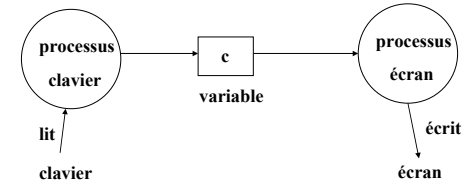


## Coopération entre processus

## Lecture d'un caractère au clavier avec écho à l'écran



## Présentation du problème

coopération =

*communication* + *synchronisation* entre processus ;

- *communication* = transmission de données entre processus;
- *synchronisation* = imposer un ordre sur l'exécution des instructions d'un ensemble de processus.

var c: char; (\* variable globale, pour la communication \*)

```

process clavier;
  var car_lu: char;
  begin
    lire un car depuis clavier dans la
    variable car_lu;
    c := car_lu;
  end clavier;

process écran;
  var car_à_écrire: char;
  begin
    car_à_écrire := c;
    écrire car_à_écrire à l'écran;
  end écran;
  
```

synchronisation: l'instruction I<sub>1</sub> doit être exécutée avant I<sub>2</sub>

## Synchronisation à l'aide d'événements

```
class événement;
survenu: boolean;
en_attente: queue de processus;
```

```
procedure attendre( );
begin
  if not survenu then
    bloquer le processus
    en queue de en_attente;
  end if;
end attendre;
```

```
procedure déclencher( );
begin
  survenu := true;
  débloquent tous les processus
  dans la queue en_attente;
end déclencher;
```

```
procedure réinitialiser( );
begin
  survenu := false;
end réinitialiser;
end événement;
```

## Synchronisation à l'aide de sémaphores

```
var c: char; (* pour la communication *)
s: sémaphore init 0; (* pour la synchronisation *)
```

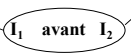
```
process clavier;
var car_lu: char;
begin
  lire un car du clavier dans la
  variable car_lu;
  c := car_lu;
  s.V;
end clavier;
```

```
process écran;
var car_à_écrire: char;
begin
  s.P;
  car_à_écrire := c;
  écrire car_à_écrire à l'écran;
end écran;
```

```
var c: char; (* pour la communication *)
e: événement; (* pour la synchronisation *)
```

```
process clavier;
var car_lu: char;
begin
  lire un car du clavier dans la
  variable car_lu;
  c := car_lu;
  e.déclencher;
end clavier;
```

```
process écran;
var car_à_écrire: char;
begin
  e.attendre;
  car_à_écrire := c;
  écrire car_à_écrire à l'écran;
  (*
end écran;
```



## Limite de l'utilisation des sémaphores

Un schéma de programmation fréquent pour exprimer l'exécution de  $s.P$  si et seulement si une condition  $cond$  est vraie est:

if  $cond$  then  $s.P$  endif

Problème: si  $cond$  est vrai au moment de l'évaluation, mais devient faux avant l'exécution de  $s.P$  !

Idée: on ajoute une section critique.

```
mutex.P;
if cond then s.P endif;
mutex.V;
```

... et on exécute  $mutex.P \dots mutex.V$  pour toute partie du code qui touche les variables qui interviennent dans  $cond$ .

Mauvaise idée: **risque d'interblocage!**

Nouvelle tentative:

```
mutex.P;
if cond then mutex.V; s.P endif;
```

Problème: *cond* peut à nouveau être vrai au moment de l'évaluation, mais devenir faux avant l'exécution de *s.P* !

On aimerait disposer d'un mécanisme qui permette de manière atomique (= indivisible) de réaliser (*mutex.V*; *s.P*).

Les **moniteurs** vont nous fournir (entre autres) un tel mécanisme.

## Exclusion mutuelle

```
monitor m;
defines proc_1, proc_2;
var v1, v2, ...

procedure proc_1(...);
begin
  ... accès aux variables v1, v2, ...
end;

procedure proc_2(...);
begin
  ... accès aux variables v1, v2, ...
end;

end m;
```

## Synchronisation à l'aide de moniteurs

Un **moniteur** est

- une *unité syntaxique* (comme un module);
- dans laquelle les procédures s'exécutent en exclusion mutuelle;
- qui fournit un mécanisme interne de synchronisation (signal).

Proposé par Hoare en 1974.

## Synchronisation

```
monitor m;
defines proc_1, proc_2;
var v1, v2, ...
signal s;

procedure proc_1(...);
begin
  ...
  s.wait;
  ...
end;

end m;
```

Bloque inconditionnellement  
le processus appelant et lève  
l'exclusion mutuelle sur le  
moniteur

```
procedure proc_2(...);
begin
  ...
  s.send;
  ...
end;
```

Le signal est implémenté à l'aide d'une queue de processus.

*s.wait;*

- bloque inconditionnellement le processus en fin de la queue associée au signal s;

*s.send;*

- débloque le processus p en tête de la queue associée au signal s;
- reprend l'exécution du processus p;
- le processus ayant exécuté *s.send* reprend son exécution lorsque p quitte le moniteur;
- permet facilement de faire des preuves: **un signal transporte une condition**.

```
process clavier;
  var car_lu: char;
begin
  lire un car du clavier dans la
  variable car_lu;
  m.déposer_car(car_lu);
end clavier;
```

```
process écran;
  var car_à_écrire: char;
begin
  m.prélever_car(car_à_écrire);
  écrire car_à_écrire à l'écran;
end écran;
```

```
monitor m;
defines proc_1, proc_2;
  var v1, v2, ...
  signal s;
```

```
procedure proc_1(...);
begin
  ...
  s.wait;
  ...
end;

procedure proc_2(...);
begin
  ...
  s.send;
  ...
end;
```

```
monitor m;
defines déposer_car, prélever_car;
var c: char; car_disponible: boolean; (* init false *)
signal s; (* transporte la condition "car_disponible = true" *)
```

```
procedure
  prélever_car(var ch: char);
begin
  if not car_disponible then s.wait
end if;
ch := c;
car_disponible := false;
end;
```

```
procedure
  déposer_car(ch: char);
begin
  c := ch;
  car_disponible := true;
  s.send;
end;
```

On a:

- *car\_disponible* = *true* lorsque *ch:=c* est exécuté dans la procédure **prélever\_char**

En effet:

- cas 1: ...
- cas 2: ...

### Solution exprimée à l'aide d'un module *tampon* (buffer)

```
module tampon;
  defines déposer, prélever;
  ...
end tampon;
```

```
process producteur;
begin
  loop
    produire un message;
    tampon.déposer(message);
  end loop;
end producteur;
```

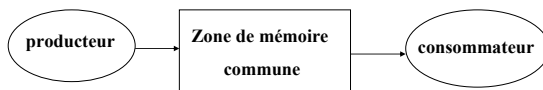
```
process consommateur;
begin
  loop
    tampon.prélever(message);
    utiliser le message;
  end loop;
end consommateur;
```

### Problème du producteur et du consommateur

Généralisation du problème de communication et synchronisation entre les processus clavier et écran

Deux classes de processus:

- *producteur* produit des messages et les dépose dans une zone de mémoire commune;
- *consommateur* prélève les messages et les utilise;
- les messages sont gérés en queue (FIFO).



### Atelier de menuiserie:

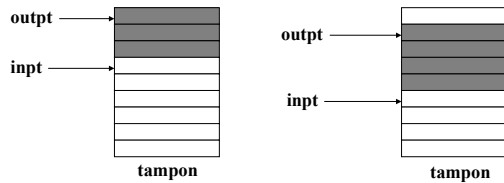
problème de producteur-consommateur entre le menuisier (producteur) et le préposé au tapis roulant (consommateur):

l'étagère joue le rôle du tampon.

## Communication

Hypothèse: tous les messages sont de même taille

- la zone de mémoire commune est implémentée à l'aide d'un *tableau* (partagé):  
un élément du tableau = un message
- les variables *inpt* et *outpt* sont utilisées pour déposer, resp. prélever, le prochain message



## Solution à l'aide d'un moniteur

```
monitor tampon;
defines déposer, prélever;

const n=100; (* taille du tampon *)

var tamp: array 1..n of message;
    nb_mess, inpt, outpt: integer; (* init: 0,1,1 *)
signal nonplein, (* transporte la condition nb_mess < n *)
        nonvide; (* transporte la condition nb_mess > 0 *)

procedure déposer(m: message);

procedure prélever(var m: message);

end tampon;
```

## Synchronisation

Problèmes à résoudre:

- tampon plein: bloquer le producteur qui veut déposer un message;
- tampon vide: bloquer le consommateur qui veut prélever un message;
- producteur bloqué et message prélevé: débloquer le producteur;
- consommateur bloqué et message déposé: débloquer le consommateur;

```
procedure déposer(m: message);
begin
  if nb_mess = n then nonplein.wait end if;
  tamp[inpt] := m;
  inpt := inpt mod n + 1; nb_mess := nb_mess + 1;
  nonvide.send;
end déposer;

procedure prélever(var m: message);
begin
  if nb_mess = 0 then nonvide.wait end if;
  m := tamp[outpt];
  outpt := outpt mod n + 1; nb_mess := nb_mess - 1;
  nonplein.send;
end prélever;
```

## Solution à l'aide de sémaphores

```

module tampon;
defines déposer, prélever;

const n=100; (* taille du tampon*)

var tamp: array 1..n of message;
    inpt, outpt: integer; (* init: 1,1 *)
    nb_mess: sémaphore init 0;
    nb_libre: sémaphore init n;

procedure déposer(m: message);

procedure prélever(var m: message);

end tampon;

```

## Questions:

- laquelle des 2 solutions est-elle correcte dans le cas de plusieurs producteurs ou plusieurs consommateurs?
- comment adapter la solution?

```

procedure déposer(m: message);
begin
    nb_libre.P;
    tamp[inpt] := m;
    inpt := inpt mod n + 1;
    nb_mess.V;
end déposer;

procedure prélever(var m: message);
begin
    nb_mess.P;
    m := tamp[outpt];
    outpt := outpt mod n + 1;
    nb_libre.V;
end prélever;

```

## Synchronisation par les données: la boîte aux lettres

Idée: mécanisme de plus haut niveau que les sémaphores et les moniteurs, qui intègre communication **et** synchronisation.

**Boîte aux lettres**: objet sur lequel sont définies les opérations *envoyer* et *recevoir* :

- var **b**: mailbox of T;
- *envoyer*(**b**, **msg**): envoi non bloquant de msg (sauf si **b** est plein);
- *recevoir*(**b**, **msg**): réception bloquante si **b** est vide.

## Problème du producteur/consommateur:

var **b**: mailbox of **message**;

```

process producteur;          process consommateur;
  var m: message;            var m: message;
  begin                      begin
    loop                    loop
      produire message m;    recevoir(b,m);
      envoyer(b,m);         utiliser m;
    end loop;              end loop;
  end producteur;          end consommateur;

```

processus P

processus Q

**Q ! 10****P ? v**

- Rendez-vous:  
**lorsque P exécute *Q ! 10* et que Q exécute *P ? V***
- la communication (envoi de 10) a lieu lors du rendez-vous (pas de tamponnage).

## Synchronisation forte par les données: le rendez-vous

- proposé par Hoare en 1978;
- deux commandes:

***P ! e*** envoi de la valeur de l'expression *e* au processus *P* ;

***P ? v*** réception d'une valeur envoyée par le processus *P* ;  
la valeur reçue est stockée dans la variable *v* .

- les deux commandes sont bloquantes.

## Problème du producteur/consommateur:

```

process producteur;          process consommateur;
  var m: message;            var m: message;
  begin                      begin
    loop                    loop
      produire message m;    producteur ? m;
      consommateur ! m;     utiliser m;
    end loop;              end loop;
  end producteur;          end consommateur;

```

NB: synchronisation forte entre producteur et consommateur: pas de tamponnage!