Chapitre 4

Outils de la Programmation Simultanée

1. Introduction

A la base de la programmation simultanée,
il y a la **coopération** entre **processus**,

ce qui engendre deux (02) **problèmes** essentiels :

- la synchronisation
- l'exclusion mutuelle

a) La Synchronisation :

Consiste à imposer un ordre sur l'exécution des instructions des processus.

Exemple : processus Clavier : lit un caractère tapé au clavier
Processus Ecran : affiche sur l'écran, le caractère lu

Le processus Ecran ne peut s'exécuter avant que le processus Clavier ne s'exécute.
Il faut donc bloquer le processus Ecran si celui-ci tente de s'exécuter avant qu'un caractère ne soit disponible :
⟶ ceci s'appelle synchroniser les processus Clavier et Ecran.

b) L'exclusion mutuelle

Peut être vue comme un cas particulier de synchronisation :

Un processus ne peut entrer en section critique qu'après le départ du processus précédent ;

La **section critique** est la partie du code d'un processus dans lequel le processus accède à une ressource critique,

Et une **ressource critique** est un objet qui doit être accédé en exclusion mutuelle.

Exemple : P1 et P2 ;

P1 est un processus qui s'occupe de lire et d'écrire sur un disque,

P2 s'occupe d'écrire sur une imprimante en cas de problème, un message doit être affiché sur l'écran.

Le **Pb** : P1 et P2 affichent au même moment sur l'écran (les caractères des messages risquent de se mélanger sur l'affichage).

Il y a donc exclusion mutuelle sur l'accès à l'écran, l'écran est appelé ressource critique (objet qui doit être accédé en exclusion mutuelle).

Il existe différents **outils** pour gérer ces problèmes

{ de **synchronisation**

et **d'exclusion mutuelle**.

Ressource critique : Ecran

Section critique : envoi du message d'erreur par P1 et P2

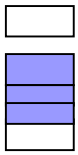
P1:	P2 :
Lire sur disque Ecrire sur disque Si Pb Alors ecrire sur Ecran fsi	Ecrire sur imprimante Si Pb Alors ecrire sur Ecran fsi

2. Les Verrous

Le verrou est l'outil le plus primitif permettant d'implémenter une section critique

Un verrou ***v*** est une variable constituée :

- d'un état (ouvert ou fermé)
- d'une liste de processus (en attente)



Initialement, le verrou est ouvert

La liste des processus est en attente devant un verrou fermé

Un processus ne peut manipuler un verrou que grâce à deux procédures :

- *verrouiller*
- et *déverrouiller*

verrouiller *:* *si l'état est ouvert,*
alors le fermer
sinon bloquer le processus dans la liste d'attente
fsi

déverrouiller *:* *si la liste d'attente n'est pas vide,*
alors débloquer un processus en attente dans la liste
sinon mettre l'état à ouvert.
fsi

Les procédures verrouiller et déverrouiller constituent elles-même des sections critiques, la ressource critique étant le verrou.

En utilisant les verrous, l'accès exclusif à l'Ecran s'exprimera :

*Var **v** : verrou*
Process P1 *:*
Début ….
Verrouiller (v) ;
Ecrire-sur-ecran (« MESSAGE : ERREUR SUR DISQUE ») ;
Déverrouiller (v) ;…..
Fin ;
et P2 s'écrira de façon analogue .

Process P2 *:*
Début ….
Verrouiller (v) ;
Ecrire-sur-ecran (« MESSAGE : ERREUR SUR IMPRIMANTE ») ;
Déverrouiller (v) ;…..
Fin

3) Les sémaphores :

- ont été introduits par Dijkstra,
- sont faciles à implanter
- et permettent de résoudre un grand nombre de problèmes de la programmation simultanée.

Le sémaphore est un puissant outil de la synchronisation

Définition : un sémaphore est une variable entière **s**, qui ne peut prendre que des valeurs positives ou nulles.

Une fois que s est initialisée, les opérations admises sur s sont :
- attendre : wait (s) et
- signaler : signal (s).

et elles sont définies par :

wait(s) : si s>0
alors s := s -1
sinon l'exécution du processus ayant appelé wait(s) est suspendue
fsi

signal(s) :si 1 processus a été suspendu lors d'1 exécution antérieure d'1 wait (s)
alors le réveiller
sinon s := s+1
fsi

* si le sémaphore ne prend que les valeurs 0 ou 1, on l'appelle sémaphore binaire, sinon, il est appelé sémaphore général.

* on ne peut ni tester, ni affecter de valeur à s sauf lors de son initialisation dans le programme principal.

* les procédures wait et signal s'excluent mutuellement. si elles se produisent en même temps, elles sont exécutées l'une après l'autre, sans que l'on connaisse l'ordre d'exécution.

* lorsque plusieurs processus sont suspendus sur un même sémaphore, la définition de signal ne précise pas quel processus est réveillé. Pour cela, il faut choisir un ordre pour choisir le processus qui sera réveillé.

* Siognal et wait sont deux instructions primitives (élémentaires) non divisibles.

Une solution de l'exclusion mutuelle à partir d'un sémaphore serait :

Program exclusion-mutuelle:

```
Var s : semaphore ; (* binaire *)
Procedure P1;
Begin
```

```
        Repeat
                Wait (s);
                Crit1;
                Signal (s);
                Rem1
        Forever
End;

Procedure P2 ;
Begin
        Repeat
                Wait (s);
                Crit2;
                Signal (s);
                Rem2
        Forever
End;
Begin   (* programme principal*)
        S:= 1;
        Cobegin
                P1 ; P2
        coend
End.
```

Expliquons le programme :

Le processus P1 (ou P2) s'exécute et réalise un wait.

S est à 1 donc il entre en section critique (crit1) ou (crit2) sinon suspendu.

L'autre processus ne peut s'exécuter car s = 0

Après avoir terminé sa section critique, signal (s) remet s à 1 et l'autre processus peut alors s'exécuter et entrer dans sa section critique.

L'exclusion mutuelle pour **n** processus est résolue par le même programme

```
Procedure mutualexclusion;

Const n := … ; (* nombre de processus *)
Var s : semaphore ; (*general*)
Procedure Process (i: integer) ;
Begin
        Repeat
                Wait (s);
                Crit;
                Signal (s);
                Rem
        Forever
End;

Begin   (* programme principal*)
        S:= n;
        Cobegin
```

```
            Process (1);
            Process (2);
            …….
            Process (n)
        coend
End.
```

Exemple pratique :
Ordinateur disposant de 2 imprimantes et pouvant donc autoriser 2 programmes à imprimer en même temps.
Le seul changement à apporter alors est d'initialiser le sémaphore à **2**

4) événements

- L'événement est l'outil le plus primitif (élémentaire) permettant de résoudre le problème de synchronisation.

- Un événement est constitué d'un booléen et d'une liste de processus en attente.

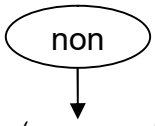
- Un événement peut avoir eu lieu ou ne pas avoir eu lieu : initialement, il n'a pas eu lieu.

- Un événement peut être manipulé grâce aux procédures
  Attendre,
  Déclencher,
  Réinitialiser

- **procédure attendre** :
  si booléen faux (non survenu), (n'a pas eu lieu)
  alors bloquer le processus dans la liste d'attente
  fsi

- procédure déclencher :
  booléen := vrai ; (survenu) et
  débloquer tous les processus en attente

- procédure réinitialiser :



booléen est initialisé à faux. (survenu = false)

En utilisant les événements, la synchronisation des processus Clavier et Ecran s'exprime ainsi :

```
Var
        c: char;
        e: evenement ; (* non survenu initialement *)

Process Clavier;
Var     car_lu: char;
debut
        c:= car_lu; (* lire un caractère dans la variable car_lu *)
        déclencher (e) ;
End;

Process ecran;
Var     car_a_ecrire : char;
```

```
debut
        attendre (e) ;
         car_a_ecrire := c ;
        ecrire le caractère car_a_ecrire sur l'écran
end;
Begin
  Repeat
        reinitialiser
        Cobegin
                Clavier;ecran
        Coend
  Forever
End
```

Le processus **Ecran** est bloqué uniquement s'il exécute *attendre(e*) avant que le processus **Clavier** n'exécute *déclencher (e).*
Une fois bloqué, le processus Ecran est débloqué lorsque le processus clavier exécute *déclencher (e).*
La synchronisation est donc correctement résolue.

Mais *l'événement* n'est pas adapté à la résolution de problèmes plus compliqués.
Il n'est pas possible de l'utiliser pour l'exclusion mutuelle.

5. les moniteurs

Le sémaphore constitue un progrès par rapport aux verrous et aux événements.

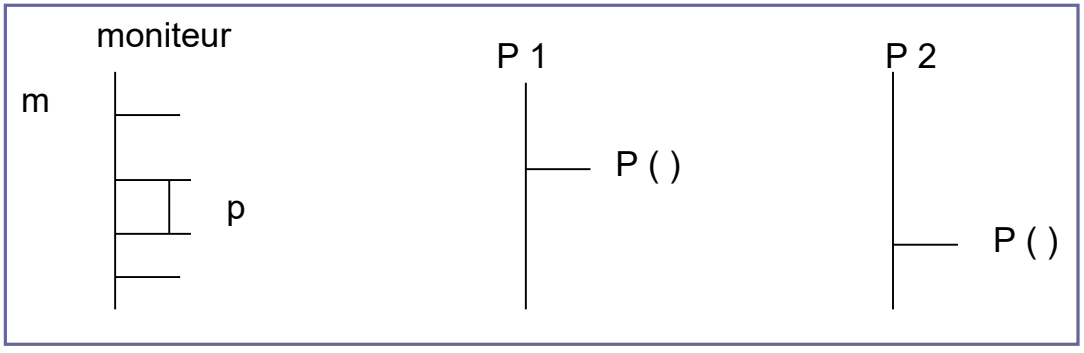
Le moniteur constitue un progrès par rapport aux sémaphores.

Moniteur = unité syntaxique,
assurant l'exclusion mutuelle **+** exprimant synchronisation

Cette notion a été proposée par Hoare en 1974

Moniteur = module permettant de regrouper
- des variables
- ainsi que les procédures agissant sur ces variables

* Le moniteur assure l'exclusion mutuelle sur les procédures déclarées dans le moniteur :
Pendant qu'un processus P1 exécute une procédure P d'un moniteur m, tout appel par un processus P2 de la procédure P ou d'une autre procédure du moniteur m est mis en attente (P2 est momentanément bloqué) l'appel de P2 sera autorisé lorsque P1 quittera le moniteur.



* La synchronisation s'exprime à l'intérieure d'un moniteur grâce à des signaux : un signal, implémenté par une liste, peut être manipulé par les procédures wait et send.

Soit S un signal : l'exécution de **S.wait** bloque inconditionnellement un processus.
Le processus bloqué est dit en attente du signal S. Il est débloqué par l'exécution de **S.send**.
Plusieurs processus peuvent être en attente du même signal : l'exécution de **S.send** n'en débloque qu'un seul le premier qui s'est mis en attente du signal (liste d'attente gérée en queue).
La procédure **send** présente une caractéristique essentielle : lorsqu'un processus se débloque par l'exclusion de **s.send**, l'exclusion mutuelle sur le moniteur est levée.

Exemples de moniteur

a) Rappel (avant)

La programmation simultanée en **Pascal S**
Le pascal séquentiel doit être enrichi des instructions découlant de la programmation simultanée. Les tâches sont écrites comme des procédures Pascal et leur nature est établie lorsqu'elles apparaissent dans le bloc.
Cobegin….coend :

Cobegin P1 ; P2 ; … ; Pn coend

La demande de l'exécution de processus en parallèle ne peut se faire que dans le programme principal et il ne peut y avoir plusieurs demandes imbriquées.

La sémantique du couple d'instructions cobegin…coend est la suivante :

L'instruction cobegin signale au système que les procédures dans le bloc ne sont pas à exécuter, mais doivent être comprises comme des processus. Quand l'instruction coend est atteinte, l'exécution du programme principal s'interrompt et les processus sont réalisés. L'imbrication des exécutions n'est pas prévisibles et peut changer d'un lancement de programme à l'autre. Quand tous les processus sont terminés, l'exécution du programme principal reprend à l'instruction suivant coend.

Nous aurons également besoin de la structure suivante :

Repeat … forever (répéter … à jamais)

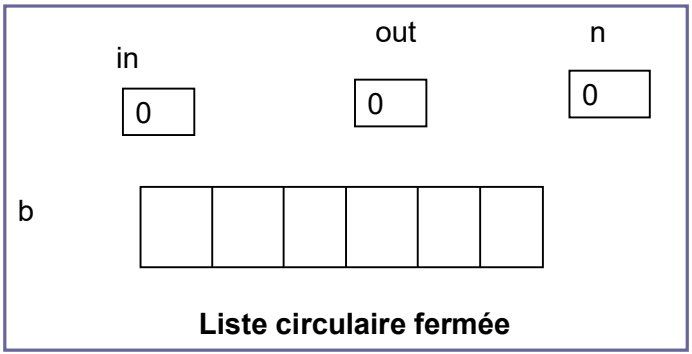
qui est exactement équivalente à

repeat … until (false)

Utiliser **repeat …forever** montre bien que les exemples simulent des programmes cycliques dans un système d'exploitation ou dans un système temps réel.

b) Exemple 1 :

Problème du **producteur-consommateur** utilisant tampon fini



Liste circulaire fermée

```
program producerconsumer ;
const sizeofbuffer=… ; (*taille du tampon*)
monitor boundedbuffer ; (* tampon fini *)
        b: array 0..sizeofbuffer of integer;
        in, out: integer;
        n: integer;
        procedure append (v: integer);
        begin
                if n = sizeofbuffer + 1 (* le tampon est plein *)
                then « attendre qu'il ne soit plus plein  » ;  (1)
                b[in] := v ;
                in := in + 1 ;
                if in = sizeofbuffer + 1 then in := 0 ;
                n:= n+1;
                « signaler que le tampon n'est pas vide »; (2)
        end ;
```

```
procedure take (var v : integer) ;
        begin
                if n = 0 (* le tampon est vide *)
                then « attendre qu'il ne soit plus vide » ; (3)
                v := b [out] ;
                out := out + 1;
                if out = sizeofbuffer + 1 then out := 0;
                n := n- 1;
                « signaler que le tampon n'est pas plein » (4)
        end ;
begin (* corps du moniteur*)
        in := 0 ;
        out := 0 ;
        n := 0 ;
end;  (* fin du moniteur *)
```

```
procedure producer ;
var v : integer ;
```

```
begin
repeat
        produce (v);
        append (v); (*appel procédure moniteur*)
    forever
end ;
```

```
procedure consumer ;
var v : integer ;
begin
    repeat
        take (v); (*appel procédure moniteur*)
        consume (v);
    forever
end ;
begin (*programme principal*)
    cobegin
        producer; consumer
    coend
end.
```

Un moniteur est un ensemble
- de déclarations de variables (globales)
- et de procédures pouvant être paramétrisées.

Le corps (begin …end) du moniteur est exécuté dès que le programme est lancé pour initialiser les variables moniteur.
Ensuite le moniteur apparaît comme un progiciel,
- ensemble de données
- et de procédures.

Les variables moniteur ne sont accessibles qu'à travers les procédures moniteur. Les communications avec l'extérieur se font par l'intermédiaire des paramètres des procédures.
La seule manière pour un processus d'accéder à une variable moniteur est d'appeler une procédure moniteur.

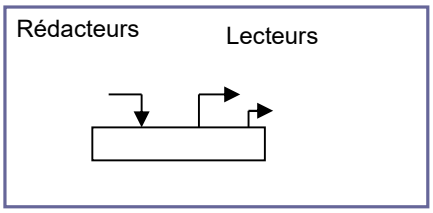
Pour synchroniser les processus on a besoin de commandes du type wait et signal.
On peut effectuer deux opérations sur une variable condition c :
wait (c) et signal (c)

Remarque :
Pour compléter l'exemple du tampon fini,
- nous devons déclarer deux conditions dans la déclaration des variables moniteur :
  Nonvide, nonplein : condition ;
- et remplacer les phrases entre parenthèses par :
  Wait (nonplein) ; (1)
  Signal (nonvide) ; (2)
  Wait (nonvide) ; (3)

Signal (nonplein) ; (4)
Respectivement.

c) Exemple 2 :

Le problème des rédacteurs et des lecteurs




```
program readersandwriters ; (* lecteurs et rédacteurs *)
monitor readwrite ; (* lire -  écrire  *)
var       readers : integer ;
          writing : boolean ;
          oktoread, oktowrite : condition;
procedure startread;  (* debut lecture *)
begin
          if writing or noempty (oktowrite)
          then  wait (oktoread) ; (*attente autorisation de lire *)
          readers := readers + 1 ;
          signal (oktoread) ( * réveil en cascade *)
end ;
```

```
procedure endread;  (* fin de lecture *)
begin
          readers := readers – 1 ;
          if readers = 0  then  signal (oktowrite )
end ;
procedure startwrite;  (* début d'écriture  *)
begin
          if readers <> 0 or writing
          then  wait (oktowrite ); (*attente autorisation d'écriture*)
          writing  := true
end ;
```

```
procedure endwrite  (* fin d'écriture *)
begin
          writing := false ;
          if noempty (oktoread) (*lecteurs en attente?*)
           then  signal (oktoread ) (* oui : autorisation de lire *)
          else signal (oktowrite) (* sinon : autorisation d'écrire *)
end ;
begin (* moniteur*)
```

```
        readers  := 0 ; (* pas de lecteurs *)
        writing  := false (* pas de rédacteurs *)
end; (* fin du moniteur *)
procedure readprocess ; (* processus lire *)
begin
        repeat
                startread;
                readthedata ;  (* opération de lecture *)
                endread
        forever
end;
```

```
procedure writeprocess ; (* processus écriture *)
begin
        repeat
                startwrite;
                writethedata ;  (*opération d'écriture *)
                endwrite
        forever
end ;
begin (*programme principal*)
        cobegin
                readprocess; readprocess;
                writeprocess; writeprocess;
        coend
end.
```

Nous avons introduit ici une nouvelle primitive pour notre modèle du moniteur: **noempty** (condition) (nonvide), fonction à valeur logique, vraie si seulement si la file d'attente sur condition contient des processus bloqués.