**République Algérienne Démocratique et Populaire**
**Ministère de l'Enseignement Supérieur et de la Recherche Scientifique**

**Université Ferhat Abbas-Sétif 1**  **جامعة فرحات عباس - سطيف1**

**Faculté des Sciences**
**Département d'Informatique**

**Polycopié de la matière**

# SYSTEME D'EXPLOITATION 2
# PROCESSUS CONCURRENTS

## Problèmes avec solutions

**Destiné aux étudiants 3ème année licence LMD**

**Pr Zibouda ALIOUAT**

**2023-2024**

# 1. Introduction

La série d'exercices et problèmes que nous proposons ci-après a pour but de montrer les difficultés rencontrées dans la mise en oeuvre de l'expression de synchronisation au travers des différents mécanismes de synchronisation discutés dans le chapitre 2. Ce but a été conduit selon un aspect pédagogique et académique. C'est pourquoi, quatre classes ou modèles de "système", les plus communément référencés dans ce contexte, ont été particulièrement présentés. Il s'agit du modèle lecteurs-rédacteurs, producteur-consommateur, allocateur de ressources et rendez-vous.

Les algorithmes proposés s'intéressent d'avantage à l'aspect spécification du problème posé plutôt qu'à l'aspect présentation de solutions élégantes et optimales. Le support algorithmique employé est un langage pseudo-pascal utilisant indifféremment les mots clés anglais que leur traduction en français tels que: ***begin (debut)***, ***end (fin), while*** (*tantque*) ***if*** *(si*), *then* (*alors*) *else* (*sinon*) *endif* (*finsi*) etc...

La syntaxe des outils de synchronisation est parfois non conforme à celle des articles originaux, car nous avons voulu privilégier les concepts basiques plutôt que l'aspect implémentation; puisque le programmeur doit de toute évidence se référer aux manuels de référence décrivant avec plus amples détails l'outil à utiliser (par exemple, l'utilisation de l'outil sémaphores est présentée dans les systèmes de type Unix de manière différente par rapport au concept abstrait, pour tenir compte de l'efficacité et l'objectivité requises par la manipulation et l'implémentation).

## 2. Problème des Lecteurs-Rédacteurs

Ce problème, dit de Lecteurs-Rédacteurs (cou et al. 1971), constitue un modèle ou classe de problèmes de synchronisation des processus séquentiels spécifiques à une politique particulière d'accès à un objet partagé. Tout autre problème de synchronisation qui s'apparente à cette politique d'accès forme une variante de ce modèle.

Ce problème traite du partage d'un objet, ici le fichier, par un ensemble d'usagers (ou processus). Il s'agit d'autoriser plusieurs accès simultanés en lecture à un fichier, et de garantir un accès exclusif relativement aux écritures dans ce même fichier. Deux types de processus communément appelés lecteurs et rédacteurs se trouvent donc en compétition pour l'utilisation d'une ressource commune: le fichier. Cette ressource possède donc N points d'accès relativement aux lecteurs et un point d'accès unique vis-à-vis des rédacteurs. Pour cela on définit plusieurs stratégies d'utilisation possibles du fichier.

### 2.1 Solutions à l'aide des sémaphores

#### 2.1.2 Priorité aux lecteurs

Un rédacteur ne peut obtenir l'accès au fichier que lorsque aucun lecteur n'ait manifesté le besoin d'y accéder.

**Une Solution possible**

Soient nblect et nbred le nombre respectif de lecteurs et rédacteurs utilisant une procédure d'accès au fichier. Le maintien de la cohérence des informations du fichier impose de définir les contraintes suivantes:

Pour pouvoir utiliser le fichier en lecture on doit satisfaire la condition suivante: nbred = 0 i.e: aucun rédacteur n'est entrain d'écrire.

La relation suivante doit être vérifiée pour pouvoir accéder au fichier en écriture: (nbred = 0) et (nblect = 0). Les accès au fichier sont soumis au respect du protocole suivant:

| Processus Lecteur: | Processus Rédacteur: |
|---|---|
| ***debut*** | ***debut*** |
| < demande de lecture >; | < demande d'écriture >; |
| < lecture >; | < écriture >; |
| < fin de lecture >; | < fin d'écriture >; |
| ***fin*** | ***fin*** |

Il est à remarquer que: l'accès au fichier en écriture est conditionné par la satisfaction de la relation précédente, donc exclusif. On peut penser à mettre l'opération d'écriture dans une section critique et la cohérence des informations du fichier est ainsi garantie. Toutefois, cette solution ne peut convenir car cela contredirait le principe selon lequel une section

critique doit être aussi courte que possible (nécessite le moins possible de temps d'exécution)!

```
 Var   nblect: integer;
 mutexlec, mutexred, mutexecr:   semaphore;
  nblect:=0;
  mutexlec:=1                  %  Sémaphore de mutuelle exclusion des lecteurs      %
                               %  sert à protéger le compteur nblect                 %
mutexecr:=1                    %  Sémaphore de mutuelle exclusion pour l'accès       %
                               %  des rédacteurs, utilisé par le premier             %
                               %  lecteur pour bloquer l'accès des rédacteurs        %
                               %   et réserver le fichier pour les lecteurs.         %
  mutexred:=1                  %  assure une plus grande priorité aux lecteurs       %
                               %  par coalition; les rédacteurs doivent attendre     %
                               %  qu'aucun lecteur n'est dans le fichier. Le seul    %
                               %  cas ou un rédacteur obtient l'accès au fichier     %
                               %  est lorsque aucun lecteur n'utilise le fichier     %
                               %  et il  n'existe  aucun  lecteur en attente.        %
Procedure   acces_lecture
  begin
    P(mutexlec)                %  Prologue pour protéger la section critique         %
    nblect := nblect+1;        %  mise à jour de la variable globale nblect.         %
    Si  nblect = 1  alors      %   test si premier lecteur  %
       P(mutexecr)             %  Blocage du premier lecteur et les suivants si le   %
    finsi                      %  fichier est déjà occupé par un rédacteur, blocage  %
                               %  des rédacteurs dans le cas contraire               %
    V(mutexlec);               %  Epilogue de la section critique                    %

    < lecture >                % l'opération de lecture est en dehors de la section %
                               %  critique car elle très coûteuse en temps           %
  P(mutexlec);
    nblect:=nblect - 1;
    Si   nblect = 0  alors     % test du dernier lecteur et réveil éventuel d'un    %
       V(mutexecr);            % rédacteur en attente                               %
    fsi
    V(mutexlec);
  end
```

```
Procedure acces_ecriture
 begin
   P(mutexred);              % sémaphore par lequel les lecteurs obtiennent la priorité%
   P(mutexecr);              % Prologue de la section critique où a lieu l'écriture %
   < écrire >;          (*)
   V(mutexecr);              %  Epilogue: libération du fichier
   V(mutexred);
 end
```

(*): L'opération d'écriture est effectuée ici à l'intérieur de la section critique. Cela peut s'avérer temporellement pénalisant si l'implémentation de base du prologue qui protège la section critique est réalisée à l'aide du masquage et démasquage des interruptions.

### 2.1.2 Priorité relative des lecteurs sur les rédacteurs

Priorité des lecteurs sur les rédacteurs si et seulement si un lecteur occupe déjà le fichier. Quand il n'y a aucun lecteur en lecture, il y a égalité des priorités. Par contre dès que un lecteur obtienne l'accès, tous les autres lecteurs peuvent y accéder et ce, quelque soit le nombre de rédacteurs en attente (possibilité de monopolisation du fichier par les lecteurs pouvant ainsi créer la famine des rédacteurs).
La solution est identique qu'en a) sans mutexred.

### 2.1.3 Priorité absolue des rédacteurs sur les lecteurs

Dès que un rédacteur réclame l'accès au fichier, il doit l'obtenir à la fin de l'utilisation du processus en cours. il y a attente des lecteurs arrivant après la demande d'un rédacteur. Il y a possibilité de privation des lecteurs provoquée par la coalition des rédacteurs.

```
Var  nblect, nbred : integer;
 mutexlec, mutexred: semaphore;        % identique que précédemment %
 mutexbloclec : semaphore;             % semaphore ayant pour utilité %
                                       % la réservation par les rédacteurs %
                                       % de l'accès au fichier.          %
mutexblocred: semaphore;               % identique à mutexecr précédent %
mutexpriored: semaphore;               % affecte une priorité absolue aux %
                                       % rédacteurs et assure qu'au plus %
                                       % un lecteur ou un rédacteur est en %
                                       % attente dans la file associée à %
                                       % mutexbloclec pendant qu'un lecteur %
                                       % met à jour le compteur nblect.  %
nblect := 0;
nbred := 0;
```

```
mutexlec := 1;
mutexecr := 1;
mutexbloclec := 1;
mutexpriored := 1;
mutexblocred := 1;

Procedure acces_lecture
 begin
   P(mutexpriored)
   P(mutexbloclec);
   P(mutexlec);
   nblect := nblect + 1;
   Si  nblect = 1  alors
       P(mutexblocred);
   fsi
   V(mutexlec);
   V(mutexbloclec);
   V(mutexpriored);
   < lecture >;
   P(mutexlec);
   nblect := nblect - 1;
   si   nblect = 0  alors
       V(mutexblocred)
   fsi
   V(mutexlec);
 end.

Procedure acces_ecriture
 begin
   P(mutexred);
   nbred := nbred + 1;
   si  nbred=1  alors
      P(mutexbloclec);
   fsi
   V(mutexred);
   P(mutexblocred);
   < écriture >;
   V(mutexblocred);
   P(mutexred);
   nbred := nbred - 1;
   si  nbred = 0  alors  ;
```

```
      V(mutexbloclec);
   fsi
   V(mutexred);
end
```

### 2.1.4 Aucune priorité

Aucune classe de processus n'a de priorité sur l'autre, l'accès au fichier est accordé selon la stratégie du « *premier arrivé, premier servi(FCFS) »,* avec toutefois l'avantage accordé aux lecteurs de se coaliser en l'absence de toute manifestation de rédacteur. Autrement dit, quand le fichier est libre, il est accordé au premier processus qui le demande sans distinction de classe. Cependant, si le fichier est déjà occupé par un lecteur, tout nouveau lecteur qui manifeste le désir d'y accéder peut le faire à condition que sa demande ne soit pas postérieure à celle d'un rédacteur. La coalition des lecteurs peut donc être interrompue par l'arrivée d'un rédacteur. Quand un rédacteur termine l'usage du fichier, il réveille éventuellement le processus qui le suit. Si ce dernier est un lecteur, il y accède et autorise, le cas échéant, l'accès des lecteurs qui le suivent immédiatement dans la file d'attente. Il y accède seul s'il est suivi par un rédacteur.
L'algorithme qui traite ce cas peut être:

```
Procedure  acces_lecture
begin
   P(mutexred);
   P(mutexlec);
   nblect:=nblect+1;
   si  nblect = 1  alors  P(mutexecr);
   fsi
   V(mutexlec);
   V(mutexred);
    < lecture >
   P(mutexlec);
   nblect := nblect - 1;
   si   nblect = 0  alors
      V(mutexecr);
   fsi
  V(mutexlec);
end
```

```
Procedure  acces_ecriture
begin
    P(mutexred);  (*)
    P(mutexecr);
    < écrire >;
   V(mutexecr);
   V(mutexred);
end
```

(*): Le sémaphore mutexred permet de bloquer les deux classes de processus dans la même file d'attente; le réveil dans cette file se fera donc selon la stratégie FIFO.

### 2.1.5 N Producteurs et N Consommateurs

Généralisez le modèle producteur-consommateur pour N producteurs et N consommateurs, et gérez correctement le buffer commun.

**Solution**

Structure de données et initialisation commune :

```
Var  head, tail: integer;
Var  buffer: array[0..N-1];
Semaphore: nplein, nvide, mutexprod, mutexcons;
        head := tail:= 0;
        mutexprod := mutexcons := 1;
Procedure Producteur(msg1)
 begin
  cycle
    Produire(msg1);  % msg1: message  produit %
    P(nvide);
    P(mutexprod);
    buffer[tail] := msg1;
    tail := tail + 1 mod(N);
    V(mutexprod);
    V(nplein);
  endcycle
 end

Procedure Consommateur(msg2)
 begin
  cycle
    P(nplein);
    P(mutexcons);
    msg2 := buffer[head];
    head := head + 1 mod(N);
    V(mutexcons);
    V(nvide);
    < consommer msg2 >;
  endcycle
 end
```

*Remarque*: Dans ces algorithmes, il n'a pas été tenu compte de consommations individualisées de messages, c'est à dire qu'un processus particulier Ci consomme un message particulier Mi. Tout a été banalisé et l'accès au buffer est exclusif.

## 2.2 Solutions à l'aide des Moniteurs

### a) Moniteur classique (Hoare)

```
Lecred.Monitor                  % priorité des rédacteurs sur les lecteurs          %
begin
   Var nblect, nbred: integer;
      lect_ok :  condition;     % lect-ok: condition associée à une lecture possible  %
      ecr_ok  :  condition;     %  ecr-ok:    ''          ''        écriture possible %
 Procedure  acces_lecture
  begin
    if nbred > 0 then  wait(lect_ok);
    nblect := nblect + 1;
    signal(lect_ok);
  end

Procedure  fin_acces_lecture;
  begin
    nblect := nblect - 1;
    if nblect = 0 then signal(ecr_ok);
  end

 Procedure  acces_ecriture;
  begin
    nbred := nbred + 1 ;
    if nblect > 0  or nbred > 1  then wait(ecr_ok);
  end

Procedure  fin_acces_ecriture;
  begin
    nbred := nbred - 1;
    if nbred > 0  then signal(ecr_ok)
    else  signal(lect_ok)
  end
 % Initialisation %
  begin
    nblect := 0;
    nbred  := 0;
  end
endmonitor.
```

**b) Moniteur de Kessels**

```
Lecred.Monitor ;
 begin
   Var   nblect, nbred : integer;
   Var   ecr_en_cours  : boolean;
        lect_ok :  condition  nbred > 0;            %  (*) condition d'attente de tout lecteur %
        ecr_ok   :  condition   nblect > 0 or ecr_en_cours;  % condition d'attente de tout
                                                              rédacteur %
   Procedure   debut_lire;
     begin
      wait(lect_ok);
      nblect := nblect + 1;
     end

   Procedure   fin_lire;
     begin
      nblect := nblect - 1;
     end

   Procedure   debut_ecrire;
     begin
      nbred := nbred + 1;          % cette incrémentation et (*)  précédent permettent %
      wait(ecr_ok);                % d'accorder une priorité aux rédacteurs %
      ecr_en_cours := vrai;
     end
   procedure fin_ecrire;
     begin
      ecr_en_cours := faux;
      nbred := nbred - 1;
     end
  % Initialisation %
  begin
    nblect := nbred := 0;
    ecr_en_cours := faux;
  end
endmonitor.
```

L'utilisation de ces moniteurs (les appels à partir des points de synchronisation) peuvent se faire par les deux classes de processus à savoir: processus lecteurs et processus rédacteurs, comme l'indique l'organisation suivante:

**processus_lecteur**
***begin***
***cycle***
**call** lecred.acces_lecture;
< lecture effective >;
**call** lecred.fin_acces_lecture;
***endcycle***
***end***

**processus_redacteur**
***begin***
***cycle***
**call** lecred.acces_ecriture;
< écriture effective >;
**call** lecred.fin_acces_ecriture
***endcycle***
***end***

Les appels éventuellement simultanés des deux classes de processus peuvent être regroupés dans un même programme principal pouvant avoir la structure de parallélisme suivante:

**main**
***cobegin ou parbegin (parallèle begin)***
processus_lecteur; processus_lecteur; ....
processus_redacteur; processus_redacteur; ...
***coend ou parend***

la structure linguistique ***cobegin*** ... ***coend*** ou parbegin …. Parend indique une simultanéité possible dans l'exécution des actions situées dans ce bloc.

### 2.3 Solution à l'aide des Régions critiques

#### 2.3.1 Priorité aux rédacteurs

```
Var v shared record
        nblect   : integer;
        nbred    : integer;
        nbredatt : integer;          % nbredatt : nombre  de rédacteurs en %
        end                          % attente de l'accès au fichier          %
nblect := 0;
nbred  := 0;
nbredatt := 0;

Procedure  acces_lecture;
 begin
   region  v  when  nbredatt ≤ 0 and nbred ≤ 0 do
                        nblect := nblect + 1;
   < lire >;
   region  v do nblect:=nblect - 1;
 end

Procedure  acces_ecriture;
 begin
  region  v  do
     nbredatt := nbredatt + 1;
  region v when  nblect ≤ 0 and nbred ≤ 0 do
       begin
         nbred := nbred + 1;
         nbredatt := nbredatt -1;
       end

  < écrire >;                        %  L'opération d'écriture est réalisée en dehors %
                                     % de la section critique de contrôle des accès    %
  region v do  nbred := nbred - 1;
 end
```

## 2.4 Solutions à des Modules de Contrôle

De la spécification du problème (voir ci-dessus), on en déduit immédiatement les conditions d'autorisation suivantes:
condition pour lire: *condition*(lire): act(écrire) = 0;
condition pour écrire: condition(écrire): act(lire) + act(écrire)=0.

En ce qui concerne la priorité, elle est spécifiée à l'aide des compteurs, en imposant une contrainte supplémentaire lors des autorisations à la classe des processus de moindre privilège. Pour une priorité des lecteurs on impose en outre: *condition*(écrire): att(lire) = 0.
Le module de contrôle peut s'écrire:

```
lecred.module_de_contrôle
begin
   lire, écrire: procedure;
   queue: lire/écrire;
   condition(lire): act(écrire) = 0;              % i.e: pour autoriser un accès en lecture, %
   condition(écrire): act(écrire)+act(lire)=0     % il est nécessaire qu'il n'y ait aucune    %
                      et att(lire) = 0;           % écriture en cours                         %
end
```

Puisque les compteurs sont des entiers positifs ou nuls; la condition pour écrire peut s'écrire plus simplement:
condition(écrire): act(écrire)+act(lire)+att(lire)=0.
Si on veut accorder une priorité aux rédacteurs, on doit changer la condition d'autorisation des lecteurs qui devient: condition(lire): act(écrire) = 0 et att(écrire) = 0 ce qui donne le mdc suivant:

```
lecred.module_de_contrôle
begin
   lire, écrire: procedure;
   queue: lire/écrire;
   condition(lire): act(écrire) + att(écrire) = 0;
   condition(écrire): act(écrire) + act(lire) = 0;
end
```

Dans le cas où l'on désire favoriser le monopole des lecteurs, le module de contrôle peut être spécifié comme suit:

```
lecred.module_de_contrôle
begin
   lire, écrire: procedure;
   queue: lire/écrire;
   condition(lire): act(écrire) = 0;
```

***condition***(écrire): act(écrire) + act(lire) = 0
***end***

*Remarque*: Dans les trois cas présentés précédemment le problème de famine est possible, ce qui est dû à l'absence de politique de gestion des files d'attente (queue).

### 2.5 Solutions à des Expressions de chemin

En l'absence de considération de priorité, le problème peut être spécifié de la manière suivant:

lecred.***path_expression***
***begin***
lire, écrire: ***procedure***;
***path*** {lire}, écrire ***end***
***end***

L'exclusion mutuelle entre tout couple de procédures (lire, écrire) et (écrire, écrire) est assurée; l'opérateur accolade permet des exécutions simultanées de la procédure (réentrante) lire. On constate donc que les règles qui spécifient la synchronisation afin d'assurer seulement l'intégrité de l'objet partagé "fichier" sont aisément exprimées. Toutefois si l'on veut introduire des priorités d'une classe de processus par rapport à une autre, le langage des expressions de chemin tel qu'il a été présenté ne permet pas de les associer facilement aux règles (contrairement au langage des mdc). Une façon de faire, consiste à contourner le problème en ajoutant des procédures "artificielles" de contrôle sur lesquelles portera la priorité désirée et qui réaliseront l'ordonnancement voulu des procédures originelles.

Ainsi, si on attribue une priorité aux rédacteurs, le problème peut être solutionné comme suit:

***path*** tentative_de_lecture ***end*** (1)
***path*** requête_lecture, { requête_écriture } ***end*** (2)
***path*** {ouvrir_en_lecture, lire}, écrire ***end*** (3)
avec les procédures intermédiaires suivantes:
tentative_de_lecture = ***begin*** requête_lecture ***end***
requête_lecture = ***begin*** ouvrir_en_lecture ***end***
requête_écriture = ***begin*** écrire ***end***
read = ***begin*** tentative_de_lecture; lire ***end***
write= ***begin*** requête_écriture ***end***

Les procédures read et write constituent l'interface entre le contrôleur de synchronisation qui analyse le chemin et les processus appelants. Ces derniers doivent d'abord obtenir la permission d'accès au fichier avant de lancer les opérations de lecture ou écriture proprement dites.

Le chemin (1) permet à tout instant l'occurrence d'une requête unique de lecture. Pendant qu'un processus initie une requête de lecture, tous les autres lecteurs doivent attendre jusqu'à ce qu'ils aient lancé une tentative de lecture. cette situation permet à une requête d'écriture d'être acceptée le plutôt possible au niveau du chemin (2). Les accolades dans le chemin (2) assurent que les requêtes d'écriture seront prises en compte au fur et à mesure de leur arrivée. Les accolades dans le chemin (3) autorisent la simultanéité des lectures.

Dans le cas où les lecteurs ont priorité sur les rédacteurs, une spécification possible du problème au moyen des expressions de chemin peut être comme suit:

***path*** tentative_écriture ***end***
***path*** {requête_lecture}, requête_écriture ***end***
***path*** {lire}, (ouvrir_en_écriture; écriture) ***end***

où requête_écriture = ***begin*** ouvrir_en_écriture ***end***
tentative_écriture= ***begin*** tentative_écriture ***end***
tentative_lecture = ***begin*** lire ***end***
read = ***begin*** requête_lecture ***end***
write= ***begin*** tentative_écriture ; écriture ***end***

## 3. Problèmes du Producteur-Consommateur (Communication interprocessus)

Soient deux classes de processus appelés Producteurs et Consommateurs, qui se communiquent de l'information (des messages) à travers une zone de mémoire commune (buffer).

Les messages transmis au consommateur par le producteur (pour simplifier, on va considérer, sauf indication contraire, un couple producteur-consommateur) ont par hypothèse, une taille constante, et le buffer est doté d'une capacité fixe de N messages (N > 0).

Les vitesses d'exécution des deux processus sont quelconques.

La communication entre processus doit respecter les règles suivantes:

- Le consommateur ne peut prélever un message en cours de dépôt par le producteur.
- Le producteur ne peut déposer un message dans le buffer lorsque ce dernier est plein; si tel est le cas, il doit attendre.
- Le consommateur doit prélever tout message une et une seule fois.
- Si le producteur (respectivement le consommateur) est en attente parce que le buffer est plein (respectivement vide) il doit être réveillé dès que cette condition devient fausse.

Les règles qui ont été imposées aux processus de production et de consommation régissent le bon fonctionnement de ce système en particulier:

- Assurer la cohérence et l'intégrité des messages déposés dans le buffer,
- Eviter la perte de messages, par effacement, dans le buffer,
- Prélèvement redondant d'un même message.

**Questions**: Ecrire les algorithmes des processus producteur et consommateur en utilisant:

a) Les sémaphores;
b) Les régions critiques;
c) Les moniteurs;

### 3.1 Solution possible à l'aide des sémaphores

```
Semaphore : nvide, nplein;     % Le sémaphore est employé ici comme compteur %
  nvide  := N;                 % de ressources. %
  nplein := 0;

Procedure producteur
begin
 cycle                         % Le mot clé cycle indique un traitement répétitif %
  begin
   < Produire un message >;
   P(nvide);                   % Existe t-il un emplacement libre dans le buffer ?%
```

```
    deposer(msg);              % procédure de dépôt de messages dans le buffer %
    V(nplein);                 % Signaler qu'un message vient d'être déposé ce  %
  end                          %  réveillera éventuellement le consommateur     %
 endcycle
end

Procedure Consommateur
begin
  cycle
   begin
     P(nplein);              %  Y a t-il  au moins un message déposé ? %
     prelever(msg);          %  si non se bloquer, si oui en prélever un %
     V(nvide);               %  et Signaler une place libre dans buffer  %
     < consommer le message >;
   end
  endcycle
end
```

Les procédures de dépôt et de prélèvement de messages peuvent être les suivantes:

```
type msg = {structure des messages}  % laissé au choix de l'usager %
ptr : [0..N-1];
var tptr : array[ptr] of msg;
mutex: semaphore init 1;
head, tail : ptr init 0;

procedure deposer (m:msg)
begin
  P(mutex);
  tptr[tail]:= m;
  V(mutex);
  tail := tail + 1 mod(N);
end;
procedure prelever (m:msg)
begin
  P(mutex);
  m := tptr[head];
  V(mutex);
  head := head + 1 mod(N);
end;
```

### 3.2 Solution à l'aide des Régions critiques

```
Var nplein :  shared integer init 0;
Procedure  Producteur
begin
  cycle
    begin
      region nplein  when nplein < N do
        begin
          nplein:=nplein + 1;
          < deposer message >;
        end
    end
  endcycle
end.
Procedure  consommateur
begin
  cycle
    begin
      region nplein  when nplein > 0 do
            begin
              nplein:= nplein - 1;
              < Prélever message >;
            end
    end
  endcycle
end.
```

### 3.3 Solution à l'aide des Moniteurs

#### 3.3.1 Moniteur classique

```
 buffer.Monitor
 begin
 Var buffer: array[0..N-1] of msg;
 Var nbrmsg: integer;  % nombre de messages dans buffer %
       nonplein, nonvide: condition;

Procedure  Producteur(m:msg);
  begin
    if nbrmsg = N then wait(nonplein);   %  Attendre qu'il y ait un emplacement libre %
```

```
   endif
   nbrmsg := nbrmsg + 1;
   deposer(m);
   signal(nonvide);                    % réveil éventuel du consommateur %
  end

Procedure consommateur(m:msg);
  begin
    if nbrmsg = 0 then wait(nonvide);     % Attendre le dépôt d'un message %
    endif
    nbrmsg := nbrmsg - 1;
    prélever(m);
    signal(nonplein);                     % Réveil éventuel du producteur   %
  end
% Initialisation %
   nbrmsg := 0;
endmonitor.
```

### 3.3.2 Variante de Kessels

```
buffer.Monitor
begin
   Var  buffer: array[0..N-1] of message;
   Var  nbrmsg: integer;
    Condition  plein: nbrmsg ≥ N;   % condition explicite d'attente  de dépôt %
    Condition  vide : nbrmsg ≤ 0;    %  ''          ''       ''  de prélèvement    %

Procedure Producteur(m:message);
  begin
    wait(plein) ;
    nbrmsg := nbrmsg + 1;      % Incrémenter le nombre de msg avant le dépôt %
    deposer(m);
  end

Procedure consommateur(m:message);
  begin
    wait(vide) ;
    nbrmsg := nbrmsg – 1;     % Décrémenter le nombre de msg avant le prélèvement %
    prélever(m) ;
  end
```

```
  % Initialisation %
    begin
     nbrmsg := 0;
    end
endmonitor.
```

## 4. Problème de l'allocateur de ressources

Soit un allocateur gérant un pool de N ressources identiques et constitué des deux procédures Allouer(n) et Liberer(n) dont les programmes ci-après sont sensés réaliser les opérations désirées.

```
mutex   : semaphore  init 1;
attente : semaphore  init 0;
nlibre  : integer    init N;
natt    : integer    init 0;
```

```
Procedure Allouer(n);
 begin
  Demande: P(mutex);
   if n > nlibre then
     begin
       natt:=natt + 1;
       P(attente);
       aller_a  Demande;
     end
   else
     nlibre:=nlibre - n;
   endif
   V(mutex);
 end
```

```
Procedure liberer(n);
begin
     P(mutex);
     nlibre:=nlibre + n;
  while natt > 0 do
   begin
     natt := natt - 1;
      V(attente)
   end
  endwhile
 V(mutex);
end
```

***<u>Question</u> 1***

Les programmes précédents sont incorrects. Dites pourquoi, et donner les modifications nécessaires pour les rendre corrects.

***<u>Question</u> 2***

Le principe de la solution présentée revient à réveiller tous les processus en attente lors de chaque libération de ressource.

Réécrire les procédures Allouer et Liberer pour ne réveiller que les processus qui seront effectivement servis.

Pour cela, on introduira un sémaphore privé *si* par processus Pi et une file d'attente f dont les éléments contiennent les couples (i,n) ou i est un numéro de processus demandeur n le nombre de ressources demandées. On définira en particulier les procédures de gestion de file que l'on jugera utiles. Donnez en une spécification complète et non l'algorithme détaillé.

### 4.1 Solution base Sémaphores

**Réponse 1**

**- Première Solution**

Parmi les procédures ci-dessus, la procedure allouer(n) est incorrecte.

En effet, l'opération P(attente) étant réalisée dans une section critique peut provoquer un blocage infini des processus appelant les procédures allouer et liberer. Il est donc indispensable que l'opération de blocage *P(attente)* se fasse en dehors de la section critique.

**- Première tentative**

```
Var abloque boolean  init false;
Procedure allouer(n);
 begin
 demander: P(mutex);
     if n > nlibre
        then begin
              natt:=natt + 1;
              abloque:=vrai;
             end
     else begin
           nlibre:=nlibre - n;
           abloque:=false;
          end
     endif
     V(mutex)
     if abloque  then
           begin
             P(attente);
             aller_a Demander;
           end
     endif
end
```

La solution précédente est incorrecte, elle peut conduire à une incohérence car le test " ***if*** abloque ***then*** ..." est hors de la section critique, donc non protégé. Elle serait correcte à condition de rendre la variable abloque privée (locale) pour chaque processus.

**- Deuxième solution**

```
Procedure Allouer(n);
 begin
 demander: P(mutex);
        if n > nlibre then
             begin
```

```
            natt:=natt + 1;
            V(mutex);
            P(attente);
            aller_a Demander;
          end
        else
          nlibre:=nlibre-n;
        endif
        V(mutex);
end
```

**Réponse 2**

Pour éviter de réveiller inutilement des processus sans pouvoir les servir (cf. question1), il est préférable pour raison d'efficacité (sauf si le nombre de processus demandeurs est restreint) de ne réveiller que ceux dont la demande sera satisfaite; c'est pourquoi, un sémaphore privé par processus est nécessaire (i.e. un tableau de sémaphores privés pour l'ensemble des processus).

***Structure de données nécessaires***

On a besoin: - Une file d'attente f de processus bloqués dont chaque élément représente le couple (p,n) avec p : identité du processus et n : le nombre de ressources demandées.

- Procedure *chaîner*(p,n): range dans la file **f** (par ordre croissant de n) le couple (p,n);
- Fonction *premier*: integer, fournit la valeur de la demande du premier processus de la file, si celle-ci est vide rend la valeur max > N.
- Procedure *extraire*(p,n): extrait le premier (au sens de la fonction premier précédente c'est-à-dire le couple (p,n) pour lequel n < nlibre) couple de la file f et restitue l'identité du processus dans p, et sa demande dans n.

```
Procedure Allouer(p,n)      % p:identité de l'appelant %
 begin                      % n: sa demande %
  P(mutex);
  if n > nlibre then chaîner(p,n);
  else
    begin
      nlibre := nlibre - n;
      V(sempriv[p]);        % signal anticipé pour éviter un blocage ultérieur %
    end
  endif
  V(mutex);
  P(sempriv[p];             % blocage seulement des processus dont la demande  %
 end                        % ne peut être satisfaite par manque de ressources  %
Procedure Liberer(n)
```

```
begin
  var q: identité_processus,
      m:  integer;
  begin
    P(mutex);
    nlibre := nlibre + n;
    while premier ≤ nlibre do     % libération de tous les processus pour  %
     begin                        % lesquels on est sûr qu'ils soient servis  %
       extraire(q,m);
       nlibre := nlibre - m;
       V(sempriv[q]);
     end
    endwhile
    V(mutex);
  end
end
```

***Remarque:*** Qu'adviendra t-il si plusieurs processus font appels aux procédures allouer et libérer de manière simultanée ? autrement dit, quelle procédure doit-on exécuter en priorité ?

Sachant qu'un système est d'autant plus performant qu'il favorise le parallélisme ou degré de multiprogrammation, il serait donc judicieux de privilégier les appels à la procédure libérer plutôt qu'à allouer.

En effet, une priorité d'exécution accordée à libérer (vis-à-vis de allouer) permettrait, grâce la remise de ressources libérées dans la réserve nlibre, de libérer le plutôt possible les processus éventuellement en attente. Ainsi, le nombre de processus en exécution dans le système augmente et les délais de réponse des processus bloqués diminueraient (plus vite ils seront réveillés, plus vite ils accompliront leurs services).

En fait, cette situation peut se rencontrer dans certains services bancaires archaïques où on privilégie les dépôts d'argent aux retraits.

Les modules de contrôle permettent d'introduire plus facilement les priorités entre procédures qu'il contrôle (voir l'exercice plus loin).

### 4.2 Solution basée régions critiques conditionnelles

Reprendre l'exercice en utilisant les régions critiques conditionnelles.

```
Var nlibre : shared integer init nbres;        % nbres= nombre de ressources à gérer %

Procedure  Allouer(n:integer);
begin
  region nlibre when n ≤ nlibre do
    nlibre := nlibre - n;
end

Procedure Liberer(n: integer);
begin
  region nlibre do nlibre := nlibre + n;
end
```

***Cas où les processus sont activés efficacement***.
Un processus n'est activé que si le nombre de ressources disponibles permet de satisfaire sa demande.

```
Var nlibre : shared integer;
Var bloquer: array[identité_processus] of boolean init false;
{ déclaration des procédures chaîner, extraire et premier }

Procedure Allouer(n:integer, p:identité_processus)
begin
   region nlibre do
   if n > nlibre
        then begin
               chaîner(p,n);
               region bloquer do
               bloquer[p] := true;   % un processus dont la demande ne peut être satisfaite%
             end                     % sera marqué à bloquer %        (1)
   else nlibre := nlibre - n
  region bloquer when not bloquer[p] do                              (2)
               bloquer[p]:= false;
end
```

(1): Tout processus dont la demande ne peut être satisfaite doit être bloqué. Mais ce blocage va se faire plutard en (2). Pour cela, on l'indique dans son entrée bloquer[p]:=true.
(2): Un processus ayant son entrée à *vrai* dans le tableau bloquer ne passera pas la condition not bloquer[p], il sera effectivement mis en attente. Par contre, un processus qui a été servi, à son entrée dans le tableau bloquer à *faux*. La condition *not bloquer*[q] étant vraie, il ne sera pas bloqué; il continuera en séquence en exécutant simplement une action inutile bloquer[p]:= ***false***.

```
Procedure liberer(n:integer)
 var q:identité_processus;
    r:integer;
begin
  region nlibre do
            begin
              nlibre:=nlibre + n;
              while premier ≤ nlibre do
                begin
                  extraire(q,r);
                  nlibre := nlibre - r;
                  region bloquer do bloquer[q] := false
                end
              endwhile
            end
end
```

### 4.3 Solution basée Moniteurs

Solution dont l'efficacité est inversement proportionnelle au nombre de processus bloqués.

#### 4.3.1 Moniteur classique (Hoare)

```
Allocateur.Monitor;
  Var nlibre : integer ;
  Var bloque : condition ;

 Procedure Allouer(n: integer);
   begin
     while n > nlibre do
       begin
         wait(bloque);            (*)
         signal(bloque);     % réveil de tous les processus %
```

```
      end                    % bloqués à  chaque libération %
    endwhile
    nlibre:=nlibre - n;
  end

 procedure Liberer(n: integer);
  begin
    nlibre:=nlibre + n;
    signal(bloque);
  end

  % Initialisation %
  begin
     nlibre:=nmax;
  end

endmonitor.
```

(*): Une primitive Wait suivie d'une primitive signal

### 4.3.2 Variante de Kessels

```
Allocateur.Monitor;
begin
  var nlibre : integer;
  var bloque : condition n > nlibre;        % Condition explicite de blocage %
Procedure allouer(n:integer)
  begin
    wait(bloque);
    nlibre := nlibre - n;
  end
Procedure liberer(n:integer);
  begin
    nlibre := nlibre + n;
  end
% initialisation %
  begin
    nlibre := nmax
  end
endmonitor.
```

***- Une autre solution plus efficace***: % réveil individuel %

```
Allocateur.monitor                    % moniteur classique %
 Var nlibre : integer;
 Var bloquer : array[identité_processus] of condition
 < déclaration des procédures chaîner, extraire, premier >

Procedure Allouer(n:integer, p:identité_processus);
 begin
  if n > nlibre then
        begin
          chaîner(p,n);
          wait(bloquer[p]);
        end
  else
      nlibre := nlibre - n;
 end

Procedure Liberer(n:integer)
 Var q : identité_processus;
 Var y : integer;
 begin
  nlibre := nlibre + n;
  while premier ≤ nlibre do
          begin
            extraire(q,y);
            nlibre := nlibre - y
            signal(bloquer[q]);
          end
 end
    % Initialisation %
      begin
        nlibre := nmax;
      end
endmonitor.
```

Compte tenu de l'absence de la primitive ***signal*** qui permet de réveiller un processus particulier via la condition associée, la solution efficace (réveil nominatif) présentée précédemment dans le moniteur de Hoare, ne peut être réalisée à l'aide de la variante de Kessels (du moins facilement sans avoir recours à un algorithme complexe de réévaluation des conditions de réveil).

### 4.4 Solution basée Module de contrôle

```
Allocateur.module_de_controle
Type structure =
  begin
    procedure:  bloquer; {corps vide};
    procedure:  lancer ; {corps vide};
    queue    :  bloquer/lancer;
    condition (bloquer): act(bloquer)+act(lancer)=0
                    and aut(lancer)-aut(bloquer)>0;
    condition (lancer) : act(bloquer)+act(lancer)=0;
  end
 < déclaration des procédures: premier, chaîner, extraire >;
condition(demander): act(demander) + act(liberer) + att(liberer) = 0;
condition(liberer) : act(demander) + act(liberer)=0;

var resdisp : integer  init N;                % ressources disponibles %
strucpriv   : array[nomprocessus] of structure;

procedure demander(n:integer, p:nomprocessus)
begin
  if n > resdisp then chaîner(p,n)
  else begin
        resdisp := resdisp - n
        strucpriv[p].lancer
      end
  endif
end

external procedure allouer(n:integer, p:nomprocessus)
  begin
    demander(n,p);
    strucpriv[p].bloquer;
  end
```

```
external procedure liberer(n)
 begin
   var q : nomprocessus;
   var y : integer;
   begin
    resdisp := resdisp + n;
    while premier ≤ resdisp do
        begin
          extraire (q,y);
          resdisp := resdisp - y;
          strucpriv[q].lancer
        end
    endwhile
   end
 end.
```

### 4.5 Solution basée Expressions de chemin

```
Allocateur.expression_de_chemin
type structure =
           begin
            procedure bloquer ; {corps vide};
            procedure lancer  ; {corps vide};
            path lancer;bloquer end;
           end
var resdisp : integer init N;
strucpriv   : array [nomprocessus] of structure;
procedure demander(n:integer, p:nomprocessus)
begin
  if n > resdisp then chaîner(p,n)
  else begin
       resdisp := resdisp - n
       strucpriv[p].lancer
      end
  endif
end
```

```
external procedure allouer(n:integer,p:nomprocessus)
  begin
    demander(n,p);
    strucpriv[p].bloquer;
  end.
external procedure liberer(n)
  begin
    var q : nomprocessus;
    var y : integer;
    begin
      resdisp := resdisp + n;
      while premier ≤ resdisp do
           begin
             extraire (q,y);
             resdisp := resdisp - y;
             strucpriv[q].lancer
           end
      endwhile
    end
  end
path demander , liberer end
```

## 5. Problème de rendez-vous

On appelle point de rendez-vous (un point de synchronisation) un point dans le programme d'un processus Pi, i ∈ [1..N], que ce dernier ne peut franchir avant que son (ou ses) partenaire(s) Pj, j ≠ i, j ∈ [1..N] aient atteint les leurs.

Cette notion a été introduite par Hoare dans le langage CSP [Hoa78]. Le mécanisme de rendez-vous revêt un caractère important dans la mesure où il convient aisément à la fois dans un environnement centralisé et réparti.

En effet, l'absence de partage de mémoire commune en réparti, impose une communication interprocessus à travers messages; ce qui permet, grâce au rdv, une évolution d'exécution cadencée au rythme d'émission de messages et attente de réponse (synchronisation entre l'émission et la réception de messages). Cette synchronisation est difficilement réalisable à l'aide d'outils tels que les sémaphores ou les moniteurs où les réveils sont assujettis aux signaux externes.

*Remarque*: Les langages ADA et CSP utilisent la notion de rendez-vous dans la communication interprocessus.

**Question**: Ecrire un algorithme réalisant la coordination de N processus sous forme d'un rendez-vous, en utilisant:

1. Sémaphores,
2. régions critiques,
3. Moniteurs,
4. Modules de contrôle,
5. Expressions de chemins.

### 5.1 Solution à l'aide des Sémaphores

```
  var nbproc : integer init 0;          % indique le nombre de processus      %
      mutex  : semaphore init 1;        % sémaphore de mutuelle exclusion     %
      attente: semaphore init 0;        %   sémaphore privé                   %
      N      : integer  init nmax;      % N: nombre de processus              %
 Procedure rdv                          % participant au rendez-vous          %
 begin
 begin
   P(mutex);                            % entrée en section critique ?        %
   nbproc := nbproc + 1;                % comptabiliser l'arrivée d'un processus. %
   if nbproc = N then begin             % le dernier arrivé doit débloquer les %
            while N = 1 do begin        % les processus précédents en attente. %
                        N := N - 1;
                        V(attente);
```

```
                    end
              endwhile
                 end
   else begin
         V(mutex);
         P(attente);
       end
   endif
end.
```

*Remarque*: Tout processus Pi, i∈[1..N], impliqué dans cette coordination doit exécuter un appel à la procédure RDV au moment où il atteint son point de synchronisation. Cet appel va donc conditionner son évolution. S'il est le dernier à atteindre son point de rendez-vous, il débloque tous ses partenaires et poursuit son exécution, sinon il sera mis en attente. Dans la procédure RDV est implicitement supposé connue l'identité de l'appelant; dans le cas contraire, il faut procéder comme suit: Changer le sémaphore privé *attente* en un tableau: attente: ***array***[identité_processus] ***of*** semaphore ***init*** 0; introduire deux procédures de manipulation de file d'attente comme dans l'allocateur et adapter cette philosophie dans RDV notamment dans le blocage et le réveil individuel.

### 5.2 Solution à l'aide des régions critiques

```
Procedure RDV
var nbproc : integer shared init N;
begin
  region nbproc do nbproc - 1 await nbproc = 0;
end
```

A noter la facilité avec laquelle est écrite la procédure RDV, et cela est dû au mécanisme d'exécution des région critiques. Le dernier arrivé, vérifiant la condition nbproc = 0, quitte la région critique (poursuit son évolution) provoquant ainsi la réévaluation de la condition de blocage. La véracité de cette condition déclenche le réveil de tous les processus en attente.

### 5.3 Solution à l'aide des Moniteurs

```
RDV.Monitor
begin
  var N       : integer;
     okarrivé: condition;
procedure arrivé
```

```
 begin
   N := N - 1;
   if N > 0 then wait(okarrivé)  % ne sont pas tous arrivés %
   endif
   signal(okarrivé);      (1)        %  réveil en chaîne %
 end

% initialisation %
 begin
   N := max;           % max: constant: nombre de processus %
 end
end RdV.
```

(1): Le dernier processus arrivé exécute ***signal****(okarrive)* et quitte le moniteur; ce qui a pour effet de réveiller un processus en attente qui, à son tour, agit comme le précédent. Ainsi, s'effectue le déblocage un à un de tous les processus en attente.

Le programme d'un processus Pi s'écrit:

```
"processus Pi"
  begin
    <avant rendez-vousi>;
      RDV.arrive;              % blocage éventuel %
    <après rendez-vousi>;
  end
```

### 5.4 Solution à l'aide de Module de contrôle

Les problèmes de coopération et d'asservissement tel la coordination par rendez-vous se prête moins bien à une expression intuitive des règles de synchronisation. Comme cela a été déjà le cas dans l'allocateur de ressources, il faut procéder par un changement du niveau d'observation de sorte à l'interpréter en termes d'accès à un objet partagé. Cela revient donc à spécifier la mise en oeuvre du rendez-vous plutôt que le rendez-vous lui-même. Plusieurs façons de faire sont possibles, voir par exemple celle donnée dans l'allocateur. On peut adopter une qui découle immédiatement de la définition suivante du rendez-vous. Celle-ci étant basée sur la trace temporelle au niveau de l'exécution d'un processus.

On dira que deux processus Pi et Pj, $i,j \in [1..N]$, sont en rendez-vous si la contrainte se synchronisation (temporelle) s'exprime par: fin(avant rendez-vousj) < début(après rendez-vousi); fin et début indique respectivement la fin et le début d'exécution des séquences d'instruction situées avant et après le point de rendez-vous.

Donc, programmer le rendez_vous de deux ou plusieurs processus coopérants, revient à transformer ce problème de coordination, peu favorable (réfractaire) aux modules de contrôle, en un problème de contrôle d'accès à un objet partagé mieux adapté. Pour spécifier les règles de synchronisation, on associe à chaque processus Pi, i∈[1..N] une procédure Pdi de mise en oeuvre du rendez-vous souhaité. Autrement dit, les procédures artificielles Pdi de mise en oeuvre joueront le rôle d'interface entre les processus Pi et le synchroniseur mdc. L'action de ce dernier sur les procédures Pdi doit réaliser, ou être conforme, à la définition du rdv, donnée précédemment.

Pour illustrer cette expression du rdv, considérons trois processus P1, P2, P3 auxquels sont associées trois procédures Pd1, Pd2, Pd3 à travers lesquelles est accomplie la coordination des Pi. Pour respecter la définition du rdv, décomposons la procédure Pd1 en deux procédures intermédiaires Pd11 et Pd12 puis Pd2 en Pd21 et Pd22. La réalisation du rdv implique l'ordonnancement des exécutions suivant:

Pd11 -> Pd21 -> Pd3 -> Pd22 -> Pd12.

Ainsi, pour tout couple de procédures, le début d'exécution de l'une précède la fin de l'exécution de l'autre. Les procédures décomposées peuvent être déclarées comme suit:

```
type Pd1 = begin
            procedure : Pd11 = null; % null = corps vide %
            procedure : Pd12 = null;
           end

type Pd2 = begin
            procedure : Pd21 = null;
            procedure : Pd22 = null;
           end
```

Rendez_vous.***mdc***

```
begin
 procedure Pd11;
 procedure Pd12;
 procedure Pd21;
 procedure Pd22;
 procedure Pd3;
 queue  Pd11, Pd12, Pd21, Pd22, Pd3;
```

Pour un rendez-vous des deux processus P1 et P2, les conditions d'exécution sont les suivantes.

```
 condition(Pd11): aut(Pd11) < 1 + term(Pd12);
 condition(Pd2) : aut(Pd2)  < term(Pd11);
 condition(Pd12): aut(Pd12) < term(Pd2);
```

Pour un rdv de trois processus P1, P2, P3, on peut écrire les conditions suivantes:
***condition***(Pd11): aut(Pd11) < 1 + term(Pd12);
***condition***(Pd21): aut(Pd21) < term(Pd11);
***condition***(Pd3) : aut(Pd3) < term(Pd21);
***condition***(Pd22): aut(Pd22) < term(Pd3) ;
***condition***(Pd12): aut(Pd12) < term(Pd22);
***end*** Rendez_vous.

### 5.5 Solution à l'aide des Expressions de chemin

Pareillement aux modules de contrôle, les règles de synchronisation qui réalisent le rendez-vous s'appliquent à la traduction de celui-ci en termes d'objet partagé (à sa mise en oeuvre). Les procédures fournies aux processus pour se synchroniser (Pd1, Pd2) se décomposent en de nouvelles procédures Pd11 et Pd12 qui respectent la trace temporelle voulue:
***procedure*** Pd1 = ***begin*** Pd11; Pd12 ***end***.

Le séquencement (point fort du mécanisme des expressions de chemin) de mise en oeuvre s'écrit, pour un rdv des processus P1 et P2:
***path*** Pd11 ; Pd2 ; Pd12 ***end***

Pour un rdv des trois processus P1, P2, P3 on peut écrire:
***path*** Pd11 ; Pd21 ; Pd3 ; Pd22 ; Pd12 ***end***

## 6. Problème de partage de fichier

Soient N processus P1, P2,...Pn concurrents pour l'utilisation exclusive d'un fichier F (ressource non partageable). Si F est libre, il peut être utilisé immédiatement autrement le processus demandeur doit attendre sa libération (par le processus détenteur). Si plusieurs processus sont en attente de F celui-ci sera attribué, dès sa libération, au processus le plus prioritaire. Un processus Pi est plus prioritaire que Pj si i<j.

**Question :** Ecrire l'algorithme du gestionnaire du fichier F en utilisant:

1: Les sémaphores,
2: Les régions critiques,
3: Les moniteurs.
4: Les modules de contrôle,
5: Les expressions de chemin.

L'appel au gestionnaire peut être comme suit:

```
"Processus Pi"
begin
  Demander(i)  i:numero du processus (I.e. son identité)
  < utiliser F >
  Libérer F
end
```

### 6.1 Solution à l'aide des sémaphores

```
  Var Foccupe  : boolean init false;
     attendre : boolean array [1..N] init false;
     mutex    : Semaphore  init 1;
     Sempriv  : array[1..N] of semaphore init 0;
Procedure Allouer(i)
  begin
    P(mutex);
    if  not Foccupe then begin
                    Foccupe := true;
                    V(sempriv[i]);
                  end
    else
      attendre(i) := true;
    endif
    V(mutex);
```

```
    P(sempriv[i]);
  end
Procedure Liberer
 begin
    P(mutex);
    i := 1;
    while i ≤ N and  not attendre(i) do  i := i + 1;
    endwhile
    if attendre(i) then begin
                      attendre(i) := false;
                      V(sempriv[i]);
                    end
    else
       Foccupe := false;
    endif
    V(mutex);
 end
```

### 6.2 Solution à l'aide des régions critiques

```
Var V : shared record
        Foccupe : boolean  init false;
        attendre : array[1..N] of boolean init false;
        autoriser: array[1..N] of boolean init false;
      end
Procedure allouer(i:integer)
 begin
  region V do begin
             if  not Foccupe then Foccupe := true
             else begin
                    attendre[i] := true;
                    await(autoriser[i]);
                  end
           end
 end.

Procedure liberer
 begin
   var i, j: integer;
 region V do begin
           i := j :=1;
```

```
        while i ≤ N and not attendre(i) do
        begin i := i + 1, j := i  end;
        endwhile
        if attendre[j] then begin
                          autoriser[j] := true;
                          attendre[j]  := false;
                        end
        else
          Foccupe := false;
        end
end
```

### 6.3 Solution à l'aide des moniteurs

```
Ressource.Monitor
 begin
   Var Foccupe : boolean;
   cond: array[1..N] of condition
   Var attendre: array[1..N] of boolean
Procedure Allouer(i)
 begin
   if Foccupe then begin
                 attendre[i] := true;
                 wait(cond[i]);
               end
   else
     Foccupe := true
   endif
 end
Procedure Liberer
 begin
   i := 1;
   while i ≤ N and  not attendre[i] do i := i + 1 endwhile;
   if attendre[i] then begin
                    attendre[i] := false;
                    signal(cond[i]);
                  end
    else
     Foccupe := false;
    endif
 end
```

```
  % Initialisation %
   begin
    for i := 1 to N do
        attendre[i] := false
    endfor
    Foccupe := false
   end
endmonitor.
```

### 6.4 Solution à l'aide des modules de contrôle

```
fichier.module_de_controle
Type structure =
  begin
    procedure  bloquer: {corps vide};
    procedure  lancer : {corps vide};
    queue      bloquer/lancer;
    condition (bloquer): act(bloquer)+act(lancer)=0
                    and aut(lancer)-aut(bloquer)>0;
    condition (lancer) : act(bloquer)+act(lancer)=0;
  end
   condition (demander): act(demander) + act(liberer)
                  + att(liberer) = 0;
   condition (liberer) : act(demander)+ act(liberer)=0;
end mdc

strucpriv          : array [nomprocessus) of structure;
priority           : array [1..N] of boolean init false;
Foccupe            : boolean init false;

 procedure demander(i:nomprocessus)
  begin
   if Foccupe then priority[i] := true;
   else begin
         Foccupe       := true;
         strucpriv[i].lancer ;
       end
   endif
  end
```

```
external procedure allouer(i)
 begin
  demander(i);
  strucpriv[i].bloquer;
 end

external procedure liberer
 begin
  var  J, K : integer;
  j := 1;
  while j ≤ N and not priority[j] do
       j := j + 1;
       k := j
  endwhile
  if priority[k] then begin
                      priority[k] := false;
                      strucpriv[k].lancer;
                     end
  else
    Foccupe := false;
  endif
 end
```

## 7. Problème de conversation entre processus

On désigne par conversation une structure linguistique permettant à un ensemble de processus, coopérant à la réalisation d'une tâche commune, de communiquer selon certaines règles. Ces règles ont pour objectif principal de coordonner ces processus de sorte à éviter le phénomène de *contamination* si l'un d'entre eux est soumis à une altération résultant d'une faute. Autrement dit, un processus coopérant, ayant un mauvais fonctionnement suite à une erreur, risque de transmettre des données corrompues (erronées) à ses partenaires. Ces derniers vont élaborer des résultats, à partir d'informations erronées reçues, qu'ils transmettront à d'autres partenaires etc...

Si l'un des processus coopérants venait à détecter l'erreur, il doit défaire tout son travail depuis l'occurrence de l'erreur jusqu'à sa détection. Il informera tous ses partenaires pour agir de même. Tout processus ayant consommé une information au moins du processus défaillant doit lui aussi annuler le travail qu'il a accompli depuis la consommation de l'information erronée jusqu'à l'instruction en cours d'exécution. Pour limiter la quantité de travail à annuler, les processus coopérants d'une application vont devoir s'organiser en conversation lors de leurs communications.

Ainsi, pour communiquer avec ses partenaires, un processus Pi doit d'abord entrer en conversation en exécutant un prologue ENTRER (Pi: nom-processus); après quoi, la communication peut s'établir à volonté.

Toutefois, si l'entrée dans une conversation se fait sans contrainte, la sortie doit se faire obligatoirement en "simultanéité". Autrement dit, c'est le dernier processus de la communauté interactive, ayant terminé son étape de travail, à qui il incombe d'autoriser ses partenaires à quitter la conversation et d'évoluer de manière indépendante. Cette sortie est donc coordonnée selon le mécanisme de rendez-vous (voir Figure 1).

<u>scénario</u>:

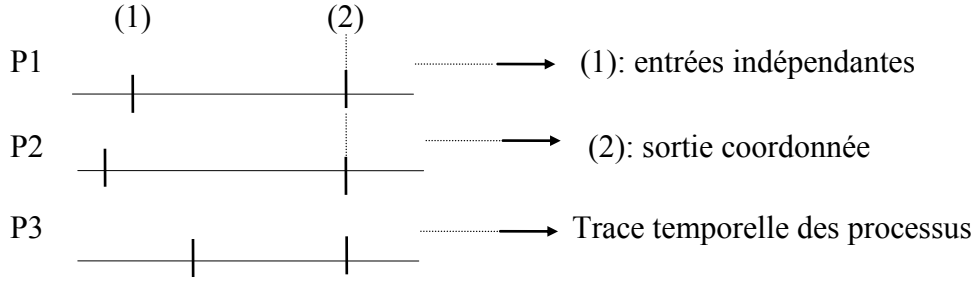



Figure 1. Conversation

<u>Remarque</u>: En réalité, la conversation est utilisée dans un système réparti où la communication interprocessus s'effectue à travers des messages. On suppose dans notre cas présent que la communication utilise des variables globales, i.e. dans un système centralisé.

**Question** : Ecrire les algorithmes des opérations
Entrer (p);
Sortir;
En utilisant:
1. les sémaphores;
2. les moniteurs;
3. les régions critiques.

### 7.1 Solution avec les sémaphores

```
var nbpart : integer init 0;      % nombre de participants %
var sauv   : integer init 0;      % variable de sauvegarde %
attente    : semaphore init 0;
mutex      : semaphore init 1;

Procedure entrer(p:nomprocessus)
 begin
   P(mutex);
   nbpart := nbpart + 1;
   sauv   := nbpart;
   V(mutex);
 end

Procedure sortir
 begin
   P(mutex);
   nbpart := nbpart - 1;
   if nbpart > 0 then begin
                       V(mutex);
                       P(attente);
                      end
   else begin
        while sauv > 0 do begin
                           sauv := sauv - 1;
                           V(attente);
                          end
        endwhile
   endif
   V(mutex);
 end.
```

## 7.2 Solution avec les moniteurs

### 7.2.1 Moniteurs classiques

```
conversation.monitor
begin
  var nbpart :   integer      % nombre de participants %
      attente :  condition    % condition d'attente    %

procedure entrer(p:identite-processus)
  begin
    nbpart := nbpart + 1
  end

procedure sortir
  begin
    nbpart := nbpart - 1;
    if nbpart > 0 then  wait(attente);
    endif
    signal(attente);
  end

 % Initialisation %
   begin
     nbpart := 0;
   end
endmonitor.
```

Remarque: On a supposé que tout processus qui entre dans une conversation doit en sortir au bout d'un temps fini. Autrement dit, il n'y a pas de déperdition (tous les processus fonctionnent correctement). Dans le cas contraire, il faudrait en tenir compte dans la solution: prendre des mesures qui s'imposent lorsque l'échéance de sortie de la conversation arrive à terme.

### 7.2.2 Variante de Kessels

```
Conversation.monitor
  begin
    var nbpart : integer;
    continue : condition  nbpart > 0;    % condition explicite d'attente %
```

```
 procedure entrer (p:nom_processus)
   begin
     nbpart := nbpart + 1;
   end

 procedure sortir
   begin
     nbpart := nbpart - 1
     wait(continue);
   end
   % initialisation %
   begin
     nbpart := 0;          % le nombre de participants n'est pas connu à l'avance, car les %
   end                     % entrées sont libres donc ne sont pas coordonnées.          %
endmonitor.
```

### 7.3 Solution avec régions critiques

```
   var nbpart : integer shared  init 0;
procedure entrer(p)
  begin
  region nbpart do nbpart := nbpart + 1;
end
procedure sortir
  begin
    region nbpart do nbpart := nbpart - 1 await nbpart < 1;
  end
```

## 8. Boutique du barbier (Dj 68)

La boutique du barbier comporte une salle d'attente et un salon comme l'illustre le schéma suivant:

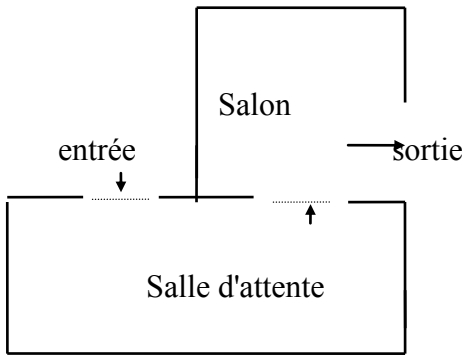



Figure 2. Boutique du barbier

La porte d'entrée et la porte de communication entre la salle d'attente et le salon du barbier peuvent laisser passer un client à la fois. Une porte coulissante (boutique japonaise !) est disposée de telle sorte que l'une de ces issues est toujours fermée (arbitrage à la manière d'un accès à une cellule mémoire). Le barbier peut adopter deux comportements possibles:

**a)** Lorsque le barbier a fini de raser un client (ce dernier s'en va par la porte de sortie), il fait entrer le client suivant. S'il n'y a aucun client dans la salle d'attente, le barbier (vieux ou paresseux) retourne dans le salon et s'endort dans son fauteuil.
Lorsque un client se présente, si la salle d'attente est vide, il réveille le barbier, sinon il attend son tour dans la salle d'attente.

**b)** Lorsque la salle d'attente est vide, le barbier s'y installe et s'endort. Si un client trouve le barbier endormi, il le réveille, sinon il attend son tour.

**Question**: Donnez l'algorithme de fonctionnement de ce système.

**Solution possible**

**a)** Dans ce cas, ce système constitue un modèle type du producteur consommateur (particulièrement si le rythme d'activité du barbier est identique à l'arrivée des clients). Le barbier étant le consommateur des éléments "clients" et la salle d'attente représente un buffer. Le client matérialise lui même le producteur. Il peut s'avérer utile de considérer la taille de la salle d'attente comme contrainte d'accès des clients. La solution est donc semblable à celle donnée précédemment pour ce modèle en respectant éventuellement la contrainte signalée.

**b)** Le client i et le barbier représentent des processus qui s'interréagissent, et l'algorithme de fonctionnement de ce système peut s'écrire comme suit:

### 8.1 Solution basée Sémaphores

```
begin
  var  nbclient : integer    init 0;
       endormi  : boolean    init true;
       mutex    : semaphore  init 1;
       réveil   : semaphore  init 0;
       clientmax: semaphore  init max;
       sempriv  : array[1..max] of semaphore init 0;
       fileatt  : queue init empty;
<   { déclaration des procédures enqueue, et dequeue }          >
%   enqueue (i,fileatt): mettre clienti dans la file d'attente        %
%   fileatt; dequeue (j, fileatt): extrait le premier processus       %
%   de  fileatt et rend son identité dans j. fileatt étant supposée   %
%   gérée en fifo.%


Processus client i
begin
  P(clientmax);                  % contrainte de taille de salle d'attente %
  P(mutex);
  nbclient := nbclient + 1;
  if endormi then begin
                  endormi := false;
                  V(réveil);           % réveiller le barbier %
                  V(sempriv[i]);
                end
  else
    queue(i,fileatt);       (1)
  endif
  V(mutex);
  P(sempriv[i]);
end.
```

(1): Selon la modélisation considérée, tout clienti (sauf celui qui trouve le barbier endormi) passe momentanément dans la file d'attente fileatt avant d'être pris par le barbier.

```
Processus barbier
 begin
 test: P(mutex);
 if nbclient ≤ 0 then begin
                    endormi := true;
                    V(mutex);
                    P(réveil);
                    goto test;      %  Chaque fois que le barbier est réveillé il s'assure qu'il %
                  end               %  qu'il y a bien un client à raser %
 else
   begin
     nbclient := nbclient - 1;
     dequeue(j, fileatt);
     V(sempriv[j]);
     V(mutex);
     V(clientmax);    % autoriser une arrivée éventuelle d'un client %
   end
   < raser >;
   goto test
end.
```

## 9. Problème d'allocation de fichiers

On considère trois fichiers f1, f2, f3 qui seront dans cette partie considérés à accès exclusif. Pour utiliser fi ($1 \leq i \leq 3$), un processus Pi ($1 \leq i \leq nmax$) doit appeler une procédure Ouvrir(fi) ayant l'effet suivant: si le fichier fi est libre, il sera alors affecté au processus demandeur et devient indisponible pour les autres; sinon le processus demandeur est mis en attente. Tout processus peut ouvrir un, deux, ou les trois fichiers pendant son execution; il ferme chacun des fichiers ouverts au bout d'un temps fini.

**Questions:**

a1) Quel type de problème engendrerait le respect des règles précédentes;
- Montrez-le à l'aide d'un exemple.
- Donnez une règle simple pour l'éviter.

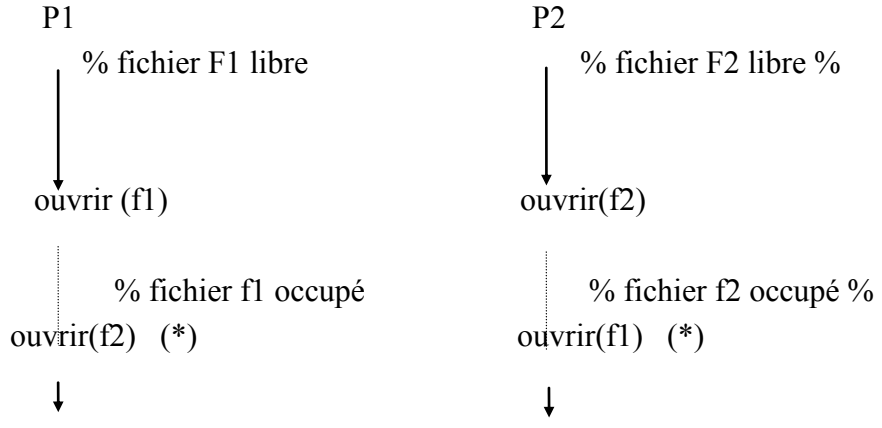
a2) Donnez un algorithme du moniteur réalisant les opérations **ouvrir**(fi) et **fermer**(fi), tout en respectant la règle donnée en a1. Une opération d'ouverture ne respectant pas la règle sera refusée, c'est-à-dire retour au processus appelant en lui délivrant un code d'erreur.

Chacun des trois fichiers peut être utilisé selon le modèle lecteur-rédacteur; les processus disposent alors de deux opérations: Ouvrirlec(fi) et Ouvrirecr(fi). chaque fichier Fi peut être ouvert par n lecteurs ou par un seul rédacteur, aucune priorité n'est établie entre les lectures et les écritures.

b1) Modifiez les réponses aux questions de a1 en fonction des nouvelles hypothèses.
b2) Un problème de famine des rédacteurs peut survenir s'il y a coalition des lecteurs; donnez le principe d'une solution évitant de telles coalitions.

**Solution possible**

a1) Le respect des règles précédentes peut engendrer un phénomène d'interblocage.
En effet, soient deux processus P1 et P2 tels que:



(*): A ce niveau, les deux processus P1 et P2 se bloquent mutuellement: P1 possédant F1 et demande F2 acquis par P2 lequel à son tour est en possession de F2 mais en demande F1.
Il existe bien entendu des algorithmes performants de prévention, et de détection et guérison de telle situation d'interblocage. Cependant dans le cas qui nous préoccupe, il s'agit de trouver une règle simple évitant ce genre de problème. Pour cela, il suffit d'établir un ordre de demande des trois fichiers. Autrement dit, tout processus doit ouvrir les fichiers désirés dans le même ordre; par exemple, un tel ordre croissant serait: (F1, F2, F3) ou bien (F2, F3) ou bien (F1, F3)... Un contre ordre serait: (F2, F1) ou (F3, F1) etc...

**a2)** Fich3.***monitor***

```
 begin
 var   occupe:    array[1..3] of boolean;
 var   condition: array[1..3] of condition;
   type processus record begin
                      ident : process_identifier;
                      F: array[1..3] of boolean; (*)
                      openbr : integer;
                     end
var process: processus;
procedure open(fi, p:processus)
  begin
   case p.openbr of                % openbr: nombre de fichiers ouverts %
   0: if ¬ occupe[i] then          % ¬ = not  %
              begin
                occupe[i] := true;
                p.F[i]:= true;
                p.openbr:= p.openbr + 1;
              end
      else   begin
                wait(condition[i]);
                p.openbr := p.openbr + 1;
             end
      endif
   1: case i of
       1: return (" Erreur: désordre ")
       2: if ¬ F(1) then return (" Erreur: désordre ")
         else if ¬ occupe[i] then
                    begin
                       occupe[i]:= true;
                       p.F[i] := true;
                       p.openbr := p.openbr + 1;
```

```
                end
            else begin
                  wait(condition[i]);
                  p.openbr:= p.openbr + 1;
                end
            endif
          endif

      3:  if ¬ occupe[i]
          then begin
                  occupe[i] := true;
                  p.F[i] := true;
                  p.openbr:= p.openbr + 1;
                end
          else begin
                  wait(condition[i]);
                  p.openbr := p.openbr +1;
                end
          endif
   2: case  i of
      3:  if ¬ occupe[i]
          then begin
                  occupe[i] := true;
                  p.F(i) := p.F(i) + 1;
                  p.openbr := p.openbr + 1;
                end
          else begin
                  wait(condition[i]);
                  p.openbr:=p.openbr + 1;
                end
      1:2:   return " Erreur: désordre "
  end


procedure fermer(fi,p:processus)
  begin
    p.openbr := p.openbr - 1;
    if ¬ empty.condition[i] then signal(condition[i])
    else occupe[i] := false;
    endif
  end
```

```
  % Initialisation %
     begin
       k := 1;
       while k ≤ 3 do occupe[k] := false;
     end
endmonitor.
```

(*): F[i] booléen indiquant l'état du fichier fi, (1≤i≤3), libre (F[i]=0) ou occupé (F[i]=1).

## 10. Problème des philosophes (E.W. Dijkstra)

Cinq moines boudhistes assis autour d'une table prennent part à un curieux rituel dont l'activité consiste (pour raison académique) à alterner entre méditer (penser) et manger. Leur existence se résume selon le schéma suivant.

***cycle***
méditer;
manger;
***endcycle***

Devant chaque moine se trouve un bol de riz (supposé toujours non vide). Quand un moine est tenaillé par la faim, il décide de manger et saisit les deux baguettes disposées de part et d'autre de son bol, voir figure ci-après (il n'y a que 5 baguettes sur la table). Un moine ne peut se sustenter que lorsqu'aucun de ses voisins n'est entrain de manger.
*Remarque*: Ce problème n'est en fait qu'un cas particulier du problème de l'allocateur de ressource précédent.

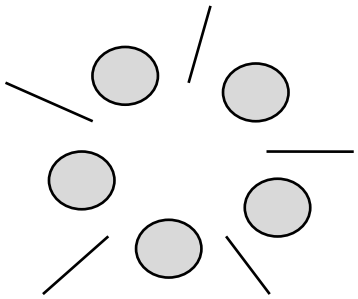


Figure 3: rituel des moines

**Question** :
Ecrire l'algorithme modélisant l'activité d'un moinei ($1 \leq i \leq 5$) en utilisant:
1. Les sémaphores;
2. les régions critiques conditionnelles;
3. les moniteurs.

### 10.1 Solution à l'aide des Sémaphores

On associe à chaque moine*i* trois états:
etat 1: quand le moinei médite;
etat 2: quand le moinei voudrait manger mais se trouve bloqué par manque de baguettes.
etat 3: lorsque le moinei mange.

d'où la structure de données suivante:
***var*** moine : ***shared array*** [1..5] ***of integer init*** 1 ;
mutex : ***semaphore init*** 1;
sempriv: ***array*** [1..5] ***of semaphore init*** 0;

Condition pour qu'un moinei à l'état "méditer" puisse manger:
(moine[i+1] ≠ 3) et (moine[i-1] ≠ 3), si cette condition n'est pas satisfaite le moinei passe à l'état moine[i] = 2 et se bloque.

```
Demande baguette(i:integer)    % i:identité d'un moine appelant %
begin
  P(mutex);
  if (moine[i+1]mod(5) ≠ 3) and (moine[i-1]mod(5) ≠ 3)
    then begin
           moine[i] := 3;
           V(sempriv[i]);
         end
  else  moine[i] := 2;
  V(mutex);
  P(sempriv[i]);
end.
```

Le passage de l'état moine[i] = 3 à moine[i]= 1 provoque le réveil des moines i+1 et i-1 si les conditions suivantes sont réalisées:
- Ces moines avaient décidé de manger, i.e. moine[k] = 2
  pour k=i+1 et k=i-1;

- On est sûr qu'ils disposeront de l'autre baguette, i.e.
  moine[k] ≠ 3 pour k=i+2 et k=i-2;

```
Liberer baguette(i)
  begin
    P(mutex);
    moine[i]:=1;
    if (moine[i+1]mod(5) = 2) et (moine[i+2]mod(5) ≠ 3)
      then begin
             moine[i+1]:=3;
             V(sempriv[i+1]);
           end
    if (moine[i-1]mod(5) = 2) et (moine[i-2]mod(5) ≠ 3)
      then begin
             moine[i-1]:=3;
             V(sempriv[i-1]);
           end
    V(mutex);
  end.
```

S'il y a coalition de certains moines au détriment d'un autre, en décidant de respecter indéfiniment l'ordre suivant: (1,4), (1,3), (5,3), (1,3), (1,4) etc..., le moine2 est condamné par ses disciples à jeûner indéfiniment.
Autre manière de modéliser ce système, voir (1) en fin de problème.

### 10.2 Solution basée sur les régions critiques

#### 10.2.1 Première tentative

```
Procedure moine(i)
begin
  var baguette : shared array[ 1 .. 5 ] of boolean init true;
  cycle
     méditer;
     region baguette(i) do
     region baguette(i+1)mod(5) do  manger;
  endcycle
end.
```

En examinant cet algorithme, on s'aperçoit qu'un interblocage est possible. Il est généré par la situation où tous les moines décident de saisir à l'unisson leur baguette de gauche et attendent que celle de droite soit libre.

#### 10.2.2 Deuxième tentative

```
procedure moine(i)
 begin
  var baguette : shared array[ 1..5] of boolean init true;
    cycle
      méditer;
      region baguette do
          begin
           await baguette((i-1)mod(5)) and
                baguette((i+1)mod(5));
           baguette(i) := false;
          end
      < manger >;
      region baguette do baguette(i) := true
    endcycle
 end.
```

Selon l'algorithme précédent, il serait possible que deux moines non adjacents ([ moine(i) et moine (i+2)] mod(5)) s'accordent de sorte qu'à tout instant un d'entre eux se trouve en état de manger alors le moine(i+1)mod(5) est condamné à jeûner jusqu'au jour du jugement dernier !

### 10.2.3 Troisième tentative

```
var baguette : shared array [1..5] of 0..2

% baguette indique le nombre de baguettes libres
% de part et d'autre de chaque bol

procedure moine(i)
 begin
  var gauche, droite :1..5;
  begin
   gauche := (i-1)mod(5);
   droite := (i+1)mod(5);
   cycle
     < méditer >;
     region baguette  when baguette[i] = 2 do
          begin
            baguette(gauche) := baguette(gauche) -1;
            baguette(droite) := baguette(droite) -1;
          end
      < manger >;
      region baguette do
           begin
             baguette(gauche) := baguette(gauche) +1;
             baguette(droite) := baguette(droite) -1;
           end
   endcycle
 end.
```

Dans la solution précédente, l'interblocage subsiste encore. Il suffit qu'un moine affamé se saisisse des deux baguettes (droite et gauche) simultanément. Pour l'éviter, il faut contraindre chaque participant à prendre les baguettes une à une.

**(1):** Une autre façon de modéliser le problème consiste à utiliser des sémaphores compteurs de ressources:

```
var       i : integer;
 baguette : array[1..5] of sémaphore init 1;
```

```
 quatre    : semaphore init 4;  % servira à éliminer l'interblocage %

procedure moine(i)
begin
  cycle
    < méditer >;
    P(baguette[i])              % tenter de prendre la baguette de gauche %
    P(baguette[i+1]mod(5))   % tenter de prendre la baguette de droite  %
    < manger >;
    V(baguette[i])              % libérer la baguette de gauche %
    V(baguette[i+1]mod(5))    % libérer la baguette de droite %
  endcycle
end.
```

Il est évident de constater qu'une telle solution simple peut conduire inévitablement à un interblocage lorsque tous les moines décident simultanément de manger. Chacun d'eux se saisi d'une baguette (gauche par exemple) sans se préoccuper de l'état de l'autre baguette, et se trouve bloquer par son voisin qui s'est emparé de cette dernière (droite). Pour éliminer cet situation, il suffit a) de supposer que les capacités stomacales des moines sont finies, i.e tout moine entrain de manger s'arrêtera au bout d'un temps fini. b) Disposer d'un nombre de baguettes > nombre de moines autour de la table. C'est l'oeuvre du sémaphore *quatre* qui régule le nombre de moines compétitifs à 4 ce qui revient à insérer ***P***(quatre) juste après <méditer>, et ***V***(quatre) juste après V(baguette[i+1] mod(5)).

### 10.3 Solution basée sur les moniteurs

```
Dîner.monitor
 begin
 var  baguette: array[0..4] of integer init 2;   % baguette[i] représente
    % le nombre de baguettes disponibles  de part et d'autre du moinei; %
 var gauche, droite: integer;
 var  baglib: array[0..4] of condition;

 procedure demand_baguette(i)
 begin
   gauche := (i-1)mod(5);
   droite := (i+1)mod(5);
   if baguette[i] ≠ 2 then wait(baglib[i]);
   baguette[gauche] := baguette[gauche] - 1;
   baguette[droite] := baguette[droite] - 1;
 end
```

```
procedure liberer_baguette(i)
begin
  gauche := (i-1)mod(5);
  droite := (i+1)mod(5);
  baguette[gauche] := baguette[gauche] + 1;
  baguette[droite] := baguette[droite] + 1;
  if baguette[gauche] = 2 then signal(baglib[gauche]);
  if baguette[droite] = 2 then signal(baglib[droite]);
end
% initialisation %
 begin
   for j = 0 to 4  do  baguette[j] := 2;
    gauche := 4;          % les valeurs initiales de gauche%
    droite := 1;          % et droite sont arbitraires %
 end
endmonitor.
```

**11. Problème des baigneurs** (Latteux 80)

Soit une piscine pouvant accueillir un nombre limité de baigneurs. Cette limite est représentée par un nombre P de paniers disponibles pour ranger les vêtements. De plus, à l'entrée comme à la sortie, les baigneurs rentrent en compétition pour l'acquisition d'une des C cabines d'habillage et de déshabillage (on suppose que P est grand devant C).

***Question***: Programmer le gestionnaire de la piscine à l'aide des outils de synchronisation suivants:

1. Les sémaphores;
2. Les régions critiques conditionnelles;
3. Les moniteurs: a) Classiques (Hoare)
   b) Variante de Kessels;
4. Les modules de contrôles;
5. Les expressions de chemins.

**11.1 Solution avec les Sémaphores**

```
var nbpanocc   : integer;      % nombre de paniers occupés   %
    nbcabocc   : integer;      % nombre de cabines occupées %
    nbpanlibre : integer;      % nombre de paniers  libres   %
    nbcablibre : integer;      % nombre de cabines  libres   %
    mutex      : semaphore;    % sem. de mutuelle exclusion  %
    nbpanocc   := 0;
    nbcabocc   := 0;
    nbcablibre := C;
    nbpanlibre := P;
    mutex      := 1;

procedure se_deshabiller
  begin
    P(nbpanlibre);
    P(nbcablibre);
    < se deshabiller >;
    V(nbcablibre);
  end.

procedure s_habiller
  begin
    P(nbcalibre);
```

```
  < s'habiller >;
  V(nbcablibre);
  V(nbpanlibre);
 end.
```

### 11.2 Solution avec les régions critiques

```
 Var pancab  shared record nbpanocc : integer;
                           nbcabocc : integer;
             end
   nbpanocc := nbcabocc := 0;

procedure se_deshabiller
 begin
  region pancab when nbcabocc < C et
       nbpanocc < P do
                  begin
                     nbcabocc := nbcabocc + 1;
                     nbpanocc := nbpanocc + 1;
                  end

      < déposer  vêtements >;

  region pancab do nbcabocc := nbcabocc - 1;

 end.

 procedure s_habiller
  begin
   region pancab when nbcabocc < C do
                nbcabocc := nbcabocc + 1;

    < remettre ses vêtements >;

   region pancab do begin
                     nbcabocc := nbcabocc - 1;
                     nbpanocc := nbpanocc - 1;
                  end
  end.
```

## 11.3 Solution avec les Moniteurs

### 11.3.1 Moniteurs classiques

```
Piscine.Monitor;
 begin
   var  nbpanocc , nbcabocc : integer;
        panlibre, cablibre  : condition;

   procedure debut_habillage;
     begin
       if nbcabocc = C then wait(cablibre);
       nbcabocc := nbcabocc + 1;
     end;

   procedure fin_habillage
     begin
       nbpanocc := nbpanocc - 1;
       nbcabocc := nbcabocc - 1;
       signal(panlibre);
       signal(cablibre);
     end;

   Procedure debut_deshabillage
     begin
       if nbpanocc = P then wait(panlibre);
          nbpanocc := nbpanocc + 1;
       if nbcabocc = C then wait(cablibre);
          nbcabocc := nbcabocc + 1;
     end;

   procedure fin_deshabillage
     begin
       nbcabocc := nbcabocc - 1;
       signal(cablibre);
     end;
   % Initialisation %
   begin
       nbpanocc := 0;
       nbcabocc := 0;
   end
```

```
 end
endmonitor.
```

### 11.3.2 Variante de Kessels

```
Piscine.Monitor;
 begin
   var nbpanocc, nbcabocc : integer;
       panlibre : condition nbpanocc < P;
       cablibre : condition nbcabocc < C;

 procedure debut_habillage
   begin
     wait(cablibre);
     nbcabocc := nbcabocc + 1;
   end.

 procedure fin_habillage
   begin
     nbcabocc := nbcabocc - 1;
     nbpanocc := nbpanocc - 1;
   end.

 procedure debut_deshabillage
   begin
     wait(panlibre);
     nbpanocc := nbpanocc + 1;
     wait(cablibre);
     nbcabocc := nbcabocc + 1;
   end.

 procedure fin_deshabillage
   begin
     nbcabocc := nbcabocc - 1;
     nbpanocc := nbpanocc - 1;
   end.

   % Initialisation %
   begin
     nbpanocc := 0;
     nbcabocc := 0;
```

***end***
***end***.
***endmonitor***.

## 12. Problème de Traitement pipe-line

On considère un système à N processus (Pi, $0 \leq i \leq N-1$) qui évoluent de la manière suivante: P0 produit des éléments ou messages du type T de taille identique qu'il communique à P1. Ce dernier procède à un certain traitement sur ces messages puis les communique à P2 qui agit de même envers P3 et ainsi de suite jusqu'à Pn-1.
Le schéma est donc le suivant: P0 ---> P1 ---> P2 ---> ... Pn-2 ---> Pn-1.
(Pour des raisons de cohérence, on suppose que Pn-1 transmet à P0 des messages vides. Le scénario de chaque processus Pi est comme suit:

```
" Processus i "
  begin
    cycle
      < recevoir message du prédécesseur >;
      < traiter message >;
      < transmettre au successeur >;
    endcycle
  end.
```

**Questions**:
**1.** En supposant que la recopie des messages est inutile, autrement dit pas d'émission et réception réelle de messages (on accèdera aux messages seulement à travers leurs indices), écrire un algorithme d'un processus Pi manipulant des messages partiellement traités. On utilisera les différents outils de synchronisation à savoir:

a) Les sémaphores;
b) Les régions critiques;
c) Les moniteurs.

**2.** Peut-il y avoir des situations d'interblocage ? Démontrer.

### 12.1 Solution avec les Sémaphores

```
var bufmes : array[0..nmsg-1] of T;              % buffer de messages          %
var disp   : array[0..N-1] of 0..nmsg-1;         % indique le nombre de msg    %
mutex      : semaphore init 1;                   % disponibles pour chaque     %
sempriv    : array[0..N-1] of semaphore init 0;  % processus                   %
begin                                            % Initialisation              %
   P(mutex);
   disp[0] := nmsg;     % P0 dispose initialement de nmsg messages %
   i := 1;
   while i ≤ N-1 do disp[i] := 0;
        i := i + 1;
```

```
   endwhile
   V(mutex);
end
            "  Processus i "
Procedure proc(i)
begin
   var j ;  % indice du msg à consommer %
   var succ : integer;
   j := 0;
   succ := (i+1)mod(N)
  cycle
    P(mutex)
    if disp[i] > 0 then begin
                    < consommer bufmes[j] >;
                     disp[i] := disp[i] - 1;
                     disp[succ] := disp[succ] + 1;
                     V(sempriv[i]);
                     V(sempriv[succ]);
                     j := (j + 1)mod(nmsg);
                    end
    V(mutex);
    P(sempriv[i]);
  endcycle
end
```

## 12.2 Solution avec les Régions critiques

```
Var Bufmes : shared array[0..nmsg - 1] of  T
Var Disp   : shared array[0..N-1] of  0..nmsg-1;

% Initialisation %
 region Disp do
  begin
     Disp[0] := nmsg ;
     i := 1
     while i ≤ N-1 do
        Disp[i] := 0;
        i := i + 1;
     endwhile
  end
```

" **Processus i** "

```
Procedure Proc(i)
begin
  j := 0 ;       % indique la trace des msg consommés %
  succ := (i+1)mod (N);
  cycle
    region Disp do
      await Disp[i] > 0;
    region Bufmes[j] do "traiter message";
    region Disp do
          begin
            Disp[i] := Disp[i] - 1
            Disp[succ] := Disp[succ] + 1
          end
    j := (j+1)mod (nmsg)
  endcycle
end
```

### 12.3 Solution avec les Moniteurs

```
Traitmsg.Monitor
begin
  var bufmes : array [0..nmsg-1] of  T;
  cond       : array [0..N-1]   of condition;
  var disp   : array [0..N-1]   of integer;

procedure traitermsg (i)
  begin
    if disp(i) ≤ 0 then wait(cond[i]);
    finsi
    disp(i) := disp(i) - 1;
  end

procedure endmsg (i,succ)
  begin
    disp(succ) := disp(succ) + 1;
    signal(cond[succ]);
  end
```

```
 " initialisation "
begin
  disp(0) := nmsg;
  i := 1;
  while i ≤ N-1 do
    disp(i) := 0;
    i := i + 1;
  endwhile
end
endmonitor.
```

Comportement d'un processus Pi:

```
" Processus Pi "
      var j;
      j := 0;
cycle
  begin
    traitermsg(i);
    " consommer bufmes(j) "
    succ := (i+1)mod (N);
    endmsg(i,succ);
    j := (j+1)mod (nmsg);
  end
endcycle.
```

### 12.4 Interblocage ?

Un interblocage peut survenir dans la situation suivante:
P0 est en attente de message transmis par Pn-1 **et**
P1 est en attente de message transmis par P0 **et**

.
.

Pn-1 est en attente de message transmis par Pn-2 et cela pour condition suivante:
disp[0] = 0 et disp[1] = 0 et ... disp[n-1] = 0. Mais puisque nmsg > 0, et que l'exclusion mutuelle satisfait l'invariant disp[i] = nmsg, cette condition ne peut se réaliser d'où l'absence d'interblocage.

## 13. Problème de rivière

Soit un pont enjambant une rivière; pour passer du coté droit vers le coté gauche et vise versa (voir figure 4. ci-après), les véhicules doivent emprunter le pont.

Toutefois, la largeur du pont ne permet que le passage à sens unique autrement dit, les véhicules peuvent se suivre mais ne peuvent se croiser sur le pont.

Ecrire l'algorithme exécuté par tout véhicule désirant franchir le pont (coté droit, coté gauche), en utilisant les mécanismes de synchronisation suivants:

1. Sémaphores,
2. Régions critiques,
3. Moniteurs.

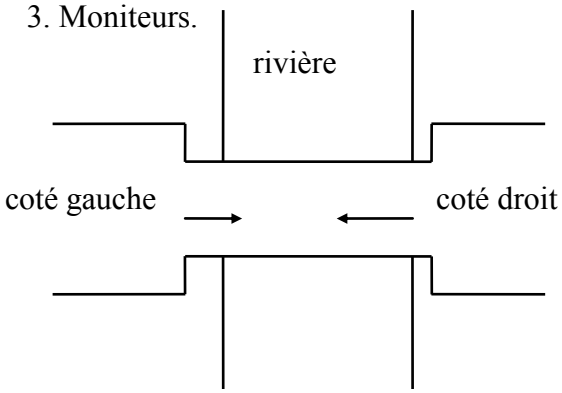



Figure 4. Variante du modèle lecteurs-rédacteurs

Ce problème constitue une variante du modèle lecteurs-rédacteurs sans spécification de priorité. Chaque véhicule modélise un processus et le pont représente la ressource partageable.

### 13.1 Solution à l'aide des Sémaphores

Voir solution du modèle lecteurs-rédacteurs.

### 13.2 Solution à l'aide des Régions critiques

```
begin
  Var pont : shared record
        droit, gauche; integer  end
        droit := gauche := 0;

  Processus VDROIT   % (véhicule droit) %
  begin
   region pont when gauche = 0  do
      droit := droit + 1;
```

```
        < passage >
    region pont do droit := droit - 1;
  end
```

De l'algorithme précédent découle, par symétrie, celui des processus VGAUCHE.

La solution précédente risque de provoquer une privation des véhicules venant d'un coté au profit de ceux du coté opposé lorsque ces derniers se coalisent.

Une manière d'éviter cette privation consiste à changer de sens de passage lorsque N véhicules ont franchi le pont, et au moins un véhicule est en attente à l'autre bout du pont.

### 13.2.1 Solution réduisant la privation

```
processus VDROIT:
  attente : nombre de véhicules en attente de passage
  passage : nombre de véhicules traversant le pont
  premier : nombre de véhicules s'engageant au bout du pont
  type sens : record
                attente, passage, premier: integer;
              end
  var pont : shared record
                       gauche, droit : sens;
                     end
  var attente := passage := premier := 0;

" processus VDROIT "

   begin
    region pont do region droit do
          begin
            attente := attente + 1;
            await gauche.passage = 0 et
                      premier < N ;
            attente := attente - 1;
            passage := passage + 1;
            if gauche.attente > 0 then
              premier := premier + 1;
          end
       < passage >
    region pont do region droit do
```

```
        begin
          passage := passage - 1;
          if passage = 0 then premier = 0;
        end
    end
```

### 13.3.3 Solution avec les Moniteurs

```
pont.monitor                    % variante de Kessels              %
begin
  var droit, gauche : integer;
  cdroit  : condition gauche = 0;  % condition de franchissement   %
  cgauche : condition droit  = 0;  %    "          "              %
procedure EVDROIT               % entrée d'un véhicule coté droit %
  begin
    wait(cgauche);
    droit := droit + 1;
  end

procedure SDROIT                % sortie du coté droit %
  begin
    droit := droit - 1;
  end

procedure EGAUCHE               % entrée coté gauche %
  begin
    wait(cdroit);
    gauche := gauche + 1;
  end

procedure SGAUCHE               % sortie coté gauche  %
  begin
    gauche := gauche - 1;
  end
 % initialisation %
 begin
   gauche := 0;
   droit  := 0;
 end
endmonitor.
```

## 14. Problème des processus dépendants

Soient trois procédures P, Q, R manipulant des données communes et pouvant être appelées par des processus distincts. Pour des raisons sémantiques des services rendus, les procédures P, Q, R doivent respecter le schéma d'exécution suivant: (P --> Q --> R)* où * : indique un facteur de répétition, autrement dit: L'exécution de P doit précéder celle de Q qui à son tour doit précéder celle de R, et le cycle peut recommencer.

**Question**: Donnez une implémentation de ce schéma d'exécution associé aux trois procédures en utilisant comme outil de synchronisation:

1. Les sémaphores;
2. Régions critiques;
3. Moniteurs;
4. Expressions de chemin.
5. Modules de contrôle;

### 14.1 Solution basée Sémaphores

```
mutexp   : semaphore init 1;
semprivq : semaphore init 0;
semprivr : semaphore init 0;

procedure P
cycle
 begin
   P(mutexp);          % entrée  exclusive dans la  procédure P %
   < corps de P>;
   V(semprivq);        % autoriser l'entrée dans la procédure Q %
 end
endcycle

Procedure Q
cycle
 begin
   P(semprivq);      % demande d'exécution de la procédure Q %
   < corps de Q >;
   V(semprivr);      % libérer l'entrée dans la procédure R       %
 end
endcycle

procedure R
cycle
```

```
 begin
   P(semprivr);       % demande d'exécution de la procédure R %
   < corps de R);
   V(mutexp);         % autoriser l'exécution de P de nouveau %
 end
endcycle
```

### 14.2 Solution basée régions critiques

```
Var  tour shared init 1..3
procedure P
cycle
 begin
  region tour when tour = 1 do
    begin
      < corps de P >;
      tour = 2;                    % autoriser exécution de Q %
    end
 end
endcycle

procedure Q
cycle
 begin
  region tour when tour = 2 do
    begin
      < corps de Q >;
      tour = 3;         % autoriser exécution de R %
    end
 end
endcycle
procedure R
cycle
 begin
  region tour when tour = 3 do
    begin
      < corps de R >;
      tour = 1;         % autoriser de nouveau P %
    end
 end
endcycle
```

## 14.3 Solution basée Moniteurs

La mise en oeuvre de l'exercice requiert dans ce cas une détermination des points de synchronisation, aussi, chaque processus doit donc invoquer le moniteur pour demander l'autorisation d'exécution de la procédure désirée, et une fois cette dernière terminée, invoquer de nouveau le moniteur pour permettre l'exécution de la procédure suivante (mettre l'indicateur tour à la bonne valeur et réveil).

### 14.3.1 Variante classique de Hoare

```
pqr.monitor      % variante classique de Hoare %
begin
  var tour  : integer;
  conditionp: condition;          %  tour = 1; %
  conditionq: condition;          %  tour = 2; %
  conditionr: condition;          %  tour = 3; %
procedure execP
 begin
   if tour ≠ 1 then
      wait(conditionp);
      tour := 4;                  % Tour = 4: évite l'exécution      %
 end                              % répétée d'une même procédure %
procedure endp
  begin
    tour = 2;
    signal(conditionq);
  end
procedure execQ
 begin
   if  tour ≠ 2 then
     wait(conditionq);
     tour := 4;
 end
procedure endQ
  begin
    tour = 3;
    signal(conditionr);
    tour := 4;
  end
procedure execR
 begin
   if tour ≠ 3 then
```

```
    wait(conditionr);
    tour := 4;
 end

procedure endR
  begin
    tour = 1;
    signal(conditionp);
  end

% Initialisation %
begin
  tour := 1;
end
endmonitor.
```

Remarque: l'algorithme précédent peut être amélioré en utilisant la variante de kessels qui permettra de réduire le nombre de points de synchronisation (voir ci-après).

### 14.3.2 Variante de Kessels

```
pqr.monitor; variante de Kessels
begin
  var tour  : integer;
  conditionp: condition  tour = 1;   % tour = I: Condition de franchissement %
  conditionq: condition  tour = 2;   % tour ≠I : Condition d'attente          %
  conditionr: condition  tour = 3;

procedure execP
 begin
   wait(conditionp);
   tour := 4;
 end

procedure execQ
 begin
   wait(conditionq);
   tour := 4;
 end
procedure execR
```

```
 begin
   wait(conditionr);
   tour := 4;
 end
procedure endpqr(retour)
  begin
    case retour of
    1: tour := 2;
    2: tour := 3:
    3: tour := 1;
  end

% Initialisation %

begin
  tour := 1;
end
endmonitor.
```

Le protocole d'appel de tout processus est le suivant:
soit proc le processus appelant;

```
   proc
    begin


  call x.pqr;      x: représente soit execP, execQ, ou execR
  call endpqr.pqr(retour);  % retour = (1,2,3) %

    end
```

### 14.4 Solution basée Expressions de chemin

La mise en oeuvre à l'aide des expressions de chemin est évidente. Il suffit donc d'utiliser les opérateurs de séquence (;) et de répétition ***path end*** comme suit: ***path*** P; Q; R ***end***
Cette simplicité montre bien la puissance et l'adéquation des expressions de chemin dans la formulation de la spécification de la sérialisation des actions.

### 14.5 Solution basée Modules de contrôle

PQR.*MdC*

```
begin
  type struc = begin
                 procedure wait ;  {corps vide};
```

```
            procedure resume; {corps vide};
            condition(wait): act(wait)+act(resume)= 0
                    and aut(wait) - aut(resume) > 0;
          condition(resume): act(wait) + act(resume) = 0;
        end
 var tour     : integer init 1;
    strucpriv: array[identite_processus] of struc;
    queuep, queueq, queuer : queue init empty;
 { déclaration des procédures enqueue et dequeue }
procedure P(id:identité de l'appelant)
  begin
    PP(id);
    strucpriv[id].wait;
  end

procedure PP(id)
  begin
    if tour = 1 then strucpriv[id].resume
    else  enqueue(id, queuep);
    endif
  end

procedure Q(id)
  begin
    QQ(id);
    strucpri[id].wait;
  end

procedure QQ(id)
  begin
    if tour = 3 then strucpriv[id].resume
    else  enqueue (id, queueq)
    endif
  end

procedure R(id)
  begin
    RR(id);
    strucpri[id].wait;
  end
```

```
procedure RR(id)
 begin
  if tour = 3 then strucpriv[id].resume
  else  enqueue (id, queuer)
  endif
 end

procedure endpqr
 var id : identité_processus;
 begin
  case tour of

  1: begin
       tour := 2;
       dequeue(id, queueq);
     end

  2: begin
       tour := 3;
       dequeue(id, queuer);
     end

  3: begin
       tour := 1;
       dequeue(id, queuep);
     end
  endcase
  sempriv[id].resume;
 end

end PQR.
```

Le protocole d'appel est identique à celui donné en 3.

## 15. Problèmes Scheduler de tête de L/E d'un disque à bras mobile

Un disque a bras mobile comporte une tête unique de lecture-écriture par face de disque. Un déplacement radial du bras porteur permet d'amener la tête au-dessus de la piste choisie. Lorsque l'unité de disques comporte plusieurs faces, tous les bras qui portent les têtes sont solidaires.
L'ensemble des pistes accessibles sans déplacement des bras est appelé un cylindre.

Plusieurs stratégies de gestion des mouvements des têtes existent, elles visent à minimiser le temps de déplacement notamment lorsque l'activité du disque est intense (plusieurs processus demandent des transferts).

Parmi ces stratégies, citons celle dite: de la plus courte distance, qui consiste à servir les requêtes la plus proche de la piste (ou cylindre) courante. Son inconvénient majeur réside dans la possibilité de conduire à des phénomènes de privation.

Pour remédier à cet anomalie, on préfère plutôt la technique de l'ascenseur dont l'objectif est de minimiser la fréquence de changement de direction mouvement des têtes de lecture-écriture. Ces dernières balayent les surfaces du disque pour servir la requêtes la plus proche du cylindre courant selon une direction donnée.
Lorsqu'il n'existe aucune requêtes à satisfaire (dans la direction courante), le balayage s'effectue suivant la direction opposée.

**Questions**:
Ecrire l'algorithme de ce scheduler qui comporte les procédures suivantes:
Demander(c) où c est un numéro de cylindre, et
Liberer;
En utilisant les mécanismes de synchronisation suivant:
a) les sémaphores;
b) Les Moniteurs : 1. classiques (Hoare);
2. variante de Kessels.

### 15.1 Solution avec Sémaphores

```
Procedure demander(c:cylindre, p:nomprocessus);
 begin
 acces: semaphore  init : 1;
 sempriv array[0..max] of semaphore init 0;   * (1)
 var   direction = {intext, extint};                * (2)
      diskoccupe: boolean  init : false;
      nbintext, nbextint: integer init: 0;
      postete:  integer init 0;                          * (3)
```

```
{déclaration des procédures chainer_inex, chainer_exin,   extraire }

P(acces)          % tentative d'entrer en section critique %
if diskoccupe
then
  if postete ≤ c and  direction = intext
    then begin
          nbintext := nbintext + 1;
          chainer_inex(p,c);
         end;
  else begin
        nbextint := nbextint + 1;
        chainer_exin(p,c);
       end
  endif
else begin
       diskoccupe := true, postete :=c;
       V(sempriv[p]);
    end
endif
 V(accès)         % libération de la section critique %
 P(sempriv[p])
end

Procedure Liberer
 begin
  P(accès);
  diskoccupe := false;
  if direction = intext then  begin
                        nbintext := nbintext - 1;
                        if nbintext = 0 then begin
                                          extraire(p,c);
                                          réveil := true;
                                         end
                        else begin
                             direction := extint;
                             nbextint := nbextint - 1;
                             if nbextint = 0
                               then begin
                                    extraire(p,c);
                                    réveil := true;
```

```
                        end
                    endif
                    if réveil
                    then begin
                        postete := c;
                        diskoccupe := true;
                        V(sempriv[p]);
                      end
                    endif
                  end
                endif
              end
else
  nbextint := nbextint - 1;
  if nbextint = 0
  then begin
       extraire(p,c);
       réveil := true;
     end
  else begin
       direction := intext;
       nbintext := nbintext - 1;
       if nbintext = 0
       then begin
            extraire(p,c);
            réveil := true;
          end
       endif
     end
  endif
  if réveil
  then begin
       postete := c;
       diskoccupe := true;
       V(sempriv[p]);
     end
  endif
endif
V(accès);
end
```

(1): sempriv[0..max]: tableau de sémaphores privés de dimension le nombre *max* de processus concurrents pour les demandes de transferts disque.
(2): intext: direction de l'intérieur (centre) vers l'extérieur (périphérie) de la surface du disque; extint: opposé de intext.
(3): postete: position courante de la tête de lecture-ecriture, initialement position de repos étant 0.
(chainer_inex(c,p): procédure qui chaîne les processus (cylindre, identité) en attente selon la direction intext. Egalement de même pour chainer_exin(c,p) selon la direction extint.

### 15.2 Solution basée Moniteurs

```
Schedtete.monitor
 begin
   postete   : cylindre;
   direction : (intext, extint);
   diskoccupe: boolean;
   haut, bas : condition;

procedure demander (dest:cylindre)
  begin if diskoccupe
        then if postete < dest or postete = dest
                        and direction = haut
           then begin
                  bas.wait(cylmax-dest);
                  diskoccupe := vrai;
                  postete := dest;
                end
  end

procedure liberer
  begin
    diskoccupe := faux;
    if direction = haut then signal(haut)
    else begin
          direction := bas;
          signal(bas);
         end
    else if bas.queue then signal(bas)
        else begin
              direction := haut;
              signal(haut);
```

```
        end
  end

    % initialisation %

    begin
      direction := haut;
      diskoccupe:= faux;
    end

endmonitor.
```

*Remarque*: constatez la différence entre les deux outils (sémaphores et moniteurs) tant au niveau de la clarté et de la lisibilité.

## 16. Communication via messages entre processus internes et terminal

On considère un système dans lequel un ensemble de processus internes P1, P2 ... Pn communique avec un opérateur ou un usager interactif U par l'intermédiaire d'un terminal.
On souhaiterait que tout processus Pi ($1 \leq i \leq N$) et l'usager correspondant puissent s'envoyer des messages unidirectionnels ou bidirectionnels. Le sens unidirectionnel (respectivement bidirectionnel) indique simplement qu'un msg ne nécessite pas de réponse (respectivement avec réponse obligatoire). Quatre types de communications sont possibles:

*1. Processus-a-Usager (PU).*
Un processus Pi envoi un msg (message) à l'usager U. cet envoi ne nécessite aucune réponse. Il est à sens unique. Ce type de msg peut contenir des informations relatives au fonctionnement du système; par exemple: une arrivée de courrier électronique de l'usager, un msg relatif à une exception (division par 0), l'espace mémoire disponible, etc...

*2. Processus-a-Usager-a-Processus (PUP).*
Un processus Pi émet un msg bidirectionnel au terminal de l'usager; ce dernier doit répondre au processus émetteur Pi. Il est permis à l'usager U de différer (retarder) sa réponse. Pour cela, il avise le système immédiatement par l'envoi d'un msg fictif par exemple un msg du type REPDIF (réponse différée).
La réponse appropriée sera émise ultérieurement au moment opportun. Par exemple un msg type " à réponse différée " est celui adressé à l'opérateur pour monter sur dérouleur une bande magnétique.
Un msg à réponse immédiate peut être la demande du prochain backup.

*3. Usager-a-Processus (UP).*
Un tel msg est à "sens unique", émis par l'usager U au processus Pi sans exiger de réponse. Par exemple, annoncer la disponibilité d'un périphérique après réparation.

*4. Usager-a-Processus-a-Usager (UPU)*:
Pour un msg du type bidirectionnel, l'usager interroge un processus et attend une réponse. Par exemple "lister les routines de ma bibliothèques privée", ou bien " quel est l'espace mémoire utilise par mon job, "quelle heure est-il", etc...

La transmission machine --> terminal est du type half-duplex (i.e les informations peuvent être émises en bidirectionnel mais seulement une transmission et une seule est possible à un instant donné). La ligne de transmission peut être un canal standard ou une ligne de communication remote-terminal.

On suppose que U est plus prioritaire vis-à-vis de Pi relativement à l'initiative d'envoi de msg, et qu'il indique son intention en positionnant un indicateur à tout instant.

Pour raison de simplicité, les processus internes constitueront deux groupes disjoints:
P1 = { Pi faisant intervenir des transmissions PU et PUP }
P2 = { Pi traitant des msg UP ou UPU }.
Les processus P1 initient des conversations avec la console de l'usager, tandis que ceux de P2 traitent des conversations initiées par l'usager.
La communication directe entre le terminal et les processus Pi est rendue difficile par le routage des msg issus du terminal vers le processus Pi adéquat. On utilisera désormais un processus *Interpréteur* de msg pour réaliser cette tache de routage, de contrôle d'erreur et de lancement des opérations d'E/S.
Le système se présente schématiquement comme l'indique la figure 5 suivante:

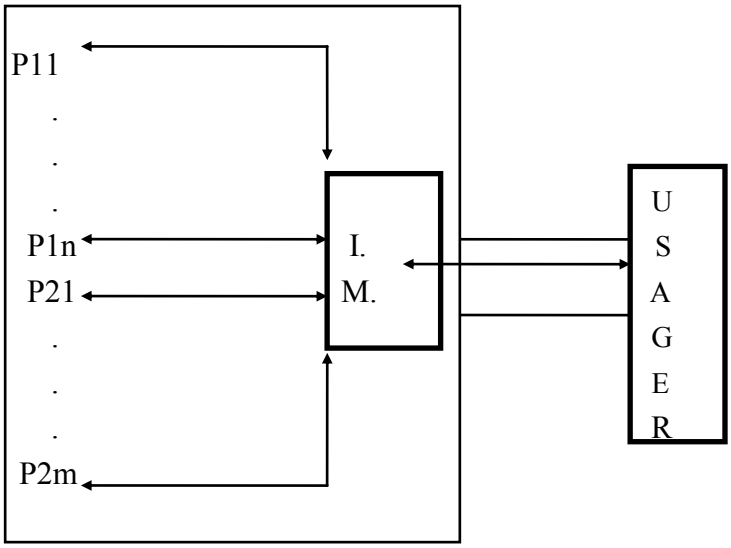



Figure 5. Communication processus internes ◄►Usager

Soit la structure de msg suivante: (p,t,m)
Où p est le nom du processus concerne, et t ∈ {UP,UPU,PU,PUP} et désigne le type du msg, et m le texte de ce msg.

Un pool de buffets est utilise pour contenir les msg; chaque buffer b est de la forme: BUF(b)=[PR(b),ty(b),text(b)] (voir structure de données ci-après). Les deux premiers paramètres permettent à l'opérateur et à l'interpréteur **IM** d'identifier le msg.

Deux listes sont maintenues dans le pool de buffers:
1. Une liste de buffers vides,
2. Une liste de buffers pleins de types PU ou PUP.

On utilisera deux routines Takebuf(x) et Addbuf(x,index) pour manipuler les buffers dans le pool; x = type $\in$ {vide,plein} et 0 $\leq$ index $\leq$ n-1, indique le numéro du buffer. Addbuf(x,index) chaîne en fin de liste de type x le buffer de numéro: index;

Takebuf(x): retire (déchaîne) le premier buffer de la liste de type x et retourne son numéro (index) comme valeur de la fonction takebuf.

On associe à chaque processus Pi $\in$ P2 une variable B(i) contenant le numéro (index) du buffer durant le traitement UP ou UPU.

Plusieurs sémaphores (à définir) sont nécessaires pour réaliser l'exclusion mutuelle, la synchronisation et le comptage des ressources.

**Question:**
Programmer les algorithmes des processus Pi $\in$ P1 et Pj $\in$ P2 et l'interpréteur de msg IM.

| Variables utilisées: | Fonction : |
|---|---|
| a | indicateur d'attention |
| up | classe de msg du type UP |
| upu | classe de msg du type UPU |
| pu | classe de msg du type PU |
| pup | classe de msg du type PUP |
| BUF(b)= | b eme buffer dans le pool |
| [pr(b), | nom du processus |
| ty(b), | type du msg |
| text(b)] | text du msg |
| empty | classe de buffers vides |
| full | classe de buffers pleins |
| F | pointeur du premier buffer plein |
| ce | sémaphore compteur de buffers vides |
| cf | sémaphore compteur de buffers pleins |
| me | sémaphore de mutuelle exclusion de buffers vides |
| mf | sémaphore de mutuelle exclusion de buffers pleins |
| s(i) | sémaphore privé du processus i |
| ans | sémaphore de synchronisation des msg up/upu |

Le squelette de l'algorithme d'un processus de Pi $\in$ P1 peut être:

```
" Processus Pi "
 begin
Li:      .
         .
         .
 % envoi de msg PU  %
 PU: b:= Dequeue(ce,me,empty);
     Buf[b]:=(i,pu,m);
     Queue(cf,mf,b,full);
         .
         .
 % lancer une transmission PUP %
PUP:  b:= Dequeue(ce,me,empty);
        Buf[b]:=(i,pup,m);
        Queue(cf,mf,b,full);
  %  attendre une réponse %
        P(s[i]);
       m:=text(b);
      Queue(ce,me,b,empty);
         .
         .
         .
   goto Li;
 end;

Integer procedure Dequeue(s1,m1,t);
semaphore s1, m1;
 begin
   P(s1);          % décrémenter nombre de buffers vides avec blocage
   P(m1);          % eventuel si ce nombre est nul
   Dequeue := Takebuf(t);        % execution en exclusion mutuelle %
   V(m1)
 end;

Procedure Queue(s1,m1,index,t);
 semaphore s1, m1;
 begin
   P(m1);
   Addbuf(t,index);
   V(m1);
   V(s1);          % réveil éventuel de processus en attente de buffers %
 end;              % vides disponibles %
```

***Forme d'un processus Pj de la classe de P2***

```
                "Processus Pj"
   begin
up_upu: P(s[j])          % attente de msg %
      b:=B[j];            % B[j]: variable contenant l'index de buffer
      m:=text[b];         % durant le traitement de msg Up ou UPU
      t:=ty[b];
      if t=upu then
            begin      % Trouver la réponse adéquate %
              text[b]:=reponse;
            end
                     % réveiller  I.M. %
       V(ans);
       goto up_upu;
   end;

" Processus I.M "     % Interpréteur de msg I.M.   %
begin
lim:
 if a
 then begin                 % U demande l'attention, lecture du msg tapé %
     receive(p,t,m);
     a :=  false ;          % remise à l'état initial de l'indicateur d'attention %
    if  t= up or t = upu
    then begin           % Entrer msg dans buffer %
         b:=B[p]:=Dequeue(ce,me,empty);
         buf[b]:=(p,t,m);
         V(s[p]);                % réveil de p %
          % attendre jusqu'à msg reçu ou reponse retournée %
         P(ans);
         if  t = upu
         then begin
                send(buf[b]);
                    % si upu, envoyer réponse %
                    % remettre le buffer dans le pool %
               Queue(ce,me,b,empty);
            end
       end
    else if  t = pup then
                  begin  % différer réponse à pup %
```

```
                    text(B[p]):=m;
                   V(s[p]);
                  end
    endif
```

*Remarque*: On suppose que les primitives de transmission ***send*** et ***receive*** ci-dessus sont offertes par le système.

```
else
   begin                % traiter pu or pup si la file full n'est pas vide  %
    if  F(full) ≠ full    % F pointeur de tête de file full
    then  begin
            b:= Dequeue(cf,mf,full);
            send(Buf[b]);
          % vérifier si c'est pup %
           if ty[b]=pup
          then begin
                 Receive(p,t,m);
                   % est-ce une réponse virtuelle %
                if m = "repdif" then B[p]:=b;
               else begin
                      text[b]:=m;
                     V(s[p]);
                    end
               endif
              end
         endif
        end
    endif
   end
endif
goto lim;
end.
```

La manipulation des buffers précédents est accomplie par l'intermédiaire de deux routines Addbuf(type,number), et Takebuf(type) où type ∈ {empty,full} et 0 ≤ number ≤ N-1.
Addbuf: ajoute buf(number) à la fin de la liste de classe type; et Takebuf: fonction qui retire le premier buffer de la liste et retourne son index comme valeur de la fonction.

```
procedure Addbuf(type,number)
type, number: value;
type, number: integer;
```

```
begin
  integer: dernier;
  dernier:= L(type);
  F(dernier):= number;        % F(i): chaîne le buffer i au suivant %
  F(number) := type;          %       du même type. %
  L(type) := number;
end;
```

## 17. Serveur de gestion de fichiers

Un serveur de gestion de fichiers est connecté à un réseau (figure 6), et peut satisfaire des demandes (ou requêtes) en provenance d'autres sites sur le réseau (clients).

Le but de ce problème est d'étudier quelques aspects de la synchronisation entre les processus qui réalisent les fonctions du serveur.

L'accès aux fichiers est réalisé sur le serveur par N processus P1, ..., Pi ,..., Pn (N fixé) appelés *processus de service,* qui peuvent s'exécuter en parallèle (le serveur pouvant être multiprocesseur). Une requête quelconque peut être traitée par n'importe lequel des processus Pi.

Un *processus récepteur* reçoit, sous forme de messages, les requêtes des clients qu'il transmet aux processus de service. Quand un processus de service a fini de traiter une requête, il transmet le résultat à un *processus émetteur*, qui est chargé de l'envoyer au client ayant émis la requête (chaque requête contient l'identité du client émetteur). les clients sont partages entre deux classes A et B. La politique de service est la suivante:

- Les requêtes des clients de la classe A sont prioritaires sur celles de la classe B (autrement dit, un processus de service Pi ne traite une requête de la classe B que s'il n'y a pas de requêtes de la classe A en attente).
- Les requêtes de deux clients de la même classe sont traitées dans leur ordre de réception par le processus récepteur. Le schéma ci-après représente l'organisation générale du serveur.

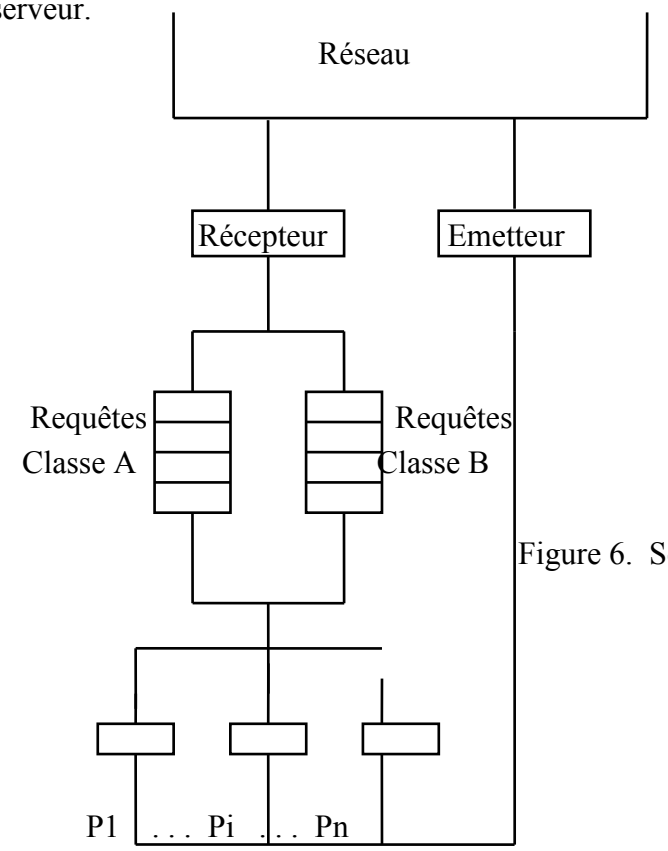



Figure 6. Schéma général du serveur

***Questions:***

**1.** Exprimer les conditions de synchronisation qui doivent être respectées par les processus Pi pour la prise en compte des requêtes.

**2.** Donnez l'algorithme du moniteur qui réalise la synchronisation du récepteur et des Pi; ce moniteur comportera deux procédures: déposer(requête), appelée par le récepteur et retirer(requête) appelée par les Pi. On suppose qu'un champs requête.classe contient la classe(A ou B) de la requête.

**3.** Le système ci-dessus comporte un certain nombre de problèmes, identifiez les, et proposez-en des solutions en maintenant le privilège accordé aux requêtes de la classe A

***Réponses***

**1.** Pour traiter une requête, tout processus Pi($1 \leq i \leq N$) doit la retirer de l'une des files A ou B. Ces mêmes files sont utilisées en concurrence par le récepteur. Pour garantir l'intégrité de ces files, les accès doivent s'effectuer en exclusion mutuelle (exclusion mutuelle entre les Pi et le récepteur). De même, toute requête doit être prélevée une et une seule fois. Les conditions de synchronisations sont:

**a)** Le dépôt de requêtes (par le récepteur) et le retrait (par les Pi) doivent s'exclure mutuellement.

**b)** Toute requête ne doit faire l'objet que d'un et un seul traitement (prélèvement par un Pi unique).

<u>*Remarque*</u>: Ce problème constitue une variante du modèle producteur-consommateur; précisément: un producteur et N consommateurs.

### 17.1 Solution basée Moniteurs

**2.** serveur.***monitor***

```
  begin
    type requête =  record
                      classe : ('A','B');
                      identité: <identité_processus>;
                      message : < structure message >;
                    end
    var   req :  requête;
          fileA , fileB : queue;
          nbreqa, nbreqb: integer;
    emptyfiles : condition
< déclaration des procédures entrer et sortir >;
```

Ces deux procédures doivent pouvoir manipuler au besoin la file A comme la file B, donc un paramètre adéquat leur permettra de distinguer entre A et B.

```
procedure deposer(req)
begin
  case req.classe of
  'A': begin
        nbreqa :=  nbreqa + 1;
        entrer(A,req);        % déposer requête dans file A %
       end
  'B': begin
        nbreqb := nbreqb + 1;
        entrer(B,req);        % déposer requête dans file B %
       end
  endcase
  while not empty emptyfiles do
       signal(emptyfiles);
  endwhile
end

procedure retirer(req)
begin
  if nbreqa = 0 then
              if nbreqb = 0
              then wait(emptyfiles)
              else begin
                    nbreqb := nbreqb - 1;
                    sortir(B,req); % sortir requête de la file B %
                   end
              endif
  else begin
        nbreqa := nbreqa - 1;
        sortir(A,req); % sortir requête de la file A %
       end
  endif
end
       % initialisation %

       begin
         nbreqa := 0;
         nbreqb := 0;
       end

endmonitor.
```

## 17.2 Problèmes possibles

**a)** Le traitement prioritaire des requêtes de classe A peut conduire à un phénomène de privation du traitement des requêtes de classe B: il suffit pour cela que des arrivées successives importantes de requêtes de classe A se produisent (cf. problème lecteurs-rédacteurs avec priorité aux lecteurs). Pour éliminer cette privation, tout en privilégiant le traitement des requêtes de type A, on peut adopter la politique suivante: Pour chaque traitement d'un nombre NA de requêtes de classe A (NA à déterminer), on force le traitement de NB requêtes de classe B tel que: NB < NA (la différence NA - NB représente le privilège accordé aux requêtes de type A).

**b)** Il n'est pas judicieux de traiter les requêtes d'une même classe dans l'ordre de réception. En effet, si des usagers dialoguent entre eux à travers le réseau par l'intermédiaire d'un fichier partagé, des occurrences d'incohérences sont possibles. De telles incohérences peuvent être, par exemple, la non correspondance d'une réponse à un message, ou bien le non respect de la spécification du modèle producteur-consommateur (si les interlocuteurs agissent selon ce modèle), etc...

## 18. Problème de circulation au niveau d'un carrefour

Soit un carrefour à deux voies A et B dont la circulation automobiles est réglée par des feux verts et rouges (voir figure ci-après). Quand le feu est vert pour une voie A (respectivement B), les véhicules qui y circulent peuvent traverser le carrefour; quand le feu est rouge, ils doivent attendre. On suppose, pour raisons de simplicité, que les véhicules traversent le carrefour en ligne droite et que les conditions suivantes sont respectées:
1) Tout véhicule arrivant à l'entrée du carrefour doit le franchir au bout d'un temps fini,
2) Les feux de chaque voie passent alternativement du vert au rouge, et chaque couleur est maintenue pendant un délai fini,
3) À un instant donné, le carrefour ne doit contenir que des véhicules empruntant une même voie.
La fonction de distribution des arrivées des véhicules sur les deux voies est quelconque.

**Question**: modélisez ce système et écrire les algorithmes des processus correspondant.
On utilisera les outils de synchronisation: Sémaphores, Moniteurs, Regions Critiques, Modules de Contrôle et Expressions de Chemins.

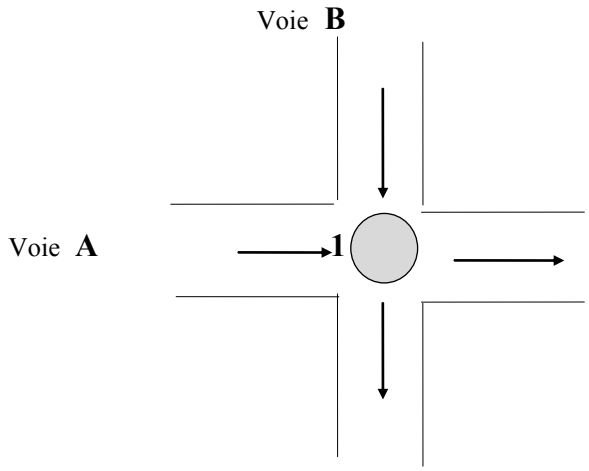



Figure 7. Interssection des voies

(1) : Intersection des deux voies
Objet à accès exclusif

### 18.1 Solution à l'aide des Sémaphores

On peut modéliser ce système comme suit:
- Une ressource critique représentée par le carrefour et celle-ci est allouée en exclusivité aux véhicules représentant des processus. Ces processus se divisent en deux catégories chacune correspond aux véhicules empruntant une voie. Deux procédures, *voieA* et *voieB*,

sont donc nécessaires pour activer ces processus. Le changement des feux est commandé par un processus appelé *Changefeux*.

Deux cas sont à considérer:

a) À un instant donné, un véhicule au plus emprunte le carrefour,

b) N véhicules au plus et à la fois empruntent le carrefour.

Le cas b) ressemble au problème des lecteurs rédacteurs précédemment étudié; considérons alors le cas a).

Pour garantir un fonctionnement correct du système, le processus *change feux* doit modifier les feux selon les impératifs suivants:

- Pour respecter la condition 2 précédente, les nouveaux véhicules qui arrivent sur une voie et trouvant le feu au rouge doivent se bloquer,
- Pour respecter la condition 3, les véhicules qui arrivent au carrefour par une $voie_i$ doivent attendre que ceux qui sont engagés sur la $voie_{3-i}$ aient quitté le carrefour.

```
Var   semaphore mutexA   init 1;          (1)
                mutexB   init 1;
                feuA     init 0;          %  Règle la voieA   %
                feuB     init 1;          %  Règle la voieB   %
 procedure Changefeux
  var switch : boolean init true;     (2)
  begin
   cycle
     timer(delai);          % Attendre un certain temps délai avant de changer les feux %
     if switch then  begin          (3)
                     P(feuB);          (4)
                     V(feuA);
                     switch := false;
                    end
     else   begin
              P(feuA);     (5)
              V(feuB);
              switch := true;
            end
     endif
   fincycle
  end
 procedure VoieA
 begin
   P(mutexA);          % n'autoriser qu'un seul véhicule à la fois dans le carrefour %
   P(feuA);            %  Si le feu est au vert alors franchir, sinon attendre        %
  < passage sur carrefour >;
   V(feuA);                %  une fois passé, libérer l'accès au carrefour  au  profit %
```

```
   V(mutexA);                % du véhicule qui suit juste derrière dans la file A %
end
procedure VoieB
begin
  P(mutexB);              % même chose que précédemment mais pour les véhicules %
  P(feuB);                % de la voie B %
   < passage sur carrefour >;
  V(feuB);                % réveil éventuel du véhicule suivant %
  V(mutexB);
end
```

(1) : Les sémaphores MutexA et MutexB ont été initialisés à 1 pour évter aux véhicules de chaque voie de se coaliser pour occuper le carrefour. L'accès à ce dernier doit se faire un par un.

(2) : La variable switch permet d'associer le caractère actif ou attente à une voie. Lorsque l'interruption de fin de delai (délai) arrive, en fonction de la valeur de switch, la voie qui était active devient en attente (le feu passe du vert au rouge) et celle qui était en attente devient active (passage de son feu du rouge au vert).

(3) : Le changement des feux, donc blocage de la voie active et réveil de la voie en attente, se fait selon l'état des voies antérieur à la fin de délai . Initialement, switch a la valeur vrai et feuA initialisé à 0, ce qui correspond à la priorité accordée à la voie B. Les véhicules de la voieA sont en attente car le sémaphore feuA est privée (valeur nulle). Lorsque l'interruption de fin de délai arrive, switch est à vrai, on fait basculer les feux de la voieB (vert rouge) et ceux de la voieA (rouge vert). Switch prend la valeur faux pour indiquer que c'est la voieA qui est active. Il prendra la valeur vrai pour signifier que la voieB est active.

(4) : P(feuB) signifie: S'il existe un véhicule *Vi* de la voieB dans le carrefour, alors se bloquer en attendant qui'il quitte le carrefour, puis réveiller, par V(feuA), les véhicules en attente sur la voieA. Si le carrefour et libre alors bloquer les véhicules de la voieB et en réveiller ceux de la voieA.

(5) : C'est l'inverse de ce qui se passe en (4)

*Remarque*: Le sémaphore feuA étant initialisé à 0 pour initialement avantager les véhicules venant de la voieB. Le choix de privilégier cette voie n'est que arbitraire, on aurait pu agir autrement en initialisant feuB à 0 et alors, ce sera les vehicules empruntant la voie A qui bénéficieraient du privilège. Ce dernier est accordé seulement à l'initialisation, après le passage est réglé par le temporisateur changefeux en fonction de *délai.*

# Bibliographie


1. André F. et al. "Synchronisation de programmes parallèles", Dunod,1983.
2. BenAri M. " Processus concurrents, introduction à la programmation parallèle", ed. masson, 1986.
3. BenAri M. `` Principles of Concurrent and Distributed Programming , Prentice Hall 1989
4. Brinch hansen " Operating system principles " ed. Prentice Hall, 1973.
5. Crocus " Systèmes d'exploitation des ordinateurs"2° ed. dunod 77.
6. Dijkstra E.W." The structure of the THE multiprogramming system" cacm, 11,5, 1968.
7. Dijkstra E.W. " A discipline of programming " Prentice Hall, 1986.
8. Habermann A.N., R.H. Campbell "The specification of process synchronization by path expressions " coll. sur les aspects théoriques et pratiques des systèms d'exploitation, IRIA, Paris 1974.
9. Hoare C.A.R. "Monitors: an operating system structuring concept, cacm 17,10, 1974.
10. Howard J.H. " Proving monitors, cacm 19,5, 1976.
11. Kessels J.L.W. "An alternative to event queues for synchronization in monitors cacm, 19,5, 1977.
12. Krakowiak S. " principes des systèmes d'exploitation"2° ed. dunod 1987.
13. Lhermitte CI. " Les systèmes d'exploitation: structure et concepts fondamentaux, ed. masson 1985.
14. Madnick S.E, J.J Donovan" Operating systems principles ", ed. Masson, 74.
15. Mossière J. " Méthodes pour l'écriture des systèmes d'exploitation " thèse d'état INP Grenoble 1977.
16. Raynal M. " Une analyse de la spécification de la coopération entre processus par variables partagées". TSI, 1,3, 1982.
17. Robert P., Verjus P. " Towards autonomous description of synchronization modules"; proc. IFIP Congress aug. 1977.
18. Shaw A.C. " The logical design of operating systems " ed. Prentice hall 1974.
19. Silberschatz A. Et al. `` Oparating System concepts , 3rd ed. Addison-Wesley, 1992.
20. Tanenbaum A.S. " Operating systems: Design and Implementation" prentice hall 1987.
21. Tsichritzis D.C., Bernstein P.A." Operating systems " ed. Computer science and applied Mathematics, 1974.
22. William Stallings, Operating Systems: Internals and Design Principles, Sixth Edition Prentice Hall 2008
23. Stallings, William.Operating systems: internals and design principles 9th edition 2018

24. Andrew S. Tanenbaum, Albert S. Woodhull, Operating Systems: Design and Implementation,3rd edition 2006
25. Andrew S. Tanenbaum, Herbert BOS,Modern Operating Systems, Fifth Edition 2023
26. Chakraborty Pranabananda, Operating systems: Evolutionary concepts and modern design principles, First edition, [2024]