opérations peuvent avoir lieu avec un fichier :

- Consultation (Lecture)
- Modification (Ecriture)

Pour garantir la cohérence des informations dans le fichier, les conditions suivantes doivent être respectées :

- Plusieurs lectures peuvent se faire en même temps.
- Pas d'écriture s'il y a au moins une lecture.
- Pas de lecture ni d'écriture s'il y a une écriture en cours.

Imaginez une grande base de données, où plusieurs processus tentent de lire et d'écrire des informations. On peut accepter que plusieurs processus lisent en même temps dans la base. Mais si un processus est en train de modifier la base en y écrivant des données, aucun autre processus, pas même un lecteur, ne doit être autorisé à y accéder.

A. Pas de priorité explicite

```
Var
```

```
mutex, W : Sémaphore ;
nl : entier ;
Init(nl, 0); /* Nombre de lecteurs en cours */
Init(W, 1); /*  */
Init(mutex, 1) ; /*Assurer l'accès en EM à nl entre lecteurs */
```

Processus Lecteur	Processus Rédacteur

Début
  Répéter
  P(mutex) ;
  nl := nl + 1 ;
  Si nl = 1 **Alors** P(W) **Fsi**;
  V(mutex) ;
  <Lecture>
  P(mutex) ;
  nl := nl - 1 ;
  Si nl = 0 **Alors** V(W) **Fsi**;
  V(mutex) ;
  Jusqu'à Faux ;
Fin.
	Début
	  Répéter
	  P(W) ;
	  <Ecriture> ;
	  V(W) ;
	  Jusqu'à Faux;
	Fin.

Commentaires

Le **fichier** est considéré comme une **ressource critique** pour un **rédacteur** ; c'est-à-dire, si un rédacteur est en cours, aucun lecteur, ni rédacteur ne peut y accéder. D'où l'utilisation du **sémaphore binaire W**.

Plusieurs lectures peuvent avoir lieu en même temps. Pour connaître le nombre de ces lectures, on utilise le compteur nl. Seul le 1er lecteur qui arrive teste l'occupation de la section critique par un rédacteur éventuel, les autres accèdent directement si le 1er réussit l'accès sinon ils se bloquent au niveau de mutex.

B. Priorité aux lecteurs

```
Var
mutex, W : Sémaphore ;
mutexp : Sémaphore ;
nl : entier ;
Init(nl, 0); /* Nombre de lecteurs en cours */
Init(W, 1); /*  */
Init(mutex, 1) ; /*Assurer l'accès en EM à nl entre lecteurs */
Init(mutexp, 1) ; /*Donner la priorité aux lecteurs*/
```

Processus Lecteur	Processus Rédacteur

Début
  Répéter
  P(mutex) ;
	Début
	  Répéter
	  P(mutexp) ;

```
    /*les autres lecteurs bloqués
    dans f(mutex)*/
    nl := nl + 1 ;
    Si nl = 1 Alors P(W) Fsi;
    /*le 1er lecteur bloqué
    derrière f(W)*/
    V(mutex) ;
    <Lecture>
    P(mutex) ;
    nl := nl - 1 ;
    Si nl = 0 Alors V(W) Fsi;
    V(mutex) ;
  Jusqu'à Faux ;
Fin.
```

```
    /*les autres Rédacteurs bloqués
    dans f(mutexp)*/
    P(W) ;
    <Ecriture> ; //Ecriture en cours
    V(W) ;
    V(mutexp) ;
  Jusqu'à Faux;
Fin.
```

mutexp empêche les rédacteurs d'attendre au niveau de W afin de donner la priorité aux lecteurs dont le 1er attend au niveau de W car une écriture est en cours.

C. Priorité aux rédacteurs

```
Var
mutex, W : Sémaphore ;
SemReserver : Sémaphore ;
nl : entier ;
Init(nl, 0); /* Nombre de lecteurs en cours */
Init(W, 1); /*  */
Init(mutex, 1) ; /*Assurer l'accès en EM à nl entre lecteurs */
Init(SemReserver, 1) ; /*Donner la priorité aux lecteurs*/
```

```
Processus Lecteur
Début
  Répéter
    P(SemReserver);
    P(mutexL) ;
    /*les autres lecteurs bloqués dans
    f(mutex)*/
    nl := nl + 1 ;
    Si nl = 1 Alors P(W) Fsi;
    /*le 1er lecteur bloqué derrière
    f(W)*/
    V(mutexL) ;
    V(SemReserver);
    <Lecture>
    P(mutexL) ;
    nl := nl - 1 ;
```

```
Processus Rédacteur
Début
  Répéter
    P(mutexR) ;
    nr := nr + 1 ;
    Si nr = 1 Alors
    P(SemReserver) Fsi;
    V(mutexR) ;
    P(W) ;
    <Ecriture> ;
    V(W) ;
    P(mutexR);
    nr := nr – 1 ;
    Si nr = 0 Alors
    V(SemReserver) Fsi ;
    V(mutexR);
```

```
    Si nl = 0 Alors V(W) Fsi;
    V(mutexL) ;
  Jusqu'à Faux ;
Fin.
```

```
  Jusqu'à Faux;
Fin.
```

4 Communication par échange de messages

L'échange de messages (**Message Passing**) permet aux processus de communiquer sans faire recours au partage et gestion explicite d'une zone commune.

Le mécanisme de communication assure la synchronisation entre les processus. Le système fournit à l'utilisateur deux primitives :

- Envoyer(Message) : Send(m).
- Recevoir(Message) : Receive(m).

c. Caractéristiques de la communication par messages

A. Synchronisation

La communication par messages entre deux processus nécessite un certain niveau de synchronisation.

Un receveur (récepteur) ne peut recevoir un message que si ce dernier a été émis par un autre processus.

Nous devons de plus spécifier ce qui se passe lorsqu'un processus exécute SEND ou RECEIVE.

- **Primitive SEND**

  Lorsqu'un processus exécute une primitive SEND, deux cas de figure sont à considérer :

  - Soit ce processus reste bloqué jusqu'à ce que le message émis soit reçu ; **Send bloquant (Emission synchrone)**.
  - Soit ce processus continue son exécution ; **Send non bloquant (Emission asynchrone)**.

- **Primitive RECEIVE**

  Lorsqu'un processus exécute une primitive RECEIVE, deux cas de figure sont aussi à considérer :

  - Si un message a été émis auparavant, le message est reçu et le processus continue son exécution.
  - S'il n'y a pas de message en attente :

    i. Le processus est bloqué jusqu'à ce que le message arrive ; **Receive bloquant**

(Réception synchrone).

ii. Le processus continue son exécution abandonnant la tentative de réception ; **Receive non bloquant**.

Ainsi, les deux primitives peuvent être bloquantes ou non bloquantes. Trois combinaisons sont communément utilisées.

1er cas : SEND bloquant / RECEIVE bloquant

Les deux processus émetteur et récepteur sont bloqués jusqu'à la réception d'un message émis (**Ex.** Communication par Rendez-Vous, RDV).

2ème cas : SEND non bloquant / RECEIVE bloquant

L'émetteur peut continuer son exécution alors que le récepteur reste bloqué jusqu'à l'arrivée d'un message (**Ex.** processus serveur).

3ème cas : SEND non bloquant / RECEIVE non bloquant

Le SEND non bloquant est la forme la plus utilisée dans la programmation concurrente (**Ex.** Demande d'impression). Mais le **problème** est qu'une erreur peut nécessiter la regénération répétitive de messages ⇒ utilisation des **acquittements (Acknowledgements)**.

La forme la plus utilisée pour la primitive RECEIVE est le RECEIVE bloquant. Généralement un processus qui demande une information, il en a besoin pour continuer son exécution. Mais le **problème** est que si l'émetteur tombe en panne, on aura un blocage infini du récepteur.

La solution est d'utiliser un RECEIVE non bloquant. Mais le **problème** est que si un message est envoyé après que le récepteur ait exécuté le RECEIVE correspondant, le message sera **perdu**.

B. Adressage

- **Communication directe**

  Dans ce cas, le correspondant est désigné directement par son nom.

  - SEND(P, Message) ; envoyer « Message » au processus P.
  - RECEIVE(Q, Message) ; Recevoir « Message » du processus Q.

  Les **sockets** (**Sockets**) sont des implémentations de la communication directe entre processus.

  Exemple (Producteur/Consommateur)

Processus Producteur	**Processus Consommateur**
Var m : Message ;	**Var** m : Message ;
Répéter	**Répéter**
Produire(m) ;	RECEIVE(Producteur, m) ;

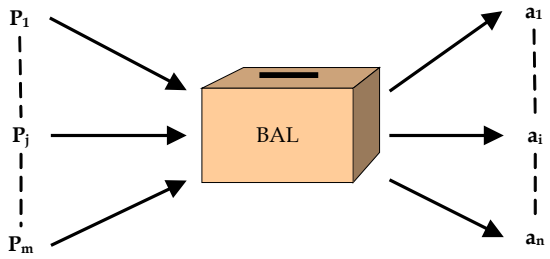
`SEND(Consommateur, m) ;` **`Jusqu'à`** `Faux.`	`Consommer(m) ;` **`Jusqu'à`** `Faux.`

- **Communication indirecte**

Dans ce cas, les messages ne sont pas envoyés directement d'un émetteur vers un récepteur, mais plutôt sont envoyés dans une structure de données partagée (une file d'attente) où sont stockés temporairement les messages.

Cette file d'attente est appelée **Boîte aux lettres** (**Mailbox**). Les **files de messages** (**Message Queues**) est une implémentation UNIX de ce concept de boîte aux lettres.

Ce type de communication permet d'avoir un découplage des processus émetteur et récepteur.



La relation entre les processus émetteurs et récepteurs peut être :

- Un à Un (**One to One**),
- Plusieurs à Un (**Many to One**),
- Un à plusieurs (**One to Many**),
- Plusieurs à Plusieurs (**Many to Many**).

Dans ce cas, les deux primitives auront la syntaxe suivante :

- SEND(B, Message) ; Envoyer « Message » à la boîte aux lettres B.
- RECEIVE(Message, B) ; Recevoir « Message » à partir de la boîte aux lettres B.

Une boîte aux lettres (BAL) peut être privée à un processus. Dans ce cas, il est le seul à l'utiliser en réception.

Une BAL peut aussi être partagée avec d'autres processus. Dans ce cas, si le message est destiné à un processus particulier, l'identité de celui-ci doit accompagner le message.

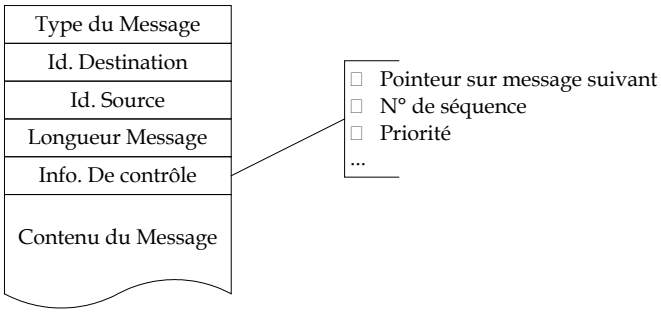
Le partage peut s'effectuer implicitement par héritage ; dès sa création, un processus partage sa BAL avec son créateur.

Un **problème** se pose dans le cas de plusieurs processus en attente de réception d'un message à partir de la même BAL. La **solution** à ce problème peut se faire soit par l'établissement d'une exclusion mutuelle sur l'exécution de la primitive RECEIVE,

soit par un choix aléatoire d'un processus.

C. Format d'un message

Le format d'un message dépend des objectifs du service de communication par messages. Avoir des messages avec un format fixe et court permet une simplicité de gestion.



Si les messages sont volumineux, le contenu est rangé dans un fichier et le message pointe seulement ce fichier.

D. Stratégie de gestion de la file

La stratégie de gestion de la file est généralement FIFO, mais elle peut s'avérer insuffisante si certains messages sont plus prioritaires que d'autres.

d. Exemple : Mise en œuvre de l'exclusion mutuelle

On suppose qu'on a un RECEIVE bloquant et un SEND non bloquant.

```
n : entier ; Init(n, nombre de processus) ;
```

```
Processus Pi
Var m : Message ;
Répéter
     Receive(mutex, m) ;
     <SCi> ;
     Send(mutex, m) ;
     <Reste de Pi>
Jusqu'à Faux.
```

```
P : /*Programme Principal*/
{
     Create_MailBox(mutex) ;
     Send(mutex, null) ;
     Parbegin P1, P2, ..., Pn Parend
}
```

Le programme principal P lance n processus concurrents après avoir créé une BAL mutex dans laquelle il a placé un message.

Le 1er processus Pi qui exécute Receive(mutex, m) entre en section critique, les suivants se bloquent tant que ce dernier est dans sa section critique. En sortant, Pi remet le message dans la BAL, ce qui permet à un autre processus bloqué sur Receive d'accéder à sa section critique.

CHAPITRE V : LES MONITEURS

1. Introduction

Les sémaphores sont des outils de synchronisation de bas niveau. Ils peuvent être utilisés pour réaliser de nouveaux outils de synchronisation.

Les sémaphores peuvent être utilisés pour exprimer différentes solutions de synchronisation de manière flexible. Cependant, certains inconvénients tels que l'interblocage ou le blocage indéfini d'un processus peuvent apparaître s'ils sont incorrectement utilisés.

Pour éviter ce problème, la conception d'outils de synchronisation de haut niveau tels que les moniteurs s'impose.

Cette solution est plus simple que les sémaphores, puisque le programmeur n'a pas à se préoccuper de contrôler les accès aux sections critiques. Mais, malheureusement, à l'exception de JAVA, la majorité des compilateurs utilisés actuellement ne supportent pas les moniteurs.

2. Les moniteurs

A. Définition

Un moniteur [HOARE 74, BRINCH Hansen 75] est un outil de synchronisation entre processus.

Il est constitué principalement :

- d'un ensemble de **variables**, dites de **synchronisation**, et
- des **procédures** qui manipulent ces variables. Certaines de ces procédures sont **internes** au moniteur et d'autres sont **externes**, donc accessibles par le processus.

Les ressources sur lesquelles se synchronisent les processus sont manipulées par les procédures externes appelées **points d'entrée (Entry Routines)** du moniteur.

Les variables de synchronisation ne sont utilisées par les processus que via ces points d'entrée.

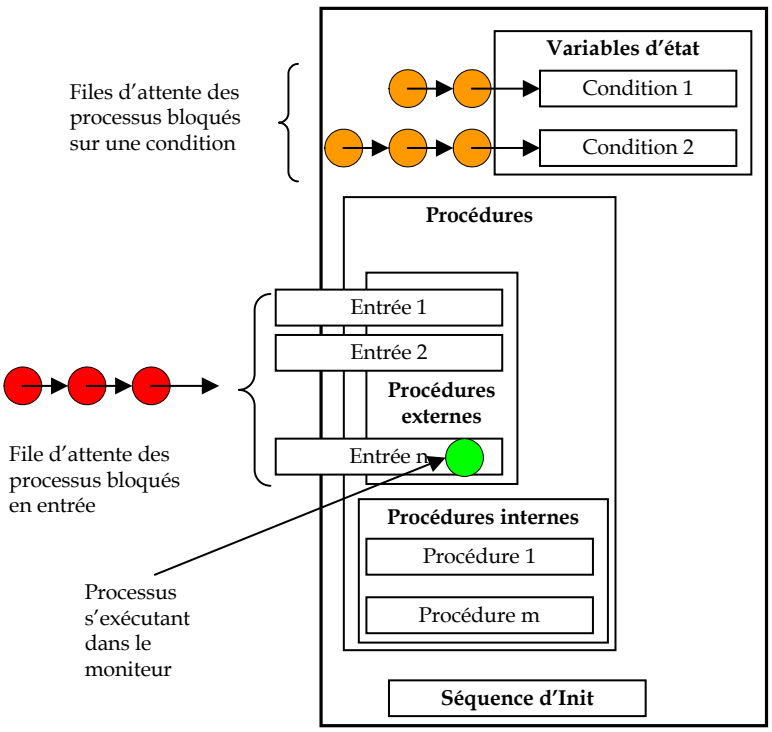
Le mécanisme de synchronisation garantit que ces variables sont utilisées en **exclusion mutuelle**.

Une **séquence d'initialisation** du moniteur permet d'affecter des valeurs initiales aux variables de synchronisation.

Le blocage et le réveil des processus s'effectuent à l'aide de trois primitives ; à savoir **SIGNAL**, **WAIT** et **EMPTY**.

Cette synchronisation s'exprime au moyen de conditions **(Condition Variables, Event Queues)**. Une **condition** est déclarée comme une variable sans valeur.

Les processus en attente définissent des files, une par condition. Ils attendent sans occuper le moniteur. Un seul processus est admis à la fois à l'intérieur du moniteur.



B. Les primitives

Primitive WAIT

Elle s'écrit **c.wait** où c est une condition.

Elle **bloque** le processus appelant en dehors du moniteur derrière la condition c.

Primitive SIGNAL

Elle s'écrit **c.signal** où c est une condition.

Elle réveille un processus bloqué derrière la condition c de manière FIFO, s'il y a lieu sinon elle est sans effet.

Primitive EMPTY

C'est une fonction booléenne, elle s'écrit **c.empty**.

Elle retourne la valeur « vrai» s'il y a au moins un processus bloqué derrière la condition c.

C. Illustrations

a. **Exemple 1** : Gestion d'une classe de ressource à N points d'entrées

Chaque processus demande une seule ressource à la fois.

Un processus P désirant accéder à une ressource aura la forme suivante :

```
Processus P
Début
     ...
     <Demande>
     ...
     <Libérer>
     ...
Fin
```

Le moniteur assurant la synchronisation entre les processus sera comme suit :

```
Allocation : moniteur ;
Var Nb : entier ; c : condition ;
Entry Procedure Demande
Begin
     Si Nb = 0 Alors c.wait Fsi ;
     Nb--
End
Entry Procedure Libérer
Begin
     Nb := Nb + 1 ;
     c.signal
End
/*Initialisation*/
Begin
     Nb := N ;
End
```

L'appel se fait ainsi :

Alloc.demande()

Alloc.liberer()

b. Exemple 2 : Producteur/consommateur

Solution 1

```
Prod_Cons : moniteur ;
Const N = ;
Var
```

```
     Tampon : tableau[0..N-1] de Tart ;
     t, q, cpt : entier ;
     prod, cons : condition ;
Entry Procedure Déposer(x : Tart) ;
Begin
     Si (cpt = N) Alors prod.wait Fsi ;
     Tampon[q] := x ;
     q := (q + 1) mod N ;
     cpt := cpt + 1 ;
     cons.signal
End
Entry Procedure prélever(Var x :Tart) ;
Begin
     Si (cpt = 0) Alors cons.wait Fsi ;
     x := tampon[t] ;
     t := (t+1) mod N ;
     cpt := cpt - 1 ;
     prod.signal
End
/*Initialisation*/
Begin
cpt := 0 ;  t :=0 ; q :=0
End
```

Problème

L'accès au tampon se fait en exclusion mutuelle entre les processus parce qu'il est déclaré à l'intérieur du moniteur.

Solution 2

Processus Producteur	**Processus Consommateur**
Var art : Tart ; **Répéter** … **Prod_cons.Dem_Dep ;** Déposer(art) ; **Prod_cons.Lib_Dep ;** … **Jusqu'à** Faux	**Var** art : Tart ; **Répéter** … **Prod_cons.Dem_Prel ;** Prélever(art) ; **Prod_cons.Lib_Prel ;** … **Jusqu'à Faux**

```
Prod_Cons : moniteur ;
Const N = ;
Var
```

```
     cpt : entier ;
     prod,cons : condition ;

Entry Procedure Dem_Dep ;
Begin
     Si cpt = N Alors prod Fsi ;
End
Entry procedure Lib_Dep ;
Begin
     cpt := cpt +1 ;
     cons.signal
End
Entry Procedure Dem_Prel ;
Begin
     Si cpt = 0 Alors cons.wait Fsi ;
End
Entry Procedure Lib_Prel ;
Begin
     cpt := cpt - 1 ;
     prod.signal
End
/*Initialisation*/
Begin
     cpt := 0
End
```

L'accès simultané au tampon est permis ; car le tampon n'est plus un objet du moniteur.

Exemple 3 : Lecteurs / Rédacteurs sans priorité

```
Lect_Red : moniteur ;
Var
Ecriture : booléen ; nl : entier ;
Lec, Ecr : condition ;
Entry Procedure Deb_Lecture ;
Begin
     Si Ecriture Alors Lec.wait Fsi ;
     nl := nl + 1 ; /*Réveiller les processus en cascade*/
     /* Il faut compter le nombre de lecteurs pour savoir quand on
     libère le fichier */
     Lec.signal
End
Entry Procedure Fin_Lecture ;
```

```
Begin
     nl := nl - 1 ;
     Si nl = 0 Alors Ecr.signal Fsi
End
Entry Procedure Deb_Ecriture ;
Begin
     Si (Ecriture ou nl ≠ 0) Alors Ecr.wait Fsi ;
     Ecriture := vrai
End
Entry Procedure Fin_Ecriture ;
Begin
     Ecriture := faux ;
     Si non Lec.empty
       Alors Lec.signal
       Sinon Ecr.signal
     Fsi
End
/*Initialisation*/
Begin
     Ecriture := faux ;
     nl := 0
End
```

D . Variantes de moniteurs

Les variantes de moniteur sont liées essentiellement à la sémantique de la primitive Signal.

Dans la définition classique, la synchronisation se situe sur deux points ; l'un est à l'exécution de signal, l'autre à l'exécution de wait.

Ce qui fait que la condition de franchissement n'est pas décrite explicitement, elle n'est représentée que symboliquement.

1. Condition de Kessels

Une variante de moniteur proposé par [Kessels 77] consiste en la redéfinition de la primitive wait, en lui associant une condition booléenne.

Dans ce cas, la primitive wait s'écrit : Wait(cond) où cond est une expression booléenne qui fait intervenir les variables de synchronisation du moniteur.

Un processus se bloque sur wait(cond) si cond n'est pas vérifiée.

A la sortie d'un processus du moniteur ou à son blocage par wait, toutes les conditions figurant dans les primitives sont **réévaluées automatiquement**.

Si au moins une condition est vérifiée, un des processus bloqués est activé. Ce qui fait que la primitive **signal** devient **inutile.**

Exemple (Producteur / Consommateur)

```
Procedure Dem_Dep
Begin
     Wait(cpt < N) ; /*se bloque si pas de cases vides*/
End
Procedure Lib_Dep
Begin
     cpt := cpt + 1;
End
Procedure Dem_Prel
Begin
     Wait(cpt > 0) ;
End
Procedure Lib_Prel
Begin
     cpt := cpt – 1 ;
End
```

2. Condition avec priorité

Certains problèmes nécessitent d'introduire un ordre de réveil qui n'est pas forcement l'ordre chronologique.

Pour répondre à ce besoin, une autre variante de moniteurs avec priorité est proposée.

Un **paramètre entier** est spécifié avec la primitive wait ; **c.wait(i)**. Ce paramètre **i** permet d'indiquer une certaine condition correspondante avec une valeur minimale de priorité.

Les processus sont donc rangés dans l'ordre croissant de priorité.

Exemple 1

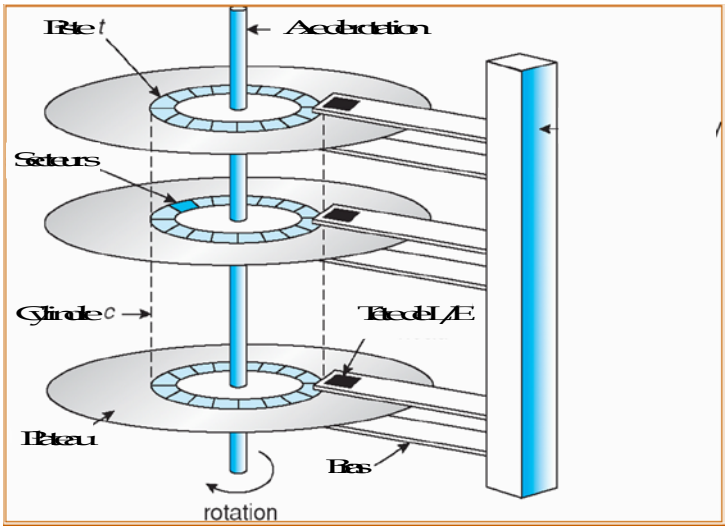
Gestion d'une ressource à N instances, dont l'allocation est faite en priorité à celui qui demande le moins d'instantes de cette ressource.

```
Pi
     Répéter
            ...
            Demande(k) ;
            <Utilisation des k instantes> ;
            Libèrer(k) ;
            ...
     Jusqu'à Faux
```

```
Ressource_k : moniteur ;
Var
     Ress : condition ;
     Dispo : entier ;
     File : file d'attente d'attribut k ;
Entry Procedure Demande(k)
Begin
     Si (Dispo < k)
       Alors
            Begin
                 Insérer(File, k) ;
                 Ress.Wait(k) ;
            End
     Fsi ;
     Dispo := Dispo - k
End
Entry Procedure Libèrer(m)
Begin
     Dispo := Dispo + m;
     Tant que (Dispo ≥ Tête(File).k)
       Faire
            Défiler(Tête(File)) ;
            Ress.Signal
       Fait
End
/*Initialisation*/
Begin
     Dispo := N ;
End
```

Pour chaque processus qui va attendre, on associe la valeur du nombre d'exemplaires demandés comme paramètre d'attente. Ainsi, le 1er servi sera celui dont la valeur est la plus petite.

Exemple 2 (Gestion du bras d'un disque)



Temps d'accès = Temps de positionnement + Temps de rotation + Temps de transfert.

Le disque est constitué de **Max** cylindres numérotés de **0** à **Max-1** et sert toutes les requêtes dans l'ordre de leur rencontre par la tête de lecture/écriture jusqu'à la dernière requête, puis change de sens (direction) et sert, de même, les requêtes jusqu'à la dernière rencontrée puis change de sens (**Politique de l'ascenseur**, **SCAN**).

On suppose que les requêtes sont générées dynamiquement.

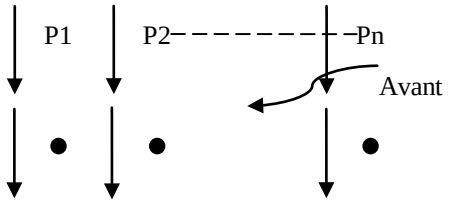
```
Disque : moniteur ;
Const Max = ... ;
Var
     Position : (0..Max-1) ;
     Direction : (Haut, Bas) ;
     Occupé : Booléen ;
     CHaut, CBas : Condition ;
Entry Procedure Demander(k : entier)
Begin









Entry Procedure Libèrer
Begin

```

```
/*Initialisation*/
Begin
     Occupé := faux ;
     Position := 0 ;
     Direction := Haut
End
```

Remarque : On peut avoir un moniteur avec une seule procédure entry

Exercice 1 (Rendez-vous Multiple)

Soient N processus P1, P2, …, Pn tous identiques. On désire que ces processus se synchronisent (s'attendent) à un point de Rendez-vous donné. Dès que tous les n processus atteignent leur point de rendez-vous commun, ils peuvent continuer leur exécution.



Point de Rendez-vous

Ecrire le moniteur qui assure la synchronisation des n processus.

```
Rendez-vous : moniteur ;
```

```
End
```

Exercice 2 (Lecteurs / Rédacteurs) Ecrire un moniteur qui implémente le modèle de communication Lecteurs / Rédacteurs en donnant la priorité aux rédacteurs.

Exercice 3 (P et V)

Ecrire un moniteur qui implémente les primitives P et V.

E. Implémentation des moniteurs au moyen des sémaphores

CHAPITRE VI : INTERBLOCAGE

5 Introduction

Dans un système informatique, l'exécution d'un processus nécessite l'utilisation d'un ensemble de ressources (**Ex.** mémoire centrale, disques, fichiers, périphériques, etc.) qui lui

sont attribuées par le système d'exploitation. L'accès aux ressources est régi par trois opérations :

- **Demande**

  Elle consiste à exprimer un besoin de certaines ressources spécifiées dans la demande.

  Elle précède toute utilisation. Si une demande ne peut pas être satisfaite, le processus est mis en attente.

- **Acquisition**

  C'est l'allocation effective de la ressource au processus. Après acquisition, le processus peut utiliser la ressource.

- **Libération**

  Les ressources préalablement acquises sont rendues disponibles.

Le nombre de ressources est limité et certaines d'entre elles ne sont pas partageables, ceci conduit à des attentes d'un ensemble de processus. Si les ressources attendues sont détenues, par des processus en état d'attente, une situation d'attente infinie peut surgir créant ainsi. Une **situation d'interblocage(deadlock)**.

6 Définition d'un interblocage

De manière générale, un ensemble de processus est dans un **état d'interblocage** (**Deadlock State**), si chacun des processus attend un événement qui ne peut être déclenché que par un autre processus du même ensemble. (L'**événement** qui nous intéresse dans ce contexte est la **libération de ressources)**.

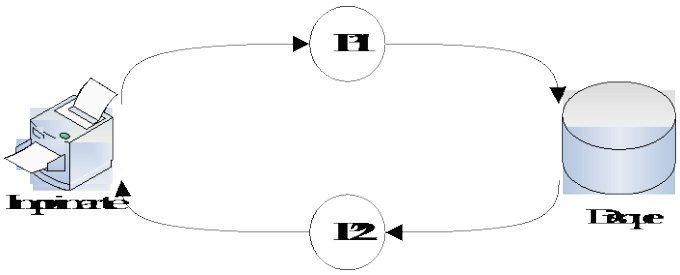
L'**interblocage** (**Deadlock**) est un état qui est dit **stable** ou **propriété stable** (**Stable Property**). Une propriété est dite stable, si une fois vérifiée elle reste toujours vraie dans les états ultérieurs du système.

7 Exemples d'interblocage

Exemple 1 (Système doté d'un disque et d'une imprimante)

Deux processus désirent imprimer un fichier. Chaque processus a besoin d'un accès exclusif au disque et à l'imprimante simultanément. On a une situation d'interblocage si :

- Le processus P1 utilise l'imprimante et demande l'accès au disque.
- Le processus P2 détient le disque et demande l'imprimante.



Exemple 2 (Accès à une base de données)

Supposons deux processus P1 et P2 qui demandent des accès exclusifs aux enregistrements d'une base de données. On arrive à une situation d'interblocage si :

- Le processus P1 a verrouillé l'enregistrement R1 et demande l'accès à l'enregistrement R2.
- Le processus P2 a verrouillé l'enregistrement R2 et demande l'accès à l'enregistrement R1.

8 Conditions d'interblocage

L'état d'interblocage est caractérisé par les **quatre** (**04**) **conditions** suivantes qui sont **nécessaires et suffisantes** [Coffman 1971]:

a. **Exclusion mutuelle** (**Mutual Exclusion**). Il existe au moins une ressource non partageable.

b. **Prendre et attendre** (**Hold and Wait**). Il existe des processus qui détiennent des ressources et demandent d'autres acquises par d'autres processus.

c. **Pas de préemption** (**No Preemption**). Des ressources acquises par un processus ne doivent pas être réquisitionnées. La libération de ressources ne peut se faire que volontairement.

d. **Attente circulaire** (**Circular Wait**). Elle consiste à avoir un ensemble de processus en attente $\{P_0, P_1, \ldots, P_{n-1}\}$, tels que P_0 attend P_1 qui attend P_2 qui ..... attend P_{n-1} qui attend P_0.

9 Représentation graphique

On peut exprimer un interblocage plus précisément en terme de théorie de graphes orienté. Un système est composé de :

Un ensemble fini de **processus** séquentiels $\mathbf{P = \{P_0, P_1, \ldots, P_{n-1}\}}$ pouvant s'exécuter de manière concurrente.

Un ensemble fini de **classes de ressources** $\mathbf{R = \{R_0, R_1, \ldots, R_{m-1}\}}$ avec un nombre de points d'accès chacune. Une classe de ressources désigne un ensemble de ressources identiques.

L'état initial du système est décrit par le vecteur « **Dispo** » indiquant le nombre d'instances de chaque ressource.

$Dispo = \begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_{m-1} \end{bmatrix}$, qui indique le nombre de ressources disponibles pour chaque classe.

$$Dem = \begin{bmatrix} d_{00} & \cdots & \cdots & d_{0m-1} \\ d_{10} & \cdots & \cdots & d_{1m-1} \\ \vdots & \cdots & \cdots & \vdots \\ d_{n-10} & \cdots & \cdots & d_{n-1m-1} \end{bmatrix} = \begin{bmatrix} D_0 \\ D_1 \\ \vdots \\ D_{n-1} \end{bmatrix},$$

Où d_{ij} est le nombre de ressources de la classe **Ri demandée** par le processus **Pj**.

$$Alloc = \begin{bmatrix} a_{00} & \cdots & \cdots & a_{0m-1} \\ a_{10} & \cdots & \cdots & a_{1m-1} \\ \vdots & \cdots & \cdots & \vdots \\ a_{n-10} & \cdots & \cdots & a_{n-1m-1} \end{bmatrix} = \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{n-1} \end{bmatrix},$$

Où a_{ij} est le nombre de ressources de la classe **Ri allouées** au processus **Pj**.

Le processus passe d'un état à un autre par l'intermédiaire de l'une des trois (03) opérations suivantes : Demander, Allouer et Libérer.

L'état du système peut être représenté par deux types de graphe orienté.

I. **Graphe d'allocation de ressources** (**Resource Allocation Graph**)

Noté **G = (V, E)**, où :

V est l'ensemble de **nœuds** (**Vertices**). Deux types de nœuds peuvent être distingués :

Les **processus**, illustrés par des **cercles** (**Circles**).

Les **ressources**, représentées par des **carrés** (**Boxes**). Les instances d'une classe de ressources sont représentées par des **points** (**Dots**).

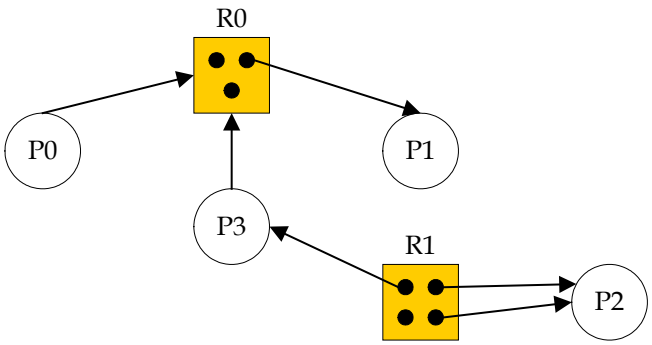
V = P + R

E est l'ensemble des **arcs** (**Edges**). Deux types d'arcs peuvent être distingués :

- **Arc requête** (**Request Edge**), dirigé de Pi vers Rj (**Pi → Rj**), indique que le processus Pi demande un exemplaire de la classe Rj.
- **Arc affectation** (**Assignment Edge**), dirigé de Rj vers Pi (**Rj → Pi**), indique qu'un exemplaire de la ressource Rj est alloué au processus Pi.

Quand Pi exprime une demande, autant d'arcs **Pi → Rj** que d'instances demandées de Rj sont créés.

Exemple



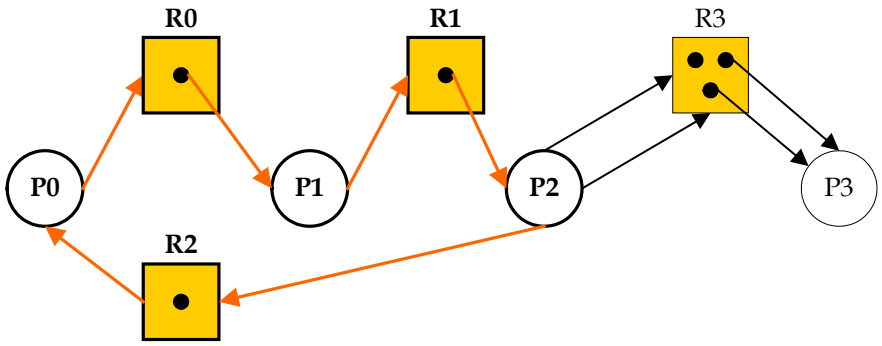
L'allocation **transforme** les arcs **Pi → Rj** en **Rj → Pi.**

La libération **supprime** les arcs issus de Pi concernés par la ressource en question.

Remarque

- Si le graphe ne contient **aucun cycle⇒ pas** de situation d'**interblocage**.
- Si le graphe contient un **cycle** :
  - Si chaque ressource existe en un **seul exemplaire**, alors un cycle indique forcément un **interblocage**.

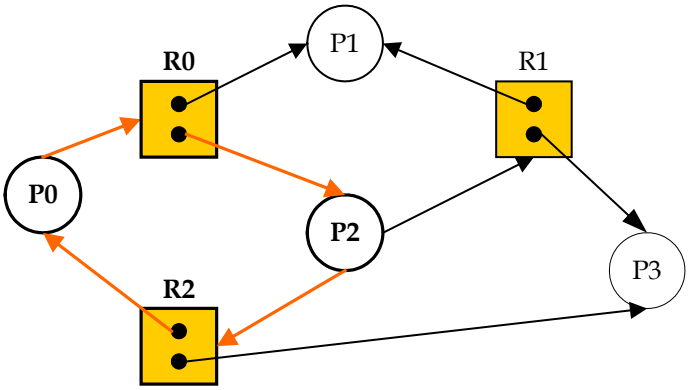
    Exemple



    P0 → R0 → P1 → R1 → P2 → R2 → P0 ⇒ Situation d'interblocage

  - Si chaque ressource existe en **plusieurs exemplaires**, alors un cycle n'indique pas forcément une situation d'interblocage. L'existence d'un cycle est dans ce cas une **condition nécessaire** mais **pas suffisante**.

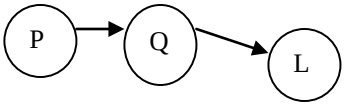
    Exemple



P0 → R0 → P2 → R2 → P0, il y a un cycle mais pas une situation d'interblocage.

II. Graphes des attentes

On exprime seulement les attentes dans ce type de graphe.



10 Traitement d'interblocage

Le traitement d'interblocage peut être effectué selon trois stratégies différentes :

1. Utiliser un protocole pour assurer que le système ne se trouve jamais dans une situation d'interblocage.

2. Laisser les processus évolués en prévoyant un protocole qui détecte l'interblocage puis le corrige(Détection guérison).

   Dans le cas positif, des techniques doivent être prévues pour lever une situation d'interblocage (On parle de **Guérison**).

3. Utiliser la politique de l'autruche qui consiste à ignorer l'existence des interblocages et donc de ne rien prévoir pour les traiter. En général, ce problème est ignoré par les systèmes d'exploitation car le prix à payer pour les éviter ou les traiter est trop élevé pour des situations qui se produisent rarement. Simplement la machine est redémarrée lorsque trop de processus sont en interblocage.

   L'interblocage ne se produira alors jamais quelle que soit la politique d'allocation de ressource adoptée.

- **L'évitement** est une opération continue (dynamique). Elle consiste à examiner l'état du système avant chaque allocation pour empêcher une allocation qui peut conduire à une situation d'interblocage.

10.1 Prévention

La prévention consiste à imposer la manière d'effectuer les requêtes des ressources, ceci revient à supprimer définitivement (de manière statique) une des conditions pouvant conduire à un interblocage.

- **Comment peut on éliminer la condition a (Exclusion mutuelle) ?**

  Eliminer la condition (a) revient à imposer que toutes les ressources soient partageable ou avoir autant de ressources que de demande. Ce qui est irréel. Sauf pour les impressions exiger des processus de « **spooler** » leurs travaux dans un répertoire spécialisé et un démon d'impression les traitera, en série, l'un après l'autre.

  Méthode non généralisable à tout type de ressources.

- **Comment peut on éliminer la condition b (Prendre et attendre) ?**

  Eliminer (b) revient à éviter qu'un processus qui détient certaines ressources d'en demander d'autres. Ceci peut se faire de deux manières :

  1/ Allouer à l'avance toutes les ressources nécessaires (avant l'exécution et non pas au fur et à mesure). Ceci ralentit considérablement les performances du système, en plus de la mauvaise exploitation des ressources.

  2/ On alloue au processus les ressources qu'il demande dynamiquement mais dés qu'il doit se mettre en attente, il doit libérer toutes les ressources dont il dispose. Cette solution peut conduire à un problème de famine.

- **Comment peut on éliminer la condition c (Pas de préemption) ?**

  Elle consiste à **réquisitionner** certaines ressources déjà allouées. Une méthode consiste à enlever toutes les ressources allouées à un processus s'il effectue une demande qui ne peut pas être satisfaite immédiatement.

  Toutes les ressources ne peuvent être réquisitionnées sans dégradation des performances du système. Cependant, on peut l'envisager pour certaines ressources dont le contexte peut être sauvegardé et restauré(registre…)

- **Comment peut on éliminer la condition d (Attente circulaire) ?**

  Eliminer (d) revient à imposer une **allocation hiérarchique** des ressources(méthodes des classes ordonnées) .

  A chaque classe de ressource , on associe un entier unique afin d'ordonner les classes.

  F : R= { R1,R2,….Rk} → N

  Exemple : F(disk) = 5 ; F(bande) = 7, F(imprimante) = 12

  Chaque processus exprime ses demandes selon la règle ci-après :

  - Un processus ne peut demander une ressource de la classe Ri que si toutes les

ressources qu'il détient sont inférieures à Ri.

- Les ressources de même niveau doivent être allouées en même temps (pas d'allocation partielle).
- Les ressources de niveau k doivent être libérées avant d'acquérir les ressources de niveau < k.

Exemple

Soient F (R1) = 2, F (R2) = 5, et F (R3) = 8. Un processus Pi doit demander les ressources dans l'ordre 2, 5, 8.

10.2 Evitement

La méthode d'évitement n'impose aucune contrainte d'allocation, ni globale, ni hiérarchique mais contrôle dynamiquement pas à pas la progression des processus de façon à garantir qu'aucun blocage ne se produira. Pour cela, le protocole correspondant étudie toute demande d'allocation en simulant l'allocation si les ressources sont disponibles. Puis teste si cette simulation ne fait pas aboutir le système à un interblocage, l'allocation sera effective (réelle). Le système est considéré dans un état **fiable**

Dans le cas contraire (état non fiable), le processus sera mis en attente (malgré la disponibilité des ressources). Pour cela le système a besoin des informations concernant les ressources disponibles, celle allouées à chaque processus et la demande maximale de chaque processus.

A. Etat fiable (fiable)

Définition

Un **état** est **fiable** (dit aussi **sûr**), si le système peut allouer des ressources à chaque processus jusqu'à son maximum.

Plus formellement, un système est dans un état fiable seulement si tous les processus peuvent terminer leur exécution ; c.-à-d., il existe une **séquence fiable** complète.

Une séquence <P1, P2, …, Pn> est fiable pour l'état d'allocation courant si pour chaque Pi, les requêtes de Pi (besoin futur) peuvent être satisfaites par les **ressources disponibles** + les ressources libérés par tous les processus qui le précède dans la séquence.

Exemple

Considérons un système avec 12 lecteurs de cassettes et 3 processus, tels que :

- P0 a besoin de 10 lecteurs,
- P1 a besoin de 4 lecteurs,
- P2 a besoin de 9 lecteurs.

A l'instant t0 :

- P0 détient 5 lecteurs,
- P1 détient 2 lecteurs,
- P2 détient 2 lecteurs.

A l'instant t0 le système est dans un état fiable ; car il y a une séquence fiable <P1, P0, P2> :

- P1 peut disposer immédiatement de tous les lecteurs dont il a besoin. A la fin de P1, 5

lecteurs de plus sont disponibles.

- P0 peut disposer immédiatement de tous les lecteurs dont il a besoin. A la fin de P0, 10 lecteurs sont disponibles.
- P2 peut disposer maintenant des 7 lecteurs dont il a besoin. A la fin de P2, 5 lecteurs de plus sont disponibles.

Notons qu'il est possible de **passer** d'un **état fiable** à un état **non fiable**.

Exemple

A l'instant t1, si P2 demande un exemplaire de plus ; l'état simulé serait

- P0 détient 5 lecteurs,
- P1 détient 2 lecteurs,
- P2 détient 3 lecteurs.

Donc, il reste 2 lecteurs disponibles.

P1 aura tous les exemplaires et se termine ⇒ 4 exemplaires disponibles.

P0 détient 5 exemplaires et il a besoin de 5 autres exemplaires ⇒ P0 ne peut pas évoluer.

P2 détient 3 exemplaires et il a besoin de 6 autres exemplaires ⇒ P2 ne peut pas évoluer.

Par conséquent, on aura un **risque** d'interblocage (état non fiable)

Conséquence

Il faut mettre P2 en attente pour éviter le risque d'interblocage.

B. Algorithme du banquier (Banker's Algorithm)

L'algorithme du banquier [Dijkstra 1965] se fait en deux phases. La première consiste à simuler l'allocation, la seconde teste l'état simulé s'il est fiable ou non.

Structures de données

n : Nombre de processus ;

m : Nombre de types de ressources ;

Dispo : Vecteur de longueur m ;

/*Dispo[j] indique le nombre d'instances disponibles de la ressource Rj.*/

Max : Matrice nxm ;

/*Max [i, j] définit le besoin maximal du processus Pi pour la ressource Rj.*/

Alloc : Matrice nxm ;

/*Alloc[i, j] définit le nombre d'instances de la ressource Rj présentement allouées au processus Pi.*/

Besoin : Matrice nxm ;

/*Besoin[i, j] indique le nombre d'instances restantes de la ressource Rj dont aurait besoin le processus Pi. **Besoin = Dem – Alloc**.*/

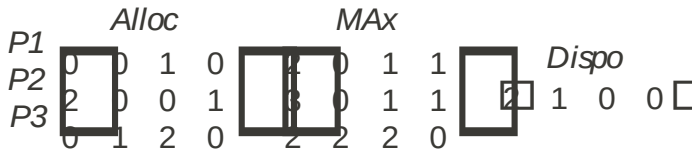
Evolue : vecteur de booléens initialisés à faux

Work : vecteur de longueur m

Exemple

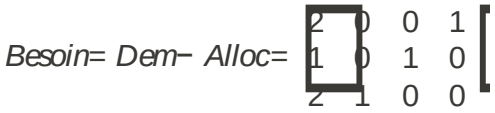
Soit un système composé de trois (03) processus {P1, P2, P3} et quatre (04) types de ressource {R1, R2, R3, R4} qui existent respectivement en 4, 2, 3 et 1 exemplaires.

L'état du système à l'instant t0 :



1. L'état courant est-il fiable ?

   Pour que l'état du système soit fiable, il faut trouver une suite ou séquence complète "qui se termine" (c'est-à-dire contient tous les processus). On calcule d'abord la matrice *Besoin* :



2. A partir de cet état, peut-on accorder une ressource de R1 à P2 ?

3. A partir de cet état, peut-on accorder deux ressources de R1 à P3 ?

C. THEOREMES

1. Si E est dans un état fiable avec Pi le 1er élément de la séquence fiable complète alors si pi exprime une requête d'allocation et les ressources sont disponibles, le processus Pi sera servi immédiatement (sans simulation car l'état sera forcement fiable).

2. Soit E1 un état fiable, si Pi exprime une requête, l'état simulé devient E2. E2 est fiable s'il existe une séquence fiable contenant Pi (pas forcement complète).

10.3 Détection – guérison

Si un système ne fait appel ni à la prévention, ni à l'évitement d'interblocage, une situation d'interblocage peut survenir.

Dans ce cas, le système doit fournir :

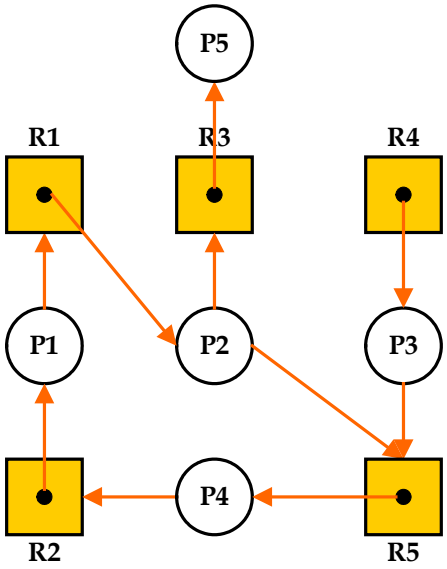
- Un algorithme qui permet de savoir si le système est dans un état d'interblocage. C'est la phase de **détection**.

- Un algorithme de recouvrement de l'interblocage. C'est la phase de **guérison**.

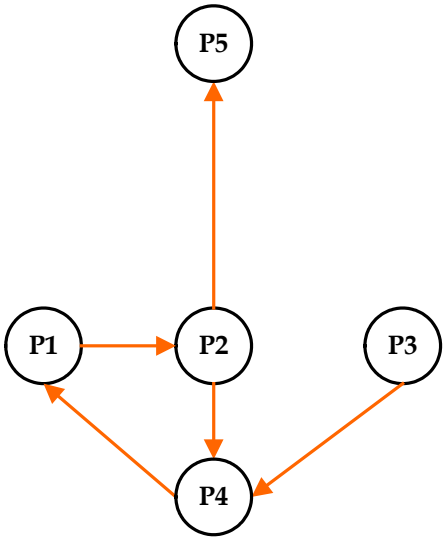
A. Recouvrement de ressource à une seule instance (Ressource Recovery)

Dans ce cas, la détection peut être basée sur une variante du graphe d'allocation de ressources, appelé **graphe des attentes**.

Il y a un interblocage si et seulement si un circuit (cycle) existe dans le graphe des attentes.



Graphe d'allocation de ressources



Graphe des attentes

B. Ressources à plusieurs instances

Le graphe des attentes n'est pas applicable dans ce cas vu que la détection d'un circuit ne conduit pas nécessairement à un interblocage.

La détection peut se faire de deux manières :

1. par réduction du graphe d'allocation
2. en utilisant un algorithme qui détermine si l'état du système est **sain**

1. REDUCTION DU GRAPHE D'ALLOCATION

Pour réduire le graphe d'allocation on utilise les trois règles suivantes :

a. Si une ressource possède seulement des flèches sortantes (allocation) ces flèches seront supprimées.

b. Si un processus possède seulement des flèches rentrantes (allocation), on les supprime

c. Si une ressource a des flèches rentrantes, si pour chaque flèche de requête il y a un exemplaire disponible, elles seront supprimées.

2. LE PROTOCOLE DE DETECTION

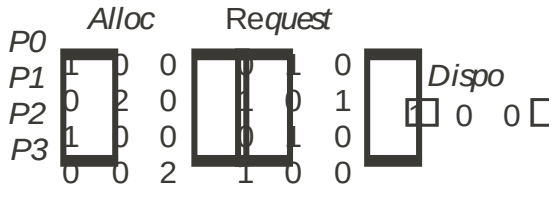
Ce protocole utilise plusieurs structures de données qui varient avec le temps : Dispo, Alloc et Requête où Requête est une matrice n*m tel que Requête [i,j] représente les requêtes non satisfaites du processus Pi % la ressource Rj.

Cet algorithme essaye de déterminer s'il existe une séquence saine complète, pour cela, il utilise le principe de marquage. A chaque fois qu'un processus peut évolué, il le marque. Un processus avec un Alloc ou Requête à 0 est marqué automatiquement vu qu'il ne peut pas être impliqué dans un interblocage

Exemple

Considérons un système où nous avons quatre processus {P0, P1, P2, P3} en cours et qui dispose de trois types de ressources {R0, R1, R2} qui existent en 3, 2 et 2 exemplaires respectivement.

Supposons qu'à l'instant t0 nous avons l'état d'allocation suivant :



C. Guérison

Quand le système est dans un état d'interblocage, le processus impliqué dans cette situation reste bloqué jusqu'à l'intervention sur le mécanisme normal d'allocation de ressources.

Il existe trois (03) façons différentes pour lever cette situation :

- La première consiste à retirer temporairement les ressources d'un processus bloqué pour les attribuer à un autre.
- La deuxième consiste à restaurer un état antérieur (retour en arrière) et éviter de retomber dans la même situation.
- La troisième consiste à détruire un ou plusieurs processus afin de briser le cycle. Il est donc nécessaire de sélectionner une victime.

Sélection d'une victime

La victime peut être le processus qui a exprimé une demande ayant provoqué l'interblocage.

La suppression d’un processus est une solution extrême.

Un processus peut être la victime de manière répétée. Ce processus ne peut donc pas terminer son exécution. Pour éviter ce problème, on utilise un compteur de sélection qui peut être utilisé comme fact