UNIVERSITE A.MIRA-BEJAIA
FACULTE DES SCIENCES EXACTES
DEPARTEMENT INFORMATIQUE

SUPPORT DE COURS
PROGRAMMATION SYSTEME
SYNCHRONISATION DE PROCESSUS

I. Notions de coopération, de compétition et de parallélisme.

1. Traitement du parallélisme :

Pour des activités qui se déroulent en parallèle (physiques ou logiques) deux types de relations existent :
- des relations d'ordre conflictuels (compétition) : lorsqu'elles partagent des ressources,
- des relations de coopération : lorsqu'elles participent à un même traitement global.

Les problèmes génériques de conflit ou de coopération existent et leurs solutions relèvent du domaine de la synchronisation. La résolution de ces problèmes de synchronisation peut utiliser différents concepts ou mécanismes qui définissent donc des modèles distincts de machines abstraites logiques et parallèles. Tous ces modèles mettent en jeu le concept fondamental de processus comme outil d'abstraction et de synchronisation et de structuration des traitements parallèles. Un processus modélise un traitement séquentiel.

Deux classes de modèles peuvent être distinguées :
- modèles centralisés : la machine à processus possède une structure avec une mémoire partagée. La synchronisation est implémentée par les sémaphores, les moniteurs, etc.
- modèles répartis : la machine à processus possède une structure décentralisée ou les processus ne peuvent communiquer que par transfert de messages.

2. Traitement syntaxique des processus :

La description explicite du parallélisme peut s'exprimer sous différentes formes syntaxiques dans un programme. On distingue trois approches :
i) primitive prédéfinie provoquant la création de processus, exemple : fork( ).
ii) introduction d'instructions parallèles dans le langage de programmation, exemple : cobegin/ coend du Concurrent Pascal.
Cobegin <bloc1> ||<bloc2>||…||<blocn> Coend → n processus
Le processus appelant attend la terminaison des processus fils avant de reprendre son exécution.
iii) introduction du type PROCESS doté d'un jeu de primitives permettant de manipuler les objets processus (exemple : Modula-2).

```
<déclaration de processus>=
        PROCESS <identificateur> [" (" <paramètres formels> ")"]" ;"
                {<déclarations locales au processus>}
        begin
                {<code du processus>}
        end <identificateur> " ;"
```

La seule primitive indispensable permettant la création de processus est :
START <identificateur de processus> ["("<arguments>")"] ;

3. Les processus communiquant :
Le seul type d'interaction possible est la communication : échange de messages. Donc l'architecture est de nature distribuée, les processus sont implantés sur différents processeurs et ces processeurs sont reliés par un réseau de communication.
Les modèles de cette classe sont caractérisés par des protocoles de communication : deux classes sont utilisées :
- les protocoles asynchrones : lorsqu'un processus envoie un message à un autre processus, il continue son exécution indépendamment de la réception du message.
- les protocoles synchrones : dans ce cas l'émetteur du message bloque jusqu'à l'arrivée d'un accusé de réception.
En général le protocole asynchrone implique une file d'attente de messages envoyés non encore reçus.

4. Programmation parallèle (concurrente) :
La programmation concurrente permet d'exploiter le travail an parallèle des processeurs et des périphériques. Les langages comme Portal, Modula-2 et ADA permettent d'écrire entièrement un programme concurrent ou en temps réel.

II. Synchronisation et communication :
1. Les modèles de problèmes :
1.1 Section critique : Une séquence d'instructions manipulant des données partagées par plusieurs processus est appelée section critique car l'entrelacement des exécutions de ces instructions par différent processus peut conduire à des incohérences. La solution est que l'utilisation de cette section critique par un processus doit exclure toues les autres processus : exclusion mutuelle.

1.2 Allocation de ressources : Dans une application parallèle (ou distribuée) les ressources mises à la disposition des processus sont en nombre limité. Les stratégies d'allocation de ressources sont :
- stratégie d'ordonnancement de requêtes
- stratégie de priorité et optimisation de nombre de requêtes satisfaites.
L'objectif est une utilisation optimale des ressources et une distribution équitable des ressources entre les processus demandeurs.
Deux solutions peuvent être envisagées :
- lors d'une requête de N ressources, le nombre de ressources est suffisant et la requête est satisfaite, alors que d'autres requêtes pour un nombre plus élevé de ressources restent en attente (risque de famine).
- lors d'une restitution de M ressources plusieurs processus demandeurs peuvent être servis.
Trois stratégies de solution peuvent être suivis :
- FIFO : les requêtes sont satisfaites suivant l'ordre d'arrivée
- Priorité aux petit-demandeurs : avec risque de famine des gros demandeurs.
- Ascenseur : les requêtes sont satisfaites selon le nombre croissant des ressources demandées. A chaque étape (étage) les requêtes en attente sont satisfaites si le nombre de ressources est suffisant. Si aucune requête n'est en attente on passe à l'étape (étage) suivant.
L'allocation de ressources peut aboutir à un problème d'interblocage.

1.3 Modèle Producteur-Consommateur : C'est le modèle de coopération de processus, cette coopération se traduit par une communication de messages de processus producteurs vers des processus consommateurs.

L'échange de messages est géré par un système de communication. Généralement un producteur crée un message et demande au système de communication que le message soit transmis à un consommateur quelconque ou bien précis.
- dans la communication par boite aux lettres d'Unix, les messages sont typés.
- le message peut être diffusé à un ensemble donné de consommateurs ou uniquement à ceux en attente au moment du dépôt.
- le langage ADA réalise la communication entre deux taches par un rendez-vous entre ces taches.

1.4 Modèle Lecteur-Rédacteur : Quand des processus partagent une structure de données (un fichier par exemple) qui peut être à un instant donné accédé par un processus lecteur qui ne fait que consulter ou par un processus rédacteur qui modifie. Les deux types de processus doivent respecter les contraintes suivantes :
- la lecture est simultanée : plusieurs lecteurs peuvent lire en même temps
- l'écriture est en exclusion mutuelle : un seul processus rédacteur à la fois peut accéder au fichier.
- Lorsque le fichier est libre, les lecteurs et rédacteurs suivent une stratégie de priorité :
variante1 : Priorité aux lecteurs tant que la ressource n'est pas occupé par un rédacteur.
variante2 : Priorité aux rédacteurs où une demande de lecture est acceptée si aucun rédacteur n'est en cours d'écriture ou en attente d'accès.
variante3 : FIFO mais en autorisant la lecture simultanée.
varainte4 : Priorité aux lecteurs sans famine de rédacteurs où une demande de lecture est acceptée si la ressource est accédée en lecture et aucun rédacteur n'est en attente. A la libération de la ressource par un rédacteur, tous les lecteurs sont réveillés.

1.5 Modèle Client-Serveur : le modèle s'applique à toute activité distribuée (répartie). L'activité parallèle se structure en 2 processus : des processus systèmes qui offrent et exécutent des services et des processus utilisateurs qui émettent des requêtes de services. La relation entre ces processus est une relation de client à serveur.
Remarque : un processus serveur peut être lui-même client.
Le protocole client serveur est implémenté par l'appel procédural à distance (RPC : remote procedure call). L'implantation des RPC est difficile car le serveur n'est pas sur le même site que le client. Le passage des paramètres de l'appel est par valeur car toute transmission d'adresse est exclue.
Le problème de fiabilité du réseau de communication oblige des sémantiques du protocole de communication. Pour être sure que la requête est bien reçue le client émet la requête à chaque délai de garde. Le serveur qui reçoit plusieurs versions d'une même requête peut envoyer plusieurs réponses. Certains protocoles assurent que le service a été exécuté au moins une fois, et d'autres assurent que le service a été exécuté une seule fois au plus.

2. Les outils de synchronisation :
2.1 Masquage des interruptions : Pour éviter l'entrelacement de l'exécution des instructions dans un environnement mono-processeur, il suffit de masquer les interruptions. Le risque est de perdre des informations véhiculées par ces interruptions. La solution est de masquer les interruptions susceptibles de conduire des incohérences dans les données. UNIX utilise cette technique pour assurer la cohérence des tables qu'il administre.

2.2 Test-And-Set (TAS) : Dans un environnement multi-processeur, masquer les interruptions est insuffisant. Pour réaliser l'exclusion mutuelle une opération qui permettrait une lecture et une écriture sur une variable d'une manière indivisible est nécessaire.
Certains processeurs possèdent une instruction atomique TAS qui effectue la lecture d'un mot mémoire et écriture en k cycles en affectant le registre d'état. La valeur à écrire est souvent

prédéfini. Cette instruction permet d'implanter un verrou au niveau Assembleur. D'autres processeurs possèdent une instruction d'échange atomique entre le contenu d'un registre et un mot mémoire tout en mettant à jour le registre d'état du processeur : XCH (exchange).

2.3 Les verrous : un verrou est une variable de type enregistrement :

```
type verrou=enregistrement {
      état : (ouvert, fermé) ;
      file : liste de processus en attente ;
      }
```

Le verrou est manipulé par deux opérations atomiques (primitives) :

```
var V : verrou ;
```

`Verrouiller(V){`
` if (V.état== ouvert) V.état=fermé ;`
` else enfiler(V.file) ;`
`}`
	`Deverrouiller(V){`
	`  if (¬filevide(V.file))`
	`  P=defiler(V.file) ;`
	`  else V.état=ouvert;`
	`}`

2.4 Les événements : Pendant l'exécution d'une application parallèle, différents événements peuvent se produire : expiration d'un délai, libération d'une ressource, envoi/réception d'un message. Certains systèmes possèdent le type événement et en général il est privé à un processeur. Un événement est une variable de type enregistrement :

```
type événement : enregistrement {
      état : (déclenché non-déclenché) ;
      file : liste de processus en attente ;
      }
```

L'événement est manipulé par 3 primitives :
- attendre( ) : bloque automatiquement les processus l'exécutant ;
- déclencher( ) : débloque un processus en attente de l'événement ;
- effacer( ) : permet à un processus de marquer un événement comme non-déclenché, si l'opération déclencher( ) ne le fait pas automatiquement (réinitialiser).

Exemple1 : un événement particulier utilisé dans les applications temps réels est l'événement « expiration de délai » (time-out). Deux primitives sont associées à cet événement :
- activer(N) : l'événement est déclenché après N unités de temps
- annuler( ) : pour annuler la dernière demande de délai du processus.

Exemple2 : UNIX (Linux) offre des primitives qui utilisent des signaux :
- pause() : bloque un processus jusqu'a réception d'un signal
- kill(processus-id, signal-id) : envoie un signal à un processus particulier
- alarm(n) : le signal SIGALARM est envoyé à un processus après n secondes.

Exemple3 : le noyau 68K KERNEL (systèmes temps réel) offre des primitives pour synchroniser des taches sur des événements. Chaque tache possède 16 événements (0..15).
-SignalEvent(Num-T :tache-id, Num-Ev :événement) ;
-WaitEvents(attendus : array[15] of Boolean, délai :integer, Var arrivés: array[15] of Boolean, Var result: (OK, time-out));
- EventsOccured(attendus: array[15] of Boolean): Boolean;
-ClearEvents(evts: array[15] of Boolean);

2.5 Les sémaphores: C'est une variable de type enregistrement :

```
type sémaphore : enregistrement {
      e : entier
      file : liste de processus en attente
      }
```

Le sémaphore est manipulé par deux primitives :

```
var S : semaphore ;
```

`Procedure P(S){`
`S.e=S.e-1;`
`if (S.e<0) enfiler(S.file);`
`}`
	`Procedure V(S){`
	`S.e=S.e+1;`
	`if (S.e≤0) defiler(S.file);`
	`}`

Généralement ils fonctionnent comme des distributeurs de tickets. Un processus demande un ticket en invoquant P(S) et s'il existe au moins un ticket (S.e>0) le processus le prends et continue son exécution sinon il est mis en attente dans une file d'attente associé au sémaphore (S.file).
Un processus qui termine rend le ticket en invoquant V(S) et si la file d'attente n'est pas vide (S.e≤0), il réveille un processus en attente.
Pour résoudre le problème de la section critique un sémaphore initialisé à 1, généralement appelé mutex, est utilisé.

Variantes de sémaphore :
- Sémaphore privé à un processus : seul ce processus effectue P (donc au plus un seul processus en attente) et V est effectué par tous les autres processus y compris le propriétaire.
- P immédiat : le processus qui demande un ticket ne sera pas bloqué dans le file d'attente s'il n y'a pas de ticket, mais recevra un message d'échec et continue son exécution.
- P temporisé : le processus qui exécute P précise le délai maximum d'attente et se ce délai expire il sera réveillé et recevra un message d'échec.
- P avec priorité : les processus en attente seront enfilés selon leur priorités.
- Sémaphore binaire : l'entier S.e ne peut prendre que deux valeurs 0 ou 1 donc pas de file d'attente.
- P à plusieurs tickets: le processus peut demander n tickets en exécutant P (n est un argument de P) et libérer m tickets en exécutant V.

2.6 Le moniteur de Hoare (1974) :
Un moniteur décrit les règles de synchronisation nécessaires à une application parallèle donnée dans une unité syntaxique et sémantique unique. Par analogie, une classe en POO permet de construire une abstraction de données, et un moniteur permet de définir une abstraction de synchronisation. Le moniteur encapsule des données partagées et définit des méthodes d'accès à ces données. Le moniteur assure un accès en exclusion mutuelle. C'est le compilateur du langage de programmation parallèle qui assure l'exclusion mutuelle et quand un processus est en attente du moniteur il est ajouté à la mémoire tampon alloué au moniteur par le système.

Syntaxe : Une seule interface mais plusieurs implémentations :

```
monitor <nom-interface> defines
      <constantes, types et procédures exportés> ;
end <nom-interface> ;
monitor <nom-implémentation> implements
      <Implantation du moniteur> ;
end <non-implémentation>.
```

où d'une manier condensé (une seule implémentation) :

```
monitor <nom-moniteur> ;
      export <liste des identificateurs> ;
      <déclarations de constantes, types et variables>
      {procédures : méthodes d'accès}
end <nom-moniteur>.
```

Une procédure du moniteur peut éventuellement appeler une autre procédure du même moniteur. Ce genre de moniteur garantit l'exclusion mutuelle mais pas la synchronisation.

Les variables condition de Hoare : ce sont des classes d'objets de synchronisation dans un moniteur. Deux opérations complémentaires s'appliquent aux variables condition :

var c : variable condition ;
wait (c) : bloque toujours le processus exécutant cette opération et est enfilé dans une file d'attente associée à la variable condition.
signal(c) : débloque un processus de la file d'attente de c (si elle n'est pas vide).

Lors de l'exécution de signal(c) deux processus sont actifs à l'intérieur du moniteur : signaleur et le signalé. La règle de Hoare dit que le processus signaleur est suspendu donnant l'accès au signaleur (prioritaire), mais pour éviter la famine du signaleur, il est prioritaire aux autres processus demandant l'accès au moniteur.

Exemple :

```
monitor mutex ;
      export P, V ;
      var libre : boolean ;
            accès : condition ;
      procedure P ;
      begin
            if ¬libre then wait
      (accès) endif
            libre:=False;
      end P;
      procedure V;
      begin
            libre:=True;
            signal(accès);
      end V ;
begin
      libre :=true ;
end mutex.
```

Utilisation:

```
Processus
begin
      Section d'entrée
      mutex.P;
      Section critique ;
       mutex.V ;
       Section restante ;
end
```

Les variables condition de Kessel: A toute variable condition est associée une expression logique lors de sa déclaration. Une seule opération est définit : wait(c) qui bloque le processus si l'expression logique est fausse. Le réveil du processus est implicite (quand c devient vrai) et entièrement gérée par le mécanisme de variable condition.

Exemple :

```
monitor mutex ;
 export P, V ;
 var libre : boolean ;
     accès : condition (libre) ;
  procedure P ;
  begin
        wait (accès) ;
        libre :=false ;
  end P ;
  procedure V ;
  begin
        libre :=true ;
  end V ;
begin
    libre:=true
end mutex.
```

3. Les mécanismes de communication :

3.1 Les tubes de communication (Pipe) : Un tube est un tampon de données (fichier) dont le quel deux processus peuvent successivement lire et écrire en exclusion mutuelle.
exemple : pipe( ) de Linux(Unix).

3.2 boite aux lettres : c'est une variable sur laquelle sont définies deux opérations indivisibles :
- send(id-boite, message) : dépose un message dans la variable id-boite
- receive(id-boite) :message : récupère un message enfilés dans la boite.

Un processus qui exécute receive est mis en attente si la boite est vide ; il est débloqué par l'exécution d'un send par un autre processus.
Un processus qui exécute un send est aussi mis en attente si la boite est remplie ; il est débloqué par l'exécution d'un receive par un autre processus.
L'outil est surtout utilisé pour la synchronisation par les données (modèle producteur / consommateur).

3.3 Sémaphore avec messages : Un processus qui exécute un V transmet un message au processus défilé. Les deux primitives P et V sont réécrites Pm et Vm et une file d'attente de message S.fm est associée au sémaphore S en plus de la file des processus et l'entier S.e
Un Vm est accompagné d'un dépôt de message dans fm et un Pm est complété d'un retrait d'un message de la file.
Pm(S, message-reçu) : le message est mis dans la variable message-reçu.
Vm(S, message) : le message est enfilé dans fm.

```
var S : semaphore ; mr, mt : message ;
```

`Pm(S, mr) {`
`S.e=S.e-1 ;`
`if (S.e<0) enfiler (S.file);`
`mr=defiler (S.fm);`
`}`
	`Vm(S, mt) {`
	`S.e=S.e+1;`
	`if (S.e ≤0) defiler( S.file);`
	`enfiler (S.fm, mt);`
	`}`

3.4 Communication par rendez-vous (RDV): proposé par Hoare en 1978 pour une synchronisation forte.
Soient P, Q : processus ; e : expression algébrique ; v : variable :
La commande d'envoi : P ! e
La commande de réception : Q ? v
Le processus P exécute l'envoi et la valeur de l'expression est mise dans v pour être reçue par le processus Q qui exécute la réception simultanément.
La commande d'envoi bloque le processus P jusqu'à réception et la commande de réception bloque le processus Q jusqu'à émission : donc un RDV est établi entre P et Q.
Les langages de programmation parallèle CSP, Modula-2 ou ADA proposent ce genre de synchronisation.

3.5 Le modèle ADA : Le langage ADA propose la notion de tache au lieu d processus (task). Chaque service assuré par une tache est un point d'entrée (entry) auquel on associe une file d'attente de taches appelantes. L'ensemble des entrées d'une tache constitue une interface et le traitement réalisé est implémenté par une unité appelé corps (body).
Une entrée d'une tache peut être connectée à une interruption matérielle qui la rend une routine d'interruption.

```
task [type] <nom-tache> [ is
      {entry <nom-entrée> ["(«"déclarations des paramètres formels ")"] ;}
end <nom-tache> ;
```

Les taches s'exécutent en parallèle et utilisent une synchronisation par RDV.

La tache appelante exécute l'appel à une entrée et reste bloqué jusqu'à satisfaction du service.

```
<tache-appelée>.<nom-entrée>["("paramètres effectifs")"] ;
```

La tache appelé indique l'acceptation du RDV et reste bloqué jusqu'à avoir le RDV.

```
accept <nom-entrée>["("paramètres formels")"]
      [do <instructions> ;]
end <nom-entrée> ;
```

Exemple : soit une tache mutex pour réaliser l'exclusion mutuelle, alors la tache appelante va bloquer sur mutex.entrer et la tache appelé va bloquer sur accept entrer jusqu'au RDV.

4. Solutions aux problèmes génériques :

4.1 Exclusion mutuelle : Pour palier au problème de la section critique.

L'exclusion mutuelle est garantie si les 4 conditions suivantes sont vérifiées simultanément :

i) Exclusion : un seul processus en SC à la fois

ii) Pas de blocage : si la SC est libre un processus en attente doit pouvoir y accéder (déroulement)

iii) Attente limitée : un processus en attente d'une SC doit pouvoir y accéder au bout d'un temps fini

iv) Equité : la solution est la même pour tous (pas de privilège).

Les processus ont cette forme avec i leur identifiant :

```
Proc(int i ) {while (TRUE){
….
 Section d'entrée ;
Section critique ;
 Section de sortie ;
….
}}
```

Solution1 : Masquage des interruptions Section d'entrée : Masquer les interruptions Section de sortie : effacer le masque	Inconvénients: i) la solution n'est valable que pour un environnement mono-processeur ii) perte d'informations véhiculées par les interruptions masquées iii) risque de famine

Solution2 : Attente active pour deux processus P0 et P1.

Algorithme1 : Une variable booléenne pour indiquer l'état de la section critique.

```
shared boolean libre=TRUE ;
Proc(int i ) { while (TRUE){
 while (!libre);
 libre:=FALSE;
Section critique;
 libre :=TRUE ;
}}
```

Algorithme2 : Une variable booléenne par processus pour indiquer leur état.

```
shared boolean demande[2] ;
Proc(int i ) {while (TRUE){
demande[i]:=TRUE;
while (demande[1-i]);
```

```
Section critique;
 demande[i]:=FALSE;
}}
```

Algorithme3 : Une variable entière pour indiquer le tour de chaque processus.

```
shared int tour ;
Proc(int i) { while(TRUE) {
 while (tour!=i);
Section critique;
 tour :=1-i ;
}}
```

Question : les trois algorithmes ne garantissent pas l'exclusion mutuelle. Trouver quelle condition n'est pas vérifiée pour chacune des solutions.

Remarque : Les constantes TRUE et FALSE n'existent pas en C.

```
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE (!FALSE)
#endif
```

Solution2 : les sémaphores	**Solution3** : les verrous

`shared semaphore mutex :=1 ;`
`Proc(int i) {while(TRUE){`
`  P(mutex);`
`  Section critique;`
`  V(mutex);`
`}}`
	`shared verrou V :=ouvert ;`
	`Proc(int i) {while (TRUE) {`
	`  verrouiler(V);`
	`  Section critique;`
	`  deverrouiler(V);`
	`}}`

Question : Vérifier que les deux solutions vérifient l'exclusion mutuelle.

4.2 Allocation de ressources multiples :

La solution est que les processus doivent respecter un protocole pour accéder à une ressource.
Les processus demandeurs ont cette forme :

```
Proc( ) {
   allouer (n ressources)
   utiliser (n ressources)
    liberer (n ressources)
}
```

Solution1 : Moniteur

Interface

```
monitor Ressources defines
      const nbr-ress=50 ;
      type Ress=0..nbr-ress ;
      procedure allouer(N :integer) ;
      procedure liberer(M :integer) ;
end Ressources ;
```

L'implémentation est suivant la stratégie :

```
monitor stratégie implements
Ressources ;
    var cpt : integer ;
         Accès : condition ;
    procedure allouer(N :integer) ;
     begin
        while (n>cpt) do wait
(Accès, N) ;
        enddo
       cpt=cpt:=cpt-N;
       signal(Accès);
   end allouer ;
```

```
     procedure liberer(M :integer) ;
        begin
            cpt:=cpt+M;
            signal(Accès);
         end liberer

begin
     cpt:=nbr-ress;
end strategie;
```

Remarque: la file d'attente associée à la variable condition Accès est trié par ordre croissant du nombre N.

Questions :
1) Quelle est la stratégie implémentée ? Justifier
2) pourquoi est-ce un while et non un if ?
3) pourquoi y'a t il un signal dans la procedure allouer ?

Solution 2 : Sémaphores

```
sempahore S :=50 ;
allouer(int N) {
for (i=0 ;i<N;i++)  P(S);
}
```

```
liberer(int M){
for (i=0;i<M; i++) V(S);
}
```

Questions:
1) Quelle est la stratégie implémentée ? Justifier
2) Un interblocage est-il possible ?

4.3 Producteur Consommateur

Producteur :

```
prod( ){ while(true){
    produire(objet)
    mettre(objet, buffer) ;
}}
```

Consommateur :

```
cons( ){while(true){
    objet=retirer( buffer) ;
    consommer(objet) ;
}}
```

Remarque : Le buffer est une ressource critique !

Solution 1 : Sémaphore

```
semaphore mutex=1 ; plein :=0 ;
vide :=n ;
Prod() {while(true){
     produire(objet) ;
      P(vide) ;
      P(mutex) ;
          mettre (objet, buffer) ;
     V(mutex) ;
     V(plein ) ;
}}
```

```
Cons( ) {while(true){
    P(plein) ;
    P(mutex) ;
         objet=retirer( buffer) ;
     V(mutex) ;
     V(vide) ;
     consommer(objet);
}}
```

Question : vérifier que les contraintes du modèle sont respectées.

Solution2 : Moniteur

```
monitor Prod-Cons ;
export mettre(objet),
retirer(objet);
     const N :=50 ;
    var vide, plein : condition ;
       cpt : integer ;
      buffer : array[50] of objets ;

    procedure mettre (objet) ;
    begin
     if cpt=N then wait(plein) endif
        buffer[cpt]=objet;
        cpt:=cpt+1;
    if cpt=1 then signal(vide) endif
    end mettre;
```

```
procedure retirer(objet);
begin
   if cpt=0 then wait(vide) endif
   objet:=buffer[cpt];
   cpt:=cpt-1;
 if cpt=N-1 then signal(plein) endif
  end retirer;

begin
    cpt:=0;
end Prod-Cons;
```

Les processus demandeurs:

Producteur :
```
begin
    produire(objet);
    Prod-Cons.mettre(objet) ;
end
```

Consommateur :
```
begin
    Prod-Cons.retirer(objet) ;
     consommer(objet) ;
end
```

Questions :

1) pourquoi l'opération produire et consommer sont en dehors du moniteur
2) peut-on utiliser une seule variable condition et pourquoi ?
3) vérifier que la solution respecte les contraintes du modèle.

4.4 Modèle lecteur-rédacteur :

Solution1 : Moniteur de Hoare

Interface :

```
monitor Lect-Redact defines
      procedure demander-lecture ;
      procedure terminer-lecture ;
      procedure demander-ecriture ;
      procedure terminer-ecriture ;
end Lect-Redact ;
```

Implémentation : suivant les variantes du modèle

```
monitor  variante implements Lect-
Redact ;
var n-lect :integer ;
      redaction : boolean ;
      lecture, ecriture :
condition ;

procedure demander-lecture ;
begin
   if redaction then wait(lecture)
endif
    n-lect:=n-lect+1;
    signal(lecture);
end demander-lecture ;

procedure terminer-lecture ;
begin
   n-lect :=n-lect-1 ;
   if n-lect =0 then
    signal(ecriture)  endif
end terminer-lecture;
```

```
procedure demander-ecriture;
begin
  if n-lect>0 or ecriture then
     wait(ecriture) endif
   redaction=true;
end demander-ecriture;

procedure  terminer-ecriture;
begin
  redaction=false;
   signal(lecture);
   if n-lect=0 then signal(ecriture)
   endif
end terminer-ecriture;

begin
   redaction=false;
   n-lect:=0;
end variante.
```

Processus utilisateurs :

Lecteur:	Redacteur :
begin variante.demander-lecture; lire( ) ; variante.terminer-lecture ; end	begin variante.demander-ecriture ; ecrire( ) ; variante.terminer-ecriture ; end

Questions :
1) quelle variante est implémentée par cette solution ?
2) vérifier que les contraintes du modèle sont respectées
3) pourquoi y'a-t-il un signal(lecture) avant un signal(ecriture) dans la procedure terminer-ecriture?

Solution2 : Sémaphore

```
semaphore mutex :=1, ecriture=1 ;
int n-lect :=0 ;

Redact( ) {while(true){
   P(ecriture);
   ecrire( );
   V(ecriture);
}}
```

```
Lect( ) {while(true){
   P(mutex)
   n-lect++;
   if (n-lect==1) P(ecriture);
   V(mutex);
   lire( );
   P(mutex);
   n-lect-- ;
   if ( !n-lect) V(ecriture) ;
   V(mutex);
}}
```

Questions :
1) Quelle variante est implémentée par cette solution ?
2) pourquoi l'opération ecrire( ) est entre un P puis V alors que l'opération lire( ) est entre un V puis P ?

4.5 Problème du carrefour :
La circulation à un carrefour à deux voies est réglée par des feux verts ou rouges. Les règles de circulations sont les suivantes :
- quand le feu est rouge sur une voie, les voitures y circulant doivent attendre ; quand le feu est vert les voitures peuvent traverser le carrefour,
- une voiture se présentant au carrefour doit le franchir au bout d'un temps fini,
- les feux de chaque voie passent alternativement du rouge au vert,
- à un instant donné, le carrefour ne doit contenir que des voitures d'une même voie,
- les voitures traversent le carrefour en ligne droite.

Ecrire les programmes correspondant aux processus :
- changement assurant la gestion de la commande des feux,
- traversée1 et traversée2 assurant respectivement la traversée d'une voiture des voies 1 et 2, quand le carrefour peut contenir une seule voiture à la fois, puis au plus k voitures à la fois.

Mme YAICI.