# Applications Concurrentes :

# Conception,

# Outils de Validation

## ACCOV_B

### Chapitre 3

## PROGRAMMATION DE LA CONCURRENCE

## COMMUNICATION PAR RESSOURCE COMMUNE

## sémaphores

## régions critiques et moniteurs

## objets protégés dans ADA 95

# COMMUNICATION PAR RESSOURCE COMMUNE

# ET VARIABLES PARTAGEES

## VARIABLES COMMUNES SIMPLES AVEC ATTENTE ACTIVE

**voir le cours Systèmes et réseaux Informatique (SRI_B ou SI_B)**
**algorithmes : Peterson, Dekker, Lamport**
**test and set**
**inconvénients de l'attente active : occupation parasite du processeur et du bus d'accès à la mémoire**

## SÉMAPHORES

**voir cours SRI_B ou SI_B**
**rappels**
**S : sémaphore et opérations associées**
**P(S) ou wait(S) (PUIS-JE?)**
**V(S) ou signal(S) (VAS-Y)**
**E(S, I) ou initial(S, I)**
**indivisibilité de l'exécution des opérations : exclusion mutuelle**
**pas de parallélisme possible, mais exécution distribuée possible**

**$\forall$ op1(S), op2(S) $\in$ {P(S), V(S), E0(S, I)}**

**(op1(S)-> op2(S)) ou (op2(S) -> op1(S))**

# RÉALISATIONS DES SÉMAPHORES

```
type sémaphore is record
        E : integer;
        F : file_de_processus_bloqués;
        end;
```

**• RÉALISATION_1 : E peut être positif, négatif ou nul**

```
P(S) :: -- API exécutée par un processus dont l'ID est donnée par EGO
   begin
   S.E := S.E -1;
   if S.E ≥ 0 then null;
              else bloquer_le_processus_EGO_et_mettre_son_ID_dans_S.F;
   end if;
end P;
```

{état(EGO) = actif et S.E≤0} P(S) {état(EGO) = bloqué et EGO ∈ S.F}

au réveil de EGO, le processus continue après P(S)

{#PF(S) := #PF(S) + 1}

```
V(S) :: -- API exécutée par un processus dont l'ID est donnée par EGO
   begin
   S.E := S.E + 1;
   if S.E > 0 then null;
              else retirer_un_processus_de_S.F_et_le_rendre_actif;
   end if;
end V;
```

{état(EGO)=actif et S.E<0 et LUI ∈ S.F} V(S) {état(LUI) = actif}

```
E0(S, I) ::
begin S.E := I; S.F := ∅; end E0; -- I ≥ 0
```

invariants :

{S.E := I + #V(S) - #P(S)},

{#PF(S) = min(#P(S), I + #V(S))},

avec #PF(S) = nombre de P(S) franchis

si S.E < 0 alors abs(S.E) = nombre d'éléments de S.F

**• RÉALISATION_2 : E peut être positif ou nul, jamais négatif**

```
P(S) :: -- API exécutée par un processus dont l'ID est donnée par EGO
   begin
   if S.E > 0 then S.E := S.E -1;
            else bloquer_le_processus_EGO_et_mettre_son_ID_dans_S.F;
   end if;
end P;
{état(EGO) = actif et S.E=0} P(S) {état(EGO) = bloqué et EGO ∈ S.F}
au réveil de EGO, le processus bloqué continue après P(S)
V(S) :: -- API exécutée par un processus dont l'ID est donnée par EGO
   begin
   if S.F ≠ ∅ then retirer_un_processus_de_S.F_et_le_rendre_actif;
            else S.E := S.E + 1;
   end if;
end V;
{état(EGO)=actif et S.F ≠ ∅ et LUI ∈ S.F} V(S) {état(LUI) = actif}
E0(S, I) ::
begin S.E := I; S.F := ∅; end E0;
{invariant : S.E ≥ 0 et S.E := I + #V(S) - #PF(S)},
{#PF(S) = min(#P(S), I + #V(S))},
      avec #PF(S) = nombre de P(S) franchis
```

**• RÉALISATION_3 :**

```
type sémaphore is record
   NP : natural := 0; -- compte les P appelées sur le sémaphore
   NV : natural := 0; -- compte les V appelées sur le sémaphore
   F : file_de_processus_bloqués;
end;
P(S) :: -- ticket est une variable locale à P
   begin
   S.NP := S.NP + 1; ticket := S.NP;
   attendre S.NV ≥ ticket; -- attente hors section critique
   end P; {#PF(S) := #PF(S) + 1} {#P(S) := #P(S) + 1}
V(S) ::
begin S.NV := S.NV + 1; end V;  {#V(S) := #V(S) + 1}
E0(S) :: begin S.NV := I; end E0; {#V(S) := I}
```

**Cette réalisation se prête à une implantation répartie car NP et NV sont des compteurs monotones croissants**

# sémaphores et équité

- **la gestion de la file d'attente n'est pas imposée; elle est donc différente d'une implantation à une autre; mais elle a une incidence sur l'équité.**
  **FIFO : équité forte**
  **RANDOM : équité faible**
  **PRIORITE : on peut avoir de la famine**

# utilisation des sémaphores

- **voir cours SI_B ou SRI_B**

# intérêt des sémaphores

- **mécanisme simple, pas d'attente active**
- **meilleur que le masquage des interruptions (conceptuellement)**
- **meilleur que l'attente active (ressources processeur et canal)**
- **mécanisme logique de base**
- **peu coûteux si bien implémenté au bon niveau (noyau)**

# critiques des sémaphores

- **A) difficile à utiliser avec fiabilité**
  **l'oubli d'un V(mutex) peut conduire à interblocage**
  **l'oubli d'un P(mutex) est une faute d'exclusion mutuelle**
  **on retrouve ces problèmes pour tous les mécanismes quand les procédures ou les appels systèmes peuvent être emboîtés**
- **B) mécanisme non structuré; la synchronisation peut apparaître n'importe où dans le code - et non pas dans des régions bien localisées - donc la détection d'erreurs et la mise au point sont plus difficile car la propagation des erreurs est très facilitée quand les sémaphores sont placés comme variables globales.**
- **C) mécanisme trop élémentaire au niveau abstraction**

## mon avis fondé sur mon expérience

**A) et B) sont de fausses critiques, car applicables à tout mécanisme; il y a une réponse facile en s'imposant de bonnes règles de programmation avec modules ou paquetages (qui sont les règles générales et bien connues de la programmation structurée "in the large" classique voir conseils des cours SI_B et SRI_B et article de C.K. dans TSI vol 13 n° 5/1994 pp. 671-696);**

**On peut alors faire des vérifications (simples ou formelles) dans les paquetages dans lesquels on confine la synchronisation et qui restent en général de petite taille. On peut avoir des outils automatisables. On peut associer des invariants à ces paquetages.**

**C) est recevable; voir la suite du cours qui apporte une réponse avec des mécanismes plus évolués sur le plan de l'abstraction (voir en particulier l'apport de l'instruction "requeue" de Ada 95 qui permet une programmation par automates, très structurée)**

## extensions simples des sémaphores

- **sémaphores avec messages (utilisés dans HB 64 renommé DPS 7)**
- **sémaphores avec P non bloquant et retour d'un compte rendu booléen**
- **sémaphores avec P bloquant seulement pendant un délai maximal**
- **sémaphores avec P(S, n) augmentant l'entier E.S de n, V(S,m) diminuant l'entier E.S de m ( m, n ≥ 0 et pas seulement n = m = 1)**

**Ce sont toutefois des bricolages qui ne répondent pas à la critique C) et qui sont rendus obsolètes par l'apparition des objets protégés associés à la clause "requeue".**

# REGIONS CRITIQUES ET MONITEURS

**régions critiques (Brinch Hansen 1972), moniteurs (Hoare 1974, utilisé dans les langages Modula puis Mesa), repris dans le langage Java, revisité et amélioré dans le langage Ada95.**

## moniteur :

- **structure de programmation associé à l'exclusion mutuelle et mise en place automatiquement par le compilateur**
- **structure modulaire (comme un objet, un paquetage, un module)**
  **les données sont internes au moniteur, encapsulées**
  **elles ne sont pas visibles, ni accessibles de l'extérieur du moniteur**
  **seules les procédures du moniteur sont exportables**
- **les accès aux données partagées restent localisés dans le moniteur**
- **la synchronisation autre que l'exclusion mutuelle (celle-ci ne règle pas tout) se fait par un type de synchronisation : condition**
  **primitives wait x et signal x où x est du type condition (primitives encore appelées "delay" et "resume")**
  **les objets de type condition sont associés à des files d'attente à l'ancienneté (FIFO) d'où fonction empty**
  **la primitive wait doit respecter l'exclusion mutuelle dans le moniteur (un seul processus dans le moniteur à tout instant, y compris ceux qui y sont bloqués); tout processus bloqué par wait doit libérer le moniteur;**
  **la primitive signal x n'a pas de mémoire : elle n'agit que sur le premier processus de la file associée à x; si la file x est vide, la primitive signal x est une opération nulle**
  **d'autre part, selon les auteurs, le processus réveillé préempte ou non celui qui le réveille en exécutant signal x. Quel que soit le choix implémenté, un seul processus s'exécute et l'autre est exclu du moniteur, donc bloqué.**

# EXEMPLE DE MONITEUR

```
Monitor BoundedBuffer {
   define QueueSize 10;
   int front = 0, back = 0;              /* location of front and back of queue */
   int count = 0;              /* number of used elements in the queue */
   int queue[QueueSize];     /* queue of integer */
   Condition NotFull, NotEmpty;

   int query(void) {return count;}     /* no mutual exclusion */

   Entry insert( int element){
      if (count == QueueSize ) Wait NotFull;  /* buffer full */
      queue[back] = elem;              /* insert element into buffer */
      back = (back + 1)% QueueSize;
      count += 1;
      Signal NotEmpty;                 /* inform tasks, buffer not empty */
   }

   Entry remove( void) {
      int element;
      if (count == 0 ) Wait NotEmpty;  /* buffer empty */
      elem = queue[front] ;  /* remove element from buffer */
      front = (front + 1)% QueueSize;
      count -= 1;
      Signal NotFull;          /* inform tasks, buffer not full */
   }
}

          /* exemple de moniteur : bounded buffer avec signal implicite */
```

# intérêts du moniteur

- **introduction de la modularité et de la systématisation de l'exclusion mutuelle et de la section critique**
- **se prête bien à une formalisation (mieux que le sémaphore) car :**
  - **la section critique est déclarée,**
  - **on peut sérialiser les sections critiques,**
  - **on peut appliquer des preuves séquentielles à cette sérialisation**
- **on peut associer des invariants à un moniteur**

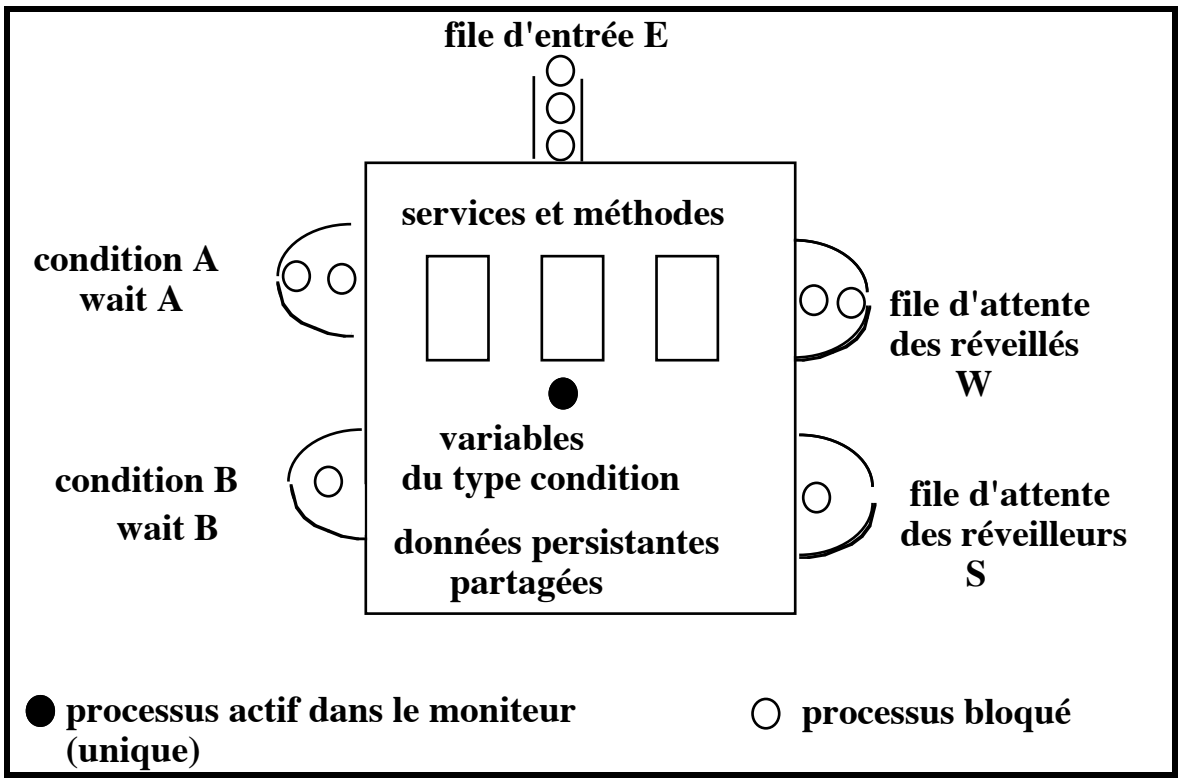
# inconvénients du moniteur

- **la synchronisation par wait et signal peut devenir complexe et se révéler d'aussi bas niveau que les sémaphores**
  **(des essais pour introduire des sortes d'expressions gardées apportent des améliorations de fiabilité, mais au coût d'une réévaluation dynamique des gardes, à des endroits non systématiques)**
- **problème des appels emboîtés = problème général des exclusions mutuelles emboîtées :**
  **- soit danger d'interblocage si appel de moniteur possible depuis un autre moniteur (P1 dans M1 appelle M2 || P2 dans M2 appelle M1)**
  **- soit un processus ne doit pouvoir n'appeler qu'un moniteur à la fois : contrainte à la programmation (explicite statique) ou à l'exécution(dynamique)**

# conclusion sur le moniteur

- **le point faible est de programmer les conditions de synchronisation dans le code du moniteur; il faudrait pouvoir vérifier ces conditions avant d'entrer dans le moniteur et en sortir systématiquement pour demander leur réevaluation, ce qui sera fait avec les objets protégés de Ada 95**

# SORTES DE MONITEURS ET LEURS SÉMANTIQUES DE CONCURRENCE




**Les processus utilisateurs du moniteur**

| Priorité relative | nom du type de moniteur |
|---|---|
| Ep = Wp = Sp | |
| Ep = Wp < Sp | Wait and Notify [Lampson 1980], repris dans Java |
| Ep = Sp <Wp | Signal and Wait [Howard 1976a] |
| Ep < Wp = Sp | |
| Ep < Wp < Sp | Signal and Continue [Howard 1976b] |
| Ep < Sp < Wp | Signal and Urgent Wait [Hoare 1974] |

**Ep : priorité de la file d'entrée dans le moniteur**
**Wq : priorité de la file des processus réveillés pour condition**
**Sp : priorité des processus qui signalent une condition**

**Typologie des principales sortes de moniteur**

**Référence : P. BUHR, M. FORTIER, M. COFFIN. Monitor Classification, *ACM Computing Surveys*, Vol.27, No.1, pp. 63-107, 1995**

# OBJETS PROTEGES (MONITEURS DE ADA 95)

**rappel : il faut distinguer deux sortes d'objets :**
**objets actifs comme les processus et les tâches**
**objets passifs comme les moniteurs, les ressources, les objets protégés**

## objets protégés et types protégés

- **ce sont des objets partageables**
- **les accès (fonctions, procédures et entrées) sont en exclusion mutuelle**
- **les fonctions (accès en lecture seule) peuvent être concurrentes (si le compilateur implante un schéma lecteurs -rédacteurs)**
- **l'exécution de procédure est contrôlable par des gardes ("barriers") et dans ce cas, on utilise la clause :**

  **entry e when expression_booléenne is...**
- **indéterminisme si plusieurs entry et plusieurs gardes sont vraies**
- **possibilité de ne pas exécuter le retour de procédure et de poursuivre l'exécution plus tard par une autre entry :**

  **instruction requeue**

  **ce qui permet par exemple d'examiner les paramètres effectifs d'un appel avant de compléter la réponse en fonction de ces paramètres**
- **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 ( cf expressions de chemin) : chaque transition demandée à l'automate est sous contrôle des gardes; le code associé à une transition s'exécute en exclusion mutuelle, c'est une section critique.**
- **facilité de preuve du genre**

  $$\frac{\{P\}\ S\ \{Q\},\ Q \Rightarrow P}{\{P \text{ et } B\}\ \text{entry X when B is begin S; end; } \{P\}}$$

  **si on sait traiter aussi l'indéterminisme, P est un invariant pour l'objet protégé (domaine à approfondir)**

# 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", de même que les appels arrivant pendant un accès en cours;**
- **si un appel est fait sur une entrée, alors la garde est évaluée;**
  - **l'évaluation de la garde est en section critique protégée par le verrou;**
  - **si la garde est vraie, l'entrée est exécutée et comme tous les accès à l'objet protégé, 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 de l'objet protégé 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) est choisie et l'accès exclusif à l'objet lui est attribué;**
  - **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 (il peut donc y avoir inversion de priorité si les tâches en attente interne ont une priorité plus faible qu'une tâche en attente externe);**
- **à la fin d'une fonction, les gardes ne sont pas réévaluées puisque la fonction ne modifie pas de valeur.**

- **attention aux appels emboîtés croisés : danger d'interblocage. Pour le limiter, il est interdit d'appeler une opération potentiellement bloquante (exemple une entrée, un accept,..) depuis une opération protégée.**

- **existence d'une clause "abort" pour avorter une tâche. Cela a des effets sur un appel d'un objet protégé**

## 1. OBJETS PROTÉGÉS : EXEMPLES INTRODUCTIFS

# Exemple simple

```
Protected variable is
        function LIRE return Item;
        procédure ECRIRE (nouveau : Item);
private
        valeur : Item;
end variable;

Protected body variable is
        function LIRE return Item is
        begin
                return valeur;
        end LIRE ;
        procédure ECRIRE (nouveau : Item) is
        begin
                valeur := nouveau ;
        end ECRIRE ;
end variable;
```

# Exemple simple amélioré

```
Protected variable_fiable is
        entry LIRE(x : out Item);
        procédure ECRIRE (nouveau : in Item);
private
        valeur : Item;
        initialisé: boolean := false;
end variable_fiable;

Protected body variable_fiable is
        entry LIRE (x : out Item) when initialisé is
        begin
                x :=valeur;
        end LIRE ;
        procédure ECRIRE (nouveau : Item) is
        begin
                valeur := nouveau ; initialisé := true;
        end ECRIRE ;
end variable_fiable;
```

# Exemple de compteur incrémental

**(horloge logique, fournisseur de nom unique)**

```
protected NUMERO is
        procédure UNIQUE(RESULTAT: out POSITIVE);
private
        COMPTE : INTEGER := 0; -- le premier résultat fourni sera 0
end NUMERO;

protected body NUMERO is
        procédure UNIQUE(RESULTAT: out POSITIVE) is
        begin
                RESULTAT := COMPTE;
                COMPTE := COMPTE + 1;
        end UNIQUE;
end NUMERO;

-- plus généralement, avec un type
protected type TYPE_COMPTEUR is
      ... -- comme ci-dessus
end TYPE_COMPTEUR ;

NUMERO_1, NUMERO_2 : TYPE_COMPTEUR;
        -- déclare deux compteurs
type BEAUCOUP is array(1..1000) of TYPE_COMPTEUR ;
X: BEAUCOUP; -- déclare un tableau de 1000 compteurs

–
exemple d'utilisation
Mon_Id: Integer;
Mon_Id := NUMERO_1.UNIQUE;
```

## 2. PARADIGME PRODUCTEURS CONSOMMATEURS

# Bounded Buffer Example

```
Protected type Bounded_Buffer is
        entry Put (X : In Item);
        entry Get (X : out Item);
private
        A : Item_Array (1 .. max);
        I, J : Integer range 1 .. max := 1;
        Count : Integer range 0 .. max := 0;
end Bounded_Buffer;

Protected body Bounded_Buffer is
        entry Put (X : In Item) when Count < max is
        begin
                A(I) := X;
                I := I mod max + 1;
                Count := Count + 1;
        end Put ;
        entry Get (X : out Item) when Count > 0 is
        begin
                X := A(J);
                J := J mod max + 1;
                Count := Count - 1;
        end Get ;
end Bounded_Buffer;


-- exemple d'utilisation
declare Z : Item; Tampon : Bounded_Buffer;
   begin prepare(Z) ; Tampon.Put(Z); ; end ;
```

**Exercices :**

**lecteurs-rédacteurs avec différentes priorités, variantes avec requeue**

**priorité aux lecteurs en attente (p 218 Burns Davies)**

# 3. ALLOCATION DE RESSOURCES BANALISEES

## exemple de "preference control" (comment traduire en français ?)

**• on illustre ici l'examen des paramètres d'une requête tout en maintenant l'appelant bloqué quand la réponse n'est pas possible immédiatement.**

**En Ada 83, il fallait plusieurs appels de l'appelant pour gérer un tel cas et cela posait des problèmes si ces appels successifs n'étaient pas respectés(exception, abort, échéance).**

**En Ada 95, ce cas est entièrement géré par l'appelé grâce à la clause requeue. C'est beaucoup plus clair et plus fiable.**

**• cette solution ne fournit que le droit de demander x ressouces; il faut donc ajouter ensuite l'allocation proprement dite par un autre objet protégé, connu ou non du processus appelant**

**si connu on aura un objet à coté du contrôleur**

**si pas connu, objet emboité, appelé par le contrôleur et défini dans sa partie private et il faut un paramètre de plus pour demander et libérer, la liste des ressources allouées**

**VARIANTE 1 : on sert autant de requêtes que possible et si une requête ne peut être servie on passe quand même à la suivante.**

**VARIANTE 2 : on sert selon l'ordre des demandes et si une requête ne peut être servie, on bloque toutes les suivantes quelles qu'elles soient.**

**VARIANTE 3 : on sert selon la priorité des tâches en attente (la priorité est utilisée par la mise en attente interne sur chaque entrée)**

**Exercice : Cette allocation peut mener à interblocage. Ajouter une méthode de prévention d'interblocage**

```
==============================================
   -- allocation de ressources controlée par un objet protégé
                  -- PROGRAMME PRINCIPAL
==============================================

with Ada.Numerics.Discrete.Random; -- pour le tirage aléatoire
with control1; use control1; -- ou control2 ou control3 selon variante

procedure resource_allocation_management is

   package R_Number_Random is new
                              Ada.Numerics.Discrete.Random(r_number);
   task type user_type(x : r_number := r_number'first);
   task body user_type is
      R : resource ;
   begin
      -- ...
      controller.request(R, x);
      --...
      controller.release(R, x);
   end user_type ;

   G : R_Number_Random.Generator;
                           -- uniforme de r_number'first à r_number'last
   user : array(1..20) of user_type
                 (x => R_Number_Random.Random(G));
begin        -- on a créé vingt tâches et elles sont activées
    null;
end resource_allocation_management ;
```

```
================================================
          -- ALLOCATEUR DE RESSOURCES
-- allocation de ressources controlée par un objet protégé
-- variante1 : on consulte les requêtes en attente selon l'ordre d'attente
-- et on sert tant qu'il y a des ressources disponibles
-- il peut y avoir famine pour de grosses requêtes
                    -- paquetage de spécification
================================================

package control1 is
   Max : constant integer := 100; -- number of resources
   subtype r_number is Integer range 1 .. Max;
   subtype free_number is Integer range 0 .. Max;
   type resource is new Integer;

   protected controller is
      entry request(R : out resource; number : r_number);
      procedure release(R : in resource; number : r_number);
   private
      entry assign(R : out resource; number : r_number);
      new_resources_released : boolean := false ;
      free : free_number := free_number'last ;
      to_try : natural := 0 ;
   end controller ;

end control1 ;
```

```
        ================================================
-- allocation de ressources controlée par un objet protégé : variante1
        ================================================
package body control1 is
protected body controller is

    entry request (R : out resource; number : r_number)
                    when free > 0 is
    begin
        if number <= free then
            free := free - number ; -- allocation
            -- allocate_resources(R) ;
        else
            requeue assign ;
        end if;
    end request ; -- request(R,1) n'est exécutée que si free>0,
-- il n'y a jamais requeue assign(R,1), l'attente minimale y est 2 ressources

    entry assign(R : out resource; number : r_number)
                    when new_resources_released and (free > 1) is
    begin
        to_try := to_try - 1 ;
        if to_try = 0 then
            new_resources_released := false; -- la file a été parcourue
        end if;
        if number <= free then
            free := free - number ; -- allocation
            -- allocate_resources(R) ;
        else
            requeue assign ; -- valable si files d'attente à l'ancienneté
        end if;
    end assign ;

    procedure release(R : in resource; number : r_number) is
    begin
        free := free + number ;  -- free resources
        -- free_resources(R);
        to_try := assign'count ;
        new_resources_released  := assign'count > 0 ;
    end release;
end controller ;

begin
    null;
end control1 ;
```

```
          ============================================
                    -- ALLOCATEUR DE RESSOURCES
-- allocation de ressources controlée par un objet protégé
-- variante 2 avec service selon l'ordre de la file de request
-- une tâche prioritaire doit attendre que la tâche en attente
-- sur assign, quelle que soit sa priorité, soit servie
-- mais elle n'attend qu'une requête (qu'une tâche)
                              -- paquetage de spécification
          ============================================
```

```
package control2 is
   Max : constant integer := 100; -- number of resources
   subtype r_number is integer range 1 .. Max;
   subtype free_number is integer range - Max .. Max;
   type resource is new Integer ;

   protected controller is
      entry request(R : out resource; number : r_number);
      procedure release(R : in resource; number : r_number);
   private
      entry assign(R : out resource; number : r_number);
      free : free_number := free_number'last ;
   end controller ;
end control2;
```

```
================================================
-- allocation de ressources controlée par un objet protégé : variante2
================================================

package body control2 is
   protected body controller is

      entry request (R : out resource; number : r_number)
                          when assign'count = 0 is
      begin
         free := free - number ; -- allocate or register request
         if free < 0 then
            requeue assign ;
         else
            -- allocate_resources(R) ;
         end if;
      end request ;

      entry assign(R : out resource; number : r_number)
                    when free >= 0 is
      begin
         -- allocate_resources(R) ;
      end assign;

      procedure release(R : in resource; number : r_number) is
      begin
         free := free + number ;  -- free resources
         -- free_resources(R);
      end release;
   end controller ;

begin
   null;
end control2 ;
```

```
==============================================
-- ALLOCATEUR DE RESSOURCES
-- allocation de ressources controlée par un objet protégé
-- variante 3 avec service selon la priorité des tâches en attente
-- Cependant à cause du modèle de l'oeuf pour les objets protégés, une
-- tâche prioritaire qui appelle request après une tâche qui fait release
-- devra attendre que toutes les tâches bloquées sur assign aient essayé
-- de prendre les ressources rendues. Si toutes les ressources sont alors
-- consommées par celles-ci, la tâche prioritaire ne pourra être servie.
                    -- paquetage de spécification
==============================================

package control3 is
   Max : constant integer := 100; -- number of resources
   type r_number is integer range 1 .. Max;
   subtype free_number is integer range 0 .. Max;
   subtype resource is new Integer ;

   protected controller is
      entry request(R : out resource; number : r_number);
      procedure release(R : in resource; number : r_number);
   private
      entry assign(R : out resource; number : r_number);
      free : free_number := free_number'last ;
      Open_Barrier : boolean := false;
   end controller ;
end control3 ;
```

```
==================================================
-- allocation de ressources controlée par un objet protégé : variante3
==================================================

package body control3 is
protected body controller is

   entry request (R : out resource; number : r_number) when true is
   begin
      if number <= free then
         if assign'count = 0 then      -- pas de tâche en attente
            free := free - number ; -- allocation possible
            -- allocate_resources(R) ;
         else                    -- une tâche en attente peut être plus prioritaire
            Open_Barrier := true;           --  que cette nouvelle tâche
            requeue assign ;
         end if;
      else
         Open_Barrier  := false;
         requeue assign ;
      end if;
   end request ;

   entry assign(R : out resource; number : r_number)
         when Open_Barrier is
   begin
      if number <= free then
         free := free - number ; -- allocation possible
            -- allocate_resources(R) ;
      else
         Open_Barrier  := false;
         requeue assign ;
      end if;
   end assign;

   procedure release(R : in resource; number : r_number) is
   begin
      free := free + number ;  -- free resources
      -- free_resources(R);
      Open_Barrier  := true;
   end release;
end controller ;
begin
   null;
end control3 ;
```

```
-- 4. REPAS DES PHILOSOPHES (REPAS ASSIS)
--protocole externe, par les clients

with scène_philo; use scène_philo; -- voir in fine
procedure philosophes_assis is   -- solution avec N-1 chaises
   N : constant integer := 5;
   subtype id is integer range 1..N;
   protected type ressource is
      entry prendre;
      procedure rendre;
   private
      pris : boolean := false;
   end ressource;
   protected body ressource is
      entry prendre when not pris is
      begin pris := true; end prendre;
      procedure rendre is
      begin
         if pris then pris := false; else raise PAS_PRISE; end if;
      end rendre;  -- lève une exception si la ressource n'est pas prise
   end ressource;
   baguette : array (id) of ressource;
   protected chaise is
      entry prendre;
      procedure rendre;
   private
      libre : natural := N-1;
end chaise;
   protected body chaise is
      entry prendre when libre > 0 is
      begin libre := libre - 1; end prendre;
      procedure rendre is
      begin
         if libre < N-1 then libre := libre + 1; else raise OH!; end if;
      end rendre;  -- lève une exception trop de chaises
   end chaise;
```

```
-- REPAS DES PHILOSOPHES (REPAS ASSIS)
-- protocole externe (suite)

   protected numéro is
      procédure unique(résultat: out id);
   private
      compte : integer := 1; -- le premier numéro fourni est 1
   end numéro;

   protected body numéro is
      procédure unique(résultat: out positive) is
      begin
         résultat := compte;
         compte := compte + 1;
      end unique;
   end numéro;

   task type philo;
   philosophe: array (id) of philo;
   task body philo is gauche, droite : integer;
   begin
      numéro.unique(gauche);
      if gauche = N then droite := 1; else droite := gauche + 1; end if;
      for I in (1.. 57) loop
         pense(gauche); -- simulation dans scène_philo
         chaise.prendre; -- pour s'asseoir
         baguette(gauche).prendre;
         baguette(droite).prendre;
         mange(gauche); -- simulation dans scène_philo
         baguette(gauche).rendre;
         baguette(droite).rendre;
         chaise.rendre;
      end loop;
   end philo;
begin null;
end philosophes_assis;
```

# SIMULATION DU COMPORTEMENT DES PHILOSOPHES

```
package scène_philo is
   procedure pense (x : in integer);
   procedure mange (x : in integer);
end scène_philo ;

with Ada.Text_IO; use Ada.Text_IO;
package Int_IO is new Integer_IO; use Int_IO;
use Ada.Numerics.Float_Random;

package body scène_philo is
   G1, G2 : Generator; -- distribution uniforme de 0.0 à 1.0

   procedure pense (x : in integer) is
      durée : Float range 0.0 .. 1.0;
   begin
      durée := Random(G1); -- nouveau tirage aléatoire d'une durée
      put ("le philosophe  "); put(x); put (" pense"); new_line;
      delay(duration(durée)); -- x pense un certain temps
      put ("le philosophe  "); put(x); put (" a faim"); new_line;
   end pense;

   procedure mange (x : in integer) is
      durée : Float range 0.0 .. 1.0;
   begin
      durée := Random(G2); -- nouveau tirage aléatoire d'une durée
      put ("le philosophe  "); put(x); put (" mange"); new_line;
      delay(duration(durée)); -- x mange un certain temps
      put ("le philosophe  "); put(x); put (" n'a plus faim"); new_line;
   end mange;

end scène_philo ;
```

## PHILOSOPHES AVEC ATTENTE ACTIVE DES BAGUETTES

```
with scène_philo; use scène_philo;
procedure repas_philosophe is
   N : constant integer := 5;
   subtype id is integer range 0..N-1;
   protected repas is
       entry demander(gauche, droite : in id);
       procedure conclure(gauche, droite : in id);
   private
       libre : array(id) of boolean := (others => true);
       notifié : boolean := false;
       entry attendre (gauche, droite : in id);
   end repas;
    protected numéro is
       procédure unique(résultat: out id);
    private
       compte : id := id'first; -- le premier numéro fourni est 0
    end numéro;
   task type philo;
   philosophe: array (id) of philo;
   task body philo is ego, la_gauche, la_droite : id;
   begin
       numéro.unique(ego); -- le numéro du philosophe
       la_gauche := ego; --la fourchette du philosophe
       la_droite := (la_gauche + 1) mod N; -- la fourchette de son voisin
       for I in (1.. 72) loop
           pense(ego); -- simulation dans scène_philo
           repas.demander(la_gauche, la_droite);
           mange(ego); -- simulation dans scène_philo
           repas.conclure(la_gauche, la_droite);
       end loop;
   end philo;
   protected body repas is separate;
   protected body numéro is separate;
begin null;
end repas_philosophe;
```

```
separate(repas_philosophe )
protected body repas is
   entry demander(gauche, droite : in id) when true is
   begin
      if libre(gauche) and libre(droite) then
         libre(gauche) := false; libre(droite) := false;
      else
         requeue attendre(gauche, droite);
      end if;
   end demander;
   entry attendre (gauche, droite : in id) when notifié is
   begin
      if attendre'COUNT = 0 then notifié := false; end if;
      requeue demander(gauche, droite);
   end attendre;
   procedure conclure(gauche, droite : in id) is
   begin
      libre(gauche) := true; libre(droite) := true;
      if attendre'COUNT > 0 then notifié := true;
   end conclure;
end repas;

separate (repas_philosophe)
protected body numéro is
   procedure unique(x : out id) is
   begin
      x := compte;
      if compte < id'last then compte:= (compte + 1); end if;
   end unique;
end numéro;
```

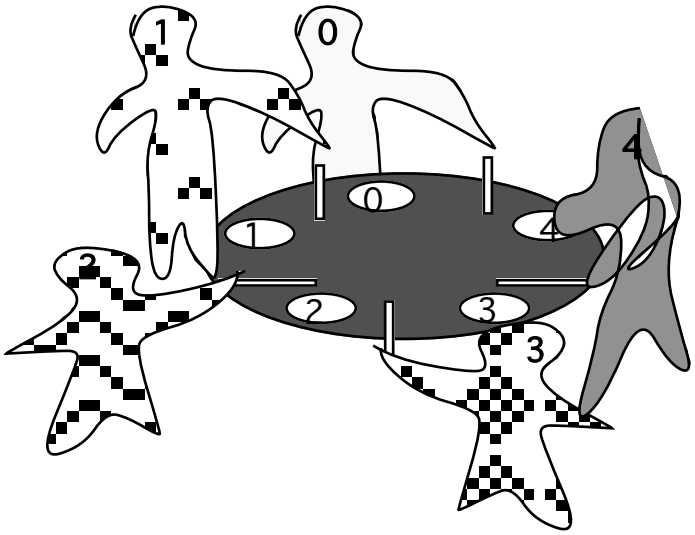
**Ce programme est correct puisque, selon la sémantique de Ada 95, tout nouvel appel de repas.demander par un philosophe ne sera pas servi avant les appels qui étaient bloqués par attendre et libérés par l'exécution de repas.conclure.**

**On peut aussi utiliser l'entrée attendre comme l'entrée assign dans l'allocation de ressources, c'est à dire y réutiliser requeue attendre**

**On a ici une programmation, à la Java, avec attente pseudo-active**

```
-- REPAS DES PHILOSOPHES
-- protocoles internes

N : Constant Integer := 5;
type philo_id is mod N;

procedure Application is

   next_id : philo_id := philo_id'first;
   function unique_id return philo_id is
   begin
      next_id := next_id + 1 ; -- addition modulo N
      return next_id ;
   end unique_id;

   task type philo(x : philo_id  := unique_id) ;
   philosophe : array(philo_id) of philo ;

   use Protected_Object_Chops
            -- ou encore use chopsticks_object
                  -- ou encore use Chopsticks_Server
   task body philo is
   begin
      loop
         thinking;
         request(x) ;
         eating ;
         release(x) ;
      end loop ;
   end philo ;

end Application ;
```

# REPAS DES PHILOSOPHES
## protocole interne ("preference control")



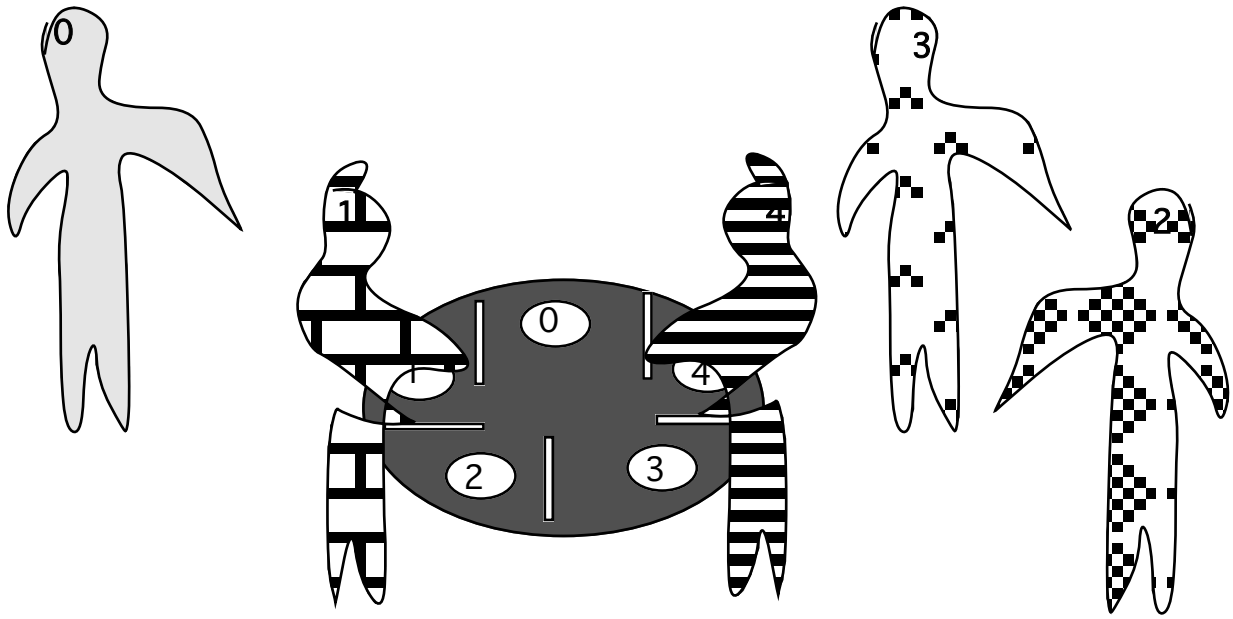

**allocation une baguette après l'autre**

**pas d'interblocage avec les objets protégés (!)**
**car protocole interne à l'objet protégé**
**et utilisation de sa sémantique**

```
package chopsticks_object is
   procedure request(me : in philo_id) ;
   procedure release(me : in philo_id) ;
end chopsticks_object ; --end of the package specification (ads)
```

```
-- REPAS DES PHILOSOPHES
-- protocole interne et une baguette après l'autre

package body chopsticks_object is
   type boolean_array is array(philo_id) of boolean ;

   protected chopsticks is
      entry get_pair(philo_id); -- famille d'entrées
      procedure release_pair(x : in philo_id);
   private
      available : boolean_array := (others => true);
      entry finish_pair(philo_id);
   end chopsticks ;

   procedure request(me : in philo_id) is
      begin  chopsticks.get_pair(me); end request;

   procedure release(me : in philo_id) is
   begin  chopsticks.release_pair(me); end release

   protected body chopsticks is
      entry get_pair(for i in philo_id) when available(i) is
      begin
         available(i) := false;
         requeue finish_pair(i + 1);
      end get_pair;

      entry finish_pair(for i in philo_id) when available(i) is
      begin
         available(i) := false;
      end finish_pair;

      procedure release_pair(x : in philo_id) is
      begin
         available(x) := true;
         available(x + 1) := true;
      end release_pair;

   end chopsticks ;

end chopsticks_object ;  -- end of package body
```

**-- REPAS DES PHILOSOPHES**
**-- protocole interne et allocation gobale**




**allocation globale**

**available (x) and available(x + 1)**
**famine**

**available (x) and available(x + 1) and not requestor(x - 1)**
**ni famine ni interblocage**
**(Courtois, Georges, On starvation prevention, RAIRO vol 11, 1977)**

```
package Protected_Object_Chops is
   procedure request(me : in philo_id) ;
   procedure release(me : in philo_id) ;
end Protected_Object_Chops ; --end of the package specification (ads)
```

```
   N : constant integer := 5 ; --number of philosophers
   type philo_id is mod N;
   type boolean_array is array(philo_id) of boolean ;
package body Protected_Object_Chops is
   protected server is
      entry get_pair(x : in philo_id);
      procedure release_pair(x : in philo_id);
   private
      available : boolean_array := (others => true); -- chopsticks
      requestor : boolean_array := (others => false); -- philosophers
      entry please_get_pair(x : in philo_id); -- private entry
      flush_count: natural := 0;
   end server;
   procedure request(me : in philo_id) is
   begin server.get_pair(me); end request;
   procedure release(me : in philo_id) is
   begin server.release_pair(me ); end release;

   protected body server is

      entry get_pair(x : in philo_id) when true is
      begin
         if available(x) and available(x + 1) and not requestor(x-1) then
            available(x) := false; available(x + 1) := false;
         else
            requestor(x) := true;
            requeue please_get_pair ;
         end if;
      end get_pair;

      entry please_get_pair(x : in philo_id) when flush_count > 0 is
      begin
         flush_count := flush_count - 1;
         if available(x) and available(x + 1) and not requestor(x-1) then
            available(x) := false; available(x + 1) := false;
            requestor(x) := false;
         else
            requeue please_get_pair ;
         end if;
      end please_get_pair;

      procedure release_pair(x : in philo_id) is
      begin
         available(x) := true; available(x + 1) := true;
         flush_count := please_get_pair'count;
      end release_pair;
   end server;
end Protected_Object_Chops ;
```

```
-- 5. TRAITE-INTERRUPTION
-- comptage des interruptions arrivées

-- comptage des interruptions : nombre pendant 10 dernieres secondes
-- voir livre Ada as a second language (Cohen)

-- le paquetage reception est une unité de bibliothèque
-- il contient l'objet protégé recepteur qui
--  recoit toutes les interruptions de SIGUSR : Interrupt_Id
--  et les fournit à la demande une à une quand il y en a

with System.OS_Interface ; use System.OS_Interface ;
package reception is
   protected recepteur is
      procedure Accepter_Ping ;
      entry Fournir_Ping ;
      pragma Attach_Handler (Accepter_Ping, SIGUSR) ;
   private
      Pings_Recus : Natural := 0;
   end recepteur ;
end reception ;

package body reception is
   protected body recepteur is
      procedure Accepter_Ping is -- appelée à chaque interruption
      begin
         Pings_Recus := Pings_Recus + 1 ;
      end Accepter_Ping ;
      entry Fournir_Ping when Pings_Recus > 0 is
      begin
         Pings_Recus := Pings_Recus - 1 ;
      end Fournir_Ping ;
   end recepteur ;
end reception ;
```

```
                        -- comptage des interruptions arrivées
-- la procedure Compte_Ping est une unité de bibliothèque
-- elle compte les Pings recus pendant les 10 dernières secondes
-- ce compte est appelé En_Dix_Secondes
-- elle rafraîchit ce compte toutes les secondes
-- et elle imprime sa nouvelle valeur toutes les secondes

with reception; use reception;
with Ada.Calendar; use Ada.Calendar;
                        -- fournit la date (Clock) et le type Time
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Compte_Ping is
   type Intervalle is mod 10; -- index de comptage dans le tableau
   Dernieres_Secondes : array (intervalle) of natural := (others => 0);
   Prochain : Intervalle := 0 ;
   Prochaine_Date : Time := Clock + 1.0;
   En_Dix_Secondes, En_Une_Seconde : natural := 0;

begin
   loop
      select
         recepteur.Fournir_Ping ;
         En_Une_Seconde := En_Une_Seconde + 1;
               -- reçoit un Ping s'il y en a, sinon attend son arrivée
               -- jusqu'à la date Prochaine_Date
      or
         delay until Prochaine_Date ;          -- la date est arrivée
         Prochaine_Date := Prochaine_Date + 1.0; -- plus une seconde
         En_Dix_Secondes := En_Dix_Secondes -
                           Dernieres_Secondes(Prochain ) +
                           En_Une_Seconde ;
               -- ancienne valeur
               -- moins le plus ancien intervalle (le dixième plus ancien)
               -- plus le nouveau
            -- prochain indique la tête et la queue du tableau circulaire
            -- car il est toujours plein
         Put(En_Dix_Secondes);
         Dernieres_Secondes(Prochain ) := En_Une_Seconde ;
            -- on écrase le dixième plus ancien intervalle par le nouveau
         Prochain  := Prochain + 1 ;
         En_Une_Seconde := 0;
      end select;
   end loop;
end Compte_Ping ;
```

```
-- ALLOCATION DISQUE AVEC TRAITE INTERRUPTION
---------- voir Ada 95 rationale p 9-19
with Ada.Interrupts; use Ada.Interrupts;
generic
   type piste is (<>); -- type discret
   type tampon is private;
protected type sous_controleur_du_disque is
   entry amorcer;
   entry écrire(où : in piste ; donnée : in tampon );
   entry lire(où : in piste ; donnée : out tampon );
   procedure interruption_venue_du_disque;
      -- appelée quand le disque a envoyé une interruption indiquant
      -- que la tête du disque a atteint la piste demandée
pragma ATTACH_HANDLER(interruption_venue_du_disque, IT_disk);
private
   entry amorcer_maintenant;
   entry écrire_maintenant(où : in piste ; donnée : in tampon );
   entry lire_maintenant(où : in piste ; donnée : out tampon );
   procedure DISK_IO_SET(x : piste);
   procedure DISK_IO_WRITE(x : in tampon);
   procedure DISK_IO.READ (x : out tampon);
   position : piste; -- piste où est placée la tête du disque
   transfert_en_cours : boolean := false; -- le disque est utilisé
   interruption_reçue : boolean := false;
      -- notification de la fin du déplacement demandé de la tête
end sous_controleur_du_disque;

protected body sous_controleur_du_disque is
   procedure DISK_IO_SET(x : piste) is
   begin null; end DISK_IO_SET; -- simulation
   procedure DISK_IO_WRITE(x : in tampon) is
   begin null; end DISK_IO_WRITE; -- simulation
   procedure DISK_IO_READ (x : out tampon) is
   begin null; end DISK_IO_READ ; -- simulation
   procedure interruption_venue_du_disque is
   begin interruption_reçue := true; end interruption_venue_du_disque;
   entry amorcer when not transfert_en_cours is
      begin
         DISK_IO_SET(piste'FIRST);
         interruption_reçue := false; transfert_en_cours := true;
            -- empêche toute autre opération avec le disque
         requeue amorcer_maintenant; -- pour attendre l'interruption
      end amorcer;
   entry amorcer_maintenant when interruption_reçue is
      begin position := piste'FIRST; transfert_en_cours := false;
   end amorcer_maintenant;
```

**-- ALLOCATION DISQUE AVEC TRAITE INTERRUPTION**

```
   entry écrire_maintenant(où : in piste ; donnée : in tampon )
      when interruption_reçue is
   begin
      position := où; DISK_IO_WRITE(donnée);
      transfert_en_cours := false;
   end écrire_maintenant;
   entry écrire(où : in piste ; donnée : in tampon )
      when not transfert_en_cours is
   begin
      if position = où then DISK_IO_WRITE(donnée); -- tête en place
      else
         DISK_IO_SET(où);
         interruption_reçue := false; transfert_en_cours := true;
            -- empêche toute autre opération avec le disque
         requeue écrire_maintenant; -- pour attendre l'interruption
      end if;
   end écrire;
   entry lire_maintenant(où : in piste ; donnée : out tampon )
      when interruption_reçue is
   begin
      position := où; DISK_IO_READ(donnée);
      transfert_en_cours := false;
   end lire_maintenant;
   entry lire(où : in piste ; donnée : out tampon )
      when not transfert_en_cours is
   begin
      if position = où then DISK_IO_READ(donnée); -- tête en place
      else
         DISK_IO_SET(où);
         interruption_reçue := false; transfert_en_cours := true;
            -- empêche toute autre opération avec le disque
         requeue lire_maintenant; -- pour attendre l'interruption
      end if;
   end lire;
end sous_controleur_du_disque;
```

**exercices**

**• 1. tenir compte du fait qu'un appelant peut être aborté par un transfert asynchrone. Utiliser requeue .. with abort. et écrire_maintenant'count = 0**

**• 2. programmer la stratégie de l'ascenseur**

## 6. AUTRES MECANISMES DE SYNCHRONISATION

# Persistent Signal Example

```
protected EVENT is
    entry WAIT;
    procedure SIGNAL;
private
    OCCURRED: BOOLEAN := FALSE;
end EVENT;

protected body EVENT is
    entry WAIT when OCCURRED is
    begin
       OCCURRED := FALSE; -- consomme le signal
    end WAIT;
    procedure SIGNAL is
    begin
       OCCURRED := TRUE;
    end SIGNAL;
end EVENT;
```

# Broadcast Signal Example 1

```
protected EVENT is
    entry WAIT;
    entry SIGNAL;
private
    entry RESET;
    OCCURRED: BOOLEAN := FALSE;
end EVENT;

protected body EVENT is
    entry WAIT when OCCURRED is
       begin null; end WAIT;
    entry SIGNAL when TRUE is
    begin
       if WAIT'COUNT > 0 then
          OCCURRED := TRUE;
          requeue RESET;
       end if;
    end SIGNAL;
-- attendre que tous les appelants bloqués sur wait aient été libérés
    entry RESET when WAIT'COUNT = 0 is
    begin
       OCCURRED := FALSE;
    end RESET;
end EVENT;
```

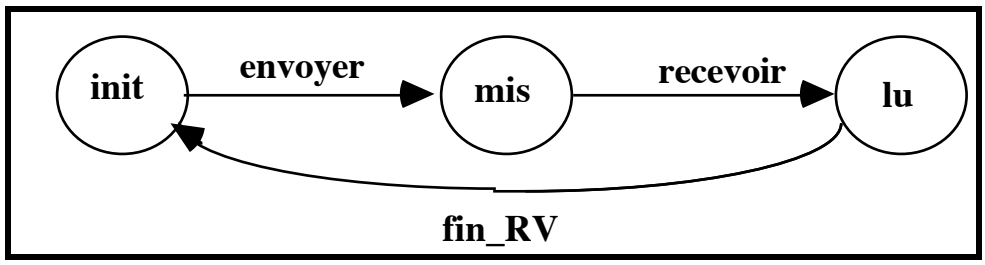
# Broadcast Signal Example 2 (another solution)

```
protected EVENT is
    entry WAIT;
    entry SIGNAL;
end EVENT;

protected body EVENT is
    entry WAIT when SIGNAL'COUNT > 0 is
    begin
       null;
    end WAIT;
    entry SIGNAL when WAIT'COUNT = 0 is
    begin
       null;
    end SIGNAL;
end EVENT;

-- c'est plus simple et plus fiable
-- car traite le cas où un appelant du wait est aborté pendant l'attente
```

**Les deux solutions sont équivalentes de par la sémantique de Ada 95.**
**Imaginons T1 et T2 bloquées sur EVENT.WAIT;**
**puis arrive T3 par EVENT.SIGNAL; T3 "réveille" T1 et T2;**
**pendant ce "réveil", appel de EVENT.WAIT par T4**
**Comme l'attente interne prime l'attente externe, l'arrivée de T3 qui met à vrai la garde de EVENT.WAIT, entraîne l'exécution de T1 et T2; puis la fin de T2 remet à zéro WAIT'COUNT, ce qui entraîne l'exécution de RESET (dans la solution 1) ou de SIGNAL dans la solution 2.**
**WAIT'COUNT compte les attentes internes et pas les attentes externes, donc on a l'ordre T1, T2, T3, T4 et T4 attend le prochain appel de EVENT.SIGNAL.**
**Si WAIT'COUNT avait compté toutes les attentes, internes et externes, T4 serait passé avant T3 et n'aurait pas été bloquée (même comportement pour les 2 solutions). Il aurait pu y avoir une sorte de famine pour T3.**

# CANAL SYNCHRONE

```
generic
   type T is private;
protected type synchrone is
   entry envoyer(x : in T);
   entry recevoir(y : out T);
private
   temp : T; type mli is (mis, lu, init); état : mli := init;
   entry fin_RV;
end synchrone;
```




```
protected body synchrone is
   entry envoyer(x : in T) when état = init is
      begin état := mis; temp := x; requeue fin_RV; end envoyer:
   entry recevoir(y : out T) when état = mis is
      begin état := lu; y := temp; end recevoir;
   entry fin_RV when état = lu is
      begin état := init; end fin_RV;
end synchrone;
```

**On a bati un rendez-vous symétrique**
**L'émetteur doit faire envoyer(x) et le récepteur recevoir(y) pour que le transfert synchrone se fasse et que l'émetteur et le récepteur soient libérés.**
**On simule à distance y := x**

# DIFFUSION ET RECEPTION
# DANS UN POOL DE N CANAUX SYNCHRONES

**pool commun à N processus**
**exemple avec deux canaux; généraliser ensuite avec N en générique.**
**On utilise les familles d'entrées et le requeue (deux fois)**

```
N : constant integer:= 2;
subtype id_canal is integer range 1..N; --pour la famille d'entrées
generic
type T is private;
protected type pool_of_canal_synchrone is
   entry envoyer(id_canal) (x : in T); -- envoi synchrone sur un canal
   entry recevoir(id_canal) (y : out T); -- lecture synchrone
   entry diffuser(x : in T); -- diffusion synchrone sur les deux canaux
   entry accepter(y : out T); -- lecture synchrone sur l'un des canaux
private
   entry fin_RV(id_canal);
   entry fin_diffuser;
   temp : array(id_canal) of T;
   type mli is (mis, lu, init);
   état : array(id_canal) of mli := (others => init);
   diffusion : boolean := false;
end pool_of_canal_synchrone;

protected body pool_of_canal_synchrone is

   entry envoyer(for i in id_canal) (x : in T)
         when état(i) = init and not diffusion is
   begin
      état(i) := mis; temp(i) := x; requeue fin_RV(i);
   end envoyer(i);
   entry recevoir(for i in id_canal)(y : out T) when état(i) = mis is
   begin
      état(i) := lu; y := temp(i);
   end recevoir(i);
   entry fin_RV(for i in id_canal) when état(i) = lu is
   begin
      état(i) := init;
   end fin_RV(i);
```

```
   entry diffuser(x : in T) when not diffusion is
   begin
      diffusion := true; requeue poster(x);
   end diffuser;
   -- réserve les canaux et attend leur disponibilité
   entry poster(x : in T) when état(1) = init and état(2) = init is
   begin
      état(1) := mis; état(2) := mis; temp(1) := x; temp(2) := x;
      requeue fin_diffuser; -- diffusion synchrone émetteur bloqué
   end poster;
   entry fin_diffuser when état(1) = lu and état(1) = lu is
   begin
      état(1) := init; état(2) := init; diffusion := false;
   end fin_diffuser;
   entry accepter(y : out T) when état(1) = mis or état(2) = mis is
   begin
      if integer(CALENDAR.Clock) mod 2 = 0 then
      -- valeur aléatoire qui introduit un certain indéterminisme
         if état(1) = mis then état(1) := lu; y := temp(1);
         elsif état(2) = mis then état(2) := lu; y := temp(2); end if;
      else
         if état(2) = mis then état(2) := lu; y := temp(2);
         elsif état(1) = mis then état(1) := lu; y := temp(1); end if;
      end if;
   end accepter;
end pool_of_canal_synchrone;
```

**exercices : 1) dans cette solution, il n'y a pas de respect du FIFO entre envoyer et diffuser; dire pourquoi. Implanter le FIFO**
**2) implanter la priorité de diffuser sur les envoyer**

**autres exercices: rendez-vous multiple 1 appelant N appelés avec réponse et réveil de l'appelant quand toutes les réponses sont revenues**
**analogue à fork-join ou à ferme de processus**

**REALISATION DES SEMAPHORES**

# Sémaphore pour exclusion mutuelle

```
Protected type mutex_sémaphore is
        procédure V;
        entry P;
private
        E : integer := 1;
end mutex_sémaphore;

Protected body mutex_sémaphore is
        procedure V is begin E := E + 1; end V ;
        entry P when E> 0 is begin E := E - 1; end P ;
end mutex_sémaphore;
```

# Sémaphore général avec compteur non négatif

```
Protected type sémaphore is
        entry V;
        entry P;
        entry E0(X : in natural); -- X ≥ 0
private
        E : natural ; -- E ≥ 0 dans cette implantation
        INIT : boolean := false;
end sémaphore;

Protected body sémaphore is
        entry V when INIT is
                begin E := E + 1; end V ;
        entry P when INIT and 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 sémaphore;
```

• exemples :

```
mutex : mutex_sémaphore;
libre, plein : sémaphore;
sempriv : array(1..5) of sémaphore;
for I in (1..5) loop sempriv(I).E0(0); end loop;
mutex.P; sempriv(3).V;
```

# Sémaphore général avec compteur classique

```
Protected type sémaphore is
      entry V;
      entry P;
      entry E0(X : in natural); -- X ≥ 0
private
      INIT : boolean := false; E : integer ; -- dans cette implantation,
-- si E<0 alors abs(E) = nombre de processus bloqués par le sémaphore
      entry P_bloquant;
      réveil : natural := 0; -- réveil : un boolean suffit si on respecte
end sémaphore; -- la priorité des attentes internes sur les externes

Protected body sémaphore is
      entry V when INIT is
            begin
                  E := E + 1; if E <= 0 then réveil := réveil + 1; end if;
            end V ;
      entry P when INIT is
            begin
                  E := E - 1; if E < 0 then requeue P_bloquant; end if;
            end P ;
      entry P_bloquant when réveil > 0 is
            begin réveil := réveil - 1; end P_bloquant;
      entry E0(X : in natural) when not INIT is
            begin E := X; INIT := true; end E0;
end sémaphore;
```

# Sémaphore général avec compteurs monotones

```
Protected type sémaphore is
      entry V;
      entry P;
      entry E0(X : in natural); -- X ≥ 0
private
      INIT : boolean := false;
      NV, NF : natural := 0; -- NF = nombre de P franchis;
            -- dans cette implantation, NF, NV ≥ 0
end sémaphore;

Protected body sémaphore is
      entry V when INIT is begin NV := NV + 1; end V ;
      entry P when INIT and NV > NF is NF := NF + 1; end P ;
      entry E0(X : in natural) when not INIT is
            begin NV := X; INIT := true; end E0;
end sémaphore;                    -- on a l'invariant NF ≤ NV
```

**exercice :** compter avec NP les appels à P et faire une implantation style compteurs d'événements et séquenceur (Reed, Kanodia, CACM 22,2 (1979)), pour préparer une implantation répartie

# Counting Semaphore Example (avec discriminant)

**Ada 95 permet de créer un type avec initialisation de valeur à l'instantiation d'un objet du type (on a aussi une valeur par défaut).**

```
Protected type Counting_Semaphore (Initial : Integer := 1) is
        function Count return natural;
        procedure Release;        -- "V" operation
        entry Acquire;      -- "P" operations
private
        Current_Count : natural := Initial;
end Counting_Semaphore;

Protected body Counting_Semaphore is
        function Count return natural is
        begin return Current_Count; end Count ;
        procedure Release is
        begin Current_Count := Current_Count + 1; end Release ;
        entry Acquire when Current_Count> 0 is
        begin Current_Count := Current_Count - 1; end Acquire ;
end Counting_Semaphore;
```

**• exemples :**

```
mutex : Counting_Semaphore(1);
libre : Counting_Semaphore(N); plein : Counting_Semaphore(0);
sempriv : array(1..5) of Counting_Semaphore(0);
mutex.Acquire; sempriv(3).Release; -- P(mutex); V(sempriv(3));
```

# OBJETS PROTEGES PROGRAMMES AVEC DES SEMAPHORE

**soit :**

```
Protected objet is
        procedure PROC (x : in ITEM);
        entry E1;
        entry E2;
private
        LOC : TYPE_LOC;
end objet ;

Protected body objet is
        procedure PROC (x :  in ITEM) is begin CORPS_PROC; end PROC;
        entry E1 when BARRIERE_E1 is begin CORPS_E1; end E1;
        entry E2 when BARRIERE_E2 is begin CORPS_E2; end E2;
end objet ;
```

## Avec des sémaphores cela se programme (en première approximation):

```
package objet is
        procedure PROC (x : in ITEM);
        procedure E1;
        procedure E2;
end objet ;

Package body objet is
        LOC : TYPE_LOC;
        mutex : sémaphore;  --  c'est le lock du rationale
        sem_E1, sem_E2 : sémaphore;
        attend_E1, attend_E2 : integer := 0;

        procedure PROC (x :  in ITEM) is
        begin
                P(mutex)
                CORPS_PROC;
                -- réévaluation des gardes dans un ordre quelconque
                if attend_E1 > 0 and BARRIERE_E1 then
                        V(sem_E1); attend_E1 := attend_E1 - 1;
                elsif attend_E2 > 0 and BARRIERE_E2 then
                        V(sem_E2); attend_E2 := attend_E2 - 1;
                -- il ne faut qu'un seul élu, qui reçoit l'objet en exclusion mutuelle
                else V(mutex);
                end if;
        end PROC;
```

```
        procedure E1 is
                OK : boolean := true;
        begin
                P(mutex);
                if BARRIERE_E1 then
                        CORPS_E1;
                        -- réévaluation des gardes dans un ordre quelconque
                        if attend_E2 > 0 and BARRIERE_E2 then
                                V(sem_E2); attend_E2 := attend_E2 - 1;
                        elsif attend_E1 > 0 and BARRIERE_E1 then
                                V(sem_E1); attend_E1 := attend_E1 - 1;
                        -- il ne faut qu'un seul élu
                        else
                                V(mutex); -- pas trouvé de processus à élire
                        end if;
                else
                        attend_E1 := attend_E1 + 1;
                        V(mutex);
                        P(sem_E1);  -- au réveil, il a la section critique
                        CORPS_E1;
                        -- réévaluation des gardes dans un ordre quelconque
                        if attend_E2 > 0 and BARRIERE_E2 then
                                V(sem_E2); attend_E2 := attend_E2 - 1;
                        elsif attend_E1 > 0 and BARRIERE_E1 then
                                V(sem_E1); attend_E1 := attend_E1 - 1;
                        -- il ne faut qu'un seul élu
                        else
                                V(mutex); -- pas trouvé de processus à élire
                        end if;
                end if;
        end E1 ;

        procedure E2 is analogue_à_E1; end E2 ;

begin
        E0(mutex, 1); E0(sem_E1, 0); E0(sem_E2,0);
end objet ;

-- à compléter avec le cas requeue
-- voir aussi l'implantation des moniteurs de Hoare en sémaphores
-- ici on a donné priorité aux "entry sur les procédures ou fonctions; à revoir en exercice
```

## Conclusion

**On a simulé un sémaphore avec un objet protégé. On a simulé un objet protégé avec des sémaphores. On peut donc dire que sémaphores et objets protégés peuvent traiter les mêmes problèmes de synchronisation. Ils ont la même puissance d'expression.**

**Hoare dans son article des CACM de 1984 a montré de même l'équivalence entre moniteurs et sémaphores.**

# COMPLÉMENTS DE LECTURE ET DE RECHERCHE POUR ADA

**Pour programmer en ADA et comprendre les commentaires des compilateurs, il faut utiliser le manuel de référence. Celui-ci a été publié en anglais et le sera en français. Toutes les éditions se valent, vous pouvez acheter la moins chère. En français, vous trouverez les éditions (de l'AFNOR - bientôt), de la société AONIX (et celle des Presses Polytechniques Romandes - bientôt). Le manuel est disponible gratuitement sur internet.**

**[ISO 95] ISO. Ada 95 Reference Manual. ISO Standard 8652/1995, ISO, Genève 1995**
**[Barnes 96] J. Barnes. Programming in Ada 95. Addison Wesley 1996 (702 p.)**
**[Bréguet 99] P.Bréguet, L.Zaffalon. *Programmation séquentielle avec Ada95,* 559 p. (Presses Polytechniques et Universitaires Romandes)**
**[Burns 95] A. Burns, A.Wellings. Concurrency in Ad. Cambrige University Prsss, 1995**
**[Cohen 95] N.Cohen. Ada as a Second Language. McGraw-Hill, 1995.**
**[Rosen 95] J.P. Rosen. Méthodes de génie logiciel avec Ada 95. InterEditions, 1995.**
**[Zaffalon 99] L.Zaffalon, P.Bréguet. *Programmation concurrente et temps réel avec Ada95,* 559 p. (Presses Polytechniques et Universitaires Romandes)**

## • ADA SUR INTERNET

**Département informatique du CNAM : deptinfo.cnam.fr/Enseignement**
**Compilateur Ada 95 gratuit (PC, Linux,..) : ftp.cnam.fr/pub/Ada/PAL/compiler/gnat/distrib**
**Free Software foundation : www.gnu.org**
**compilateur GNAT :www.gnat.com**

**Groupe Ada-France : ada-france.org/**
**Groupe Ada Europe page : www.ada-europe.org**
**Adalog perso.: wanadoo.fr/adalog**
**Composants concurrents pour Ada et java : www.averstar.com/tools**
**Ada Resources for Educators and Students : www.acm.org/sigada/education**
**Ada and Software Engineering (CDROM) : ase.isu.edu**
**Repository of free ada software : www.informatik.uni-stuttgart.de/ifi/ps/ada-software/**
**Home of Brave Ada Programmer : www.adahome.com ou lglwww.epfl.ch/Ada**
**Ada Yacc, Adaface : host ftp.cs.tut.fi/pub/src/ASENTO**
**PAL library of Ada and VHDL : www.cdrom.com/pub/ada/pal.html**
**Cours : http://www.asset.com/WSRD/abstracts/ABSTRACT_825.html**
**http://www.asset.com/WSRD/abstracts/ABSTRACT_826.html**