

CM70 Les Moniteurs

Lundi 29.11.2021

Sémaphores

- **wait(S)** : $s > s - 1$ — sinon on se bloque
- **signal(S)** : $s + 1$ OU débloquer un processus en attente
- *Maintenant, supposons qu'on veut les implémenter. On doit utiliser quelque chose qui va être de plus bas niveau qui va assurer l'exclusion mutuelle car ces primitives sont supposer atomique.*
- *Pour implémenter ça faut utiliser au plus bas niveau, quelque chose qui permet un accès atomique a une certaine variable.*
- *On a besoin de primitives de type **T & S = Test and Set***
- **T & S** d'une variable Boolean « lock », ça revient à faire :

Atomique {
Test and Set(lock) {
 v := lock
 lock := true
 return v
}
}

Offert par le matériel.

- Si la valeur est 0, c'est bon, on a pris le lock et la valeur passe de 0 à 1, le prochain qui va tester va voir qu'il obtient une valeur de 1.
- Donc : c'est prendre la valeur et en même temps la modifier : si elle est 0, elle passe à 1. Si elle est à 1 : on attend.
- Avec ça on peut faire de l'attente active : attendre que le lock en paramètre devient 0.
- Du coup, on aura un lock sur notre sémaphore S, faut s'assurer que cela ce passe de manière atomique (Test and Set).
- Dans certains langages : on a une notion qui permet à une certaine section de s'exécuter de manière Oatomique. Alors cela passe à la charge du compilateur...

Exemple d'implémentation

Au début : $l = \text{faux}$ // le lock est libre (pas pris)

Wait(S)

```

done := false
do
  // On attend le lock pour accéder à S : On attend que le retour de T&S sera false (l = false)
  if (¬ T&S(l) ) then // Quand on fait T&S on avait 0, maintenant il devient 1.
    if ( S > 0) then // On termine la boucle, on cherche plus
      done := true
      S--
      l := false // Je lâche le lock pour que d'autres accèdes à S
while(¬ done)

```

- Ici la queue est déjà implémenté ; Attente active que S devienne positif.
- On va attendre qu'on aura le droit de décrémenter S.

*L'implémentation de **signal(S)** est plus simple :*

Signal(S) // Faut qu'il incrémente le S

// Dit qu'il y a la possibilité d'incrémenter le S, il lâche le lock et s'en va

```
while(T&S(l)) do {} // Tant que T&S réussie : j'attends  
S++ ;  
l := false //Il lâche le lock
```

Exclusion mutuelle

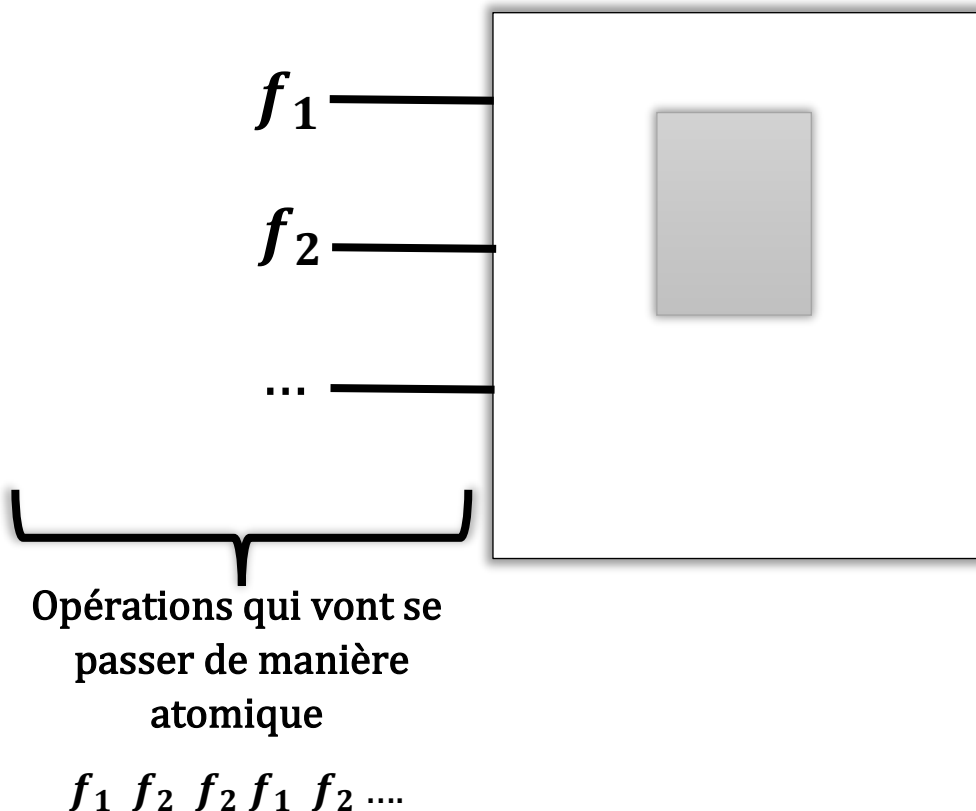
- Attente active
- Attendre que quelqu'un nous réveille
- *L'Attente active a du sens lorsque on a plusieurs processus. Sinon ça a pas de sens.*
- *D'une certaine manière elle peut couter moins cher que le blocage d'un thread.*

Cela est vrai au bas niveau.

- *Mais, au niveau logique , on a plutôt envie des mécanismes qui nous permette de pas se tromper.*
- *Le sémaphore : un mécanisme de bas niveau, difficile à maîtriser.*
- *Pour cela, il existe un autre concept, de plus haut niveau :*
les moniteurs.

Les moniteurs

- Disons qu'on a un certain nombre de structures qu'on veut manipuler, si ces structures sont partagées, faudra régler tous les problèmes de synchronisation sur cette mémoire partagée.
- Par exemple, le problème du Producteur-Consommateur avec les actions **append()** et **take()**.
- Une façon simple est de programmer de manière séquentielle et faire un **lock** sur la structure entière avant de faire une certaine opération.
- Le problème est que cette méthode prend beaucoup de temps...
- Ce que nous voulons : 2 opérations qui s'exécute de manière atomique.
- C'est pour cette raison que les langages de programmation propose des différentes structures (piles, files, etc.) qui permette de réduire la latence (l'attente) due à la synchronisation.
- A notre niveau, on fait de l'abstraction.
- Les moniteurs (qui est un vieux concept qui est un peu comme la POO) : c'est ce que nous donne cette notion d'atomicité élevée.



- C'est bien, mais ça ne suffit pas car l'exécution d'une certaine fonction a besoin de se synchroniser avec les autres exécutions.
- Donc, il y a un mécanisme qui ressemble à **wait()** et **signal()** : **blocage / réveille**.
- C'est quelque chose de plus abstrait : on va se bloquer sur une condition.

$$C_1 \ C_2 \ .. \ C_n$$

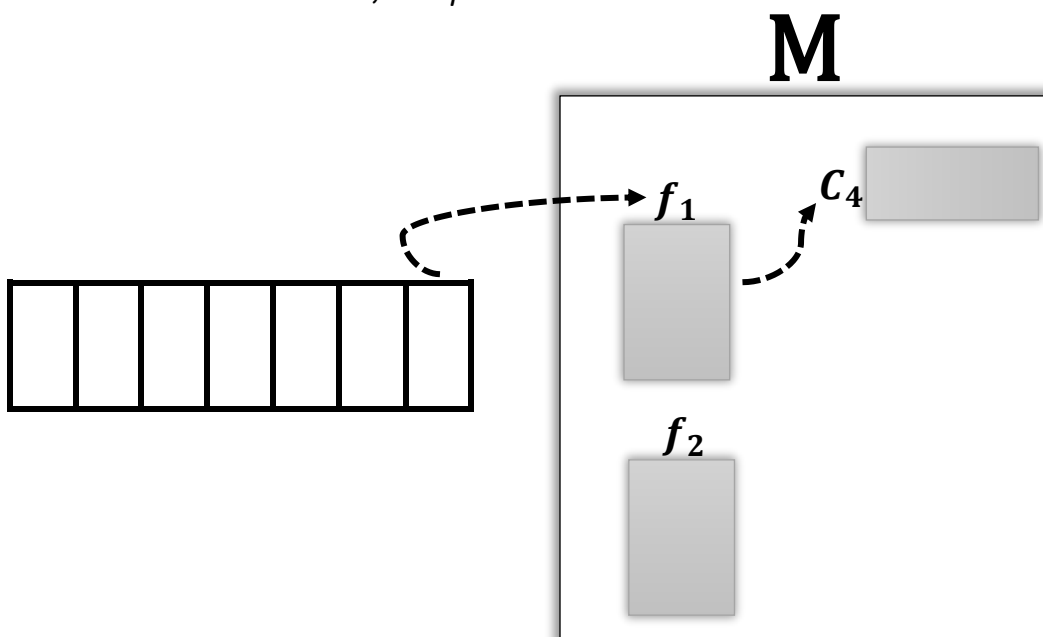
- Disons qu'il y a un processus qui attend que quelque chose devient **true** (vrai), donc il va se mettre dans une file d'attente, et en suite, le processus qui va changer la valeur à « Vrai », va devoir réveiller le processus qui se trouve dans la file d'attente.
- Exemple : j'attends que le buffer contient quelque chose, donc le processus qui va mettre quelque chose dans le buffer va devoir l'informer (le réveiller).
- On peut imaginer qu'on a des files d'attentes FIFO pour chaque condition.
- Les indices des fonctions $f_1 \ f_2 \$ non rien a voir avec les conditions $C_1 \ C_2 \$
-

Différence avec les sémaphores

- Lorsque ils attende, un processus arrive et fait **wait**.
Si il peut pas le décrémenter, il attend.
- Lorsqu'un processus fait signal, quelqu'un d'autres peut passer. Mais : aucun supposition sur l'ordre (l'ordre dans le quelle nous réveillons les processus en attente). Or, nous voulons que le réveiller soit par l'ordre.
- Sémaphore pas positif \Rightarrow je peut pas décrémenter \Rightarrow j'attend qu'il soit positif \Rightarrow l'ordre de l'attente est pas défini, en particulier l'implémentation que nous avons fait pour les sémaphores.
- Dans les moniteurs on suppose qu'il y a FIFO. C--à-d : on peut imaginer que chaque condition est une file d'attente. Initialement la file d'attente

est vide, toujours on réveille le processus qui a dormi le plus de temps, les files des conditions sont ordonnées (FIFO), donc on réveille les processus dans cet ordre.

- Pour un certain moniteur : à un moment donné une fonction s'exécute. La supposition qu'on fait est que l'appel des fonctions assure l'atomicité.
- On peut imaginer que le moniteur lui-même a une file d'attente, et ensuite ils entrent, un par un.

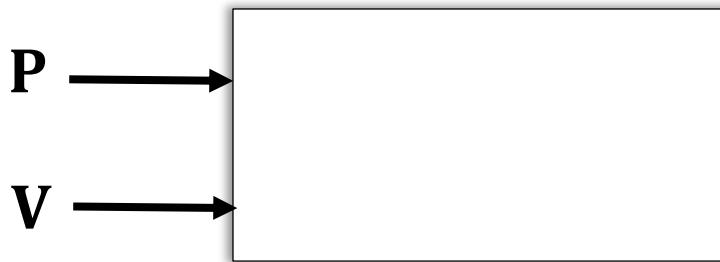


- Si le premier qui rentre est celui qui veut retirer du buffer, mais le buffer est vide \Rightarrow il va attendre, mais comme il est déjà à l'intérieur faut le suspendre, on va donc le mettre dans une certaine file, C_4 par exemple. Et on passe au processus suivant.
- Or, au moment où il va réveiller le premier processus, on a un problème et cela car il y a 2 processus dans le moniteur, alors qu'on avait dit qu'il peut y avoir que 1.
- Là, on peut adopter plusieurs stratégies, par exemple que le processus qui a réveillé l'autre va céder au processus qui s'est réveillé et va revenir lorsque celui-ci va terminer.

- Dans l'hypothèse que le processus 2 à exécuter f_2 , f_2 n'a pas été exécuté jusqu'au bout. Donc on obtient un mélange entre les exécutions des fonctions, mais l'exclusion mutuelle sur la structure de données qu'on veut protéger, est toujours assurée.
- Donc, à chaque moment qu'une fonction s'exécute, mais cela ne veut pas dire qu'elle s'exécute totalement.
- Supposons même que f_2 arrive à s'exécuter jusqu'au bout, il peut maintenant réveiller le processus 1, mais vu qu'il y a personne dans le moniteur, on a le choix entre le réveiller ou de faire entrer un autre processus. Or, tant qu'il y a un processus qui peut être réveiller, les processus qui se trouvent dans la file à l'extérieur, attendent.
- Un processus qui fait **signal** sur une condition est suspendu, jusqu'à ce que le processus réveiller termine ou alors se bloque sur une condition.
- Que se passera-t-il si ce processus réveiller va lui-même réveiller un autre processus ? il va également être suspendu. Donc, on peut avoir une chaîne de processus suspendus.
- Si le processus réveiller, réveille un autre (= fait **signal**), alors il est suspendu aussi.
- Un processus réveiller a la priorité sur un processus externe qui est en attente d'accès au moniteur.

Simulation des sémaphores avec les moniteurs

- Pour ne pas mélanger les 2 types de **wait** et **signal**, pour les sémaphores on va utiliser **P()** et **V()**.
- Un moniteur a une variable à l'intérieur, et on va avoir les 2 opérations **P()** et **V()**.



- On va définir notre moniteur « *Sem – simul* » (Sémaphore simule). On va définir une variable et une condition :

Moniteur Sem – simul Simulation d'un sémaphore binaire

var busy : boolean , initialisé à FAUX
notbusy : condition

Procédure P

```
if busy then wait(notbusy)
busy := true ;
```

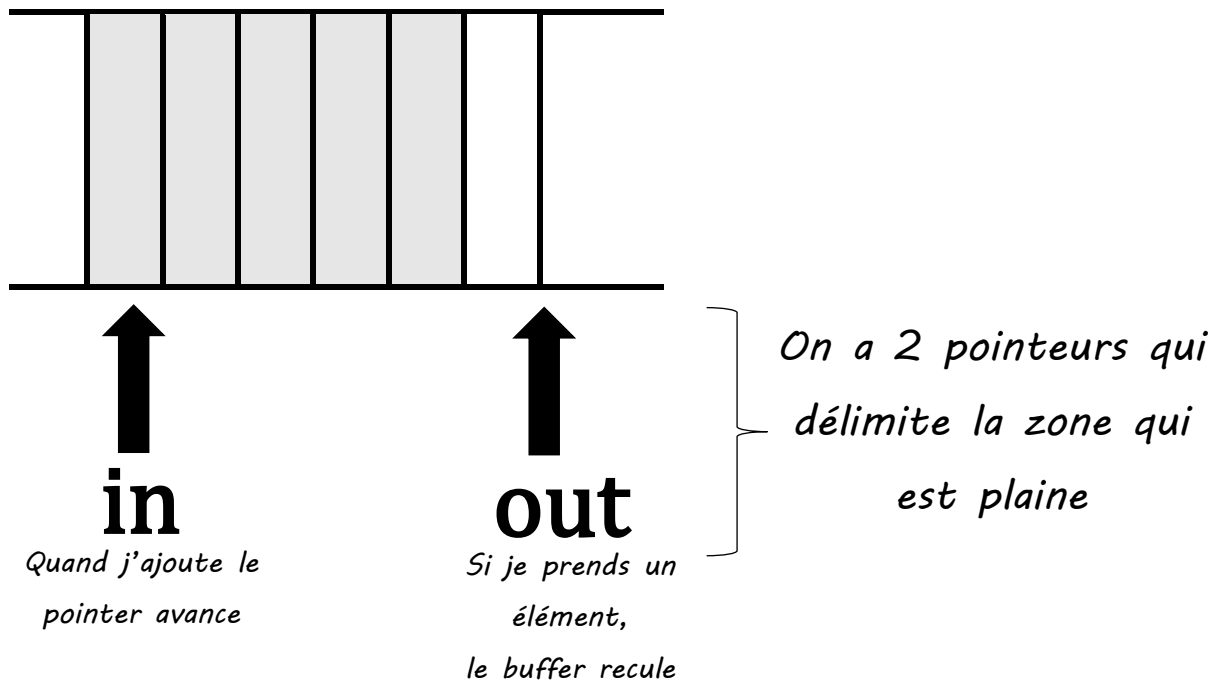
Procédure V

```
busy := false ;
signal(notbusy)
```

- Donc, avec les moniteurs, on peut faire tous ce qu'on peut faire avec les sémaphores.

Producteur - Consommateur avec les moniteurs

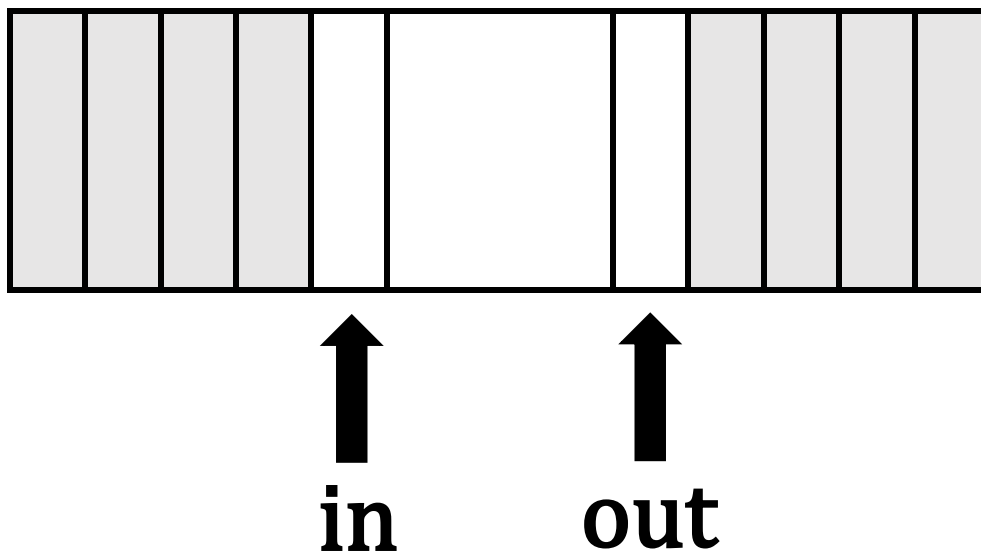
Un buffer circulaire

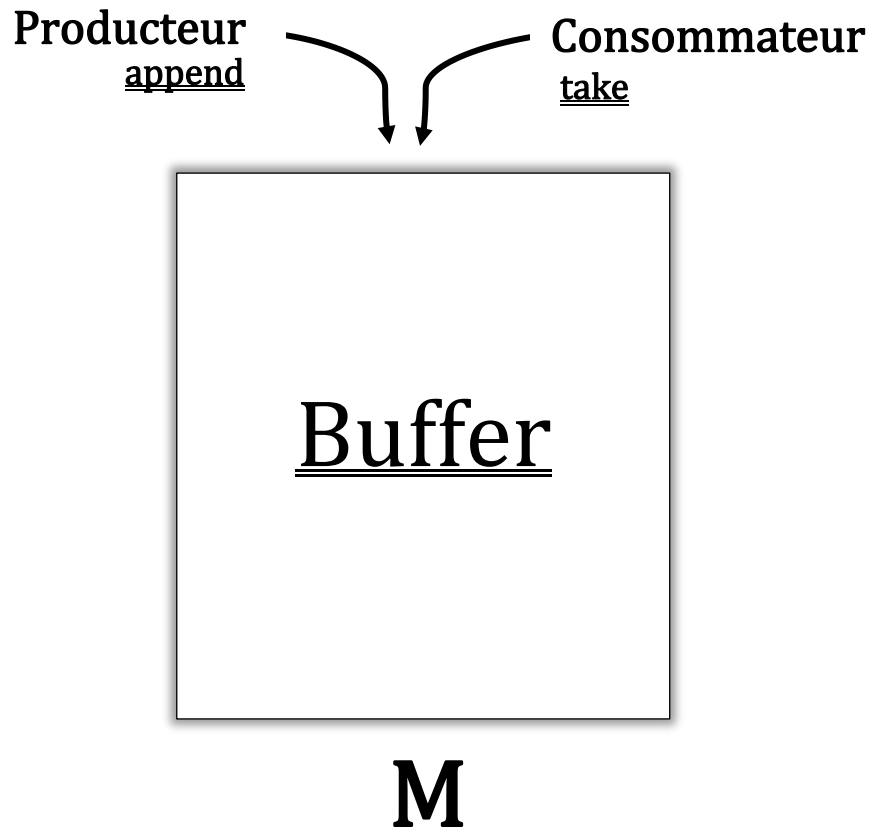


Si les 2 pointeurs se croise : le buffer est vide.

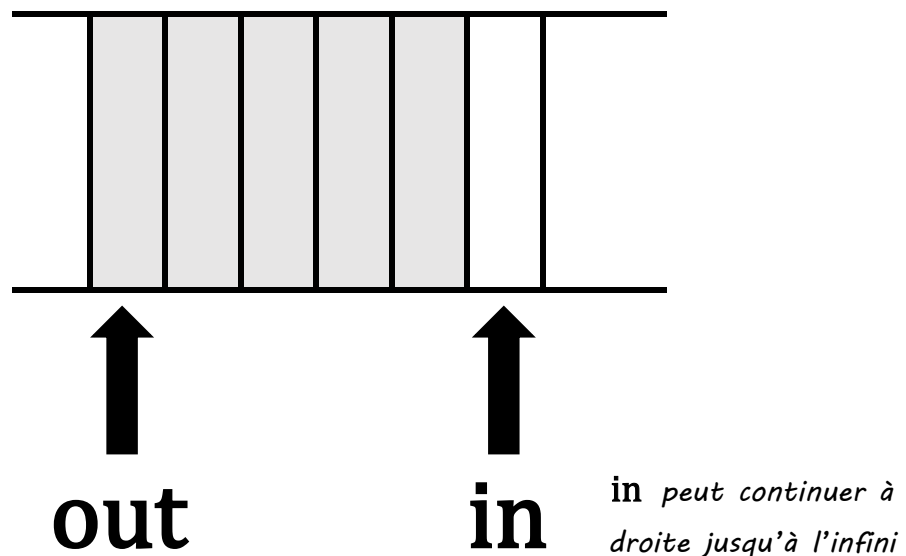
Taille du buffer : **B** , lorsque on atteint cette borne on s'arrête.

Le buffer peut aussi être de la forme :





On peut commencer, dans un premier temps, par un buffer infini qu'on peut toujours l'ajouter des éléments.



- *Donc, on ne s'occupe pas d'un cas où le buffer risque d'être plein.*
- *On a 2 fonctions à implémenter : **ajouter** et **retirer**.*
- *Dans le moniteur on va avoir un tableau et 2 variables : **in** et **out**.*

Moniteur buffer

b : array[0,-] // Tableau non borné of integer

in, out : integer // in, out : initialisé à 0

nonempty : condition

Procédure append(v : integer)

■ b[in] := v ;

in := in + 1 ;

■ signal(nonempty) // réveiller les processus en attente (si il y a de tel processus..)

Procédure take : return integer

// Soit le buffer est non-vidé : alors on peut prendre un element,

// soit le buffer est vidé : faut bloquer le processus

■ if (in = out) then // Si les 2 pointeurs se croise : le buffer est vidé.

■ wait(nonempty)

tmp := b[out] ;

out := out + 1 ;

■ return tmp

Vu que le buffer est non-borné, le seul problème qu'on doit gérer est la situation où le buffer est vidé.

Producteur - Consommateur : buffer borné

Moniteur buffer

```
n : integer // initialisé à 0  
notfull : condition  
b : array[0,B] // Tableau borné of integer  
in, out : integer // in, out : initialisé à 0  
nonempty : condition
```

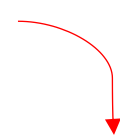
Procédure append(v : integer)

```
if (n == B+1) then wait(notfull)  
b[in] := v ;  
in := in + 1 ;  
if (n == B+1) then in := 0 ;  
n := n + 1 ;  
signal(nonempty) // réveiller les processus en attente (si il y a de tel processus..)
```

Procédure take : return integer

```
// Soit le buffer est non-vidé : alors on peut prendre un element,  
// soit le buffer est vide : faut bloquer le processus
```

```
if (n = 0) then // Si les 2 pointeurs se croise : le buffer est vide.  
    wait(nonempty)  
    tmp := b[out] ;  
    out := out + 1 ;  
    if (out = B + 1) then out := 0 ;  
    n := n - 1 ;  
    signal(notfull)  
    return tmp
```



*Si il fait **signal**, un autre processus va se réveiller et **take()** va pas faire return ⇒ il fera return plus tard.*

- Sémaphore : *notion de compteur*.
- Moniteur : *j'attends / un procesuus me réveille ; Une condition c'est une file d'attente*.