

## LES SÉMAPHORES

```

type Semaphore is record
    E : Integer;
    F : File_de_Processus_Bloqués;
end Semaphore;

procedure P(S : in out Semaphore) is
begin
    S.E := S.E - 1;
    if S.E < 0 then
        -- EGO pid du processus élu, exécutant P(S)
        état(EGO)=bloqué ; EGO ∈ S.F ;
        l'allocateur d'UC élit un autre processus
    end if;
end P;

procedure V(S : in out Semaphore) is
begin
    S.E := S.E + 1;
    if S.E <= 0 then
        LUI = pid du processus réveillé dans S.F
        LUI ∉ S.F ; état(LUI) = prêt;
    end if;
end V; -- LUI et EGO sont prêts tous deux;
    -- un seul est élu en monoprocesseur
    -- selon implantation de politique de choix

procedure E0(S : in out Semaphore; I in Natural) is
begin S.E := I; S.F := ∅; end E0; -- I ≥ 0

```

## PROPRIÉTÉS DES SÉMAPHORES

$$(R1) \quad \boxed{S.E = I - NP(S) + NV(S)}$$

$$\text{soit } NBLOC(S) = NP(S) - NF(S)$$

$$(R2) \quad \boxed{NBLOC(S) = \text{Max}(0, -S.E)}$$

**vrai initialement car**  $E0(S, I) \Rightarrow S.E \geq 0$   
**ensuite par récurrence, l'exécution conserve R2**  
 $\{(R2)\} P(S) \{(R2)\}$   
 $\{(R2)\} V(S) \{(R2)\}$

$$(R3) \quad \boxed{NF(S) = \text{Min}(NP(S), I + NV(S))}$$

$$\begin{aligned}
 NBLOC(S) &= NP(S) - NF(S) \\
 -NF(S) &= NBLOC(S) - NP(S) \\
 &= \text{Max}(-NP(S), -S.E - NP(S))
 \end{aligned}$$

**ou**  
 $NF(S) = \text{Min}(NP(S), S.E + NP(S))$   
**or d'après (R1) :**  
 $S.E + NP(S) = I + NV(S) \quad \text{c.q.f.d.}$

**notation :**  
 $NP(S)$  : nombre d'appels à la primitive  $P(S)$   
 $NV(S)$  : nombre d'appels à la primitive  $V(S)$   
 $NF(S)$  : nombre de sorties de la primitive  $P(S)$

**franchissement (la demande est comptée) :**  
 $P(S)$  when  $S.E \geq 0$ , soit when  $NP(S) \leq I + NV(S)$   
 $V(S)$  when True

**EXCLUSION MUTUELLE MODULAIRE**

avec un composant générique  
en Ada95

```
generic
  with procedure CodeSectionCritique;

package ExecEnSC is
  procedure AppelCodeSc;
end ExecEnSC;
```

```
with Semaphores; use Semaphores; --importation
package body ExecEnSC is
  MutexSC : Semaphore; -- d'exclusion mutuelle

  procedure AppelCodeSc is
  begin
    P(MutexSC); -- entrée en section critique
    CodeSectionCritique;
    V(MutexSC); -- sortie de section critique
  end AppelCodeSc;

begin
  E0(MutexSC, 1); -- initialisation du paquetage
end ExecEnSC;
```

```
with Text_Io; with ExecEnSC;
procedure Main is
```

```
  CompteClient : Integer := 0;
  procedure CodeExemple is
    X : Integer;
  begin
    X := CompteClient;
    X := X + 1;
    delay(0.01); --delai pour forcer commutation
    CompteClient := X;
  end CodeExemple ;
```

```
  package SC is new ExecEnSC(CodeExemple);
```

```
  task type Un_P; task body Un_P is
  begin
    for I in 1 .. 16 loop
      delay(0.01); --Actions_hors _section_critique;
      SC.AppelCodeSc; --code en section critique
    end loop;
  end Un_P;
```

```
  P, Q : Un_P; -- deux processus concurrents
```

```
begin
  delay(1.0); -- attendre la fin de P et Q
  -- on fera mieux quand on connaîtra les signaux
  Text_Io.Put(Integer'Image(CompteClient));
  -- on affiche la valeur finale qui doit être 32
end Main;
```

**COHORTE**

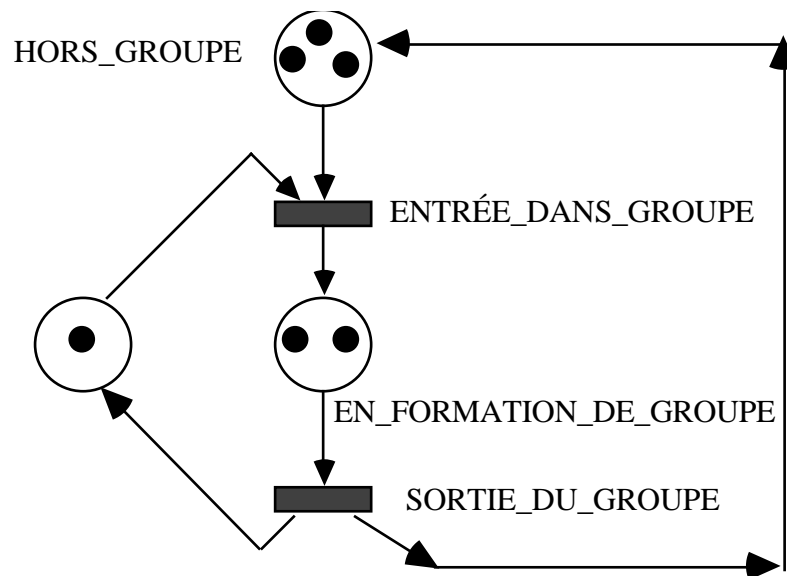
```

procedure Cohorte is
  S_C : Semaphore ;
  task type Processus;

  task body processus is
  begin loop
    Actions_hors_Groupe;
    P(S_C); -- entrée contrôlée dans le groupe
    -- Actions en cohorte de 3 processus au plus
    V(S_C); -- sortie annoncée du groupe
  end loop end processus;

  Application : array (1..5) of Processus;
  begin E0(S_C, 3); end Cohorte;

```

**COHORTE MODULAIRE**

```

package Mon is procedure Service; end Mon ;

package body Mon is
  S_C : Semaphore ;
  procedure body Service is
  begin
    P(S_C); -- entrée contrôlée dans le groupe
    -- Actions en cohorte de 3 processus au plus
    V(S_C); -- sortie annoncée du groupe
  end Service;
  begin E0(S_C, 3); end Mon ;

  procedure Cohorte_Modulaire is

  task type Processus;

  task body processus is
  begin loop
    Actions_hors_Groupe;
    Mon.Service; -- Actions en cohorte
  end loop end processus;

  Application : array (1..5) of Processus;

  begin
    null;
  end Cohorte_Modulaire;

```

## LOT DE RESSOURCES BANALISÉES

### Exemple

Chaque processus demande deux fois le droit d'utiliser une ressource. Puis quand il a les deux droits, il va rechercher dans Stock deux ressources disponibles ; cette recherche doit se faire en exclusion mutuelle

Stock contient 6 ressources, cela évite de tomber en interblocage. Si on avait un stock plus petit et toujours 5 processus, il pourrait y avoir un interblocage et il faudrait utiliser une méthode de prévention.

```

procedure Gestion_Cooperative is
  S_C, Mutex : Semaphore;
  Stock : array(1..6) of Integer := (others => 0);
  task type Processus;

  task body processus is
    I, J : Integer;
  begin loop
    Actions_hors_Groupe;
    P(S_C); P(S_C); -- droits pour deux ressources

    P(Mutex); I := 1;
    while Stock(I) /= 0 loop I := I + 1; end loop;
    Stock(I) := 1; J := I + 1;
    while Stock(J) /= 0 loop J := J + 1; end loop;
    Stock(J) := 1;
    V(Mutex);

    -- utilisation des ressources I et J acquises
    -- retour des ressources I et J

    P(Mutex); Stock(I):=0; Stock(J):=0; V(Mutex);
    V(S_C); V(S_C); -- retour des droits
  end loop end processus;

  Application : array (1..5) of Processus;

begin
  E0(S_C, 6); E0(Mutex, 1);
end Gestion_Cooperative;

```

## LOT DE RESSOURCES BANALISÉES solution modulaire

## SOLUTION MODULAIRE (suite)

```

package Deux is
  procedure Demander(I, J : out Integer);
  procedure Rendre(I, J : in Integer);
end Deux ;

package body Deux is
  S_C, Mutex : Semaphore;
  Stock : array(1..6) of Integer := (others => 0);

  procedure body Demander(I, J : out Integer) is
  begin
    P(S_C); P(S_C); -- droits pour deux ressources
    P(Mutex); I := 1;
    while Stock(I) /= 0 loop I := I + 1; end loop;
    Stock(I) := 1; J := I + 1;
    while Stock(J) /= 0 loop J := J + 1; end loop;
    Stock(J) := 1; V(Mutex);
  end Demander;

  procedure body Rendre(I, J : in Integer) is
  begin
    P(Mutex); Stock(I):=0; Stock(J):=0; V(Mutex);
    V(S_C); V(S_C); -- retour des droits
  end Rendre;

begin E0(S_C, 6); E0(Mutex, 1); end Deux ;

```

```

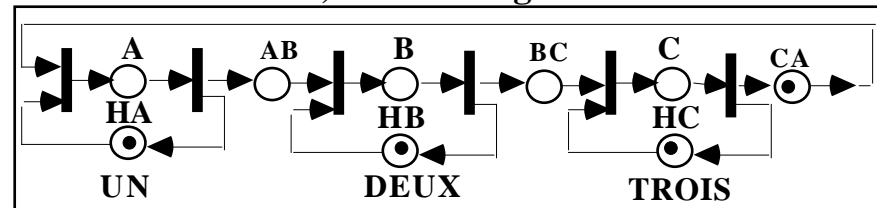
procedure Gestion_Modulaire is

  task type Processus;
  Application : array (1..5) of Processus;

  task body processus is
    X, Y : Integer; -- Id des ressources
  begin
    loop
      Actions_hors_Groupe;
      Deux.Demander(X, Y);
      Utiliser_les_Ressources; -- deux ressources
      Deux.Rendre(X, Y);
    end loop
  end processus;

begin null; end Gestion_Modulaire;

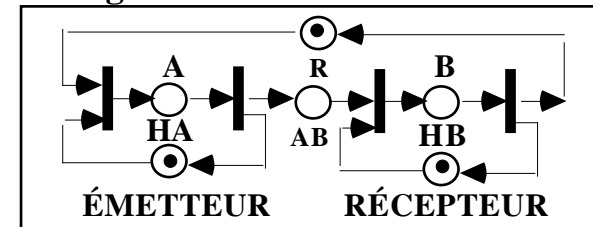
```

**PASSAGE DE TÉMOIN****1) envoi de signaux**

```
generic
package Un_Signal is
  procedure Envoyer;
  procedure Attendre;
end Un_Signal ;
```

```
generic
package Une_SC is
  procedure Entree ;
  procedure Sortie ;
end Une_SC ;
```

```
with Un_Signal ;
procedure Main is
  package AB is new Un_Signal;
  package BC is new Un_Signal;
  package CA is new Un_Signal;
  task UN is
    begin loop
      CA.Attendre; A; AB.Envoyer; HA;
    end loop; end UN;
  task DEUX is
    begin loop
      AB.Attendre; B; BC.Envoyer; HB;
    end loop; end DEUX;
  task TROIS is
    begin loop
      BC.Attendre; C; CA.Envoyer; HC;
    end loop; end TROIS;
  begin CA.Envoyer; -- déclenche le processus UN
end Main;
```

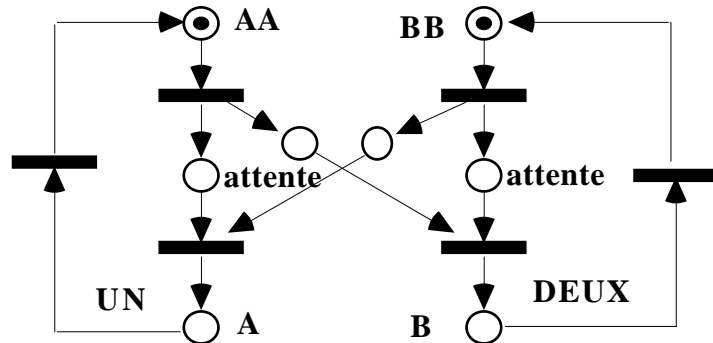
**2) Passage du droit d'accès à une ressource**

```
with Un_Signal; with Une_SC;
procedure Main is
  package AB is new Un_Signal;
  package R is new Une_SC;
  task Emetteur is
    begin loop
      HA; R.Entree; A; AB. Envoyer;
    end loop; end Emetteur;
  task Recepteur is
    begin loop
      HB; AB.Attendre; B; R.Sortie;
    end loop; end Recepteur;
  begin null; end Main;
```

```
package body Un_Signal is
  S : Semaphore; -- initialisé à 0
  procedure Envoyer is begin V(S); end Envoyer;
  procedure Attendre is begin P(S); end Attendre;
  begin E0(S, 0); end Un_Signal;
```

```
package body Une_SC is
  S : Semaphore; -- initialisé à 1
  procedure Entree is begin P(S); end Envoyer;
  procedure Sortie is begin V(S); end Attendre;
  begin E0(S, 1); end Une_SC;
```

### 3) Rendez-vous entre deux processus

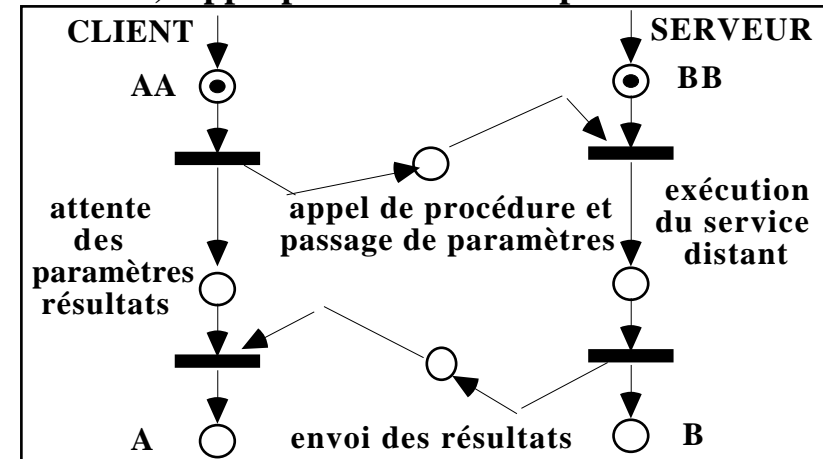


```

with Un_Signal ;
procedure Main is
package AB is new Un_Signal;
package BA is new Un_Signal;
task UN is
  begin loop
    AA; AB.Envoyer; BA.Attendre; A;
    -- en décapsulant : AA; V(AB.S); P(BA.S); A;
  end loop; end UN;
task DEUX is
  begin loop
    BB; BA.Envoyer; AB.Attendre; B;
    -- en décapsulant : BB; V(BA.S); P(AB.S); B;
  end loop; end DEUX;
begin null;
-- en décapsulant E0(AB.S, 0); E0(BA.S, 0);
end Main;

```

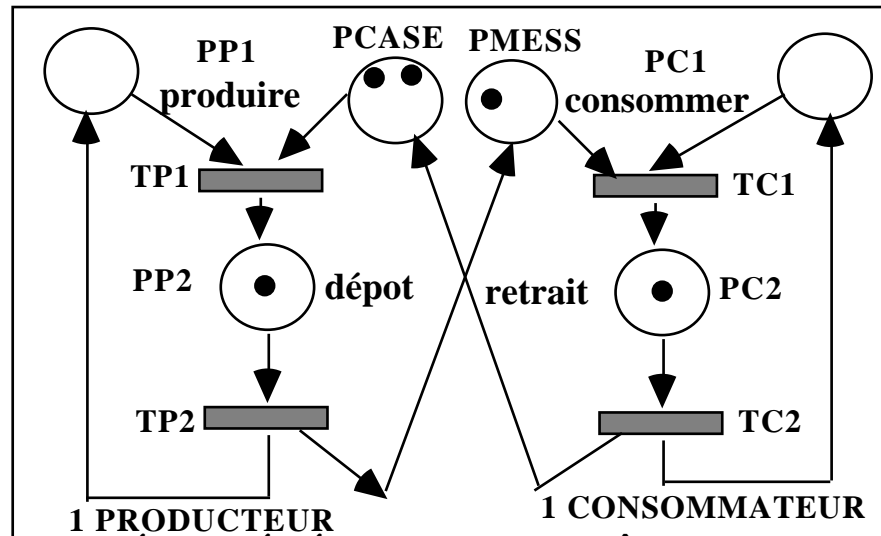
### 4) Appel procédural entre processus



```

with Un_Signal ;
procedure Main is
package Appel is new Un_Signal;
package Reponse is new Un_Signal;
task CLIENT is
  begin loop
    AA; Ecrire_Message; Appel.Envoyer;
    Reponse.Attendre; Lire_Resultats; A;
    -- en décapsulant : Ecrire; V(Appel.S);
    -- P(Reponse.S); Lire_Resultats;
  end loop; end CLIENT;
task SERVEUR is
  begin loop
    BB; Appel.Attendre; Lire_Resultats;
    Traiter_requete; Ecrire; Reponse.Envoyer; B;
    -- en décapsulant : BB; P(Appel.S); Lire;
    -- Traiter_requete; Ecrire; V(Reponse.S); B;
  end loop; end SERVEUR ;
begin null; end Main;

```



SCHEMA GÉNÉRAL DE CONTRÔLE DE FLUX

Nplein, Nvide : Semaphore;  
 E0(Nplein, 0); E0(Nvide, N);

<b>task Producteur is</b>	• <b>task Consommateur is</b>
X : Message;	Y : Message;
<b>begin</b>	• <b>begin</b>
<b>loop</b>	• <b>loop</b>
Produire (X);	• P(Nplein);
P(Nvide);	• Retrait(Y);
Depot(X);	• V(Nvide);
V(Nplein);	• Consommer(Y);
<b>end loop;</b>	• <b>end loop;</b>
<b>end Producteur;</b>	• <b>end Consommateur;</b>

PRODUCTEUR(S)- CONSOMMATEUR(S)

## CONSTRAINTES

- (1) : contrôle de flux :  $\#TP1 \leq N + \#TC2$   
 $\#TC1 \leq \#TP2$
- (2) : respect de l'ancienneté pour déposer
- (3) : respect de l'ancienneté pour retirer
- (4) : cohérence des données du tampon  
 (#TP1 : nombre de franchissements de TP1)

## SOLUTIONS

- (1) : par allocation de ressources banalisées :  
 cases vides contrôlées par le sémaphore Nvide  
 messages contrôlés par le sémaphore Nplein
- (2), (3) : par accès séquentiel
- (4) : par exclusion mutuelle

## GESTION DU TAMPON

- à l'ancienneté avec des index séparés
- 1 producteur et 1 consommateur :  
 schéma 1
- p producteur et c consommateur ( $p + c > 2$ ) :  
 schéma 2
- à l'ancienneté avec un compte commun
- toute autre politique de service (priorité, ...)  
 toute autre gestion du tampon :  
 schéma 3 = schéma général



## CONFLIT IMPOSSIBLE SUR UN MESSAGE entre un producteur et un consommateur

soit #TP1 : nombre de franchissements de TP1

numérotons les messages à l'émission, à TP1

#(messages émis) = #TP1

comptons les messages en réception, à TC1

#(messages reçus) = #TC1

Raisonnons par l'absurde

(E1) : si accès concurrent en PP2 et PC2

prod. cyclique en PP2      #TP1 = #TP2 + 1

contrôle de flux après TC1      #TC1 ≤ #TP2

#TC1 < #TP1

(E2) : si accès au même message

=> #émis = #reçu

#TC1 = #TP1

conclusion : E1 et E2 => contradiction

=> impossibilité

## IMPOSSIBILITÉ D'INTERBLOCAGE

soit M(PP1) : marquage de PP1, etc.

invariants du système

M(PP1) + M(PP2) = 1 -- évolution producteur

M(PC1) + M(PC2) = 1 -- évolution consommateur

M(PCASE) + M(PP2) + M(PMESS) + M(PC2) = N

-- évolution des cases selon 4 états

Il y aurait interblocage si on avait :

M(PP1) = 1 -- le producteur demande l'accès

M(PC1) = 1 -- le consommateur demande l'accès

M(PCASE) = 0 -- pas de case libre

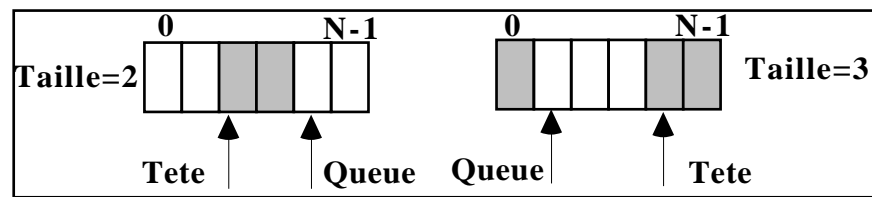
M(PMESS) = 0 -- pas de message prêt

Alors, le dernier invariant :

M(PCASE) + M(PP2) + M(PMESS) + M(PC2) = N

devient : 0 + 0 + 0 + 0 = N

=> impossibilité car N ≠ 0 par construction

**1 PRODUCTEUR - 1 CONSOMMATEUR****schéma 1****Nplein, Nvide : Semaphore;****E0(Nplein, 0); E0(Nvide, N);****type Tampon is array(0..N-1) of Message;****T : Tampon;****task Producteur is****X : Message;****Queue : Mod N := 0;****begin****loop****Produire (X);****P(Nvide);****T(Queue) := X;****Queue := Queue + 1;****V(Nplein);****end loop;****end Producteur;****task Consommateur is****Y : Message;****Tete : Mod N := 0;****begin****loop****P(Nplein);****Y := T(Tete);****Tete := Tete + 1;****V(Nvide);****Consommer(Y);****end loop;****end Consommateur;****p PRODUCTEUR - c CONSOMMATEUR****schéma 2 : (p + c > 2)****Nplein, Nvide : Semaphore;****E0(Nplein, 0); E0(Nvide, N);****type Tampon is array(0..N-1) of Message;****T : Tampon; Tete, Queue : Mod N := 0;****Mutexprod, Mutexcons : Semaphore;****E0(Mutexprod, 1); E0(Mutexcons, 1);****task Producteur is****X : Message;****begin****loop****Produire (X);****P(Nvide);****P(Mutexprod);****T(Queue) := X;****Queue := Queue + 1;****V(Mutexprod);****V(Nplein);****end loop;****end Producteur;****task Consommateur is****Y : Message;****begin****loop****P(Nplein);****P(Mutexcons);****Y := T(Tete);****Tete := Tete + 1;****(Mutexcons);****V(Nvide);****Consommer(Y);****end loop;****end Consommateur;****Notation : type Mod N est modulo N****a + b signifie: (a + b) modulo N**

## PRODUCTEURS - CONSOMMATEURS

### schéma 3 : schéma sans concurrence d'accès

**Nplein, Nvide : Semaphore;**  
**E0(Nplein, 0); E0(Nvide, N);**

**type Tampon is array(0..N-1) of Message;**  
**T : Tampon;**  
**Tete: Mod N := 0; Taille : Integer := 0;**  
**Mutex : Semaphore; E0(Mutex, 1);**

**task Producteur is**

**X : Message;**

**begin**

**loop**

**Produire (X);**

**P(Nvide);**

**P(Mutex);**

**T(Tete+Taille) := X;**

**Taille := Taille + 1;**

**V(Mutex);**

**V(Nplein);**

**end loop;**

**end Producteur;**

**task Consommateur is**

**Y : Message;**

**begin**

**loop**

**P(Nplein);**

**P(Mutex);**

**Y := T(Tete);**

**Tete := Tete + 1;**

**Taille := Taille - 1;**

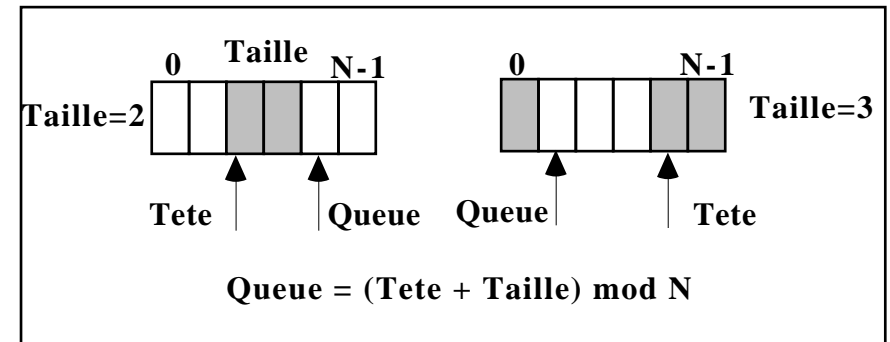
**V(Mutex);**

**V(Nvide);**

**Consommer(Y);**

**end loop;**

**end Consommateur;**



$$Tete + Taille = (Tete + Taille) \bmod N$$

**TAMPON MODULAIRE ET GÉNÉRIQUE****-- corps**

```

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

```

<pre> <b>Procedure Deposer</b>   (X : in Message) is begin   <b>P(Nvide);</b>   P(Mutex);   T(Tete+Taille) := X;   Taille := Taille + 1;   V(Mutex);   <b>V(Nplein);</b> end Deposer; </pre>	<ul style="list-style-type: none"> <li>• <b>Procedure Retirer</b></li> <li>• (Y : out Message) is</li> <li>• begin</li> <li>• <b>P(Nplein);</b></li> <li>• P(Mutex);</li> <li>• Y := T(Tete);</li> <li>• Tete := Tete + 1;</li> <li>• Taille := Taille - 1;</li> <li>• V(Mutex);</li> <li>• <b>V(Nvide);</b></li> <li>• End Retirer;</li> </ul>
--	---

```

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

```

**TAMPON MODULAIRE ET GÉNÉRIQUE****-- interface et utilisation**

```

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

```

**Procedure Main is**

```

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

```

**task Client is**

```

  R : String; Val : Integer;

```

**begin loop**

```

  Faire_Ailleurs_Preparation(R);

```

```

  Requete.Deposer(R); Reponse.Retirer(Val);

```

```

end loop; end Client;

```

**task 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;

```

## LECTEURS-RÉDACTEURS schéma de base

Mutex\_A, Mutex\_L : Semaphore;  
E0(Mutex\_A,1);E0(Mutex\_L, 1);  
NL : Natural := 0;

task Lecteur is  
begin

Faire\_Ailleurs;

**P(Mutex\_L);**

NL := NL + 1;

if NL = 1 then

**P(Mutex\_A);**

end if;

**V(Mutex\_L);**

Suites\_De\_Lectures;

**P(Mutex\_L);**

NL := NL - 1;

if NL=0 then

**V(Mutex\_A);**

end if ;

**V(Mutex\_L);**

end Lecteur;

• task Redacteur is

• begin

• Faire\_Ailleurs;

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

• end Redacteur;

**P(Mutex\_A);**

Ecritures\_Lectures;

**V(Mutex\_A);**

## Spécifications du comportement de base

- lectures et écritures forment une suite d'accès
- lectures par groupe de lecteurs concurrents
- écritures cohérentes en exclusion mutuelle
- exclusion entre groupe de lecteurs concurrents et tout rédacteur

Schéma de base acceptable car le rédacteur qui donne l'accès aux lecteurs par V(Mutex\_A) n'a pas à entrer en section critique pour le faire

=> possibilité de famine pour des rédacteurs

## spécifications additionnelles en cas de conflit

- pas famine de rédacteurs => l'arrivée d'un rédacteur arrête l'entrée de nouveaux lecteurs
- priorité aux lecteurs en attente avant nouvelle rédaction
- priorités égales entre lecteur et rédacteurs
- au plus N lecteurs concurrents consécutifs

**LECTEURS-RÉDACTEURS (priorités égales)**

**Mutex\_A, Mutex\_L, Fifo : Semaphore;**  
**E0(Mutex\_A,1);E0(Mutex\_L, 1);E0(Fifo, 1);**  
**NL : Natural := 0;**

<b>task Lecteur is</b>	• <b>task Redacteur is</b>
<b>begin</b>	• <b>begin</b>
<b>Faire_Ailleurs;</b>	• <b>Faire_Ailleurs;</b>
<b>P(Fifo);</b>	•
<b>P(Mutex_L);</b>	•
<b>NL := NL + 1;</b>	•
<b>if NL = 1 then</b>	• <b>P(Fifo);</b>
<b>P(Mutex_A);</b>	• <b>P(Mutex_A);</b>
<b>end if;</b>	• <b>V(Fifo);</b>
<b>V(Mutex_L);</b>	•
<b>V(Fifo);</b>	•
<b>Suites_De_Lectures;</b>	• <b>Ecritures_Lectures;</b>
<b>P(Mutex_L);</b>	•
<b>NL := NL - 1;</b>	•
<b>if NL=0 then</b>	•
<b>V(Mutex_A);</b>	• <b>V(Mutex_A);</b>
<b>end if ;</b>	•
<b>V(Mutex_L);</b>	•
<b>end Lecteur;</b>	• <b>end Redacteur;</b>

• toutes les requêtes d'accès des processus sont mises en séquence sous le contrôle de Fifo

**Une trace....**

**Lecteur1 : P(Fifo); P(Mutex\_A);    -- E(Fifo) = 0**  
**Lecteur1 : Debut\_Lire;**  
**Lecteur1 : V(Fifo);                -- E(Fifo) = 1**  
**Lecteur1 : Suites\_De\_Lectures;**  
**Lecteur2 : P(Fifo);                -- E(Fifo) = 0**  
**Lecteur2 : Debut\_Lire;**  
**Lecteur2 : V(Fifo);                -- E(Fifo) = 1**  
**Lecteur1 et Lecteur2 : Suites\_De\_Lectures;**  
**Redacteur3 : P(Fifo);             -- E(Fifo) = 0**  
**Redacteur3 : P(Mutex\_A); -- bloquant**  
**Lecteur4 : P(Fifo); -- bloquant E(Fifo) = -1**  
**Redacteur5 : P(Fifo); -- bloquant (E= -2)**  
**Lecteur6 : P(Fifo); --bloquant (E= -3)**  
**Lecteur2 : Fin\_Lire;**  
**Lecteur1 : Fin\_Lire; V(Mutex\_A);**  
**Redacteur3 : V(Fifo);             -- E(Fifo) = -2**  
**Lecteur4 :P(Mutex\_A);**  
**Redacteur3 : Ecritures\_Lectures;**  
**Redacteur3 : V(Mutex\_A);**  
**Lecteur4 : V(Fifo);                -- E(Fifo) = -1**  
**Lecteur4 : Suites\_De\_Lectures;**  
**Lecteur4 : V(Mutex\_A);**  
**Redacteur5 : Debut\_Ecrire;**

**LECTEURS-RÉDACTEURS MODULAIRES****-- lecteurs rédacteurs modulaires (suite)**

**package body MaBase is**

**Mutex\_A, Mutex\_L, Fifo : Semaphore;**  
**NL : Natural := 0; -- variable persistante**

```
procedure Debut_Lire is
begin
  P(Fifo); P(Mutex_L);
  NL := NL + 1;
  if NL = 1 then
    P(Mutex_A);
  end if;
  V(Mutex_L); V(Fifo)
end Debut_Lire;
```

```
• procedure
  Debut_Ecrire
• is
• begin
•   P(Fifo);
•   P(Mutex_A);
•   V(Fifo);
• 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;**

**package MaBase is**

**procedure Debut\_Lire;**  
**procedure Fin\_Lire;**  
**procedure Debut\_Ecrire;**  
**procedure Fin\_Ecrire;**

**end 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;**

**LE REPAS DES PHILOSOPHES**

**repas assis avec chaise pour 4 seulement**

```

procedure Repas_des_Philosophes is
  Baguette : array (0..4) of Semaphore;
  for I in 0..4 loop E0(Baguette(I), 1); end loop;
  Chaise : Semaphore; E0(Chaise, 4);

```

<b>task Philosophe(I) is</b>	•	<b>task Philosophe(J) is</b>
<b>begin loop</b>	•	<b>begin loop</b>
Penser;	•	Penser;
P(Chaise);	•	P(Chaise);
P(Baguette(I));	•	P(Baguette(J));
P(Baguette(I + 1) ;	•	P(Baguette(J + 1));
Manger;	•	Manger;
V(Baguette(I));	•	V(Baguette(J));
V(Baguette(I + 1) ;	•	V(Baguette(J + 1));
V(Chaise);	•	V(Chaise);
<b>end loop;</b>	•	<b>end loop;</b>
<b>end Philosophe(I)</b>	•	<b>end Philosophe(J);</b>

```

begin null; end Repas_des_Philosophes;
-- I et J prennent une valeur entre 0 et 4, et I ≠ J
-- + : calcul modulo 5; par exemple : 4 + 1 = 0
-- ni interblocage, ni famine

```

**VERSION MODULAIRE**

```

package Repas is
  procedure Demander(X : in Id);
  procedure Conclure(X : in Id);
end Repas;
procedure Main is
  type Id is mod 5; -- d'où calculs modulo 5

```

<b>task Philosophe(I) is</b>	•	<b>task Philosophe(J) is</b>
<b>begin loop</b>	•	<b>begin loop</b>
Penser;	•	Penser;
Repas.Demander(I);	•	Repas.Demander(J);
Manger;	•	Manger;
Repas.Conclure(I);	•	Repas.Conclure(J);
<b>end loop;</b>	•	<b>end loop;</b>
<b>end Philosophe(I)</b>	•	<b>end Philosophe(J);</b>
<b>begin null; end Main;</b>		

```

package body Repas is
  Chaise : Semaphore;
  Baguette : array(id) of Semaphore;
  procedure Demander(X : in Id) is
    begin
      P(Chaise); P(Baguette(X)); P(Baguette(X + 1));
    end Demander;
  procedure Conclure(X : in Id) is
    begin
      V(Baguette(X)); V(Baguette(X + 1)); V(Chaise);
    end Conclure;
  begin
    E0(Chaise, 4);
    for x in id loop E0(Baguette(x), 1); end loop;
  end Repas;

```



## STRUCTURE TYPE DES COMPOSANTS DE CONTRÔLE DE SYNCHRONISATION À GROS GRAINS

mécanismes reposant sur de la mémoire partagée

### BASE

Données persistantes de contrôle partagées

cohérence de ces données : => toute suite d'accès aux données de contrôle doit être une section critique

=> procédures d'accès en exclusion mutuelle

### SYNCHRONISATION

**ATTENTE** : Si Condition1 alors laisser continuer le processus appelant sinon le bloquer

**SIGNAL** : Si Condition2 alors alerter un processus quelconque (que ce soit le processus qui utilise le composant de contrôle, un processus bloqué ou un processus non bloqué).

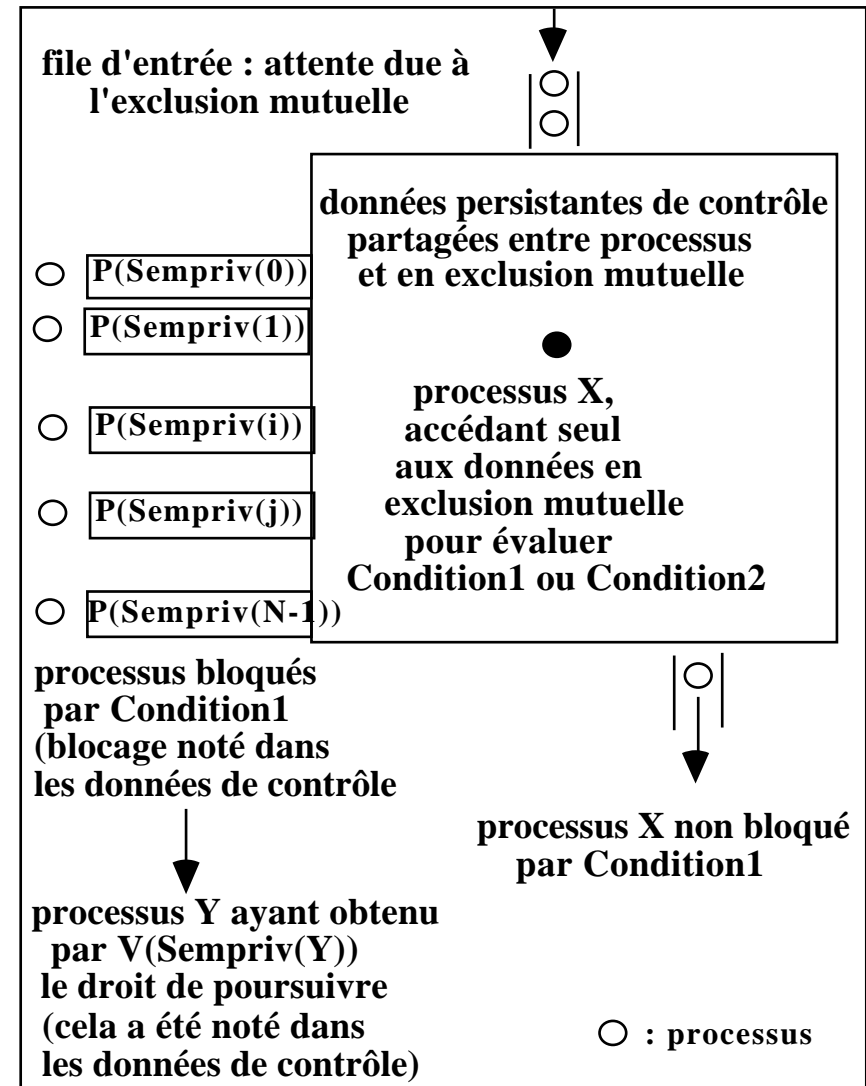
**CONTRAINTES** : respect de l'exclusion mutuelle sur les données de contrôle, donc pas de blocage de processus en section critique (=> interblocage)

### EXEMPLES D'OBJETS DE CONTRÔLE

**MONITEUR** : mis en place par le compilateur  
synchronized method en java,  
objet protégé en Ada

**SÉMAPHORES**: à programmer explicitement par la technique des sémaphores privés

## SCHÉMA DES SÉMAPHORES PRIVÉS



**COMPOSANT AVEC SÉMAPHORE PRIVÉ**

**Mutex : Semaphore; E0(Mutex, 1);**  
**Sempriv:array() of Semaphore; E0(Sempriv(),0);**  
**A, B, C, ... : Variables\_Partagées**

**Demande de ressource avec éventuel blocage**

```
P(Mutex) ; ....
-- le processus X a obtenu l'accès au composant
-- le boolean Ok est local à X, Ok est non partagé
if Cond1(A, B, C) then Ok := True; end if;
--Cond1 : condition de non blocage de X
V(Mutex);
if not Ok then P(Sempriv(X)); end if;

-- blocage (éventuel) hors section critique
-- X va continuer tout de suite ou plus tard
-- Il faut donc noter dans les données de contrôle
-- si X se poursuit ou s'il est bloqué
-- Le processus Z qui réveillera X devra noter,
-- avant de le réveiller, que X a changé d'état
```

**Libération de ressource avec éventuel réveil**

```
P(Mutex) ; ....
-- le processus Z a obtenu l'accès au composant
if Cond2(A, B, C) then V(Sempriv(Y)); end if;
--Cond2 : condition de réveil de processus bloqué
-- le processus Y est validé. Z doit le noter
-- dans les données de contrôle
-- car Y ne le fera pas après son réveil
V(Mutex);
```

**COMPOSANT AVEC SÉMAPHORE PRIVÉ****Variante historique (Dijkstra)****Demande de ressource avec éventuel blocage**

--Cond1 : condition de non blocage de X  
 --Cond2 : condition de réveil de processus bloqué

```
P(Mutex) ;
-- le processus X a obtenu l'accès au composant
-- le boolean Ok est local à X, Ok n'est pas partagé
if Cond1(A, B, C) then
  Ok := True; V(Sempriv(X));
else
  Ok := False;
end if;
V(Mutex);
P(Sempriv(X));
-- blocage (éventuel) hors section critique
```

**Libération de ressource avec éventuel réveil**

```
P(Mutex) ; ....
-- le processus Z a obtenu l'accès au composant
if Cond2(A, B, C) then
  V(Sempriv(Y)); -- après le choix d'un Y possible
  -- Z doit aussi noter le nouvel état de Y
end if;
V(Mutex);
```

**LE REPAS DES PHILOSOPHES**

Un philosophe reçoit deux baguettes ou aucune  
 Allocation globale des baguettes  
 si et ssi les deux baguettes sont libres  
 i.e. si aucun des voisins du demandeur ne mange

```
task Philosophe(I : Id) is Ok : Boolean;
begin loop
  PENSER;

  P(Mutex); -- composant de contrôle
  Etat(I) := demande;
  if Etat(I + 1) /= mange and Etat(I - 1) /= mange
    then Etat(I) := mange; Ok := True;
  else Ok := false; end if;
  V(Mutex) ;
  if not Ok then P(Sempriv(I)); end if;

  MANGER;

  P(Mutex); -- composant de contrôle
  Etat(I) := pense;

  -- le philosophe I essaie de réveiller ses voisins
  if Etat(I+1) = demande and Etat(I+2) /= mange
  then Etat(I+1):=mange;V(Sempriv(I+1)); end if;

  if Etat(I-1) = demande and Etat(I-2) /= mange
  then Etat(I-1) :=mange;V(Sempriv(I-1)); end if;
  V(Mutex) ;
end loop; end Philosophe(I) ;
```

**variables partagées utilisées pour le contrôle**

```
type Id is mod 5; -- d'où calculs modulo 5
type pdm is (pense, demande, mange);
Etat : array(Id) of pdm := (others => pense);

task Philosophe(J : Id) is
  Ok : Boolean;
begin loop
  PENSER;

  P(Mutex); -- composant de contrôle
  Etat(J) := demande;
  if Etat(J + 1) /= mange and Etat(J - 1) /= mange
    then Etat(J) := mange; Ok := True;
  else Ok := false; end if;
  V(Mutex) ;
  if not Ok then P(Sempriv(J)); end if;

  MANGER;

  P(Mutex); -- composant de contrôle
  Etat(J) := pense;

  -- le philosophe J essaie de réveiller ses voisins
  if Etat(J+1) = demande and Etat(J+2) /= mange
  then Etat(J+1):=mange;V(Sempriv(J+1)); end if;

  if Etat(J-1) = demande and Etat(J-2) /= mange
  then Etat(J-1) :=mange;V(Sempriv(J-1)); end if;
  V(Mutex) ;
end loop; end Philosophe(J) ;
```

**LE REPAS DES PHILOSOPHES****Allocation globale des baguettes****programmation modulaire****procedure Main is****type Id is mod 5; -- d'où calculs modulo 5**

<b>task Philosophe(I:Id) is</b>	•	<b>task Philosophe(J:Id) is</b>
<b>begin</b>		<b>begin</b>
<b>loop</b>		<b>loop</b>
<b>Penser;</b>	•	<b>Penser;</b>
<b>Repas.Demander(I);</b>	•	<b>Repas.Demander(J);</b>
<b>Manger;</b>	•	<b>Manger;</b>
<b>Repas.Conclure(I);</b>	•	<b>Repas.Conclure(J);</b>
<b>end loop;</b>	•	<b>end loop;</b>
<b>end Philosophe(I)</b>	•	<b>end Philosophe(J);</b>

**begin null; end Main;****contrôle pour autorisation de manger****type pdm is (pense, demande, mange);****Etat : array(Id) of pdm := (others => pense);****function test (x : 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****package body Repas is****type pdm is (pense, demande, mange);****Etat : array(Id) of pdm := (others => pense);****Mutex : Semaphore;****Sempriv : array(Id) of Semaphore;****function test (X : Id) return boolean;-- ci-contre****procedure Demander(X : in Id) is Ok : boolean;****begin****P(Mutex);****Etat(X) := demande; Ok := test(X);****if Ok then Etat(X) := mange; end if;****V(Mutex);****if not Ok then P(Sempriv(X)); end if;****-- au réveil mangera sans prévenir****end Demander;****procedure Conclure(X : in Id) is****begin****P(Mutex); Etat(X) := pense;****if test(X + 1) then -- voisin de droite****Etat(X + 1) := mange; V(Sempriv(X + 1));****end if; -- on a noté le nouvel Etat de X + 1****if test(X - 1) then -- voisin de gauche****Etat(X - 1) := mange; V(Sempriv(X - 1));****end if; -- on a noté le nouvel Etat de X - 1****V(Mutex);****end Conclure;****begin E0(Mutex, 1);****for Y in id loop E0(Sempriv(Y), 0); end loop;****end Repas;**

**LOT DE RESSOURCES BANALISÉES****Lot de ressources banalisées**

```

package Lot is
  --LOT
  procedure Prendre(I:Id; X:Integer; Y:out Liste);
  procedure Rendre(X: Integer; Y : Liste);
  -- I : nom du processus demandeur
  -- X : nombre de ressources demandées
  -- Y : liste nominative des ressources reçues
end Lot ;
  --LOT

```

**procedure Avec\_Ressources\_Banalisees is**

```

  type Id is Integer range 1..Nb_Processus;
  task type Processus (I : Id := Nom_Unique);
  Application : array(Id) of Processus;

  task body Processus is
    --PROCESSUS
    Plusieurs : Integer;
    Les_Ressources: Liste;
  begin loop
    Calculer(Plusieurs); -- Plusieurs est le résultat
    Lot.Prendre(I, Plusieurs, Les_Ressources);
    Utiliser(Les_Ressources);
    Lot.Rendre(Plusieurs, Les_Ressources);
  end loop; end Processus;

  begin null; end Avec_Ressources_Banalisees;

```

**Les processus demandent plusieurs ressources qui leur sont allouées globalement sous forme d'une liste de ressources élémentaires**

**Allocation sous contrôle d'un composant réalisé par la méthode des sémaphores privés.**

**Les données du composant sont**

**Amas : tableau de ressources allouables**

**Amas(J) = 1 quand la ressource J est allouée**

**Stock : nombre de ressources non allouées**

**Initialement il y a Stock\_Initial ressources**

**La liste L contient les processus en attente**

**La liste V indique les ressources qu'ils attendent**

**-- voir Annexe 1 pour la gestion de listes en Ada**

**on utilise les sémaphores**

**Mutex , Mutex\_Amas: Semaphore;**

**Sempriv: array(Id) of Semaphore;**

```

package body Lot is
    --LOT BANALISÉ
    type R_Id is Integer range 1.. Stock_Initial;
    Mutex , Mutex_Amas : Semaphore;
    Sempriv : array(Id) of Semaphore;
    Amas : array(R_Id) of Integer := (others => 0);
    Stock : Integer:= Stock_Initial; L,V: Liste := Ø;
    -- ce sont les données persistantes du module

```

```

-- PRENDRE

```

```

procedure body Prendre

```

```

    (I : Id; X: Integer; Y: out Liste) is

```

```

    Ok : Boolean; J : R_Id ; -- propres à chaque appel

```

```

begin
    -- demander le droit d'en prendre X

```

```

    P(Mutex);
    if X <= Stock then
        Stock := Stock -X; Ok := True;
    else
        Ajouter(L,I); Ajouter(V,X); Ok := False;
    end if;
    V(Mutex);

```

```

    if not Ok then P(Sempriv(I)); end if;

```

```

-- I a obtenu le droit de prendre ses X ressources

```

```

    P(Mutex_Amas); J := 1; Y := Ø; -- liste vide
    for K in 1..X loop
        while Amas(J) /= 0 loop J := J + 1; end loop;
        Amas(J) := 1; Ajouter(Y,J); J := J + 1;
    end loop;
    V(Mutex_Amas);

```

```

end Prendre;

```

```

-- RENDRE

```

```

procedure body Rendre(X: Integer; Y: Liste) is

```

```

    Lui : Id; K : Integer;

```

```

begin
    -- le processus remet X ressources

```

```

    P(Mutex_Amas); -- restituer les ressources
    for J in 1..X
        loop Oter(Y, J); Amas(J) := 0; end loop;
    V(Mutex_Amas);

```

```

-- et le dit au contrôle

```

```

    P(Mutex); Stock := Stock + X;
    while not EstVide(L) loop
        Premier(L, Lui); Premier(V, K);
        exit when K > Stock;
        Oter(L, Lui); Oter(V, K);
        Stock := Stock - K;
        -- on doit noter que Lui peut se servir
        V(Sempriv(Lui));
    end loop;
    V(Mutex);

```

```

end Rendre;

```

```

begin E0(Mutex_Amas,1);

```

```

E0(Mutex, 1);

```

```

    for I in Id loop E0(Sempriv(I),0) ; end loop;

```

```

end Lot;

```

# CONTRÔLE DE CONCURRENCE ET SYNCHRONISATION DES PROCESSUS

- l'exclusion mutuelle ne suffit pas pour traiter tous les aspects de la synchronisation et le contrôle entre les processus concurrents
- il faut aussi pouvoir bloquer un processus et réveiller un processus

## SÉMAPHORES

On a vu qu'on peut traiter tous les cas avec le seul mécanisme des sémaphores

## MONITEURS (ET JAVA)

- le moniteur ajoute le type condition et une file d'attente associée à chaque variable de type condition ainsi que des primitives wait x, signal x, avec x : condition

Un processus se bloque par wait et libère le moniteur

Un processus réveille un autre processus par signal, mais un seul processus utilise le moniteur, l'autre doit attendre la libération du moniteur

- En java cela s'appelle wait(), notify(), notifyAll()

et il n'y a qu'une condition anonyme avec une seule file d'attente par moniteur

## OBJETS PROTÉGÉS

- les conditions sont remplacées par des gardes sur les appels de procédure, transformées en entrées gardées par des expressions booléennes. Un processus appelant une procédure ou une entrée s'exécute sans préemption, en exclusion mutuelle. La fin d'une exécution d'entrée soit s'accompagne d'un retour du processus vers la procédure appelante, soit bloque à nouveau le processus si l'instruction requeue nom\_d\_entrée est programmée. L'objet protégé est libéré pour un autre processus. Avec les gardes et le requeue, on peut programmer l'utilisation d'un objet protégé comme un automate synchronisé.

**//PRODUCTEUR CONSOMMATEUR EN JAVA//**

```
class Buffer{  
    protected final int max;  
    //"protected" : accessible dans la classe  
    // "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();  
    }
```

```
    public synchronized Object get() {  
        while (count == 0) {  
            try {wait(); }  
            catch (InterruptedException e){}  
        Object result = data(nextOut)); ;  
        nextOut = (nextOut+1) % max;  
        count--;  
        notify();  
        return result;  
    }  
} // fin de la déclaration de la classe Buffer  
  
// noter l'emploi de notify car il n'y a qu'une  
// cause de blocage à un instant donné  
// valable même si on a plusieurs producteurs ou  
// consommateurs  
// chaque moniteur n'a qu'une condition anonyme
```



```

class Producer implements Runnable{
    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) );
            } // fin du bloc où est levée l'exception
        } catch(InterruptedException e) {return}
    }
// fin de la classe Producer

class Consumer extends Thread {
    protected final Buffer buffer;
    Consumer(Buffer buffer){this.buffer=buffer;}
    public void run(){
        try{
            for (int j=1; j<=100; j++){
                Integer p = (Integer) (buffer.get());
                System.out.println("consommé" + p) ;
                Thread.sleep(1000);
            } // fin du bloc où est levée l'exception
        } catch(InterruptedException e) {return}
    }
// fin de la classe Consumer

```

**// voici le programme principal**

```

public class ProducerConsumer{
    static Buffer buffer = new Buffer(20);
    public static void main (String[] args) {

        // on crée un producteur et un consommateur
        // on doit leur passer les objets partagés
        // c'est à dire ici l'objet buffer

        Producer p = new Producer(buffer);
        Thread pt = new Thread(p);
        // car le producteur est Runnable

        Consumer c = new Consumer(buffer);
        pt.start() ;
        c.start();
    }

//FIN DU PRODUCTEUR CONSOMMATEUR //
// EN JAVA //

```

## OBJETS PROTEGES DE ADA 95

### objets protégés et types protégés

- ce sont des objets partageables
- les accès sont en exclusion mutuelle
- les fonctions (accès lecture seule) peuvent être concurrentes (si le compilateur l'implante)
- l'exécution de procédure est contrôlable par des gardes ("barriers") avec la clause :  
     entry e when expression\_booléenne is...
- indéterminisme si plusieurs entry et plusieurs gardes sont vraies
- peut ne pas exécuter de retour de procédure et aller attendre sur une autre entry :  
     instruction requeue
- permet par exemple d'examiner les paramètres effectifs d'un appel
- requeue renvoie à une entrée d'objet protégé et ne termine pas l'entry qui se termine de façon habituelle (sans libérer l'appelante)
- la partie privée peut contenir des données et des entry:  
     clause private
- avec ces gardes et requeue, on peut programmer l'accès concurrent à un objet protégé comme un automate l'automate est sous contrôle des gardes;

## sémantique d'un objet protégé

- chaque accès (fonctions, procédures et entrées) est une section critique protégée par un verrou ("lock"); il y a un verrou par objet protégé; s'il y a plusieurs appels concurrents, un seul est pris, les autres sont en "attente externe", ainsi que les appels arrivant pendant un accès ;
- si une entrée est appelée, sa garde est évaluée;
- l'évaluation de la garde est en section critique
- si la garde est vraie, l'entrée est exécutée et l'exécution se fait en exclusion mutuelle;
- si la garde est fausse, la tâche appelante est mise en "attente interne" et est placée dans une file d'attente associée à l'entrée appelée;
- quand une procédure ou une entrée se termine, toutes les gardes des entrées qui ont une tâche en attente interne sont évaluées;
- si une garde au moins est vraie, l'une des tâches en attente interne sur cette entrée (et une seule) reçoit l'accès exclusif à l'objet protégé;
- le choix d'une tâche parmi plusieurs possibles est non déterministe;
- les tâches en attente interne ont priorité sur les tâches en attente externe; en particulier, un nouvel appel d'entrée ne peut pas évaluer de garde tant que l'appel en cours (et tous les appels qui sont en attente interne et dont les gardes seront vraies à la fin de l'appel en cours) n'est pas terminé.

```

generic   max : positive; type Item is private;
package Composant is
  Item_Array is array(Integer range < >) of Item;

```

```

Protected type Canal is -- partie spécification
  entry Deposer (X : In Item);
  entry Retirer (X : out Item);
private -- variables persistantes pour chaque objet
  A : Item_Array (1 .. max);
  I, J : Integer range 1 .. max := 1;
  Compte : Integer range 0 .. max := 0;
end Canal ;

```

```

Protected body Canal is -- partie réalisation

  entry Deposer (X : In Item) when Compte < max is
  begin
    A(I) := X;
    I := I mod max + 1;
    Compte := Compte + 1;
  end Deposer ;

  entry Retirer (X : out Item) when Compte > 0 is
  begin
    X := A(J);
    J := J mod max + 1;
    Compte := Compte - 1;
  end Retirer ;
end Canal ;
end Composant ;

```

## ADA : OBJET PROTÉGÉ

Producteurs-Consommateurs  
avec composant générique

Procedure Main is

```

Package A is new Composant (6, String);
Package B is new Composant (2, Integer);

```

```

Requete : A.Canal
Reponse : B.Canal;

```

**task** Client is

```

  R : String; Val : Integer;

```

**begin loop**

```

  Faire_Ailleurs_Et_Preparer(R);

```

```

  Requete.Deposer(R); Reponse1.Retirer(Val);

```

```

end loop; end Client;

```

**task** 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;

**TYPE SEMAPHORE EN ADA95**

```

--specification : ads
package Semaphores is
-----
-- ce paquetage implémente le type Semaphore
-- et les primitives classiques P, V and E0
-- chaque sémaphore déclaré est réalisé
-- par un objet protégé
-- en Ada, un objet limité privé ne peut être
-- ni copié ni modifié en dehors du paquetage

type Semaphore is limited private;
procedure P (S : in out Semaphore);
procedure V (S : in out Semaphore);
procedure E0 (S : in out Semaphore; I : in Natural);

private

protected type Semaphore is
  entry V;
  entry P;
  entry E0(X : in Natural); --  $X \geq 0$ 
private -- variables persistantes de chaque Semaph.
  E : Natural; --  $E \geq 0$  dans cette implantation
  INIT : Boolean := false; -- E0 fait avant P ou V
end Semaphore ;
end Semaphores ;
-- fin de l'ads

```

```

-- corps : adb
package body Semaphores is

protected body Semaphore is

  entry V when INIT is
    begin E := E + 1; end V;

  entry P when INIT and then E > 0 is
    begin E := E - 1; end P;

  entry E0(X : in natural) when not INIT is
    begin E := X; INIT := true; end E0;

  end Semaphore ;

  procedure P (S : in out Semaphore ) is
    begin S.P; end P;

  procedure V (S : in out Semaphore ) is
    begin S.V; end V;

  procedure E0
    (S : in out Semaphore; I : in Natural) is
    begin S.E0(I); end E0;

end Semaphores ;
-- fin de l'adb

```