

Applications Concurrentes :

Conception,

Outils de Validation

ACCOV_B

Chapitre 2

PARADIGMES DE LA CONCURRENCE DES PROCESSUS

Paradigmes de la concurrence

exclusion mutuelle

la cohorte

producteurs et consommateurs

lecteurs et rédacteurs

le repas des philosophes

la terminaison d'un traitement coopératif

l'élection d'un coordinateur

la diffusion de messages

Interblocage

LES PARADIGMES DE LA CONCURRENCE

Briques de base pour toute étude, analyse ou construction de système ou d'application coopérative. ("design patterns")

PARADIGMES

- **Exemples-type qui permettent de modéliser des classes de problèmes réels fréquemment rencontrés et présents à tous les niveaux dans les systèmes et dans les applications concurrentes.**
- **Acceptés par la communauté pour leur capacité à fournir des schémas de conception**

ARCHÉTYPES

- **Solutions des paradigmes pouvant servir de schémas de construction de programmes.**

PRINCIPAUX PARADIGMES DE LA CONCURRENCE

- **l'exclusion mutuelle** qui modélise l'accès cohérent à de ressource partagées,
- **la cohorte** qui modélise la coopération d'un groupe de taille maximale donnée,
- **les producteurs-consommateurs**, exemple qui modélise la communication par un canal fiable,
- **les lecteurs-rédacteurs** exemple qui modélise la compétition cohérente,
- **le repas des philosophes** schéma qui modélise l'allocation de plusieurs ressources.
- **la terminaison d'un traitement coopératif**
- **l'élection d'un coordonnateur**
- **la diffusion de messages** fiable et avec ordre de réception (total, causal) identique pour tous les processus

Voir aussi le polycopié du cours Systèmes et réseaux informatiques

◆ exclusion mutuelle cohérence de suites d'actions concurrentes

EXEMPLE 1 : des processus P et Q produisent des suites de lignes pour une imprimante partagée

P1 P2 Q1 P3 Q2 P4 P5 Q3 P6 Q4 Q5 -- mauvais

Q1 Q2 Q3 Q4 Q5 P1 P2 P3 P4 P5 P6 -- bon

P1 P2 P3 P4 P5 P6 Q1 Q2 Q3 Q4 Q5 -- bon

Une impression parmi d'autres avec LES_MAUX_DE_L_IMPRESSION

L'homme est capable de faire ce qu'il est incapable d'imaginer.

Terre arable IMPRESSION Sa tête du songe!

DE DEUX POEMES sillonne Qui parle CONTEMPORAINS

de bâtir? la galaxie

de l'absurde J'ai vu la terre distribuée

Rene en de vastes espaces

Char et ma pensée n'est point distraite du navigateur

Saint-John Perse

EXEMPLE 2 : deux processus P et Q modifient une variable commune TOTAL, chacun faisant +1 sur TOTAL

{TOTAL= 100} P1 Q1 Q2 P2 Q3 P3 {TOTAL= 101} --mauvais

{TOTAL= 100} Q1 Q2 P1 P2 P3 Q3 {TOTAL= 101} --mauvais

{TOTAL= 100} Q1 Q2 Q3 P1 P2 P3 {TOTAL= 102} --bon

{TOTAL= 100} P1 P2 P3 Q1 Q2 Q3 {TOTAL= 102} --bon

EXEMPLE 3: SNCF avec section de ligne à voie unique. Trains dans un sens ou dans l'autre, mais pas dans les deux sens en même temps.

EXEMPLE 4: nom unique avec plusieurs demandes concurrentes

transactions concurrentes pour l'obtention d'un nom unique

SCi(xi) ; A1i : lire N; A2i : N := N + 1; A3i : Xi := N;

{N=100}(A11->A21->A31)->(A12->A22->A32) {N=102, X1 =101, X2=102}

{N=100}(A12->A22->A32)->(A11->A21->A31) {N=102, X1 =102, X2=101}

toute autre exécution est fautive, comme par exemple :

{N=100}(A11->A21->A12->A22->A31->A32) {N=102, X1 =102, X2=102}

◆ exclusion mutuelle et transactionnel

Dans le contrôle de concurrence en transactionnel, on veut maintenir la cohérence ("consistency", une des propriétés A.C.I.D)

Soit par exemple à maintenir l'invariant $Y=2X$ et des états cohérents $\{X=0,Y=0\},\{X=10,Y=20\},\{X=20,Y=40\},\{X=30,Y=60\},\dots$

EXEMPLE 5 : TRANSACTIONS T1 et T2 avec écritures cohérentes

T1 : $\{Y=2X\}$ T11 : écrire (X,10); T12 : écrire(Y,20) $\{Y=2X\}$
 T2 : $\{Y=2X\}$ T21 : écrire (X,30); T22 : écrire(Y,60) $\{Y=2X\}$

SPÉCIFICATION : T1 \rightarrow T2 ou T2 \rightarrow T1 mais pas T1 || T2

Traces :

$\{X=0,Y=0\}$ T1 \rightarrow T2 $\{X=30,Y=60\}$

$\{X=0,Y=0\}$ T2 \rightarrow T1 $\{X=10,Y=20\}$

$\{X=0,Y=0\}$ T11 \rightarrow T21 \rightarrow T22 \rightarrow T12 $\{X=30,Y=20\}$ incohérent $Y \neq 2X$

notons que T1 = (T11 || T12) et que T2 = (T21 || T22)

la spécification T1 \rightarrow T2 ou T2 \rightarrow T1 devient :

(T11 || T12) \rightarrow (T21 || T22) ou (T21 || T22) \rightarrow (T11 || T12)

regardons ce que donne (T11 || T12) || (T21 || T22)

$\{X=0,Y=0\}$

T1 || T2

$\{X=10,Y=20\}$ ou $\{X=30,Y=60\}$ ou $\{X=10,Y=60\}$ ou $\{X=30,Y=20\}$

EXEMPLE 6 : TRANSACTIONS P1 et P2 avec lectures et écritures : lectures cohérentes

P1 : $\{Y=2X\}$ P11 : $W := \text{lire}(X)$; P12 : $Z := \text{lire}(Y)$ $\{Y=2X, Z=2W\}$

P2 : $\{Y=2X\}$ P21 : écrire (X,30); P22 : écrire(Y,60) $\{Y=2X\}$

on note que P11 || P12 et que P21 || P22

SPÉCIFICATION : P1 \rightarrow P2 ou P2 \rightarrow P1 mais pas P1 || P2

Trace

$\{X=10,Y=20\}$ P1 \rightarrow P2 $\{X=30,Y=60, W=10, Z=20\}$

$\{X=10,Y=20\}$ P2 \rightarrow P1 $\{X=30,Y=60, W=30, Z=60\}$

concurrence cohérente :

Trace

$\{X=10,Y=20\}$ P11 \rightarrow P21 \rightarrow P22 \rightarrow P12 $\{X=30,Y=60, W=10, Z=60\}$

concurrence incohérente car $Z \neq 2W$

TERMINOLOGIE

• **RESSOURCE CRITIQUE** : entité partagée dont l'utilisation ne doit être faite que par un processus à la fois -en accès exclusif- : imprimante, COMPTE_CLIENT, bus ethernet, section à voie unique.

• **EXCLUSION MUTUELLE**: condition de fonctionnement garantissant à un processus l'accès exclusif à une ressource pendant une suite d'opérations avec cette ressource

• **SECTION CRITIQUE** : séquence d'actions d'un processus pendant lesquelles il doit être le seul à utiliser la ressource critique

P1 P2 P3 P4 P5 P6 ou Q1 Q2 Q3 Q4 Q5 de l'exemple 1

P1 P2 P3 ou Q1 Q2 Q3 de l'exemple 2

REMARQUE : une exclusion mutuelle, une section critique pour chaque ressource critique utilisée.

Soit SC_i la section critique pour le processus P_i

• spécification : $\forall P_j, P_k$ voulant entrer en section critique

$SC_j \rightarrow SC_k$ ou $SC_k \rightarrow SC_j$ mais pas $SC_j \parallel SC_k$

PROBLÈMES :

Attention aux sections critiques emboîtées, car cela peut conduire à interblocage (problème de sûreté)

Pour cette raison, on doit interdire des appels récursifs d'une section critique (on doit faire l'appel récursif à l'intérieur de la SC , mais pas demander récursivement à entrer en SC)

HYPOTHÈSES POUR L'EXCLUSION MUTUELLE

H1: Les vitesses relatives des processus sont quelconques.

H2: Les priorités ou les droits des processus sont quelconques.

H3: Tout processus sort de sa section critique au bout d'un temps fini; en particulier ni panne ni blocage perpétuel ne sont permis en section critique.

SPÉCIFICATION DE COMPORTEMENT

C1: Un processus au plus en section critique. Peu importe l'ordre d'accès.

C2: Pas d'interblocage actif ou passif. Si aucun processus n'est en section critique et que plusieurs processus attendent d'entrer dans leur section critique, alors l'un d'eux doit nécessairement y entrer au bout d'un temps fini. (contreexemple: déclaration et politesse)

C3: Un processus bloqué en dehors de section critique, en particulier un processus en panne, ne doit pas empêcher l'entrée d'un autre processus dans sa section critique. (contreexemple: accès à l'alternat)

C4: La solution ne doit faire d'hypothèse ni sur les vitesses, ni sur les priorités relatives des processus. De ce point de vue la solution doit être symétrique.

RÈGLES D'ATTENTE--PARFOIS CONTRADICTOIRES

R1: Pas de coalition voulue ou fortuite entraînant la famine d'un processus. Aucun processus qui demande à entrer en section critique ne doit attendre indéfiniment d'y entrer. C'est la propriété d'équité. (Danger présent avec des priorités fixes)

R2: Le processus le plus prioritaire du système entre en premier en section critique

R3: Pas de règle particulière d'équité ou de temps de réponse. C'est le cas le plus fréquent

REMARQUE 1. Toute solution viable doit supposer le respect permanent de l'hypothèse H3 ou plutôt en reconstituer les conditions, en cas de panne matérielle ou logicielle. Ce rôle est assuré par le noyau du système. Par exemple, les fichiers ouverts par un programme utilisateur qui est arrêté pour cause d'erreur ou de dépassement de temps doivent être fermés. Quand le noyau tombe en panne, le système est aussi en panne.

REMARQUE 2. Il est souhaitable que l'exécution d'une section critique soit traitée comme une action atomique: elle est exécutée soit entièrement, soit pas du tout.

SPÉCIFICATION FORMELLE DE L'EXCLUSION MUTUELLE

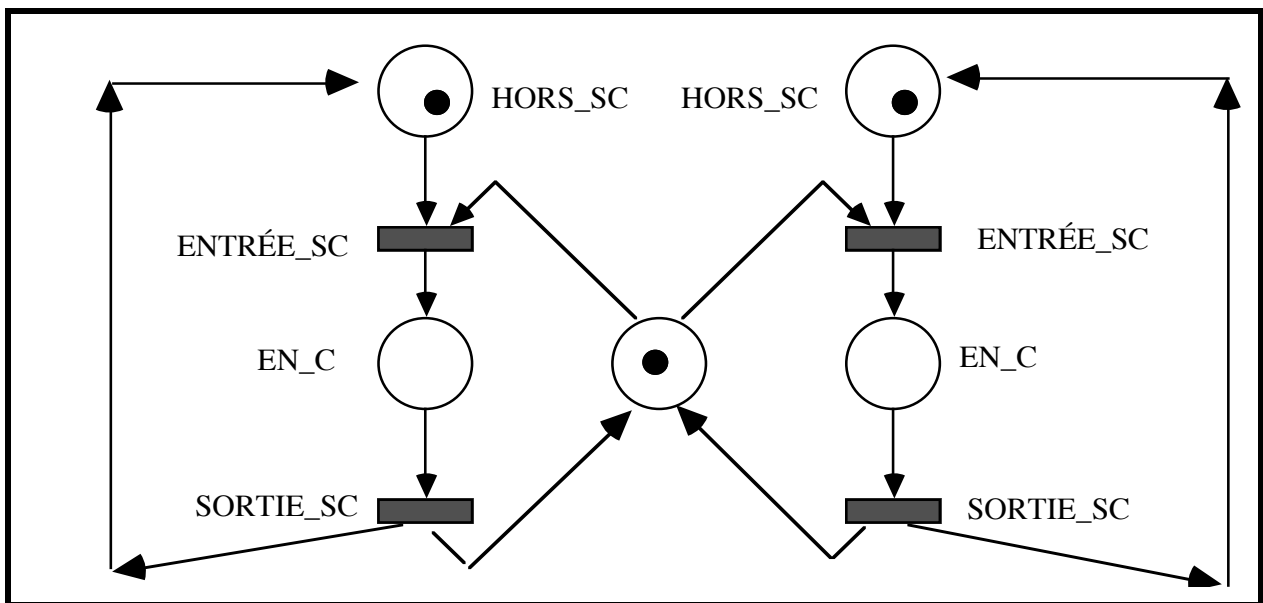
1. SCHÉMA D'EXECUTION

Protocole d'initialisation

procédures de contrôle pour entrer et sortir de section critique :

ENTREE_SCi;
 section_critique_i;
 SORTIE_SCi;

2. RÉSEAU DE PETRI POUR DEUX PROCESSUS



◆ importance de l'exclusion mutuelle

- impose l'exécution séquentielle, la sérialisation, d'un sous-ensemble d'actions concurrentes
 - c'est souvent nécessaire pour la cohérence
 - c'est toujours suffisant pour la cohérence
- on peut utiliser les techniques de preuve de programmes séquentiels sur les actions en section critique
- la compréhension des problèmes, la programmation et les preuves des solutions sont moins complexes

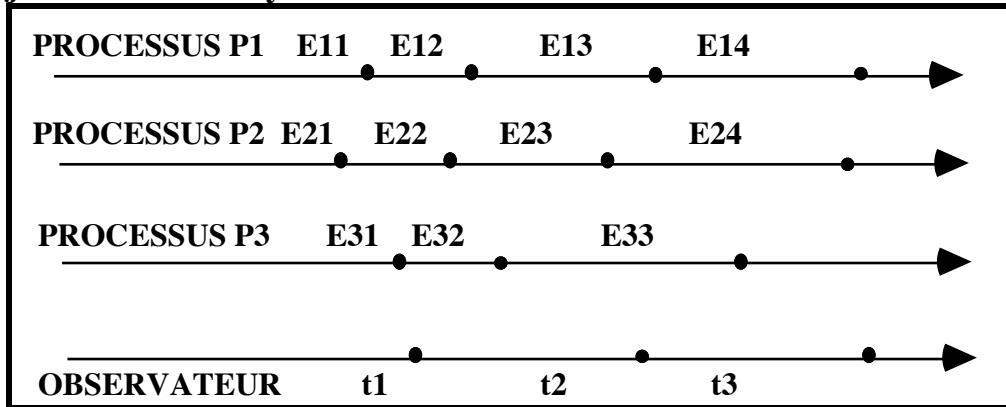
la combinatoire de la concurrence est grande :

Si chaque processus est une séquence de m états

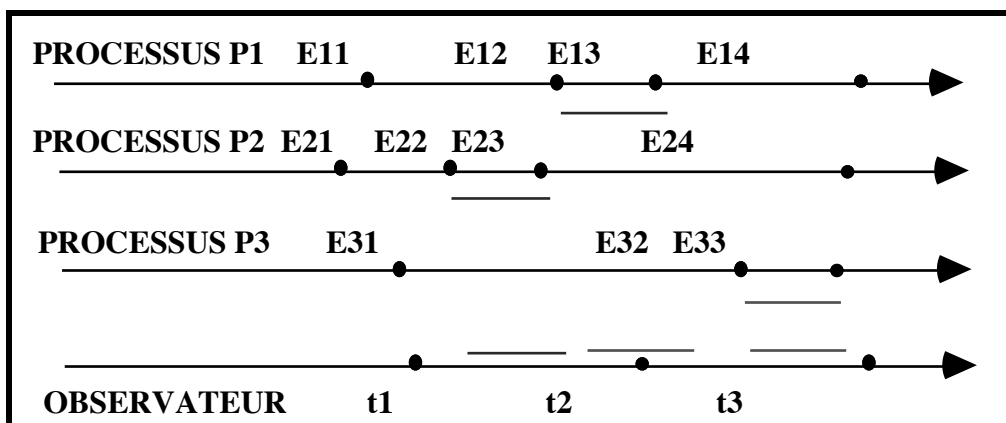
$E1 \rightarrow E2 \rightarrow \dots \rightarrow E_m$,

alors un système de n processus concurrents comprend m^n états

s'ajoute encore l'asynchronisme des états et des observations



si on contrôle la concurrence, on enlève de la complexité



- c'est la base des mécanismes de synchronisation
 - sémaphore, moniteur, objet protégé
 - gestion des queues de messages quand ils arrivent en parallèle (la scrutation séquentielle "polling" est une exclusion mutuelle)

◆ Exercices pour l'exclusion mutuelle**Exercice 1:**

théâtre et deux caisses d'accès : compter les entrées au théâtre

Exercice 2

La concurrence peut créer des incohérences s'il y a des invariants à conserver entre deux bases de données, par exemple $BD1(n) + BD2(n) = C$. On note n la version courante et cohérente.

Par exemple soit la transaction T1 qui ajoute X dans BD1(n) puis retranche X dans BD2(n), ce qui donne BD1(n+1) et BD2(n+1). Soit la transaction T2 qui lit BD2 puis BD1. La lecture de BD2(n) et BD1(n) est cohérente, celle de BD2(n+1) et BD1(n+1) l'est aussi puisque $BD1(n) + BD2(n) = C$. Mais si T2 lit BD2(n) puis BD1(n+1), c'est incohérent.

Spécifier les exécutions concurrentes valides de T1 et T2 :

T11 : $BD1 := BD1 + X$;

T12 : $BD1 := BD1 - X$;

T21 : $A := BD1$;

T22 : $B := BD2$;

$BD1 + BD2 = C$

$A + B = C$

EXCLUSION MUTUELLE MODULAIRE AVEC SÉMAPHORE

```

procedure Exclusion_Mutuelle_Modulaire_Par_Semaphore is
    -- valable quel que soit le nombre de processus

    package Compte is
        procedure Section_critique;
    end Compte;

    package body Compte is
        -- on encapsule la donnée critique et son sémaphore de contrôle
        COMPTE_CLIENT : Integer := 0;      -- variable commune persistante
        S : Semaphore;      -- à initialiser avec une seule autorisation E.S = 1

        procedure body Section_critique is
            X : Integer := 0; -- variable locale
        begin
            P(S) ; -- ENTREE_SC1
            X := COMPTE_CLIENT;      -- P1
            X := X + 1;                -- P2
            COMPTE_CLIENT := X;      -- P3
            V(S); -- SORTIE_SC1
        end Section_critique ;

    begin
        E0(S,1) ; -- initialisation du sémaphore avant utilisation du paquetage
    end Compte; -- fin du module partagé, unité de bibliothèque

    task type TP ;

    task body TP is
        begin
            for I in 1 .. 16 loop
                actions_hors_section_critique;
                Compte.Section_critique;
            end loop;
        end TP;

    P,Q, R ; TP ; -- on déclare des processus
begin les processus P, Q, R et le main sont concurrents
    null;
end Exclusion_Mutuelle_Modulaire_Par_Semaphore ;

```

EXCLUSION MUTUELLE MODULAIRE EN JAVA

```

public class Compte {
    // on encapsule la donnée critique, contrôle d'exclusion mutuelle implicite
    private int COMPTE_CLIENT = 0
        // variable locale à chaque objet instancié de la classe
        // pour une variable globale à la classe, il faudrait static
        // mais cela n'est pas permis avec synchronized – à vérifier
    public synchronized void Section_critique (){
        int X = COMPTE_CLIENT;    -- P1
        X := X + 1;                -- P2
        COMPTE_CLIENT := X;       -- P3
    } // fin de Section_critique ;
} // fin de la classe Compte

public class TP extends thread{
    private Compte compte;
    TP(Compte compte){this.compte = compte;}
    public void run(){
        for (int I=1; I <16; I++){
            actions_hors_section_critique;
            Compte.Section_critique;
        }
    } // fin de TP

public class Exclusion_Mutuelle_Modulaire_En_Java {
    // valable quel que soit le nombre de processus
    static Compte c = new Compte();
    public static void main(String[] args) {
        // programme principal
        TP p = new TP(c); TP q = new TP(c);
        // on doit passer en paramètre les objets partagés
        p.start(); q.start(); // lance les processus
        try {
            p.join();
            q.join();
        } catch(InterruptedException e) {
            System.out.println("un fils ne repond pas");
        }
        System.out.println(c); // imprime le résultat
    } // fin du programme principal main
} // fin de Exclusion_Mutuelle_Modulaire_En_Java

```

EXCLUSION MUTUELLE MODULAIRE AVEC OBJET PROTÉGÉ EN ADA

```
procedure Exclusion_Mutuelle_Modulaire_En_Ada is
    -- valable quel que soit le nombre de processus

protected Compte is
    procedure Section_critique;
private
    COMPTE_CLIENT : Integer := 0;    -- variable commune persistante
    -- on encapsule la donnée critique
end Compte;

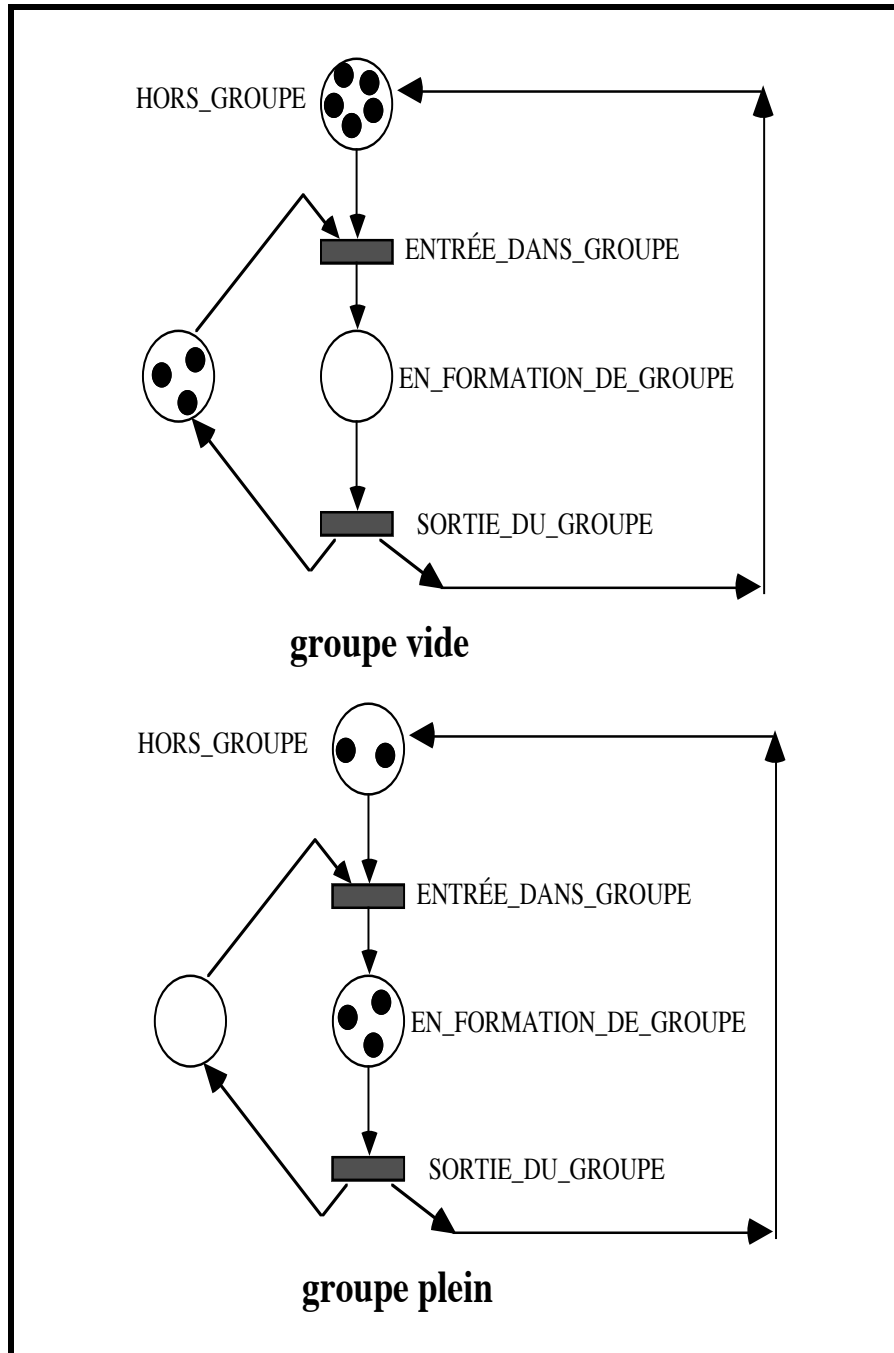
protected body Compte is
    procedure body Section_critique is
        X : Integer := 0; -- variable locale
    begin
        X := COMPTE_CLIENT;    -- P1
        X := X + 1;            -- P2
        COMPTE_CLIENT := X;    -- P3
    end Section_critique ;
end Compte;    -- fin de l'objet protégé

task type TP ;

task body TP is
begin
    for I in 1 .. 16 loop
        actions_hors_section_critique;
        Compte.Section_critique;
    end loop;
end TP;

P, Q, R ; TP ; -- on déclare des processus
begin les processus P, Q, R et le main sont concurrents
null;
end Exclusion_Mutuelle_Modulaire_En_Ada;
```

COHORTE OU GROUPE DE TAILLE MAXIMALE DONNÉE



RESEAU DE PETRI

CE GROUPE N'ACCEPTÉ QUE TROIS PROCESSUS AU PLUS

COOPERATION ENTRE PROCESSUS CONCURRENTS : LES PRODUCTEURS ET LES CONSOMMATEURS

◆ Problème classique en systèmes centralisés

EXEMPLE 1: des processus utilisateurs préparent des fichiers de texte et déposent des requêtes au service d'impression. Ces requêtes sont satisfaites par l'un des processus de ce service, à un moment où une des imprimantes est disponible, ni en panne, ni en maintenance.

EXEMPLE 2: un processeur d'entrée-sortie, un canal, un organe d'accès direct à la mémoire, reçoivent des données externes, caractère par caractère, et les engrangent à la vitesse des organes périphériques dans une zone de mémoire tampon. Quand le tampon est plein, un processus spécialisé est alerté pour qu'il puisse les exploiter .

EXEMPLE 3: un processus d'un site émetteur dans un réseau envoie des messages à destination d'un processus d'un site récepteur; celui-ci doit être prêt à recevoir ces messages; toutefois l'émetteur ne doit pas les envoyer plus vite que le récepteur ne peut les recevoir. Il faut assurer un contrôle de flux entre sites.

EXEMPLE 4: un service de messagerie permet aux abonnés d'un réseau d'envoyer du courrier électronique; un abonné édite son courrier et le dépose dans la boîte aux lettres du destinataire, que celui-ci soit en session ou non; un abonné lit son courrier, que l'émetteur soit ou non en session.

EXEMPLE 5: une cascade de relations de coopération peut aider à structurer une application. Ainsi dans la gestion d'un atelier de fabrication, un processus d'acquisition de messages reçoit les données des ponts roulants, des stocks et des machines outils et recueille des messages en provenance d'autres calculateurs de l'atelier. Ces données et ces messages sont déposés dans une file d'entrée; un processus analyseur les en extrait pour les filtrer, les mettre en forme et les déposer dans un tampon de requêtes; un processus interpréteur se charge de faire traiter le service demandé en appelant les sous-programmes adéquats; il prépare des réponses qu'il dépose dans une file de sortie ou bien il génère de nouvelles requêtes qu'il se dépose à lui-même pour plus tard dans le tampon des requêtes. Enfin un quatrième processus met en forme les réponses et regroupe dans un message toutes celles qui concernent le même destinataire.

LES PRODUCTEURS ET LES CONSOMMATEURS

LA TERMINOLOGIE

La coopération comprend des **producteurs** et des **consommateurs** qui se partagent un **tampon** ou une voie de communication, avec un nombre maximal, taille du tampon ou crédit de messages. Le **contrôle de flux** freine les producteurs s'ils vont trop vite pour les consommateurs.

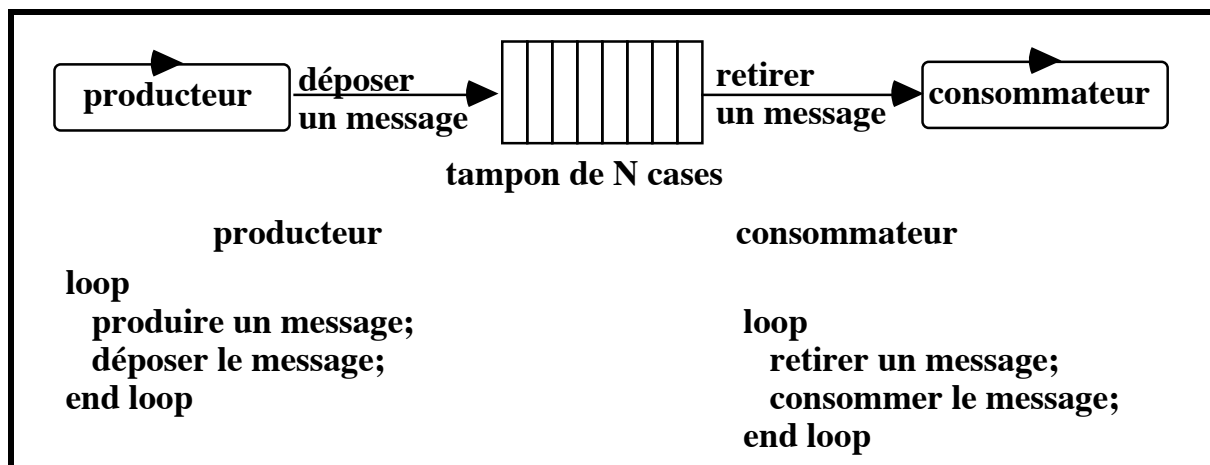
HYPOTHÈSES GÉNÉRALES

H1: Les vitesses relatives des processus sont quelconques.

H2: Les priorités des processus sont quelconques.

H3: Tout processus met un temps fini pour déposer ou retirer un message; en particulier ni panne ni blocage perpétuel pendant ces opérations.

SPÉCIFICATION DE COMPORTEMENT



OBJECTIF

asservir la vitesse moyenne de production à la vitesse moyenne de consommation en ralentissant le moins possible le producteur

HYPOTHÈSES DU PARADIGME PRODUCTEURS-CONSOMMATEURS

- 1) vitesses relatives quelconques, débit irrégulier**
- 2) tampon de taille fixe : N cases (une case = un message) et tampon vide initialement**
- 3) tout message déposé est lu une fois et une seule**
- 4) communication fiable**

CONSÉQUENCES

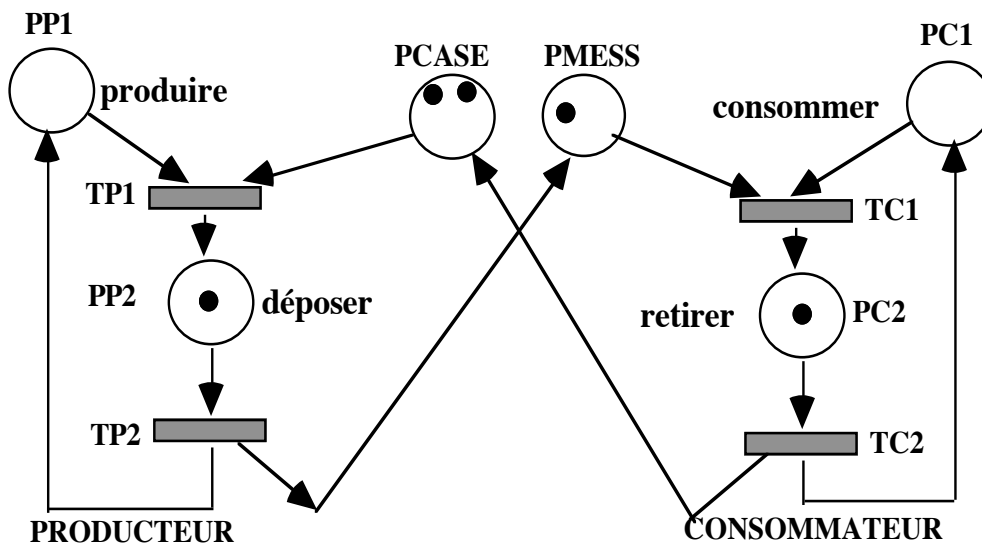
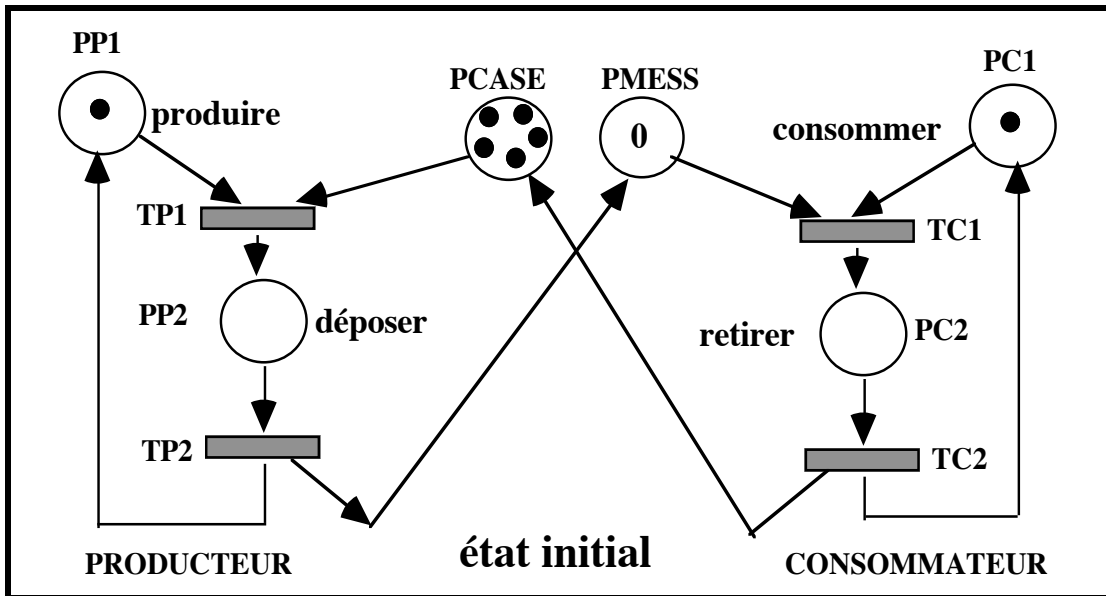
- 1) exclusion mutuelle d'accès aux messages (le message peut être lu ou écrit en plusieurs opérations**
- 2) le producteur s'arrête s'il veut déposer alors que le tampon est plein. Il est réveillé quand cette condition n'est plus vraie.**
- 3) le consommateur s'arrête s'il veut retirer alors que le tampon est vide. Il est réveillé quand cette condition n'est plus vraie.**
- 4) pas d'interblocage**

PARAMÈTRES D'IMPLANTATION

- nombre de producteurs et de consommateurs se partageant le tampon**
 - ordre de prélèvement des messages (premier déposé, dernier déposé, selon nature du message, ...)**
- structuration du tampon (tableau, liste chaînée, fichier,...)**

LES PRODUCTEURS ET LES CONSOMMATEURS

RÉSEAU DE PETRI



**état : deux cases vides, un message en dépôt,
un message en retrait, un message en stock**

PRODUCTEUR CONSOMMATEUR avec Sémaphores solution GÉNÉRIQUE

```

generic                    -- modèle de module
  N: Positive:= 5; type Message is private;
package UnTamponDe is
    procedure Deposer(X : in Message);
    Procedure Retirer(Y : out Message);
end UnTamponDe;      -- fin de l'interface

package body UnTamponDe is      -- corps
  Nplein, Nvide, Mutex : Semaphore;
  type Tampon is array(0..N-1) of Message;
  T : Tampon; -- variables persistantes
  Tete: Mod N := 0; Taille : Integer := 0;

  Procedure Deposer(X : in Message) is
  begin
    P(Nvide);
    P(Mutex);
    T(Tete+Taille) := X; Taille := Taille +1;
    V(Mutex);
    V(Nplein);
  end Deposer;

  Procedure Retirer(Y : out Message) is
  begin
    P(Nplein);
    P(Mutex);
    Y := T(Tete); Tete := Tete + 1; Taille := Taille - 1;
    V(Mutex);
    V(Nvide);
  End Retirer;

begin
  E0(Nplein, 0); E0(Nvide, N); E0(Mutex, 1);
end UnTamponDe;

```

PRODUCTEUR CONSOMMATEUR avec Sémaphores solution GÉNÉRIQUE (suite)

Procedure Main is

```
package Requete is new UnTamponDe(6, String);  
package Reponse is new UnTamponDe(2, Integer);
```

```
task Client ;
```

```
task body Client is
```

```
    R : String; Val : Integer;
```

```
begin
```

```
    loop
```

```
        Faire_Ailleurs_Preparation(R);
```

```
        Requete.Deposer(R);
```

```
        Reponse.Retirer(Val);
```

```
        Exploiter_Reponse ;
```

```
    end loop;
```

```
end Client;
```

```
task Serveur ;
```

```
task body Serveur is
```

```
    S : String; V : Integer;
```

```
begin
```

```
    loop
```

```
        Requete.Retirer(S);
```

```
        Traiter_Requete(S, V);
```

```
        Reponse.Deposer(V);
```

```
    end loop;
```

```
end Serveur;
```

```
begin
```

```
    null;
```

```
end Main;
```

// PRODUCTEUR CONSOMMATEUR// ProducerConsumer solution JAVA

```

class Buffer{
    protected final int max;
        // "protected" : accessible dans la classe et sous-classes
        // "final" : variable non mutable
    protected final Object[] data;           // tableau d'Objet
    protected int nextIn=0, nextOut=0, count=0;

    public Buffer(int max){
        this.max = max;
        this.data = new Object[max];
    }
    public synchronized void put(Object item)
        while (count == max)
            try {wait(); }
            catch (InterruptedException e){}
        data[nextIn] = item ;
        nextIn = (nextIn+1) % max;
        count++;
        notify();           // reveille un processus en attente s'il y en a
    }
}
    public synchronized Object get()
        while (count == 0) {
            try {wait(); }
            catch (InterruptedException e){}

        Object result = data(nextOut))) ;
        nextOut = (nextOut+1) % max;
        count--;
        notify();           // reveille un processus en attente s'il y en a
        return result;
    }
}
// noter l'emploi de notify car il n'y a qu'un blocage à la fois
// valable même si on a plusieurs producteurs ou consommateurs
    
```

```

// PRODUCTEUR CONSOMMATEUR// ProducerConsumer.java (suite)
class Producer implements Runnable{ // type processus producteur
    protected final Buffer buffer;
    Producer(Buffer buffer){ this.buffer=buffer; }
    public void run(){
        try{
            for (int j=1; j<=100; j++){
                Thread.sleep(500);
                buffer.put( new Integer(j) ); // crée un objet de classe Integer
            } // fin du bloc où est levée l'exception
        } catch (InterruptedException e) {return} // en réponse à l'exception
    }
} // fin de Producer

class Consumer extends Thread { // type processus consommateur
    protected final Buffer buffer;
    public Consumer(Buffer buffer){ this.buffer=buffer; }
    public void run(){
        try{
            for (int j=1; j<=100; j++){
                Integer p = (Integer) (buffer.get());
                int k = p.intValue(); // conversion vers le type int
                Thread.sleep(1000);
            } // fin du bloc où est levée l'exception
        } catch (InterruptedException e) {return} // en réponse à l'exception
    }
} // fin de Consumer

public class ProducerConsumer{ // objet programme principal
    static Buffer buffer = new Buffer(20); // tampon commun
    public static void main (String[] args) {
        Producer p = new Producer(buffer);
        Thread pt = new Thread(p); // car le producteur est Runnable
        Consumer c = new Consumer(buffer); // on passe l'objet à partager
        pt.start() ;
        c.start();
    } // fin de la méthode main
} // fin de la classe ProducerConsumer

```

PARADIGME PRODUCTEURS CONSOMMATEURS solution ADA

Bounded Buffer Example

```

generic                    -- modèle de module
  N: Positive:= 5; type Message is private;
package UnTamponDe is
    procedure Deposer(X : in Message);
    procedure Retirer(Y : out Message);
end UnTamponDe;      -- fin de l'interface

package body UnTamponDe is
    type Index is mod N ;
    type Message _Array is array (Index) of Message;
    protected Bounded_Buffer is
      entry Put (X : In Message);
      entry Get (X : out Message);
    private
      A : Message _Array;
      I, J : Index := 0;
      Count : Integer range 0 .. N := 0;
    end Bounded_Buffer;

    protected body Bounded_Buffer is
      entry Put (X : In Message) when Count < N is
      begin
          A(I) := X;
          I := I + 1; Count := Count + 1;
      end Put ;
      entry Get (X : out Message) when Count > 0 is
      begin
          X := A(J);
          J := J + 1; Count := Count - 1;
      end Get ;
    end Bounded_Buffer;

    procedure Deposer(X : in Message) is
    begin Bounded_Buffer.Put(X) ; end Deposer ;

    procedure Retirer(Y : out Message) is
    begin Bounded_Buffer.Get(Y) ; end Retirer ;

end UnTamponDe ;

procedure Main
-- etc... voir le Main dans solution avec semaphores

```

COMPETITION D'ACCES

LES LECTEURS ET LES REDACTEURS

LA SITUATION CONCRETE

EXEMPLE1: une entreprise veut gérer en temps réel une base d'informations; certains processus modifient cette base ou y font des adjonctions; d'autres processus ne font que la lire, sans modifications. On souhaite avoir le maximum de concurrence pendant ces lectures.

EXEMPLE 2: dans tout système informatique, on maintient de nombreux catalogues pour répertorier les clients, les abonnés, les groupes d'utilisateurs, les utilisateurs en session, les autorisations diverses, les processus, les ressources disponibles, les ressources en maintenance,...; des processus mettent ces catalogues à jour, en général à fréquence faible; d'autres processus les lisent, en général assez fréquemment; pour une meilleure efficacité, il est souhaitable de faire les lectures en parallèle.

EXEMPLE 3: tout système informatique comprend une horloge qui sert à dater l'arrivée et le départ d'un utilisateur, à mesurer son temps d'utilisation des ressources du système, à dater les versions successives d'un même fichier,...; cette horloge est incrémentée par un compteur à quartz, toutes les dix millisecondes par exemple; ce compteur active un processus qui fait progresser des indicateurs de année, mois, jour, heure, minute, seconde, 1/100e de seconde; d'autres processus lisent ces valeurs; l'efficacité prône une lecture concurrente; il faut toutefois éviter de lire pendant la progression des indicateurs, en particulier quand l'heure incrémentée est 31 jours, 23 heures, 59 minutes, 59 secondes, 99/100e, car comment garantir qu'on ne lira pas de valeur intermédiaire du calcul, comme 31jours, 23 heures, 0 minute, 0 seconde, 0/100e.

- concentrateur de lignes asynchrones en acquisition temps réel
- gestion de caches multiples dans les multiprocesseurs
- mémoire virtuelle répartie
- contrôle de transactions concurrentes

LA TERMINOLOGIE

La compétition d'accès met en présence des processus lecteurs et des processus rédacteurs qui se partagent des données. La synchronisation (contrôle de concurrence) garantit la cohérence des données lues

LECTEURS REDACTEURS solution par Semaphores

```

package MaBase is
  procedure Debut_Lire;
  procedure Fin_Lire;
  procedure Debut_Ecrire;
  procedure Fin_Ecrire;
end MaBase;

package body MaBase is
  Mutex_A, Mutex_L : Semaphore;
  NL : Natural := 0; -- variable persistante
  procedure Debut_Lire is
  begin
    P(Mutex_L);
    NL := NL + 1; if NL = 1 then P(Mutex_A); end if;
    V(Mutex_L);
  end Debut_Lire;
  procedure Debut_Ecrire is
  begin P(Mutex_A); end Debut_Ecrire;
  procedure Fin_Lire is
  begin
    P(Mutex_L);
    NL := NL - 1; if NL=0 then V(Mutex_A); end if ;
    V(Mutex_L);
  end Fin_Lire;
  procedure Fin_Ecrire is
  begin V(Mutex_A); end Fin_Ecrire;
begin
  E0(Mutex_A,1);E0(Mutex_L, 1); E0(Fifo, 1);
end MaBase;

with Mabase ; use Mabase ;
Procedure Main is
  task type Un_Client;
  Client : array(1..50) of Un_Client;
  task body Un_Client is
  begin loop
    Faire_Ailleurs_La_Preparation;
    MaBase.Debut_Ecrire;
    Acces_Exclusif_En_Ecriture;
    MaBase.Fin_Ecrire;
    Faire_Ailleurs_Un_Autre_Calcul;
    MaBase.Debut_Lire;
    Acces_Concurrent_Possible_En_Lecture;
    MaBase.Fin_Lire;
  end loop;
  end Un_Client ;
begin null; end Main;

```

//LECTEURS ET RÉDACTEURS solution Java //

```
public class Database{  
  public Database(){  
    readerCount = 0 ; Reading = false; Writing = false;  
  }  
  
  public synchronized int startRead() {  
    while (Writing == true) {  
      try{ wait(); }  
      catch(InterruptedException e) {}  
    }  
    ++readerCount; // readerCount = readerCount + 1  
    // le premier lecteur indique que la base est en accès en lecture  
    if (readerCount == 1) Reading = true;  
    return readerCount;  
  }  
  
  public synchronized int endRead() {  
    --readerCount // readerCount = readerCount -1  
    // le dernier lecteur indique que la base n'est plus en lecture  
    if (readerCount == 0) Reading = false;  
    notifyAll(); //réveille tous les processus en attente  
    return readerCount;  
  }  
  
  public synchronized int startWrite() {  
    while (Reading == true || Writing == true) {  
      try{ wait(); }  
      catch(InterruptedException e) {}  
    }  
    // tout rédacteur indique que la base est en accès en écriture  
    Writing = true;  
  }  
  
  public synchronized int endWrite() {  
    Writing = false;  
    notifyAll(); //réveille tous les processus en attente  
  }
```

// suite des LECTEURS RÉDACTEURS solution Java//

```
public static final int NAP_TIME = 5;

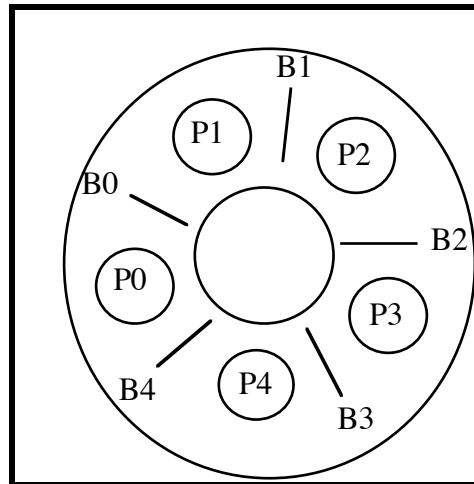
private int ReaderCount ;
private boolean Reading;           // indique que la base est en lecture
private boolean Writing ;         // indique que la base est en écriture

// les lecteurs et rédacteurs appellent cette méthode pour dormir
public static void napping(){
  int sleeptime= (int) (NAP_TIME * Math.random());
  try{Thread.sleep(sleeptime * 1000);} catch(InterruptedException e) {}
}
}

public class Reader extends Tread {
  public Reader(Database db) {server = db};
  public void run() {
    int c ;
    while (true) {
      Database.napping();
      c = server startRead();
      Database.napping();
      c = server endRead();
    }
  }
  private Database server;
}

public class Writer extends Tread {
  public Writer (Database db) {server = db};
  public void run() {
    while (true) {
      Database.napping();
      erver startWrite();
      Database.napping();
      server endWrite();
    }
  }
  private Database server;
}
```

LE REPAS DES PHILOSOPHES



HYPOTHÈSES

1. Chaque philosophe a une place fixe
2. Chaque assiette a une place fixe
3. Chaque baguette a une place fixe
4. accès à chaque baguette en exclusion mutuelle

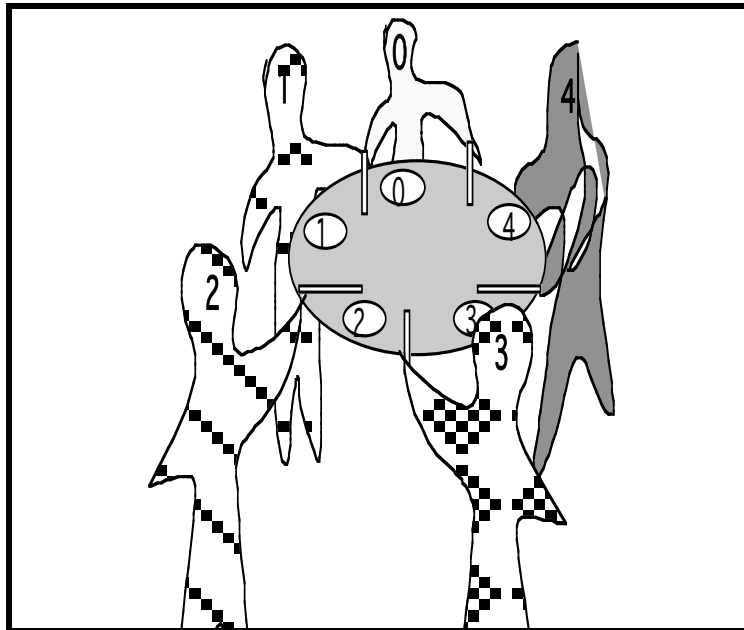
PROTOCOLE

1. Pour manger, un philosophe doit utiliser deux baguettes, la sienne et celle de son voisin de droite
2. Tout philosophe ne mange que pendant un temps fini, puis restitue les deux baguettes

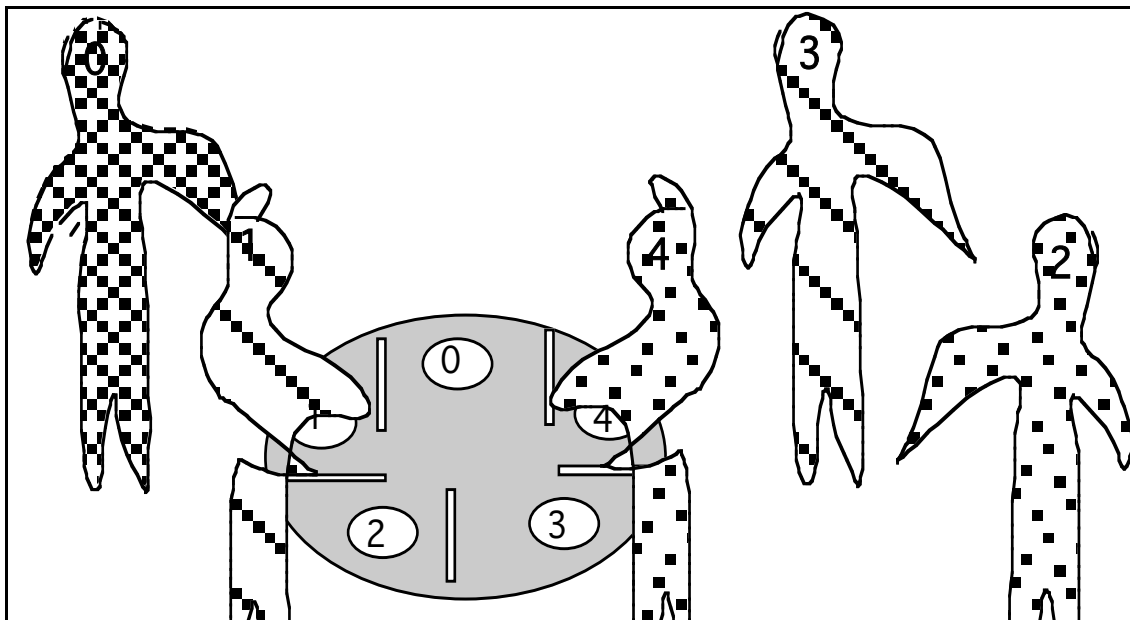
PROPRIÉTÉS

- 1 pas d'interblocage
- 2 pas de coalition (famine)

LE REPAS DES PHILOSOPHES



interblocage



famine : ((1,3)->(1,4)->(2,4)->1,4))*

LE REPAS DES PHILOSOPHES MainPhilo en ADA

```
with LesBaguettes ; use LesBaguettes ;
procedure MainPhilo is
    N : constant Integer := 5;
    type Philo_Id is mod N;
    -- simulation du comportement des philosophes

    procedure Pense (X : in Philo_Id) is begin nul ;end Pense;
    procedure Mange (X : in Philo_Id) is begin nul ;end Mange;

    package Nom is
        function Unique return Philo_Id;
    end Nom;
    package body Nom is
        Next_Id : Philo_Id := Philo_Id'Last; -- protégé par le package
        function Unique return Philo_Id is
        begin
            Next_Id := Next_Id + 1; -- addition modulo N
            return Next_Id;
        end Unique;
    end Nom;

    task type Philo (X : Philo_Id := Nom.Unique);

    Philosophes : array (Philo_Id) of Philo;

    task body Philo is
    begin
        for I in 1 .. 65 loop
            Pense (X);
            LesBaguettes.Prendre (X);
            Mange (X);
            LesBaguettes.Rendre (X);
        end loop;
    end Philo;

begin
    nul;
end MainPhilo ;
```

REPAS DES PHILOSOPHES

contrôle pour autorisation de manger avec Semaphores privés

```

package LesBaguettes is
  procedure Prendre (PourMoi : in Philo_Id);
  procedure Rendre (PourMoi : in Philo_Id);
end LesBaguettes;

package body LesBaguettes is
  type pdm is (pense, demande, mange);
  Etat : array(Philo_Id) of pdm := (others => pense);

  function test (x : Philo_Id) return boolean is
  begin
    return
      Etat(x) = demande
      and then Etat(x + 1) /= mange
      and then Etat(x - 1) /= mange;
  end test; -- +, -, sont des opérations modulo 5
  Mutex : Semaphore;
  Sempriv : array(Philo_Id) of Semaphore;
  procedure Prendre (PourMoi : in Philo_Id); is Ok : boolean;
  begin
    P(Mutex);
    Etat(PourMoi) := demande; Ok := test(PourMoi);
    if Ok then Etat(PourMoi) := mange; end if;
    V(Mutex);
    if not Ok then P(Sempriv(PourMoi)); end if;
    -- au réveil mangera sans prévenir
  end Demander;

  procedure Rendre (PourMoi: in Philo_Id) is
  begin
    P(Mutex); Etat(PourMoi) := pense;
    if test(PourMoi + 1) then -- voisin de droite
      Etat(PourMoi + 1) := mange; V(Sempriv(PourMoi + 1));
    end if; -- on a noté le nouvel Etat de PourMoi + 1
    if test(PourMoi - 1) then -- voisin de gauche
      Etat(PourMoi - 1) := mange; V(Sempriv(PourMoi - 1));
    end if; -- on a noté le nouvel Etat de PourMoi - 1
    V(Mutex);
  end Rendre;

begin E0(Mutex, 1);
  for Y in Philo_Id loop E0(Sempriv(Y), 0); end loop;
end LesBaguettes;

```

```
// DINER DES PHILOSOPHES//      Philo.java  
// Programme général //
```

```
public class Philo extends Thread{      // type processus Philo  
  
    private Server chops ;  
    private int x ;  
  
    Philo(int x ; Server : chops) {  
        this.x = x ;  
        this.chops = chops ;  
        new Thread(this).start();  
    }  
    private void thinking(){  
        System.out.println(x + " is thinking");  
        yield();  
    }  
    private void eating(){  
        System.out.println(x + " is eating");  
        yield();  
    }  
  
    public void run() { // code exécutable de Philo  
        while (true) {  
            thinking ;  
            chops.request(x) ;  
            eating ;  
            chops.release(x) ;  
        }  
    }  
  
public class Main {  
    public static void main (String[] args) {  
        final int N = 5;          // 5 philosophes 5 baguettes  
        Server chops = new Server(N) ;          // objet partagé de type Server  
        for (int i = 0 ; i < N ; i++) {new Philo(i, chops) ; }  
    }  
}
```

```
// Server.java ) avec possibilité de famine  
// allocation globale : available (x) and available(x + 1)
```

```
public final class Server {  
  
    private int N ;  
    private boolean available [];  
  
    Server (int N) {  
        this.N = N ;  
        this.available = new boolean[N] ;  
        for (int i =0 ; i < N ; i++) {  
            available[i] = true ;  
        }  
    }  
  
    public synchronized void request (int me) {  
        while  
        ( available [me] == false || available [(me + 1)% N] == false )  
        {  
            try { wait() ; }  
            catch (InterruptedException e) {}  
        }  
        available [me] = false ; available [(me + 1)% N] = false ;  
    }  
  
    public synchronized void release (int me) {  
        available [me] = true ; available [(me + 1)% N] = true ;  
        notifyAll ;           // on réveille tous le processus bloqués  
    }                       // car on n'a qu'une variable condition (implicite)  
}
```

**-- REPAS DES PHILOSOPHES une des solutions en Ada avec Objet protégé
-- protocole interne et une baguette après l'autre**

```
package LesBaguettes is
  procedure Prendre (PourMoi : in Philo_Id);
  procedure Rendre (PourMoi : in Philo_Id);
end LesBaguettes;

package body LesBaguettes is

  type Tableau_De_Booleens is array (Philo_Id) of Boolean;

  protected Baguettes is
    entry Saisir (Philo_Id); -- famille d'entrées
    procedure Restituer (X : in Philo_Id);
  private
    Disponible : Tableau_De_Booleens := (others => True);
    entry Completer (Philo_Id); -- autre famille d entrees
  end Baguettes;

  protected body Baguettes is
    entry Saisir (for I in Philo_Id) when Disponible (I) is -- famille d'entrées
    begin
      Disponible (I) := False;
      requeue Completer (I + 1);
    end Saisir;

    entry Completer (for I in Philo_Id) when Disponible (I) is
    begin
      Disponible (I) := False;
    end Completer;

    procedure Restituer (X : in Philo_Id) is
    begin
      Disponible (X) := True;
      Disponible (X + 1) := True;
    end Restituer;

  end Baguettes; -- fin du body de l'objet protégé Baguettes

  procedure Prendre (PourMoi : in Philo_Id) is
  begin Baguettes.Saisir (PourMoi); end Prendre;

  procedure Rendre (PourMoi : in Philo_Id) is
  begin Baguettes.Restituer (PourMoi); end Rendre;

end LesBaguettes;
```

NOUVEAUX PARADIGMES POUR LA CONCURRENCE EN UNIVERS REPARTI

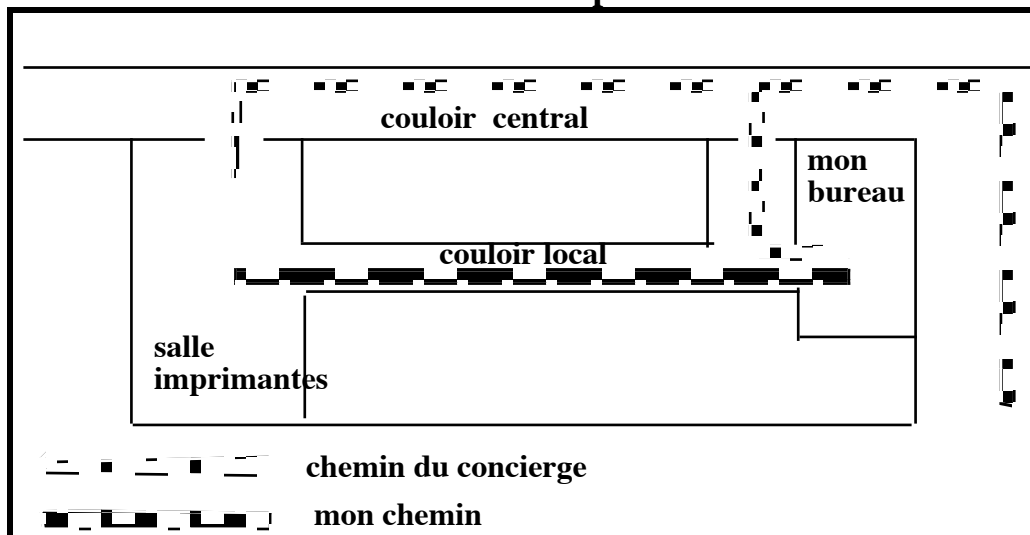
LA TERMINAISON D'UN TRAITEMENT COOPÉRATIF

- **création dynamique de processus pour un traitement coopératif :**
 - processus clients séquentiels, chacun en charge d'un aspect**
 - begin -- réalise une partie du travail; end client;**
 - processus serveurs cycliques : communication; gestion des ressources communes; présentation IHM, pilotes de périphériques**
 - begin loop -- code du serveur end loop; end serveur;**
- **détecter la fin du travail coopératif, c'est détecter que :**
 - tous les clients ont terminé (état terminé)**
 - tous les serveurs sont en attente de requête (état bloqué)**
 - il n'y a pas de requête en transit**
 - en Ada , utilisation de la clause "terminate" dans le "select"**
 - en réparti, importance des messages dans les canaux asynchrones**
- **Exemple**
 - **Le concierge du CNAM est chargé de couper l'électricité le soir après la fin des cours quand il n'y a plus d'enseignant dans le bâtiment Vaucanson.**
 - **Pour le concierge, il n'y a plus personne quand tous les bureaux et l'escalier sont éteints. Pour s'en assurer, il passe une fois dans les bureaux. S'il trouve un bureau vide et allumé, il éteint la lumière du bureau. Puis il sort vers le bureau suivant. Quand il a tout éteint, il coupe le courant.**
 - **Hélas, nous sommes plusieurs à nous retrouver souvent dans le noir, sans lumière. Pourquoi?**

Comment améliorer l'algorithme utilisé par le concierge.

◆ exemple de terminaison

Entre mon bureau et la salle des imprimantes, il y a deux couloirs, avec deux portes et deux chemins concurrents. Je prends l'un et le concierge prend l'autre et nous ne nous croisons pas.



• suite des actions

KA : je quitte mon bureau et parfois j'éteins la lumière

KB : je traverse le couloir local que j'allume

KC : j'allume la salle des imprimantes

KD : je sors de la salle des imprimantes après avoir éteint

CA : le concierge éteint la salle des imprimantes

CB : le concierge passe par le couloir central

CC : le concierge éteint mon bureau ou le voit éteint

KA -> KB -> KC -> KD || CA -> CB -> CC`

• une histoire (une trace, une exécution) explicative

{bureau allumé}

{bureau éteint}

{salle allumée} CA -> CB -> KA -> KB -> KC -> CC {salle allumée}

LA DIFFUSION DE MESSAGES

Dans un groupe de processus, chacun peut diffuser des messages aux autres processus du groupe.

Comment garantir que :

- chaque message soit bien diffusé à tout le groupe
 - il y ait un ordre entre les messages émis par des processus différents
 - cet ordre soit connu de tous
 - les messages soient reçus (ou réordonnables) selon cet ordre par tous les processus
- Ordres : causal, total quelconque, total et causal,
 - cas d'un ordre total de diffusion

$\forall P_i, \forall P_j, \forall P_k, \forall m_1, \forall m_2, \forall m_3$

DIFFUSION_i(m₁) -> DIFFUSION_j(m₂) -> DIFFUSION_k(m₃)

$\Rightarrow \forall P_a, \text{RECEPTION}_a(m_1) -> \text{RECEPTION}_a(m_2) -> \text{RECEPTION}_a(m_3)$

- illustration

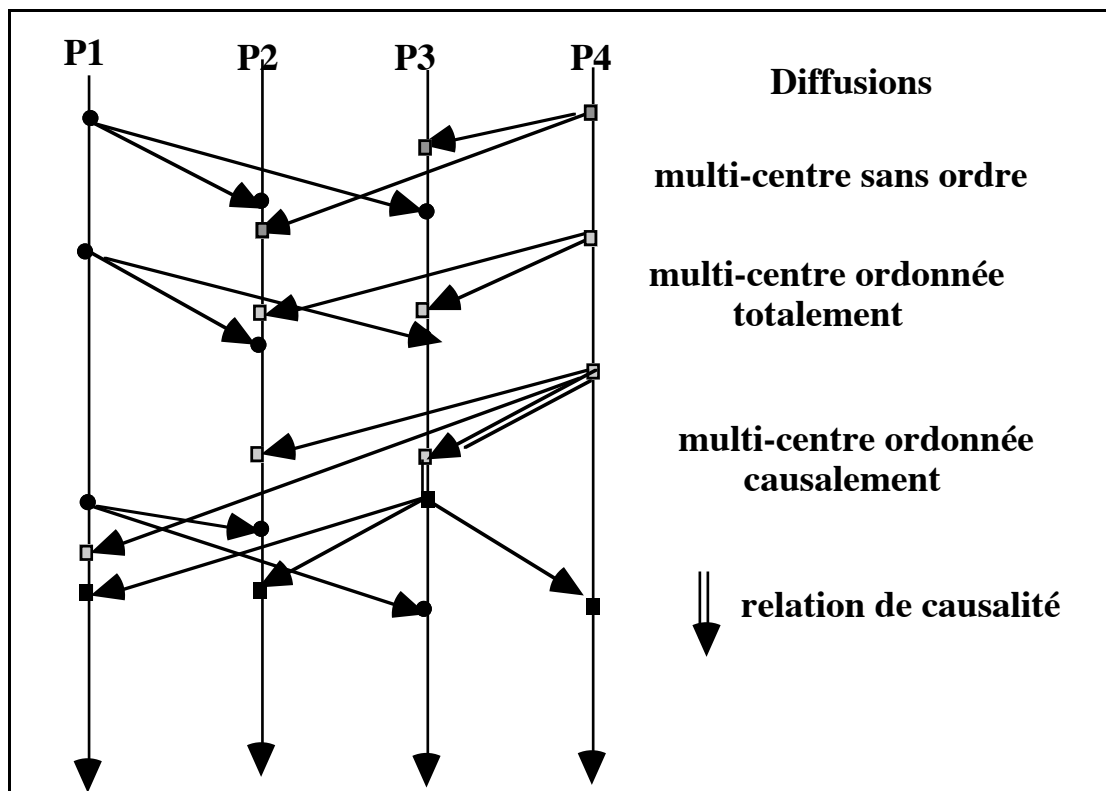


TABLEAU DE BORD ET CONNAISSANCE IMPARFAITE

TABLEAU DE BORD CHEZ LE CLIENT

MOI : CLIENT		LUI : MON SERVEUR	
mes demandes	9	requêtes acquittées	8
mes retours	7	réponses reçues	7
connaissance locale exacte	ce que je sais de lui, avec retard		

TABLEAU DE BORD CHEZ LE SERVEUR

LUI : MON CLIENT		MOI : SERVEUR	
ses demandes reçues	8	requêtes traitées	8
ses retours reçus	7	réponses faites	8
ce que je sais de lui, avec retard	connaissance locale exacte		

demandes \geq requêtes \geq réponses \geq retours

TABLEAU DE BORD CHEZ UN TIERS, UN OBSERVATEUR

LE CLIENT		LE SERVEUR	
demandes envoyées	7	requêtes traitées	8
retours reçus	7	réponses faites	7
ce que je sais de lui, avec retard	ce que je sais de lui, avec retard		

tous les chiffres sont connus avec retard

pas de connaissance commune instantanée

ELECTION D'UN COORDINATEUR

Dans une application répartie, il se peut qu'un processus soit utilisé comme coordinateur d'un groupe de processus

A l'initialisation du groupe, il faut élire ce coordinateur

Si son site tombe en panne, il faut élire un nouveau coordinateur

Décisions par consensus distribué

L'élection peut être lancée par plusieurs processus

Tous doivent se mettre d'accord pour n'en élire qu'un seul,

Mais il faut en élire un au bout d'un temps fini

Tous doivent être avertis de la fin de l'élection

et du nom du coordinateur

Contexte

Le groupe d'électeurs est fini, mais tous ne votent pas

chaque processus a un nom unique

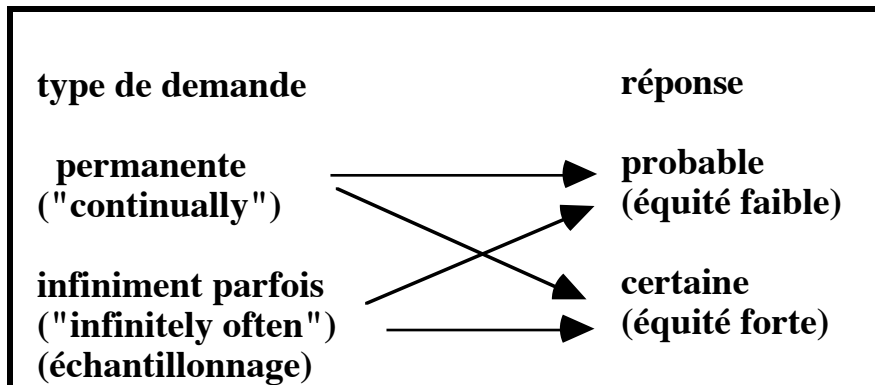
PROPRIETES GLOBALES D'UNE APPLICATION CONCURRENTE

◆ sûreté

"safety" : garantir que rien de "mauvais" ne peut se produire
 états interdits pour des raisons sécuritaires en fonctionnement normal ou
 en cas de panne
 respect de l'exclusion mutuelle
 absence d'interblocage (processus actifs ou passifs)

◆ vivacité

"liveness" : garantir que quelque chose de "bon" se produira
 équité faible ou forte



◆ famine

philosophes : si allocation globale des baguettes
 exclusion mutuelle : si file d'attente avec priorité
 producteurs consommateurs : si service en pile (LIFO) ou priorités
 lecteurs rédacteurs : si priorité aux lecteurs (ou aux rédacteurs)

◆ notions de priorité

importance (criticité) : critère issu des besoins applicatifs et s'appliquant à
 un sous-ensemble de processus traitant une requête
 urgence : critère associé à des échéances temporelles
 priorité (fixe ou variable): critère local à un processus utilisé comme moyen
 de gestion de ressource
 inversion de priorité,
 héritage de priorité

NOTIONS LIEES A LA TOLERANCE AUX PANNES ET A LA SURETE DE FONCTIONNEMENT

◆ atomicité

soit $A \rightarrow (B \rightarrow C \rightarrow D) \rightarrow \text{succès}$

soit $A \rightarrow \text{nil} \rightarrow \text{succès}$

soit on exécute toute la séquence $(B \rightarrow C \rightarrow D)$ soit rien
donc si une panne entre C et D, il faut revenir en A : point de reprise
(propriété A de A.C.I.D dans le transactionnel)

s'il y a concurrence comme par exemple :

$A \rightarrow (B1 \rightarrow C1 \rightarrow D1 \mid \mid B2 \rightarrow C2 \rightarrow D2 \mid \mid B3 \rightarrow C3 \rightarrow D3) \rightarrow \text{succès}$

ou $A \rightarrow \text{nil} \rightarrow \text{succès}$

il faut être sûr de la fin de D1 et D2 et D3 avant de continuer
c'est la phase de validation ("commitment")

Par exemple, on doit s'assurer que D1, D2, D3 laissent la base de données dans un état cohérent

◆ pannes temporelles

notion d'échéance et de fenêtre temporelle de validité d'un résultat, d'un calcul, d'une commande envoyée à l'organe piloté

exemple : cas d'un avion qui atterrit sur un porte-avion : il faut couper les gaz au bon moment

trop tôt l'avion va à la balle

trop tard l'avion va à la balle

au bon moment, il atterrit sur le pont et est freiné par le filet

◆ autres notions non développées

- protection ou contrôle d'accès

méfiance mutuelle, cheval de Troie, intrusions

- pannes malicieuses multiples

généraux byzantins

(pas de solution asynchrone, il faut une suite d'étapes synchronisées)

CONDITIONS D'APPARITION DE L'INTERBLOCAGE

1. Accès exclusif à chaque ressource
2. Les clients (en général, les processus) qui demandent de nouvelles ressources
 - gardent celles qu'ils ont déjà acquises
 - attendent l'arrivée de leur demande
3. Pas de réquisition possible des ressources déjà allouées
4. Les clients en attente des ressources déjà allouées forment une chaîne circulaire d'attente.

INTERBLOCAGE

- ◆ interblocage dû à une attente circulaire de ressources
 - LIVELOCK (processus actif) : "attente moteur en marche"
 - DEADLOCK (processus bloqué) : "attente moteur coupé"
- ◆ interblocage dû à une attente circulaire de messages sur l'un des N canaux d'entrée de message (= attente de l'une ou l'autre ressource)
- interblocage = état stable: une fois en interblocage, le système y reste
- politiques pour l'interblocage
 - détection suivie de guérison chirurgicale
 - prévention (statique): allocation globale ou classes ordonnées
 - évitement (dynamique): algorithme du banquier, transactions ordonnées
- ensemble de solutions pour le paradigme du repas des philosophes
 1. Allocation globale des deux baguettes (mais famine)
 2. Classes ordonnées : Baguette de plus petit numéro d'abord
 3. Prévention dynamique
 - On s'arrange pour qu'un philosophe au moins puisse toujours recevoir sa deuxième baguette : un philosophe doit manger assis et prendre d'abord une chaise. Or il n'y a que quatre chaises...
 4. Détection d'interblocage. Guérison brutale
 - On sacrifie un philosophe, et on lui reprend une baguette
 5. Prévention pessimiste par estampillage des transactions (repas)
 - On date les transactions. Quand un philosophe demande une baguette déjà allouée à un autre philosophe, il n'attend que si la date de sa transaction est plus ancienne que celle de l'utilisateur actuel. Sinon, il n'attend pas, rend l'autre baguette et recommence sa transaction.

PRÉVENTION STATIQUE PAR PRÉCAUTION PRÉALABLE

On se donne assez de ressources pour qu'il n'y ait jamais interblocage.

Avec

$$X > \Sigma(C_i - 1)$$

alors, même au pire cas, celui où tous les processus sont bloqués avec le maximum de ressources qu'ils peuvent ne pas rendre, $(X - \Sigma(C_i - 1))$ processus ne sont jamais bloqués

Le test du banquier est inutile (aucun test n'est à faire).

=====

PRÉVENTION DYNAMIQUE PAR L'ALGORITHME DU BANQUIER CAS D'UNE CLASSE DE RESSOURCES

- Chaque processus annonce le max du total de ses demandes
- Examen de la requête du processus P_k s'il peut être servi $R \geq req(k)$
Après service, il aurait A_k . Cela garde-t-il le système fiable?
 - Trier les processus par Dist croissant : $Dist_j = annonce_j - alloc_j$
 - on regarde successivement tous les processus selon cet ordre pour voir s'ils peuvent atteindre leur annonce et un processus P_j qui l'atteint rend ses ressources au profit des suivants d'où :

$$Total := Total + A_j$$

- On itère sur tous les processus et à chaque itération,
 - deux tests, l'un sur k , l'autre sur candidat = premier(T)
 - un seul test sur candidat car

$$Dist(candidat) > Total \Rightarrow Dist(succ(candidat)) > Total$$
- • échec si $Dist(premier(T)) > Dist(k)$
-- car aucun des succ(i) ne peut faire mieux que i
- • succès si $Dist(k) \geq Total$ -- car on part d'un état fiable

Complexité en $O(n)$ si on ne compte pas le tri, en $O(n \log n)$ avec le tri

ALGORITHME POUR UNE CLASSE DE RESSOURCES

```

type IdProc is range 1..n; type Matrice is array(IdProc) of Natural ;
type Permutation is array(IdProc) of IdProc;

```

```

C, A : Matrice; Dist : Matrice := C - A;

```

```

    -- contiennent les valeurs du nouvel état à tester

```

```

R : Natural ;    -- résidu disponible après allocation à Pk

```

```

Rang : Permutation; -- suite de processus à valeurs Dist croissantes:

```

```

    -- Dist(rang(1)) ≤ ... ≤ Dist(rang(n))

```

```

-- l'allocateur passe A, Dist, Rang et R

```

```

-- qui correspondent à l'état résultant après allocation à Pk

```

```

function EtatFiable(A, Dist: Matrice; Rang: Permutation; R: Natural;
                    k: IdProc) return boolean is

```

```

    Total : Natural := R ; -- R une fois le processus k servi

```

```

    candidat, j : IdProc := IdProc'First; -- pour créer la suite fiable S

```

```

begin

```

```

    j := 1 ; candidat:= Rang(j) ; -- le candidat a plus petite Dist

```

```

    while Dist(k) > Total and Dist(candidat) <= Total loop

```

```

        Total := Total + A(candidat);

```

```

            -- on prépare Total pour le processus suivant de S

```

```

        j := j + 1 ; -- le rang suivant dans l'ordre des Dist croissants

```

```

        candidat := Rang(j) ; -- le processus correspondant

```

```

    end loop;

```

```

    return (Dist(k) <= Total);

```

```

        -- si Pk est dans la suite fiable, l'état est fiable

```

```

        -- Si elle existe, la suite fiable comprend :

```

```

        -- une sous suite : Rang(1 .. j-1),

```

```

        -- Pk,

```

```

        -- les autres processus qu'on se dispense d'examiner

```

```

end EtatFiable;

```

CAS D'UNE CLASSE DE RESSOURCES

CAS PARTICULIER : toutes les annonces sont égales : $C_i = C_0$

Il suffit de garder assez de ressources pour que dans le nouvel état (après allocation de $req(k)$ au processus demandeur), un processus J au moins puisse encore atteindre son annonce max, c'est à dire $Dist(J) = 0$:

Condition : le nouvel état est fiable s'il y existe au moins un processus J tel que

$$Dist(J) \leq R$$

Le test du banquier devient, pour un processus k qui demande $req(k)$ ressources en plus et dont la distance actuelle est $Dist(k)$

```

function EtatFiable(k : in IdProc) return Boolean is
    OK : Boolean;
    -- on peut encore servir toute l'annonce d'au moins un processus
begin
    if req(k) > R then
        return False ; -- on ne peut même pas passer dans le nouvel état
    else
        R := R - req(k) ;
        -- on attribue les req(k) ressources à k, pour simuler le nouvel état
        Dist(k) := Dist(k) - req(k) ;

        for J in 1..IdProc loop
            OK := (Dist(J) <= R);
            -- un processus au moins pourra atteindre son annonce
            exit when OK; -- on en a trouvé 1, c'est bon et cela suffit
        end loop;

        R := R + req(k) ; -- on retourne à l'état initial
        Dist(k) := Dist(k) + req(k) ;
        return OK;
    end if;
end EtatFiable;
    
```