COURS

SYSTEMES CONCURRENTS ET DISTRIBUES

CHAFIKA BENZAID

LSI-Département Informatique,
Faculté d'Electronique & Informatique, USTHB
Email: benzaid@hotmail.com

TABLE DE MATIERES

PARTIE I

SYSTEMES CONCURRENTS

CHAPITRE I : 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='abcd' ;      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 ('abcdABCD' 'aAbBcCdD', 'abcABCDd', …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)  Load  R1    PLACE_DISPO
(B2)  Cjump R1    Plein  % Saut à Plein si R1 = 0%
(B3)  Sub   R1    1
(B4)  Store R1    PLACE_DISPO
(B5)  Repondre    (‘Place réservée’)
(B6)  Jump        Fin
(B7)  Plein 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 COMPTEUR.

```
P1 :                                    P2 :
1  Load  R1   COMPTEUR                  1'   Load  R1   COMPTEUR
2  Add   R1   1                         2'   Sub   R1   1
3  Store R1   COMPTEUR                  3'   Store R1   COMPTEUR
```

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

3 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**.

3.1 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

3.2 Conditions de la section critique

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

1. **Exclusion mutuelle ;** Si un processus Pi est dans sa section critique, aucun autre ne doit être dans sa section critique,

2. **Progression ;** Si un processus opère en dehors de sa section critique, il ne doit pas empêcher un autre d'entrer dans sa section critique, s'il le désire. En d'autres termes, il ne décidera pas lequel va entrer en section critique,

3. **Absence de blocage mutuel ;** Si deux ou plusieurs processus essayent d'entrer dans leurs sections critiques, au moins un réussira.

4. **Attente bornée** ; un processus ayant demandé d'entrer en section critique doit y accéder au bout d'un temps fini. Autrement dit, les processus autorisés à accéder en section critiquer doivent y entrer un nombre limité de fois, si un autre processus a demandé la section critique.

4 Mécanismes d'exclusion mutuelle

4.1 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.

4.2 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 ≠ i) faire <rien> fait ;
                  <SCi>
            tour := j
      Jusqu'à faux
```

L'exclusion mutuelle est assurée ; car l'entrée en section critique est conditionnée par la valeur de **tour** qui doit être d'une part identique à l'identité du processus et d'autre part tour n'est modifiée qu'à la sortie de la section critique.

PB : La solution engendre l'**alternance** quant à l'accès à la section critique.

Cette solution indique seulement lequel des processus est en section critique.

- **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] := v ;
            Tant que drapeau[j] faire <rien> fait ;
                  <SCi>
            drapeau[i] := f ;
            <Section restante>
      Jusqu'à faux
```

Quand le processus Pi désire entrer en section critique, il met drapeau[i] à vrai. Pi teste ensuite si l'autre processus ne désire pas entrer en section critique; si c'est le cas, il entre en section critique, sinon il boucle sur ce test tant que l'autre est en section critique.

L'exclusion mutuelle est assurée car chaque processus Pi avant de tester drapeau[j], il prend la précaution de mettre drapeau[i] à vrai et la mise à jour de drapeau[i] n'est réalisée qu'à la sortie de la section critique.

Un blocage mutuel est possible si les vitesses des deux processus sont égales et ces derniers arrivent en même temps.

- **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
```

Si un processus Pi désire entrer en section critique, il met **drapeau[i]** à **vrai** et **tour** à **j**. Pour entrer en section critique, Pi doit constater que **drapeau[j] = faux** (l'autre processus ne désire pas entrer en section critique) ou **tour ≠ j** (Pj arrive après Pi).

Si les deux processus arrivent en même temps, celui qui modifie tour en premier entre en section critique ; tour tranche entre les deux processus.

- L'**exclusion mutuelle** est **assurée**. Chaque processus Pi entre en section critique si drapeau[j]= faux ou tour = i. D'autre part, les deux processus sont en section critique en même temps si drapeau[i] = drapeau[j] = vrai. Ceci implique que les deux processus ne peuvent franchir en même temps la boucle car tour prend une seule valeur i ou j. Donc, un seul processus, par exemple Pi, franchit la boucle et l'autre (qui a exécuté en dernier tour := i) doit attendre jusqu'à ce que Pi mette drapeau[i] à faux à sa sortie de la section critique.

- La **progression** est **assurée** car si un processus Pi seul désire entrer en section critique (drapeau[j]=faux), il entre en section critique. Si les deux processus désirent entrer en section critique, tour tranche lequel va entrer, car la seule condition qui prévient la section critique est drapeau[j]=vrai et tour =j.

- L'**attente bornée** est **assurée.** Si un processus Pi est en attente de la section critique, sachant que Pj est dans la section critique ; ceci veut dire que tour =j et drapeau[j]=vrai. En sortant, Pj met drapeau[j]=faux. Après quoi, quelle que soit sa vitesse Pi va entrer en section critique en constatant que drapeau[j]=faux sinon si Pj est rapide et postule une autre fois, il met tour = i et ainsi Pi franchit la boucle avec succès.

4.3 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**).

4.4 Les sémaphores

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

- Elles ne peuvent être généralisées à un nombre **illimité** de processus,
- 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
Début
   S := n ;
   f(s) := ∅ ;
Fin
```

```
P (S)  /* P pour Proberen */
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) /* V pour Verhogen */
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

- L'exécution de P et V est **atomique**.
- 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**.

B. Sémaphores binaires

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
```

C. Sémaphores privés

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.

Exemple

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>
```

D. Sémaphores compteurs (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 (**S = -1**).

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%
```

E. Problèmes de la mise en œuvre

L'utilisation des sémaphores pose principalement deux problèmes ; à savoir l'**interblocage** et la **famine**.

- **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) $\leftrightarrow$ R1 = 0

P2 exécute P(R2) $\leftrightarrow$ R2 = 0

P1 exécute P(R2) $\leftrightarrow$ R2 = -1 $\Rightarrow$ P1 se bloque

P2 exécute P(R1) $\leftrightarrow$ R1 = -1 $\Rightarrow$ P2 se bloque

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

- **Famine (Starvation)**

  La famine a lieu si la file d'attente associée au sémaphore est gérée en LIFO (Last in First Out : Dernier entrée premier servi).

4.5 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 II : 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.

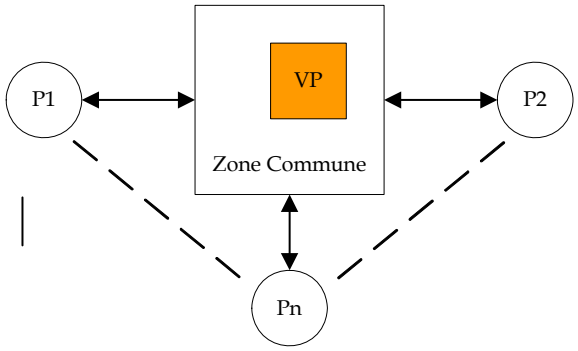
2 Types de communication

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.

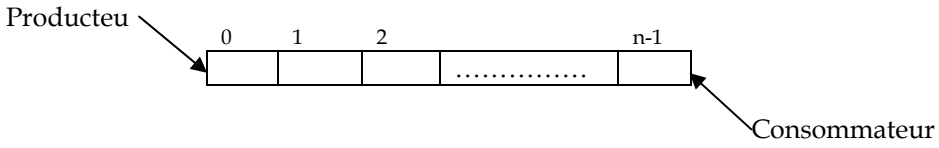


3.1 Modèle du Producteur/Consommateur

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
Var art : Tart
Début
   Répéter
      Produire(art) ;
      Tq (nb = n) Faire <Rien>
      Fait ;
      Déposer(art) ;
      nb := nb + 1 ;
      <Suite>
   Jusqu'à Faux ;
Fin.
```

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

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

```
    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.

3.2 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) ;
```

```
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 ;
```

```
    nl := nl - 1 ;
    Si nl = 0 Alors V(W) Fsi;
    V(mutexL) ;
  Jusqu'à Faux ;
Fin.
```

```
    V(mutexR);
  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).

4.1 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**

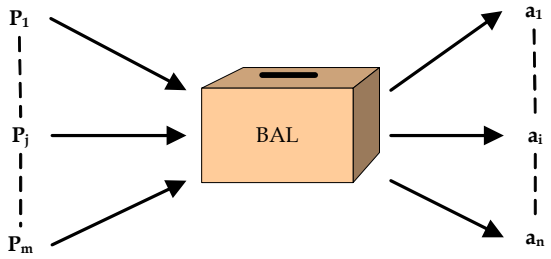
`Produire(m) ;`
`SEND(Consommateur, m) ;`
`Jusqu'à` `Faux.`
	`RECEIVE(Producteur, m) ;`
	`Consommer(m) ;`
	`Jusqu'à` `Faux.`

- **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.

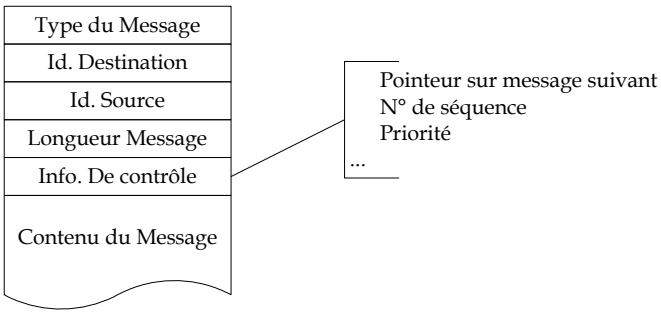
Le partage peut s'effectuer implicitement par héritage ; dès sa création, un processus partage sa BAL avec son créateur.

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.

4.2 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	**P : /*Programme Principal*/**
Var m : Message ;	**{**
Répéter	Create_MailBox(mutex) ;
Receive(mutex, m) ;	Send(mutex, null) ;
<SCi> ;	**Parbegin** P1, P2, ..., Pn **Parend**
Send(mutex, m) ;	**}**
<Reste de Pi>	
Jusqu'à Faux.	

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 III : 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

2.1 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**.

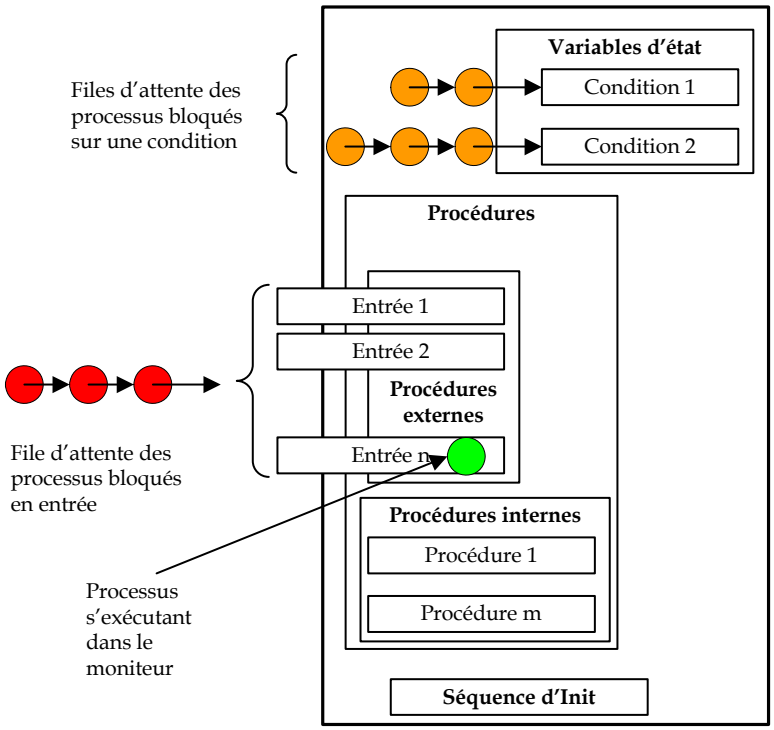
Une **séquence d'initialisation** du moniteur permet d'affecter des valeurs initiales aux variables de synchronisation.

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 en dehors du moniteur.

Un seul processus est admis à la fois à l'intérieur du moniteur.



2.2 Les primitives

A. 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.

B. 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.

C. Primitive EMPTY

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

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

3 Illustrations

3.1 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 N : entier ; c : condition ;
Procedure Demande
Begin
     Si N = 0 Alors c.wait Fsi ;
     N := N - 1
End
Procedure Libérer
Begin
     N := N + 1 ;
     c.signal
End
/*Initialisation*/
Begin
     N := k ;
End
```

3.2 Exemple 2 : Producteur/consommateur

A. Solution 1

```
Prod_Cons : moniteur ;
Const N = ;
Var
     Tampon : tableau[0..N-1] de Tart ;
     t, q, cpt : entier ;
```

```
     nonvide, nonplein : condition ;
Procedure Déposer(x : Tart) ;
Begin
     Si (cpt = N) Alors nonplein.wait Fsi ;
     Tampon[q] := x ;
     q := (q + 1) mod N ;
     cpt := cpt + 1 ;
     nonvide.signal
End
Procedure prélever(Var x :Tart) ;
Begin
     Si (cpt = 0) Alors nonvide.wait Fsi ;
     x := tampon[t] ;
     t := (t+1) mod N ;
     cpt := cpt - 1 ;
     nonplein.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.

B. Solution 2

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

```
Prod_Cons : moniteur ;
Const N = ;
Var
     cpt : entier ;
     nonplein, nonvide : condition ;
```

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

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

3.3 Exemple 3 : Lecteurs / Rédacteurs sans priorité

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

```
     Si nl = 0 Alors E.signal Fsi
End
Procedure Deb_Ecriture ;
Begin
     Si (Ecriture ou nl ≠ 0) Alors E.wait Fsi ;
     Ecriture := vrai
End
Procedure Fin_Ecriture ;
Begin
     Ecriture := faux ;
     Si L.empty
       Alors L.signal
       Sinon E.signal
     Fsi
End
/*Initialisation*/
Begin
     Ecriture := faux ;
     nl := 0
End
```

4 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.

4.1 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(c) où c est une expression booléenne. c fait intervenir les variables de synchronisation du moniteur.

Un processus se bloque sur wait(c) si c 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
```

4.2 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

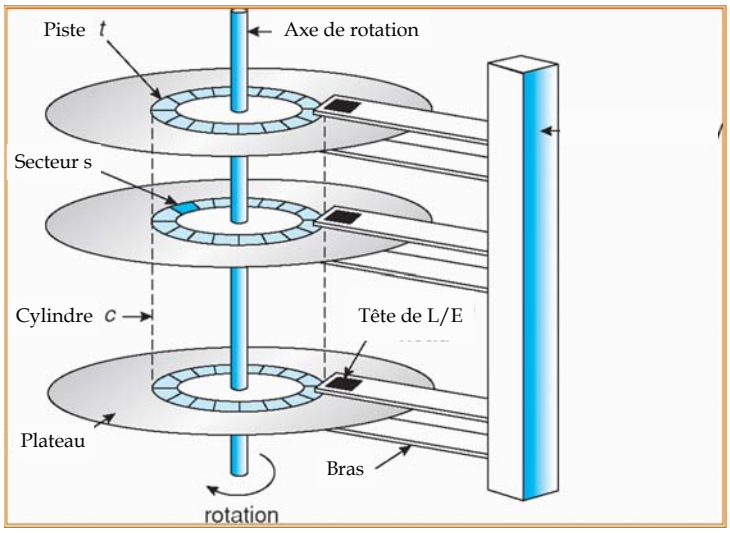
Gestion d'une ressource à N instances, dont l'allocation est faite en priorité à celui qui demande le moins d'instantes de cette ressource.

```
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 ;
Procedure Demande(k)
Begin
     Si (Dispo < k)
       Alors
            Begin
                 Insérer(File, k) ;
                 Ress.Wait(k) ;
            End
     Fsi ;
     Dispo := Dispo - k
End
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**).

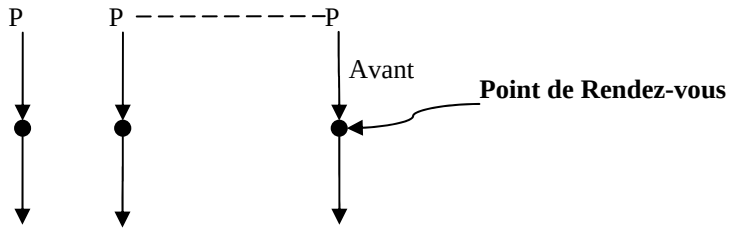
On suppose que les requêtes sont générées dynamiquement.

```
Disque : moniteur ;
Const Max = ... ;
Var
     Position : (0..Max-1) ;
     Direction : (Haut, Bas) ;
     Occupé : Booléen ;
     CHaut, CBas : Condition ;
Procedure Demander(k : entier)
Begin
     Si Occupé Alors
       Si (Position < k) ou (Position=k et Direction=Bas)
            Alors Chaut.Wait(k)
            Sinon Cbas.Wait(Max-k)
       Fsi
     Fsi ;
     Occupé := vrai ;
     Position := k ;
End
Procedure Libèrer
Begin
     Occupé := faux;
     Si (Direction = Haut)
```

```
        Alors
             Si Chaut.Empty
                  Alors Chaut.Signal
                  Sinon
                       Begin
                           Direction := Bas ;
                           CBas.Signal
                       End
             Fsi
        Sinon
             Si CBas.Empty
                  Alors CBas.Signal
                  Sinon
                       Begin
                           Direction := Bat ;
                           Chaut.Signal
                       End
             Fsi
     Fsi
End
/*Initialisation*/
Begin
     Occupé := faux ;
     Position := 0 ;
     Direction := Haut
End
```

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.



Ecrire le moniteur qui assure la synchronisation des n processus.

```
Rendez-vous : moniteur ;
Const N = ... ;
```

```
Var
     Cpt : entier ;
     Point : Condition ;
Procedure Avant
Begin
     Cpt ++ ;
     Si Cpt < N Alors Point.Wait Fsi ;
     Point.Signal ;
End
/*Initialisation*/
Begin
     Cpt := 0 ;
End
```

Exercice 2 (Lecteurs / Rédacteurs)

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

```
Lect-Red : moniteur ;
Var
     Lisant, VoulantEcrire : entier ;
     Okpour-lire, Okpour-écrire : Condition ;
Procedure Deb_Lecture
Begin
     Si VoulantEcrire > 0
       Alors
            Début
                  Okpour-lire.Wait ;
                  Okpour-lire.Signal ;
            Fin
     Fsi ;
     Lisant := Lisant + 1 ;
End
Procedure Fin_Lecture
Begin
     Lisant := Lisant - 1 ;
     Si Lisant = 0 Alors Okpour-écrire.Signal Fsi ;
End
Procedure Deb_Ecriture
Begin
     VoulantEcrire := VoulantEcrire + 1 ;
     Si (Lisant > 0) ou (VoulantEcrire > 1
```

```
        Alors Okpour-écrire.Wait
     Fsi ;
End
Procedure Fin_Ecriture
Begin
     VoulantEcrire := VoulantEcrire - 1 ;
     Si VoulantEcrire = 0
        Alors Okpour-lire.Signal
        Sinon Okpour-écrire.Signal
     Fsi ;
End
/*Initialisation*/
Begin
     Lisant := 0 ;
     VoulantEcrire := 0 ;
End
```

Exercice 3 (P et V)

Ecrire un moniteur qui implémente les primitives P et V.

```
Semaphore : moniteur ;
Var
     Valeur : entier ;
     Libre : Condition ;
Procedure P
Begin
     Valeur := Valeur - 1 ;
     Si Valeur < 0 Alors Libre.Wait Fsi ;
End
Procedure V
Begin
     Valeur := Valeur + 1 ;
     Si Valeur <= 0 Alors Libre.Signal Fsi ;
End
/*Initialisation*/
Begin
     Valeur := Valeur initiale ;
End
```

CHAPITRE VI : INTERBLOCAGE

1 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. Par conséquent, l'accès à ces ressources risque de créer une **situation de blocage**.

2 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**.

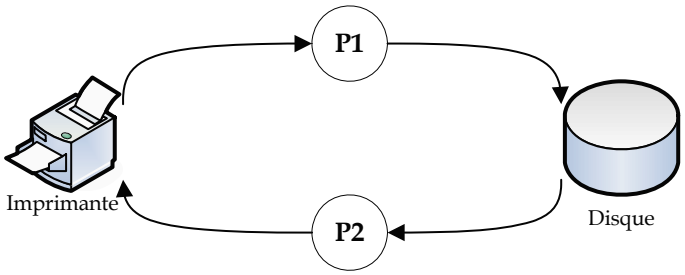
L'**interblocage** (**Deadlock**) est un état qui est dit **stable** ou **propriété stable** (**Stable Property**). Une propriété est dite stable, si une fois vérifiée elle reste toujours vraie dans les états ultérieurs du système.

3 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.

4 Conditions d'interblocage

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

- **Exclusion mutuelle** (**Mutual Exclusion**). Il existe des ressources non partageables.
- **Détenir et attendre** (**Hold and Wait**). Les processus détiennent des ressources et demandent d'autres acquises par d'autres processus.
- **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.
- **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.

5 Formalisation

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 (ou banalisées).

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

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

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

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

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

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.

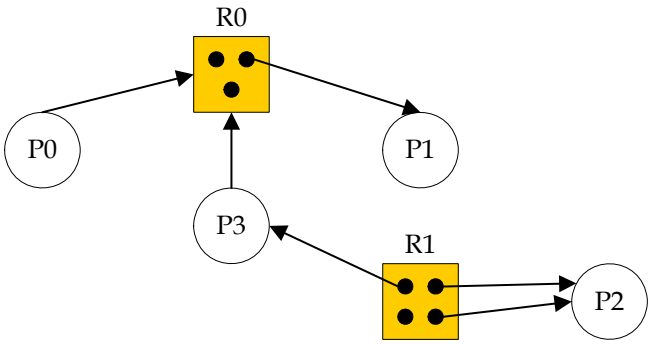
6 Représentation graphique

L'état du système peut être représenté par un graphe orienté, dit **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**).
- **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



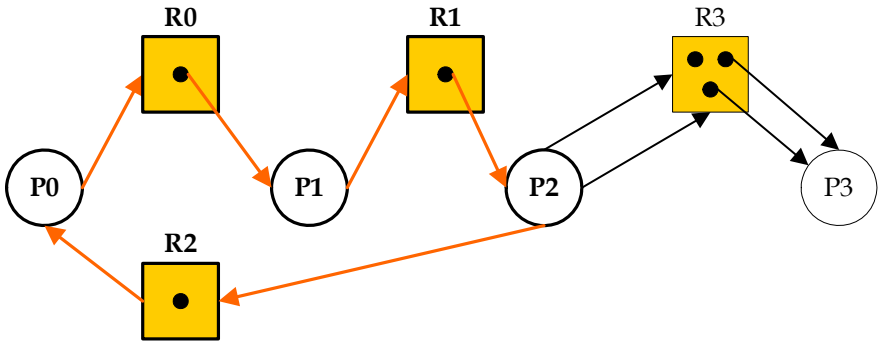
L'allocation **transforme** les arcs **Pi → Rj** en **Rj → Pi.**

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**.

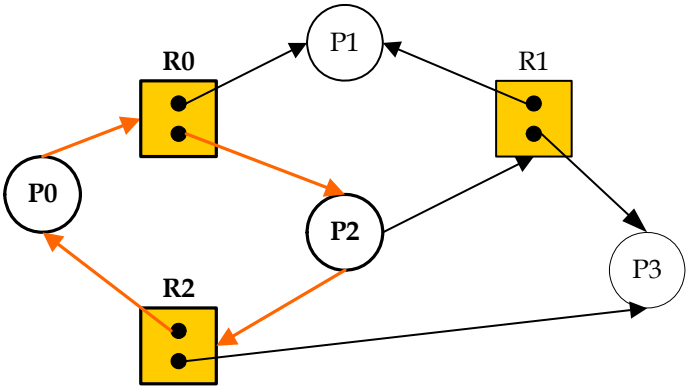
    Exemple



    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.

7 Traitement d'interblocage

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

- **La prévention** qui consiste à supprimer définitivement (de manière statique) une des conditions pouvant conduire à un 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.

- **La détection** qui consiste à laisser le système évoluer sans contrainte et essayer de vérifier périodiquement l'existence ou non d'une situation d'interblocage.

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

- **La politique de l'autruche** qui consiste à nier 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.

7.1 Prévention

Pour prévenir les interblocages, on doit éliminer une des quatre conditions nécessaires à leur apparition.

- **Condition 1 (Exclusion mutuelle)**

  Par exemple, pour les imprimantes, les processus « **spoolent** » 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 (**Ex.** Processeur).

- **Condition 2 (Détenir et attendre)**

  Il faut éviter qu'un processus qui détient certaines ressources d'en demander d'autres. Une solution est que le processus demande toutes les ressources avant l'exécution. Le problème avec cette solution est la mauvaise utilisation des ressources qui peut conduire à un ralentissement du système, en plus de la difficulté de réalisation de cette solution dans la pratique car l'allocation est, en général, dynamique.

- **Condition 3 (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 (**Ex.** Processeur).

- **Condition 4 (Attente circulaire)**

  Dans ce cas, la méthode consiste à imposer un ordre total sur les demandes de ressources. 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 Rj.

  A chaque classe Ri, on associe un **numéro unique F(Ri)** qui permet leur comparaison. Quand un processus Pi demande initialement la ressource Ri, ultérieurement si Pi demande une autre ressource Rj, on a forcement **F(Ri) < F(Rj)**. Ou si Pi demande Rj tel que F(Ri) > F(Rj), alors il doit au préalable **libérer** toutes les ressources Ri telle que F(Ri) < F(Rj).

  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.

Les techniques de prévention permettent d'empêcher les situations d'interblocage en imposant des contraintes sur la manière de formuler les requêtes d'allocation de ressources.

Au moins une des conditions d'interblocage ne pourra pas se produire. Ceci peut cependant conduire à une mauvaise utilisation des ressources, qui peut à son tour avoir une incidence sur les performances du système.

7.2 Evitement

La méthode d'évitement nécessite des informations additionnelles sur la manière dont les ressources sont demandées.

Par exemple, dans un système avec une ressource R1 et une ressource R2, on doit savoir qu'un processus P demande en premier R1 ensuite R2, et que le processus Q demande R2 en premier ensuite R1.

La connaissance des séquences complètes des demandes et libérations de chaque processus nous permet alors de décider, pour chaque demande, si le processus **devra ou non attendre**. Ce qui permet de savoir si une demande peut être satisfaite ou mise en attente pour éviter un interblocage.

Autrement dit, lorsqu'un processus demande une ressource, le système doit déterminer si l'attribution de la ressource est **sûre.**

A. Etat sain (fiable) (Safe State)

Définition

Un **état** est **fiable** (dit aussi **sûr**), si le système peut allouer des ressources à chaque processus dans un certain ordre en évitant un interblocage.

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** d'allocation de ressources qui permet à tous les processus de se terminer.

Exemple

P0 → P1 → P2 → P3, P0 s'exécute et libère les ressources qui sont demandées par P1, et P1 peut s'exécuter …etc.

Une séquence fiable <P1, P2, …, Pn> veut dire : dans l'état courant du système, pour chaque processus Pi, les ressources que Pi peut encore demander peuvent être satisfaites par les **ressources disponibles** ; C'est-à-dire, les ressources détenues par tous les processus Pj prédécesseurs de Pi dans la séquence. Ceci veut dire que si les ressources demandées par Pi ne sont pas disponibles immédiatement, alors Pi attendra jusqu'à ce que tous les Pj aient terminé.

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, P2 détient un exemplaire de plus ; c'est-à-dire :

- 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 est mis en attente.

P2 détient 3 exemplaires et il a besoin de 6 autres exemplaires ⇒ P2 est mis en attente.

Par conséquent, on aura une situation d'interblocage entre P0 et P1.

Conséquence

Il fallait mettre P2 en attente lorsqu'il a demandé un exemplaire de plus.

B. Algorithme du banquier (sûreté) (Banker's Algorithm)

L'algorithme du banquier [Dijkstra 1965] permet la construction d'une séquence fiable.

```
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.*/

Dem : Matrice nxm ;

/*Dem[i, j] définit la demande max 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.*/
```

Algorithme

1. Soient Work et Finish deux vecteurs de longueur respective m et n.

   Initialement

   Work := Dispo ;

   Finish[i] := False, ∀ i =1,n ;

2. Trouver un i **tel que** ((Finish[i] = False) et ($Besoin_i \leq$ Work))

   Si un tel i n'existe pas **Alors** aller à 4 ;

3. Work := Work + $Alloc_i$;

   Finish[i] := True;

   Aller en 2;

4. **Si** Finish[i] = True, ∀ i =1,n

      Alors le système est dans un **état fiable**

      Sinon le système est dans un état non fiable

C. Algorithme de demande de ressources

Soit REQUESTi le vecteur de demande du processus Pi.

REQUESTi[j] indique le nombre d'instances de la ressource Rj demandées par Pi.

Algorithme

Lors de la demande de ressources effectuée par Pi **Faire**

1. **Si** $REQUEST_i \leq Besoin_i$

      Alors Aller en 2

      Sinon Erreur « Processus a dépassé son max »

   Fsi ;

2. **Si** $REQUEST_i \leq$ Dispo

      Alors Aller en 3

      Sinon Pi doit attendre puisque les ressources demandées ne sont pas disponibles

   Fsi ;

3. Supposons que le système alloue les ressources demandées à Pi :

```
Dispo := Dispo – REQUESTi ;

Allocᵢ := Allocᵢ + REQUESTᵢ ;

Besoinᵢ := Besoinᵢ – REQUESTᵢ ;

Si l'état d'allocation résultant est sûr

   Alors allouer les ressources demandées

   Sinon Mettre en attente Pi ; Restaurer l'état antérieur ;

Fsi ;
```

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 :

$$\begin{matrix} & \textit{Alloc} \\ P1 \\ P2 \\ P3 \end{matrix}\begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix} \quad \begin{matrix} & \textit{Dem} \end{matrix}\begin{pmatrix} 2 & 0 & 1 & 1 \\ 3 & 0 & 1 & 1 \\ 2 & 2 & 2 & 0 \end{pmatrix} \quad \begin{matrix} \textit{Dispo} \end{matrix}\begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

1. L'état courant est-il sûr ?

   Pour que l'état du système soit sain, 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* :

$$\textit{Besoin} = \textit{Dem} - \textit{Alloc} = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

   Soient :

   - $\textit{Finish}[i] = \textit{false}$ pour i=1,3.
   - $\textit{Work} := \textit{Dispo} = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$.

$$\textit{Besoin}_3 = \begin{pmatrix} 2 \\ 1 \\ 0 \\ 0 \end{pmatrix} \le \textit{Work} = \begin{pmatrix} 2 \\ 1 \\ 0 \\ 0 \end{pmatrix} \Rightarrow \text{Servir P3} \Rightarrow \begin{cases} \textit{Work} = \textit{Work} + \textit{Alloc}_3 = \begin{pmatrix} 2 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 2 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ 2 \\ 0 \end{pmatrix} \\ \textit{Finish}[3] = \textit{true} \end{cases}$$

$$Besoin_2 = \begin{pmatrix}1\\0\\1\\0\end{pmatrix} \le Work = \begin{pmatrix}2\\2\\2\\0\end{pmatrix} \Rightarrow \text{Servir P2} \Rightarrow \begin{cases} Work = Work + Alloc_2 = \begin{pmatrix}2\\2\\2\\0\end{pmatrix} + \begin{pmatrix}2\\0\\0\\1\end{pmatrix} = \begin{pmatrix}4\\2\\2\\1\end{pmatrix} \\ Finish[2] = true \end{cases}$$

$$Besoin_1 = \begin{pmatrix}2\\0\\0\\1\end{pmatrix} \le Work = \begin{pmatrix}4\\2\\2\\1\end{pmatrix} \Rightarrow \text{Servir P1} \Rightarrow \begin{cases} Work = Work + Alloc_1 = \begin{pmatrix}4\\2\\2\\1\end{pmatrix} + \begin{pmatrix}0\\0\\1\\0\end{pmatrix} = \begin{pmatrix}4\\2\\3\\1\end{pmatrix} \\ Finish[1] = true \end{cases}$$

D'où la suite P3 P2 P1 est une suite complète et se termine ⇒ l'état du système est sûr (fiable).

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

   P2 demande $\text{Re}quest_2 = (1 \quad 0 \quad 0 \quad 0)$:

   Puisque $\text{Re}quest_2 \le Besoin_2$ et $\text{Re}quest_2 \le Dispo \Rightarrow$ la demande de P2 peut être satisfaite, mais on doit d'abord vérifier que le système reste dans un état fiable après la satisfaction de cette demande. Si on sert P2, on aura l'état suivant :

$$Dispo = Dispo - \text{Re}quest_2 = \begin{pmatrix}2\\1\\0\\0\end{pmatrix} - \begin{pmatrix}1\\0\\0\\0\end{pmatrix} = \begin{pmatrix}1\\1\\0\\0\end{pmatrix}$$

$$Alloc_2 = Alloc_2 + \text{Re}quest_2 = \begin{pmatrix}2\\0\\0\\1\end{pmatrix} + \begin{pmatrix}1\\0\\0\\0\end{pmatrix} = \begin{pmatrix}3\\0\\0\\1\end{pmatrix}$$

$$Besoin_2 = Besoin_2 - \text{Re}quest_2 = \begin{pmatrix}1\\0\\1\\0\end{pmatrix} - \begin{pmatrix}1\\0\\0\\0\end{pmatrix} = \begin{pmatrix}0\\0\\1\\0\end{pmatrix}$$

   Après la satisfaction de cette requête, aucun autre processus ne pourra être servi ; le résultat de l'application de l'algorithme du banquier est une suite vide. ⇒ Le système sera bloqué.

   Par conséquent, la ressource de R1 ne sera pas accorder à P2. D'où, P2 va se bloquer et on revient alors à l'état précèdent du système (c'est-à-dire l'état à l'instant t0).

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

   P3 demande $\text{Re}quest_3 = (2 \quad 0 \quad 0 \quad 0)$:

Puisque $\text{Re}\,quest_3 \le Besoin_3$ et $\text{Re}\,quest_3 \le Dispo \Rightarrow$ la demande de P3 peut être satisfaite, mais on doit d'abord vérifier que le système reste dans un état fiable après la satisfaction de cette demande. Si on sert P3, on aura l'état suivant :

$$Dispo = Dispo - \text{Re}\,quest_3 = \begin{pmatrix}2\\1\\0\\0\end{pmatrix} - \begin{pmatrix}2\\0\\0\\0\end{pmatrix} = \begin{pmatrix}0\\1\\0\\0\end{pmatrix}$$

$$Alloc_3 = Alloc_3 + \text{Re}\,quest_3 = \begin{pmatrix}0\\1\\2\\0\end{pmatrix} + \begin{pmatrix}2\\0\\0\\0\end{pmatrix} = \begin{pmatrix}2\\1\\2\\0\end{pmatrix}$$

$$Besoin_3 = Besoin_3 - \text{Re}\,quest_3 = \begin{pmatrix}2\\1\\0\\0\end{pmatrix} - \begin{pmatrix}2\\0\\0\\0\end{pmatrix} = \begin{pmatrix}0\\1\\0\\0\end{pmatrix}$$

En appliquant l'algorithme du banquier, on peut trouver la séquence complète suivante P3 P2 P1. Par conséquent, les deux ressources de R1 peuvent être accordées à P3.

7.3 Détection

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

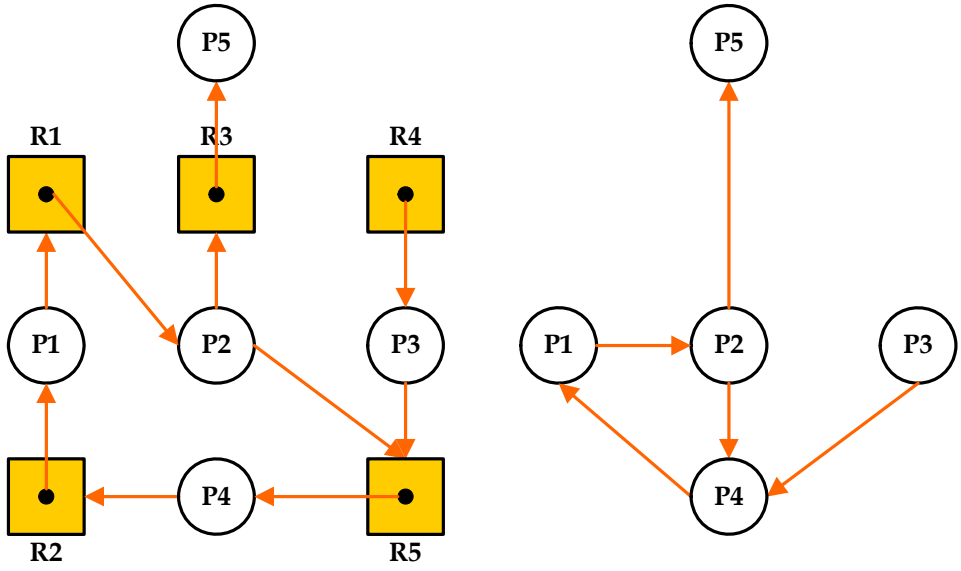
Dans ce cas, le système doit fournir :

- 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.

Nous utiliserons un algorithme basé sur des structures de données identiques à celles de l'algorithme du banquier (Dispo, Alloc et REQUEST).

```
Algorithme

1. Soient Work et Finish deux vecteurs de dimensions respectives m et n.
   Initialement
      Work := Dispo ;
      For i := 1 to n Do Finish[i] := false ;
2. Trouver un index i tel que ((Finish[i] = false) et (Request_i ≤ Work))
   /*Request est une matrice nxm.*/
   //Request[i, j] indique le nombre d'instances de Rj demandé par Pi.*/
   Si un tel index n'existe pas Alors aller à 3 ;
   Work := Work + Alloc_i ;
   Finish[i] := true ;
   Aller à 2;
3. Si Finish[i] = false, pour i ≤ i ≤ n
      Alors le système est dans un état d'interblocage ;
```

```
Si Finish[i] = false Alors le processus Pi est interbloqué.
```

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 :

$$\begin{array}{c} \\ P0 \\ P1 \\ P2 \\ P3 \end{array} \begin{array}{c} \textit{Alloc} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 2 \end{pmatrix} \end{array} \quad \begin{array}{c} \text{Re}\textit{quest} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \end{array} \quad \begin{array}{c} \textit{Dispo} \\ \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \end{array}$$

Soient :

- $Finish[i] = false$ pour i=0,3.
- $Work := Dispo = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$.

$$\begin{cases} \text{Re}\,quest_3 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \le Work = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \\ Finish[3] = false \end{cases} \Rightarrow \begin{cases} Work = Work + Alloc_3 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} \\ Finish[3] = true \end{cases}$$

$$\begin{cases} \text{Re}\,quest_1 = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \le Work = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} \\ Finish[1] = false \end{cases} \Rightarrow \begin{cases} Work = Work + Alloc_1 = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} + \begin{pmatrix} 0 \\ 2 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix} \\ Finish[1] = true \end{cases}$$

$$\begin{cases} \text{Re}\,quest_0 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \le Work = \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix} \\ Finish[0] = false \end{cases} \Rightarrow \begin{cases} Work = Work + Alloc_0 = \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix} \\ Finish[0] = true \end{cases}$$

$$\begin{cases} \text{Re}\,quest_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \le Work = \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix} \\ Finish[2] = false \end{cases} \Rightarrow \begin{cases} Work = Work + Alloc_2 = \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 2 \end{pmatrix} \\ Finish[2] = true \end{cases}$$

D'où la suite P3 P1 P0 P2 est une suite complète et se termine $\Rightarrow$ le système n'est pas dans un état interbloqué.

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, **Rollback**) 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. La suppression d'un processus est une **solution extrême**.

  La victime peut être un ou plusieurs processus appartenant à l'ensemble des processus interbloqués (**Ex.** le processus qui a exprimé une demande ayant provoqué l'interblocage), comme elle peut être un processus qui ne se trouve pas dans l'ensemble de processus interbloqués afin de libérer ses ressources.

  Un processus peut être la victime de manière répétée ; ce qui fait que ce processus peut ne jamais terminer son exécution. Pour éviter ce problème, on utilise un **compteur de sélection** qui peut être utilisé comme facteur de coût.

PARTIE II

SYSTEMES DISTRIBUES

CHAPITRE V : PRINCIPES DES SYSTEMES REPARTIS

1 Définitions

Il existe différentes définitions au terme système distribué, entre autres :

- **Tanenbaum** : "A distributed system is a collection of **independent computers** that appears to its users as a **single coherent system**".

  Ensemble d'ordinateurs indépendants connectés en réseau et communiquant via ce réseau. Cet ensemble apparaît du point de vue de l'utilisateur comme une unique entité.

- **Coulouris** : "…one in which **hardware** or **software components** located at **networked computers** communicate and coordinate their actions only by **message passing**".

  Un système réparti est un système formé de composants matériels ou logiciels localisés sur des ordinateurs en réseau qui communiquent et coordonnent leurs actions uniquement par échange de messages, le tout en vue de répondre à un besoin donné.

- **Lamport** : "A distributed system is one on which I cannot get any work done because some machine I have never heard of has crashed".

  Un système distribué est un système où la défaillance d'une machine dont l'utilisateur ignore jusqu'à l'existence peut rendre sa propre machine inutilisable.

Dans notre cours, on opte pour la définition suivante :

- Un système réparti (**Distributed System**) est un système comprenant un **ensemble d'ordinateurs** et un **système de communication.**

Un système réparti = {Ordinateurs} + Système de Communication

2 Motivations

2.1 Partage de ressources (Ressource Sharing)

La répartition est un moyen de mise à disposition et donc de partage de ressources et services.

Exemples

- Partage et impression de fichiers localisés sur des sites distants. Un service de gestion de fichiers "répartis" est un des services de base des systèmes d'exploitation répartis.

- Traitement d'information sur une base de données distribuée

- Partage de périphériques distants. Une imprimante peut être rendue accessible à tous les usagers d'une architecture répartie. Elle peut même constituer un nœud du réseau.

2.2 Partage et décentralisation du travail

Ceci est possible s'il y a une communication entre les membres du groupe.

Exemples

- Travail collaboratif supporté par ordinateurs (**Computer-Supported-Collaborative Work, CSCW**) tel que l'édition coopérative.

2.3 Coût et puissance de calcul

La connexion de machines en réseau permet d'obtenir à moindre coût une puissance de calcul importante.

Exemples

- 1000 µp à 100MIPS dépassent la performance brute du supercalculateur le plus rapide.

2.4 Performance (Performance)

Beaucoup d'applications sont parallèles. Pour accélérer l'exécution de ces applications, il suffit de répartir le traitement sur un ensemble d'ordinateurs du réseau. On parle dans ce cas de la **répartition de charges** (**Load Balancing**)

Exemples

- Calculs scientifiques distribués sur un ensemble de machines.

2.5 Fiabilité (Reliability)

La **réplication** (**Replication**) de l'information et des services sur plusieurs sites augmente leur **disponibilité** et par conséquent la fiabilité du système.

Exemples

- Existence de deux serveurs de fichiers dupliqués, avec sauvegarde.

2.6 Flexibilité (Flexibility)

Une architecture répartie est par nature **modulaire**. Il est donc plus facile d'ajouter ou d'enlever un nœud connecté au réseau, le système global continuant à être disponible. Cette caractéristique joue un rôle important avec la mobilité des usagers.

Cette flexibilité se situe aussi au niveau logique. Un nouveau service peut être installé sur un nœud sans nécessiter une reconfiguration des autres nœuds.

3 Défis

L'apparition des systèmes distribués soulève de nouveaux défis propres à ces environnements :

3.1 Transparence

Le but est de cacher l'architecture, le fonctionnement de l'application ou du système distribué pour apparaître à l'utilisateur comme une application unique cohérente. On peut distinguer plusieurs types de transparence :

- **Transparence d'accès (Access transparency) :** Accès identique aux données locales ou distantes et indépendamment de leur format de représentation.
- **Transparence d'emplacement (Location transparency) :** L'utilisateur ne connaît pas où sont situées les ressources.
- **Transparence de migration (Migration (mobility/relocation) transparency) :** Les ressources peuvent être déplacées sans modification de leur nom.
- **Transparence de duplication (Replication transparency) :** L'utilisateur ne connaît pas le nombre de copies existantes.
- **Transparence de concurrence (Concurrency transparency) :** Plusieurs utilisateurs peuvent partager en même temps les mêmes ressources.
- **Transparence de parallélisme :** Des tâches peuvent s'exécuter en parallèle sans que les utilisateurs le sachent.

3.2 Souplesse

Offrir au système la facilité de modification, de configuration et d'extension.

3.3 Fiabilité

A. Disponibilité

Implique de savoir maintenir la cohérence des copies

B. Sécurité

La nature d'un système distribué fait qu'il est beaucoup plus sujet à des attaques. Les ressources doivent être protégées contre des utilisations abusives et malveillantes. En particulier le problème de piratage des données sur le réseau de communication.

C. Tolérance aux pannes

Le système doit être conçu pour résister aux pannes et les masquer aux utilisateurs. La panne de certains serveurs (ou leur réintégration dans le système après la réparation) ne doit pas perturber l'utilisation du système en terme de fonctionnalité.

4 Exemples de systèmes distribués

On peut citer :

- World Wide Web
- Systèmes de fichiers distribués (NFS)

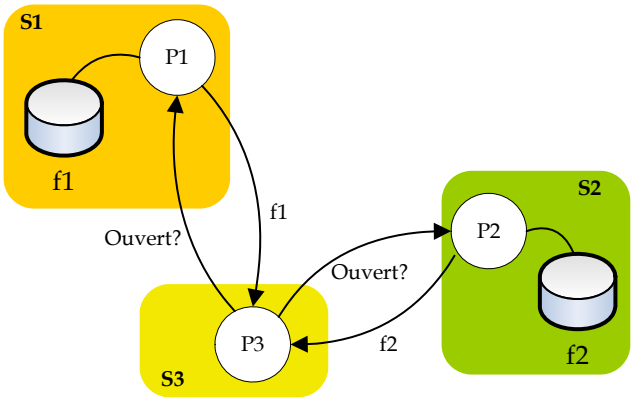
- Bases de données distribuées
- Systèmes P2P
- Réseaux mobiles (Cellulaires, Ad-hoc)
- Réseaux de capteurs (Sensor networks)

5 Caractéristiques des systèmes répartis

5.1 Absence de mémoire commune

Impossibilité de capter instantanément l'**état global** (**Global State**) d'un système réparti au moyen d'un ensemble de variables partagées.

Exemple



P3 veut savoir si P1 ou P2 ont ouvert des fichiers ?

Connaissance instantanée **impossible**

5.2 Absence d'une horloge physique globale

Deux éléments distincts du système peuvent avoir une perception différente de l'état d'un troisième élément ou sous-système et de l'ordre des événements qui s'y produisent.

5.3 Variabilité des délais de transmission des messages

Il n'y a aucune supposition sur les délais de transfert des messages qui sont **finis** mais **imprévisibles**. Cette absence des limites temporelles rend le système distribué **asynchrone**.

L'exécution asynchrone des processus et le caractère fini et non borné des délais de communication confèrent aux systèmes répartis un comportement **non déterministe**.

6 Algorithmique répartie

6.1 Définition

Un algorithme réparti est un algorithme parallèle composé d'un ensemble fini de processus séquentiels, communiquant entre eux par échange de messages.

Algorithme réparti = {Processus} + Voies de communication

Un algorithme réparti peut être modélisé par

- L'ensemble des processus **P = {P0, P1, …, Pn}**, et
- L'ensemble des canaux de communication reliant ces processus **C = {C0, C1, …, Cm}**.

6.2 Eléments de base

A. Processus

Les processus d'un système s'exécutent en parallèle. Ils représentent une entité logicielle effectuant une tâche ou un calcul. Un processus peut recevoir plusieurs messages extérieurs entraînant des traitements spécifiques.

Ceci nécessite de construire ces processus à l'aide d'une structure de contrôle non déterministe leur permettant d'être en attente d'un événement parmi plusieurs possibles.

Exemple

```
Lors d'Evt1 Faire Trait1 Fait ;
Lors d'Evt2 Faire Trait2 Fait ;
…
Lors d'Evntn Faire Traitn Fait ;
```

Un processus possède une mémoire locale, un état local (Ensemble de ses données et des valeurs de ses variables locales), un identifiant qu'il connaît et Pas ou peu de connaissance des autres processus du système et de leur état

B. Voies de communication

La topologie est un facteur important pour écrire un algorithme de votre solution ; i.e. pour écrire un algorithme il faut définir une topologie bien définie.

- **Topologie physique**

  De P0 à P1, il existe plusieurs chemins. C'est pour ça les délais de messages sont différents ; i.e. un message m1 peut prendre un chemin court et m2 prendre un chemin long.

- **Topologie logique**

  Les voies de communication définissent une topologie logique donnée :

- **Maillage en anneau (Ring Topology)** : le réseau est un cycle, tout processus possède deux voisins et il ne peut communiquer qu'avec ceux-ci.

- **Maillage en étoile (Star Topology)** : un processus particulier peut communiquer avec tous les autres qui, réciproquement, ne communiquent qu'avec le processus particulier.

- **Maillage complet (Full Mesh Topology)** : tout processus peut communiquer avec tout processus.

- **Arbre (Tree Topology)** : la panne de tout processus non terminal déconnecte le système logique.

- Quelconque…

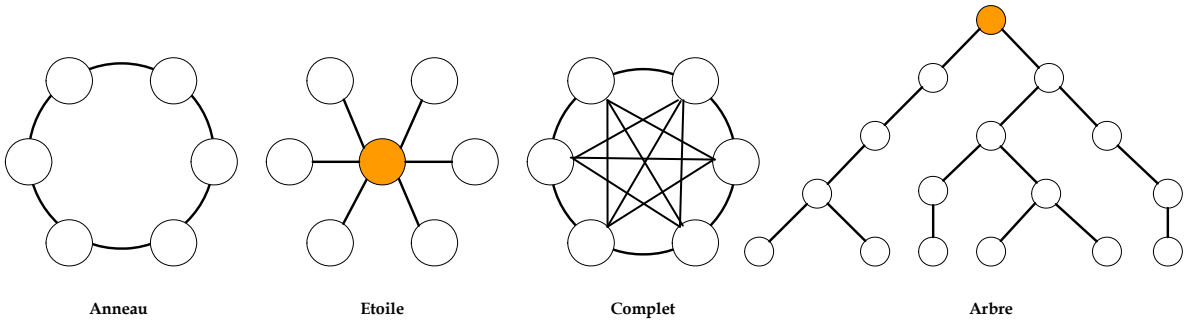


Figure : Types de Topologies Logiques

Il y a une dépendance étroite entre un algorithme réparti et la topologie sous-jacente des voies de communication.

On peut être amené à rajouter une "couche" sur un réseau X pour simuler un maillage Y. Il ne faut pas confondre réseau physique et maillage logique (Ex. l'anneau peut n'être que virtuel).

6.3 Quelques problèmes de contrôle réparti

Il existe deux types d'algorithmes répartis :

- **Algorithmes de contrôle** qui font partie des fonctions de base d'un système réparti, telles que l'exclusion mutuelle, terminaison, détection d'interblocage, …etc.

- **Algorithmes de calcul** qui font un calcul et rendent un résultat, tels que le calcul d'une arborescence couvrante d'un graphe, calcul du flux minimal, élection, …etc.

A. Exclusion mutuelle

Il s'agit d'attribuer un privilège **équitablement** à un ensemble de processus communiquant qui coopèrent à la réalisation d'un but commun (un privilège ne peut être détenu indéfiniment par un même processus).

B. Election

C'est le choix d'un processus et d'un seul dans un ensemble de processus équivalents afin de lui attribué un privilège donné.

C. Interblocage

C'est la situation dans laquelle se trouve un ensemble de processus (au moins deux) telle que chaque processus de l'ensemble attend l'occurrence d'un événement qui ne peut être produit que par un autre processus de ce même ensemble.

Cet événement peut être :

- Soit la libération d'une ressource (Interblocage de compétition).
- Soit l'arrivée d'un message (Interblocage lié à la communication où l'état de tous les processus + les canaux de transmission sont tous vides).

D. Terminaison

Comment peut-on détecter la terminaison d'un algorithme réparti ?!

Cette détection aurait été facile si on pouvait disposer d'un état global instantané décrivant l'état des processus et des voies de communication.

Un algorithme réparti sera dit terminé si tous les processus sont passifs et il n'y a aucun message en transit.

6.4 Qualité d'un algorithme réparti

Des hypothèses doivent être faites sur le comportement des voies de communication. Les caractéristiques les plus importantes à considérer sont :

- L'ordre de réception des messages. Autrement dit, y a-t-il ou non **déséquencement** possible des messages ?
- Est-il possible que certains messages se trouvent **dupliqués** au cours de leur transmission ?
- Les messages peuvent-ils être **altérés** ?
- Les messages peuvent-ils être **perdus** ? (**Délai** d'acheminement **fini**)
- Le **délai** d'acheminement des messages est-il **borné**, c'est-à-dire peut-on être certain d'une valeur maximum du délai entre émission et réception ?

Parmi les qualités qu'on peut souhaiter pour un algorithme réparti :

- Minimiser le trafic engendré (complexité en nombre de messages).
- Minimiser le temps d'exécution.
- Bonne résistance aux pannes, c'est-à-dire que le système contrôlé ne soit pas irrémédiablement interrompu si l'un des processeurs qui assurent le contrôle devient défectueux.

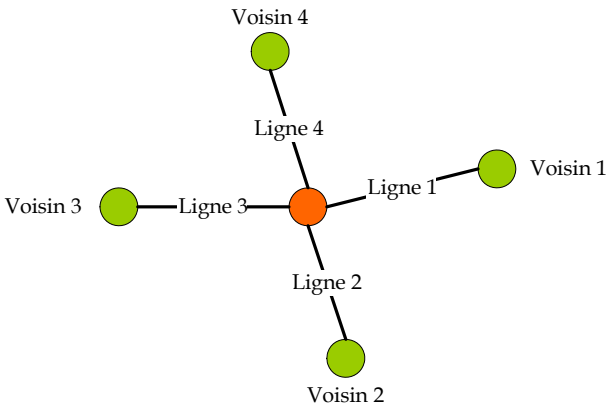
CHAPITRE VI : EXEMPLES D'ALGORITHMES REPARTIS

1 Diffusion d'une information dans un réseau

1.1 Contexte d'un processus

A un instant donné, tout site (processus) du réseau possède une connaissance de son environnement. Par exemple :

- Ports ou lignes auxquels il est connecté.
- Identités des processus auxquels il est directement connecté ; c'est-à-dire, il connait ses **voisins directs (Direct Neighbors)**.



- Nombre de sites dans le réseau.
- Topologie du réseau :
  - Nature de cette topologie (i.e. Anneau, Complètement Maillée, Quelconque, …etc.), ou
  - Graphe qui modélise le réseau, **G = (E, X)** où **E** est l'ensemble des sommets du graphe et représente les processus et **X** est l'ensemble des arrêtes du graphe et représente les canaux de communication.

1.2 Notre problème

Le processus $\mathbf{P_{init}}$ veut diffuser une information **m** vers tous les autres processus.

Deux (02) idées de solutions :

- **Diffusion parallèle** (**Parallel Broadcast**) : quand on reçoit **m**, on le diffuse à **tous** ses voisins.
- **Diffusion séquentielle** (**Sequential Broadcast**) : quand on reçoit **m**, on l'envoie à **un** de ses voisins.

Dans ces deux cas, le **contexte** de **Pi** est le suivant :

- L'ensemble des identités des canaux reliant Pi à ses voisins, noté **Canaux$_i$**.
- **Structure de données** servant à mémoriser m.

On peut écrire les algorithmes suivants :

A. Diffusion parallèle

```
Lors de réception de m
Faire
      Mémoriser m ;
      ∀ C ∈ Canaux_i : Envoyer(m) sur C ;
Fait
```

Cet algorithme assure la correction partielle mais ne se termine pas.

B. Diffusion séquentielle

```
Lors de réception de m
Faire
      Mémoriser m ;
      Soit C ∈ Canaux_i : Envoyer(m) sur C ;
Fait
```

Cet algorithme n'assure ni la correction partielle, ni la terminaison.

1.3 Etude de la diffusion parallèle

La non terminaison des solutions précédentes est dû au fait qu'aucun **contrôle** n'est appliqué.

Le principe de la solution ci-dessous repose sur :

- L'utilisation d'une variable booléenne locale à chaque processus, notée **Reçu$_i$**, permettant de savoir si le processus Pi a déjà reçu le message ou pas encore. **Reçu$_i$** est **initialement** égale à **faux**.
- Transport avec chaque message de l'information concernant son canal d'arrivée, notée **C$_{ij}$**.

```
Algorithme de Pi
Lors de réception de m sur C_ij
Faire
      Si ¬ Reçu_i Alors
```

```
            Reçu_i := vrai ;
            Mémoriser m ;
            ∀ C ∈ Voisins – {C_ij} : Envoyer(m) sur C ;
      Fsi ;
Fait
```

Remarque

- Nature du contrôle
  - Purement local (connaissance jamais transmise).
  - Peut être mis en œuvre avec une connaissance de l'environnement purement locale (noms des canaux).
  - Contrôle exercé à **postériori** (à la réception).
  - La **terminaison** est **assurée**.

A. Contrôle à priori

- **But**

  Eviter que Pi ne transmette m à un voisin Pj dès lors que ce dernier à déjà reçu ou va recevoir (d'un autre de ses voisins) cette information.

- **Conditions**
  - Pi doit pouvoir discriminer ses voisins par leur identité. Ceci qui fait que la connaissance locale de l'environnement, n'est plus réduite aux canaux de communication.
  - Pi doit apprendre une connaissance de contrôle qu'il ne possède pas initialement. Ceci est possible en véhiculant une information de contrôle dans les messages échangés.

- **Solution**
  - Pi est doté d'une constante **Voisins$_i$** qui implémente sa connaissance locale de son environnement. Cette constante représente l'ensemble des identités des voisins de Pi.
  - La connaissance de contrôles est :
    - Mise en œuvre localement par le booléen **Reçu$_i$**.
    - Transportée par les messages dans un champ **Z**, où Z est un sous ensemble de processus ayant reçu ou allant recevoir l'information.

```
Algorithme de Pi
```

```
Voisins_i : identités des voisins de Pi ;

Reçu_i : booléen ; init(Reçu_i, faux) ;

Lors de réception de (m, Z)

Faire

      Si ⌉ Reçu_i Alors

            Reçu_i := vrai ;

            Mémoriser m ;

            Y := Voisins_i – Z ;

            /*Y contient les identités des voisins qui ne sont pas
            dans Z*/

            Si Y ≠ ∅ Alors ∀ k ∈ Y : Envoyer (m, Z ∪ Y) à Pk  Fsi ;

      Fsi ;

Fait ;
```

2 Construction d'une arborescence couvrante de racine donnée

2.1 Problème

$\mathbf{P_{init}}$ souhaite construire une arborescence couvrante dont il sera la racine.

A l'issue de l'algorithme, chaque processus Pi doit connaître :

- Son père $\mathbf{Pere_i}$ dans l'arborescence (pour $i \neq j$).
- Ses fils dans l'arborescence (ensemble $\mathbf{Fils_i}$). Pour deux processus Pi et Pj donnés, leurs ensembles $Fils_i$ et $Fils_j$ doivent vérifier la condition suivante :

$$\forall\ \mathbf{i, j, i \neq j : Fils_i \cap Fils_j = \varnothing}$$

Cette solution sera construite par : **Diffusion + Acquittements**.

2.2 Arborescence par diffusion parallèle

- Le premier message reçu par Pi est le seul pris en compte. Ceci établit $\mathbf{Pere_i}$ (= émetteur du message).
- Pi propage l'exploration vers un sous-ensemble de ses voisins. Ceci établit $\mathbf{Fils_i}$.
  - **Problème**

Un même processus est atteint par plusieurs messages. Il est alors considéré comme fils pour tous les émetteurs.

– **Solution**

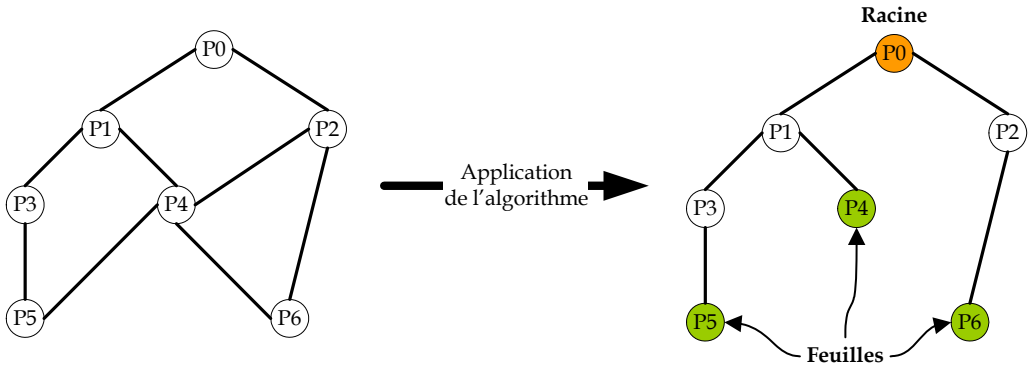
Renvoi systématique d'un message d'acquittement à l'émetteur de tout message d'exploration.

- Le **1er** porte la réponse **Oui**. Cette réponse n'est renvoyée que lorsque tous les acquittements attendus sont arrivés.
- Les **autres** portent la réponse **Non**. Cette réponse est renvoyée immédiatement.

```
Algorithme de Pi

Const Voisins_i = …;

Var

      Pere_i : Nom Init nil ;

      Fils_i : Ensemble de noms Init ∅ ;

      Nback_i : entier Init 0 ; /*compte le nombre d'acks attendus par
      Pi*/

Lors de réception de (Explorer, Z) depuis Pj

Faire

      Si Reçu_i

            Alors Envoyer(Acquitter, Non) à Pj

            Sinon

            Reçu_i := vrai ;

            Pere_i := j ;

            Soit Y := Voisins_i – Z ;

            Cas Y = ∅ : Envoyer(Acquitter, Oui) à Pj ;

                Y ≠ ∅ : ∀k ∈ Y, Envoyer(Explorer, Z ∪ Y) à Pk ;

                        Fils_i := Y ;

                        NbAck_i := Card(Y) ;

            FCas ;

      Fsi ;

Fait ;
```

```
Lors de réception de (Acquitter, X) depuis Pj
Faire
      Si X = Non Alors Fils_i := Fils_i – {j} Fsi ;
      NbAck_i := NbAck_i – 1 ;
      Si NbAck_i = 0 Alors
            Cas i ≠ init : Envoyer(Acquitter, Oui) à Pere_i ;
                i = init : "Algorithme terminé" ;
            FCas ;
      Fsi ;
Fait ;
```

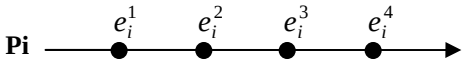
Exemple



CHAPITRE VII : TEMPS LOGIQUE

3 Introduction

L'exécution d'un algorithme réparti est vue comme une succession d'**événements (Events)**, chacun d'eux se produisant sur un site (processus) donné. Donc, on peut modéliser l'**activité** d'un processus Pi participant au calcul réparti comme une séquence d'événements e_i^j (**i.e.** événement j du processus i).



Dans un système distribué, deux **activités** générales peuvent prendre place :

- Les **activités locales** qui sont réalisées de manière indépendante par chaque processus.
- Les **activités de synchronisation** durant lesquelles deux ou plusieurs processus agissent entre eux et échangent des messages.

Ces activités peuvent être décomposées en **trois** (03) types d'événements. Donc, un événement e_i^j peut être soit :

- Evénement d'**émission** d'un message *m*.
- Evénement de **réception** d'un message *m*.
- Evénement **interne**.

L'absence d'un référentiel de temps globale et le caractère fini et non borné des délais de transmission, rendent difficile, voir impossible, de déterminer si un évènement s'est produit avant un autre.

Cependant, l'ordre dans lequel surviennent les événements est important dans bien des cas. Ainsi :

- Pour une entrée dans une section critique, on ne peut, par exemple, prendre comme critère d'autorisation la date de la demande : une machine dont la date à 1 heure d'avance sur les autres machines sera fortement avantagée ;
- De même, la date d'arrivée de réception d'un message peut du coup être inférieure à la date d'émission !
- Lorsqu'une donnée est modifiée sur deux sites, laquelle de ces modifications est la plus récente et donc la dernière ?

Il devient donc nécessaire de réaliser l'ordre entre les événements **sans avoir recours à l'heure qui se trouve dans les sites**.

Dans un système réparti de n sites (n processus) connectés par des canaux de communication :

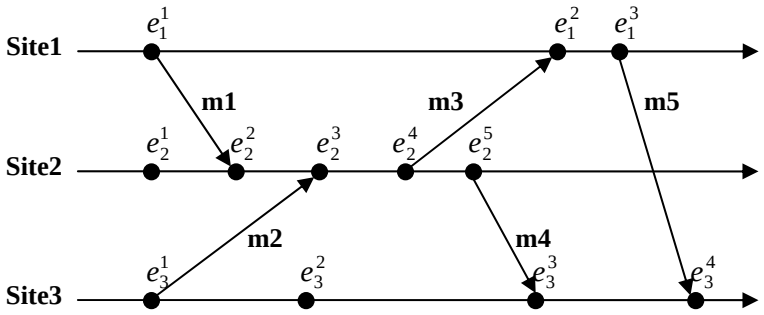
R1 : tous les événements sur un même site (processus) sont totalement ordonnés.

R2 : pour chaque message, l'événement d'émission précède l'événement de réception.

La fermeture transitive de ces relations de dépendance causale, notée $\rightarrow$, sur l'ensemble des événements produits par une exécution répartie est la relation dite « **happened before** » définit par L. Lamport (1978).

Cette relation définit un **ordre partiel** sur les événements. Deux événements distincts, e et e', sont dits **concurrents** ou non comparables, noté **e || e'**, si $\neg$**(e** $\rightarrow$ **e')** et $\neg$**(e'** $\rightarrow$ **e)**.

Une manière adéquate pour visualiser les calculs distribués est les diagrammes de temps :



L'application de la relation **happened before** sur cet exemple conduit à tirer les résultats suivants :

- En appliquant la première règle, on déduit les relations suivantes entre les événements internes à chaque processus :
  - Sur le site 1 : $e_1^1 \longrightarrow e_1^2 \longrightarrow e_1^3$
  - Sur le site 2 : $e_2^1 \longrightarrow e_2^2 \longrightarrow e_2^3 \longrightarrow e_2^4 \longrightarrow e_2^5$
  - Sur le site 3 : $e_3^1 \longrightarrow e_3^2 \longrightarrow e_3^3 \longrightarrow e_3^4$
- L'application de la deuxième règle introduit les relations suivantes entre les événements d'émission et de réception :
  - Pour le message m1 : $e_1^1 \longrightarrow e_2^2$
  - Pour le message m2 : $e_3^1 \longrightarrow e_2^3$
  - Pour le message m3 : $e_2^4 \longrightarrow e_1^2$
  - Pour le message m4 : $e_2^5 \longrightarrow e_3^3$
  - Pour le message m5 : $e_1^3 \longrightarrow e_3^4$

- Parmi les relations d'ordre partiel qu'on puisse définir sur les événements de ce calcul distribué, on trouve :

  - $e_1^1 \longrightarrow e_2^2 \longrightarrow e_2^3 \longrightarrow e_2^4 \longrightarrow e_1^2 \longrightarrow e_1^3 \longrightarrow e_3^4$
  - $e_3^1 \longrightarrow e_2^3 \longrightarrow e_2^4 \longrightarrow e_2^5 \longrightarrow e_3^3 \longrightarrow e_3^4$

- Parmi les événements concurrents, on trouve :

  - $e_1^1 \,||\, e_2^1$, $e_2^1 \,||\, e_3^1$, $e_1^1 \,||\, e_3^1$, $e_1^3 \,||\, e_2^5$, …etc.

4 Horloges logiques

4.1 Problème

Trouver des mécanismes qui permettent d'associer des dates aux événements concernés tels que :

Si **a** $\rightarrow$ **b** alors la date associée à **a** doit être relativement à un temps logique global avant la date associée à **b**.

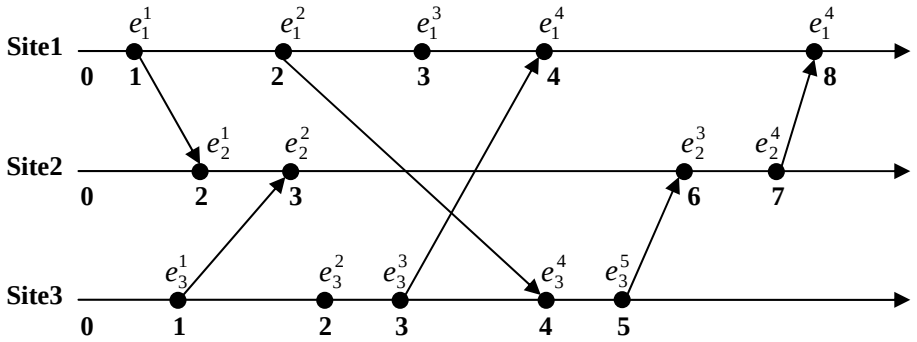
4.2 Horloges linéaires (Lamport 78)

A. Principe

A chaque site (processus) **i** est associée une variable entière $\mathbf{h_i}$ ayant des **valeurs croissantes**. Cette variable est **initialisée** à **0** au lancement du système.

```
Lors d'occurrence d'un événement e /*interne ou émission*/
Faire
    h_i := h_i + d (d > 0)
    Associer h_i à e.
Fait ;
Lors de réception (m, h)
Faire
    h_i := max(h_i, h) + d ;
    Associer h_i à l'événement de réception.
Fait ;
```

B. Exemple



C. Propriétés

- Il est possible d'utiliser ce mécanisme d'estampillage pour construire un **ordre total**, noté $\rightarrow_T$.

  L'**estampille** d'un événement e est composée de sa **date d'occurrence** et de l'**identité du site** qui a produit e.

  La règle de comparaison des dates de deux événements e et e' estampillés respectivement par (hi, i) et (hj, j) est alors :

  $$\mathbf{e \rightarrow_T e' \Leftrightarrow (h_i < h_j \text{ ou } (h_i = h_j \text{ et } i < j))}$$

  En appliquant cette règle sur l'exemple précédent, on déduira la séquence d'ordre total suivante entre les événements du système :

  $e_1^1 \rightarrow_T e_3^1 \rightarrow_T e_1^2 \rightarrow_T e_2^1 \rightarrow_T e_3^2 \rightarrow_T e_1^3 \rightarrow_T e_2^2 \rightarrow_T e_3^3 \rightarrow_T e_1^4 \rightarrow_T e_3^4 \rightarrow_T e_3^5 \rightarrow_T e_2^3 \rightarrow_T e_2^4 \rightarrow_T e_1^5$

- Les horloges linéaires vérifient la propriété suivante :

  $$\mathbf{e \rightarrow e' \Rightarrow h(e) < h(e')}$$

  Mais, la relation h(e) < h(e') **n'est pas suffisante** pour savoir s'il existe une relation de causalité entre e et e'.

  La relation h(e) < h(e') implique uniquement que e' ne précède pas causalement e ; il est possible que e précède causalement e' ou bien que e et e' soient concurrents.

  C'est le cas des événements e_3^2 et e_1^3. En comparant leurs estampilles, on trouve que $\mathbf{h(e_3^2) < h(e_1^3)}$ mais les deux événements sont concurrents $\mathbf{(e_3^2 \,||\, e_1^3)}$.

4.3 Horloges vectorielles (Mattern 89, Fidge 89)

Il est quelquefois utile de pouvoir déterminer s'il y a ou non dépendance causale entre deux événements, par exemple pour la recherche des causes potentielles d'une erreur.

Le mécanisme des horloges vectorielles a été introduit pour caractériser la dépendance causale.

A. Principe

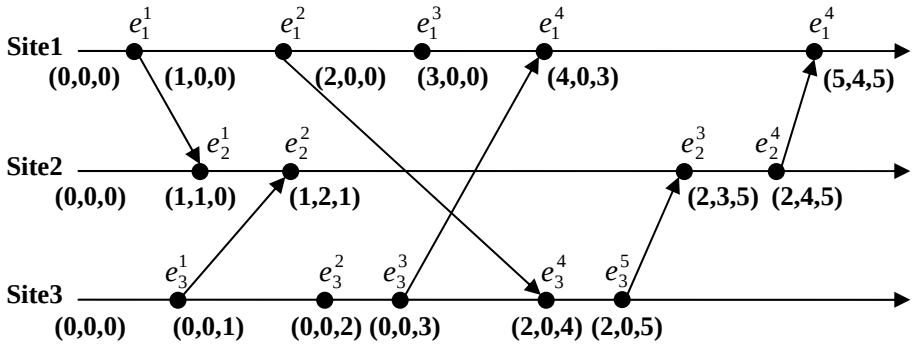
A chaque site (processus) **i** est associé un vecteur $\mathbf{V_i}$ de dimension **n** (n est le nombre de processus dans le système) où :

- $\mathbf{V_i[i]}$ décrit l'évolution du temps logique du site S_i (i.e. l'horloge logique locale de S_i). Elle prend des valeurs croissantes générées localement.
- $\mathbf{V_i[j]}$ représente la connaissance qu'a le site S_i sur l'évolution du temps sur le site S_j.

Ce vecteur est **initialisé à 0** au lancement du système.

```
Lors d'occurrence d'un événement e /*interne ou émission*/
Faire
    Vi[i] := Vi[i] + d (d > 0)
    Associer Vi à e.
Fait ;
Lors de réception (m, V)
Faire
    Vi[i] := Vi[i] + d (d > 0)
    ∀j≠i, Vi[j] := max(Vi[j], V[j]);
    Associer Vi à l'événement de réception.
Fait ;
```

B. Exemple



C. Propriétés

- Pour **d = 1**, la valeur de la i^{ème} composante de l'estampille vectorielle d'un événement **e** ($\mathbf{V_e[i]}$) est exactement le **nombre d'événements** du site S_i qui appartiennent au **passé** de e.
- La relation d'ordre suivante peut par ailleurs être définie sur les estampilles vectorielles:

- **$V_e \leq V_{e'} \Leftrightarrow \forall i, V_e[i] \leq V_{e'}[i]$.**
- **$V_e < V_{e'} \Leftrightarrow V_e \leq V_{e'}$ et $\exists i, V_e[i] < V_{e'}[i]$.**
- **$V_e \,||\, V_{e'} \Leftrightarrow \neg(V_e < V_{e'})$ et $\neg(V_{e'} < V_e)$.**

- Le système de datation par estampilles vectorielles a la remarquable propriété de **refléter exactement la relation de précédence causale entre événements**. Soit V_e et $V_{e'}$ les estampilles des événements **e** et **e'** respectivement :
  - **$e \rightarrow e' \Leftrightarrow V_e \leq V_{e'}$.**
  - **$e \,||\, e' \Leftrightarrow V_e \,||\, V_{e'}$.**

CHAPITRE VIII : ETAT GLOBAL ET COUPURES

1 Introduction

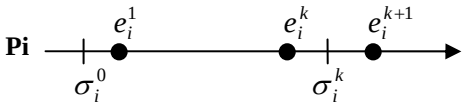
L'absence d'état global est une des caractéristiques essentielle des systèmes répartis. Un processus n'a qu'une connaissance partielle de l'état global du système.

L'intérêt de calculer un état global d'un calcul réparti est de :

- Trouver des états cohérents à partir desquels on peut reprendre un calcul distribué en cas de plantage du système. Il s'agit donc de définir des **points de reprise** (**Checkpoints**) potentiels.
- Détecter la **terminaison** : il s'agit de savoir si l'algorithme qu'on observe est terminé, c'est-à-dire si tous les processus sont passifs relativement à cet algorithme, et s'il n'y a plus de messages en transit s'y rapportant.
- Détecter l'**interblocage** : le but est ici de détecter si le système est bloqué (plusieurs processus en attente de messages venant d'autres processus, lesquels sont dans un état d'où ils ne peuvent les envoyer).
- Détecter des **propriétés stables**, du respect d'invariants.
- Détecter la perte de jeton (**Token**).
- Ramasse miette (**Garbage Collection**).

2 Notations

- Soient $h_i = e_i^1 e_i^2 \ldots$ l'**histoire locale** (**Local History**) du processus Pi pendant le calcul réparti et $H = h_1 \cup h_2 \cup \ldots \cup h_n$ l'**histoire globale** (**Global History**) du calcul réparti.
- Soit $h_i^k = e_i^1 e_i^2 \ldots e_i^k$ un **préfixe initial** (**Initial Prefix**) de l'histoire locale h_i contenant les premiers **k** événements de cette histoire.
- Soit σ_i^k l'**état** (**State**) du processus Pi juste après l'occurrence de l'événement e_i^k.
- Soit σ_i^0 l'**état initial** (**Initial State**) de Pi.



3 Définitions

3.1 Etat global (Global State)

L'**état global** d'un calcul distribué est défini par le tuple $\sum = (\sigma_1^{k_1}, \sigma_2^{k_2}, \ldots, \sigma_n^{k_n})$ qui représente l'ensemble des états locaux des différents processus participant au calcul distribué.

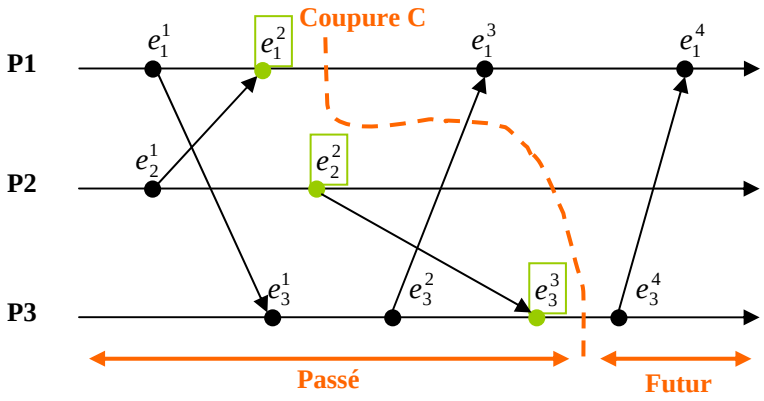
3.2 Coupure (Cut)

Une **coupure** (**Cut**) d'un calcul distribué est un **sous-ensemble C** de son histoire globale **H**, qui inclut un préfixe initial de chacune des histoires locales.

Une coupure $\boldsymbol{C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_n}}$ est définie par un tuple **(c_1, c_2, …, c_n)**, où c_i est le nombre d'événements de l'histoire locale h_i. Elle correspond à l'état global $(\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots, \sigma_n^{c_n})$

L'ensemble des événements $\boldsymbol{(e_1^{c_1}, e_2^{c_2}, \ldots, e_n^{c_n})}$ inclus dans la coupure **(c_1, c_2, …, c_n)** définit la **frontière** (**Frontier**) de la coupure.

Exemple



La coupure C est définie par le tuple (2, 2, 3) et sa frontière est définie par l'ensemble d'événements (e_1^2, e_2^2, e_3^3)

La notion de coupure est utilisée pour évaluer une **propriété globale** sur un calcul distribué (**Ex.** x=10 et y < 0, interblocage, calcul terminé, . . . etc.). Une stratégie générique pour l'évaluation d'une propriété globale consiste en expression de cette propriété sous forme de **prédicat Φ** qui sera **évalué** sur un **état global ∑**.

Une stratégie naïve pour le calcul d'un état global

Soit un processus P0, en dehors du système, demandant à chaque processus Pi son état local σ_i. Le processus construit l'état global $\sum = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ et évalue Φ sur $\sum$.

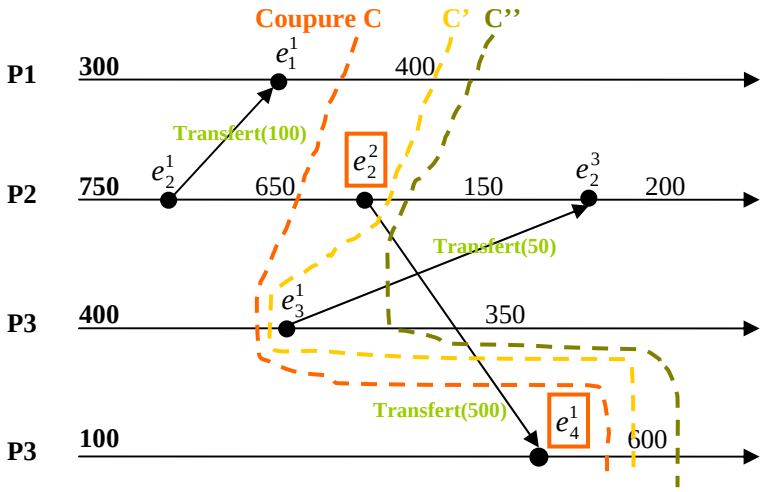
Exemple

Les processus gèrent des comptes bancaires. Chaque processus gère un seul compte.

Invariant (Prédicat à évaluer)

La somme d'argents qui se trouvent dans les comptes ne doit jamais dépasser la somme initiale.

Considérant l'état global qui correspond à la coupure C définie par le tuple (1, 1, 0, 1).



L'état global est défini par l'ensemble des états locaux (400, 650, 400, 600). La somme des comptes correspondant à cet état est égale à 2050 qui est supérieure à la somme initiale (=1550).

Ceci est dû à l'événement e_4^1 de la coupure C qui correspond à la réception d'un message dont l'émission (i.e. événement e_2^2) est effectuée dans le futur. On parle dans ce cas d'une **coupure incohérente** (**Inconsistent Cut**).

3.3 Coupure cohérente (Consistent Cut)

Une coupure C est dite cohérente ssi pour tous événements e et e', on a :

$$(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$$

Exemple

La coupure **C'** est cohérente. L'état global, correspondant à **C'**, est défini par l'ensemble des états locaux (400, 150, 400, 600). La somme des comptes correspondant à cet état est égale à 1550 qui est égale à la somme initiale (=1550).

La coupure **C''** est cohérente. Cependant, la somme des comptes, correspondant à l'état global défini par cette coupure (400, 150, 350, 600), est égale à 1500 < 1550. Cette différence est due au non prise en compte de l'**état des canaux** dans le calcul de l'état global.

3.4 Etat global cohérent (Consistent Global State)

Un **état global cohérent** correspond à une **coupure cohérente** et est défini par l'**ensemble des états locaux** des différents processus participant au calcul distribué ainsi que les **états des différents canaux** reliant ces processus.

Exemple

L'état global cohérent, correspondant à la coupure **C''**, est défini par l'ensemble des états locaux (400, 150, 350, 600) et des états de canaux (50).

4 Algorithme de Chandy & Lamport pour le calcul d'état global (The Chandy-Lamport Snapshot Algorithm)

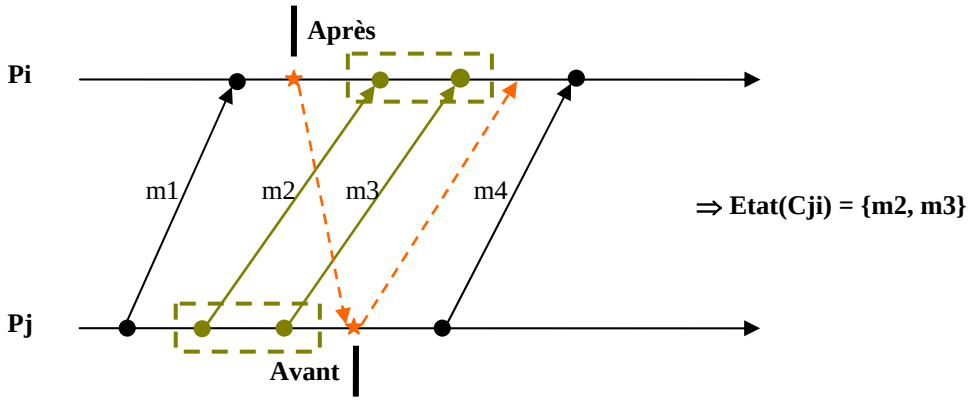
4.1 Hypothèses

- Les canaux de communication sont unidirectionnels et FIFO.
- Le système est fiable ; i.e. pas de plantage ou de perte de message.
- La topologie de connexion est fortement (voire totalement) connexe.
- L'algorithme peut être démarré par un des *n* processus concernés par ce calcul ou par un processus supplémentaire, dit **processus moniteur** (ou **collecteur**).

4.2 Principe

Le calcul de l'état global repose sur l'utilisation d'un message particulier, appelé « **marqueur** ». Le processus émetteur du message « marqueur », enregistre son état local et envoie le message « marqueur » sur ses canaux sortants. Le récepteur de ce message enregistre son état local ainsi que l'état de ses canaux entrants jusqu'à ce qu'il reçoit le message « marqueurs » sur ces canaux.

L'état du canal C_{ji} est défini par la séquence de messages envoyés par l'émetteur du marqueur (i.e. **Pj**), avant que son état soit enregistré, et non encore reçu quand l'état du récepteur (i.e. **Pi**) était enregistré.



Pour un processus Pj, l'algorithme est fini quand il a reçu un marqueur sur chacun de ses canaux. L'état enregistré pour Pj est composé de :

- Son état local
- Les états de tous ses canaux entrants

L'état global est donc constitué de l'ensemble des états enregistrés par les processus.

4.3 Structures de données

Chaque processus P_i, détient les structures de données suivantes :

- σ_i : permet de mémoriser son état local.
- **Etat_Memo(i)** : variable booléenne **initialisée** à faux et permet de dire si P_i a enregistré son état local.
- Pour tout processus P_j :
  - **Etat(C_{ji})** : liste de messages reçus par P_i depuis P_j.
  - **Recu_Marq(j)** : variable booléenne **initialisée** à faux et permet de dire si P_i a reçu un marqueur depuis P_j.

4.4 L'algorithme

```
Pinit

/*Pinit est le processus initiateur du calcul de l'état global. Pinit
est un des N processus*/

Lors de démarrage de l'algorithme

Faire

      σinit := état local de Pinit ;

      Etat_Memo(init) := vrai ;

      Pour tout canal entrant Cjinit Faire Etat(Cjinit) := ∅ Fait;

      Envoyer « Marqueur » à tous les autres processus ;

Fait ;

Pi

/*Pi est un des N processus concernés par le calcul distribué*/

Lors de réception d'un « Marqueur » depuis Pj sur Cji

Faire

   Si (Etat_Memo(i) = Faux)
```

```
        Alors
          Début
             σi := état local de Pi ;
             Etat_Memo(i) := vrai ;
             Pour tout canal entrant Cji Faire Etat(Cji) := ∅ Fait;
             Envoyer « Marqueur » à tous les autres processus ;
          Fin
    Fsi ;
    Reçu_Marq(j) := vrai ;
    Si (pour tout canal entrant Cji, Reçu_Marq(j))
      Alors Envoyer (σi , {Etat(Cji)}) à Pinit ;
      /*{Etat(Cji)} est l'ensemble des états des canaux de Pi*/
    Fsi
Fait
Lors de réception d'un message m ≠ « Marqueur » depuis Pj
Faire
    Si (Etat_Memo(i) = vrai) et (Recu_Marq(j) = faux)
      Alors Etat(Cji) := Etat(Cji) ∪ {m} ;
    Fsi
Fait
```

Pour un processus Pi, la phase d'enregistrement de l'état local et d'envoi du marqueur se fait de manière atomique ; c'est-à-dire, l'exécution de la séquence d'instructions ci-après doit se faire de manière atomique :

```
σi := état local de Pi ;
Etat_Memo(i) := vrai ;
Pour tout canal entrant Cji Faire Etat(Cji) := ∅ Fait;
Envoyer « Marqueur » à tous les autres processus ;
```

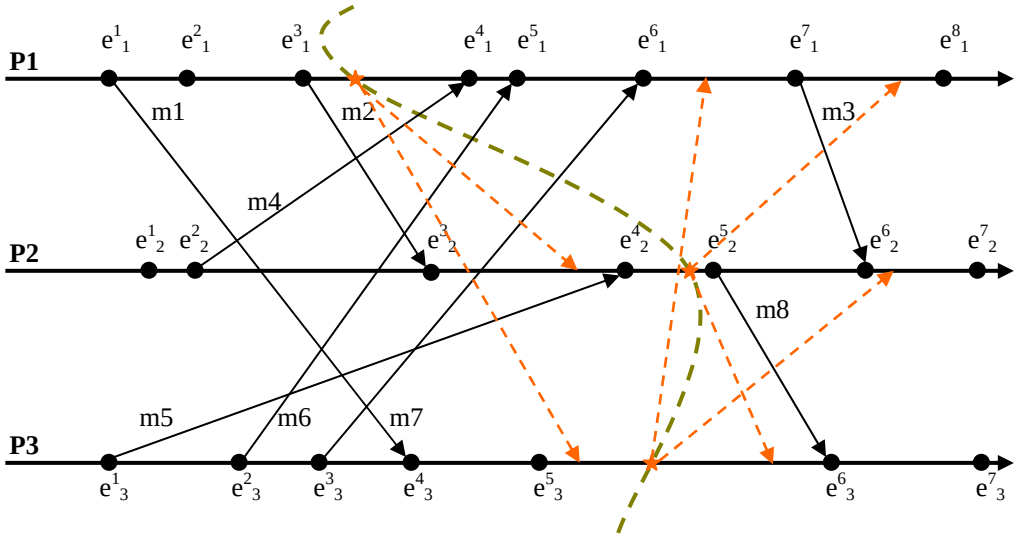
4.5 Propriétés

- L'algorithme définit une **coupure** dont la **frontière** est formée pour chaque processus par l'événement d'enregistrement de l'état local et de diffusion du marqueur aux autres processus.

- La **coupure** définie est **cohérente**. Ceci est dû au fait que les **canaux** sont **FIFO** (i.e. les messages arrivent dans l'ordre de leur émission, aucun message ne peut en doubler un autre). Donc, pas de message venant du futur.

- L'état global calculé correspond à un **état global cohérent**.

4.6 Exemple



Le processus P1 est le processus initiateur du calcul de l'état global.

La coupure, correspondant au calcul de cet état global, est définie par la frontière $(e_1^3 e_2^4 e_3^5)$

L'état enregistré par P1 est :

$\sigma_1 = \sigma_1^3,$
$Etat(C_{21}) = \{m_4\},\ Etat(C_{31}) = \{m_6, m_7\}.$

L'état enregistré par P2 est :

$\sigma_2 = \sigma_2^4,$
$Etat(C_{12}) = Etat(C_{32}) = \Phi.$

L'état enregistré par P3 est :

$\sigma_2 = \sigma_3^5,$
$Etat(C_{13}) = Etat(C_{23}) = \Phi.$

Par conséquent, l'**état global** obtenu est :

$$\sigma_1 = \sigma_1^3, \quad \sigma_2 = \sigma_2^4, \quad \sigma_3 = \sigma_3^5$$
$$Etat(C_{21}) = \{m_4\}, \; Etat(C_{31}) = \{m_6, m_7\},$$
$$Etat(C_{13}) = Etat(C_{23}) = Etat(C_{12}) = Etat(C_{32}) = \Phi$$