Théorie et pratique de la concurrence

CM9 Exclusion mutuelle

Bakery algorithm, sémaphores

Lundi 22·11·2021

Peterson (suite du CM 8)

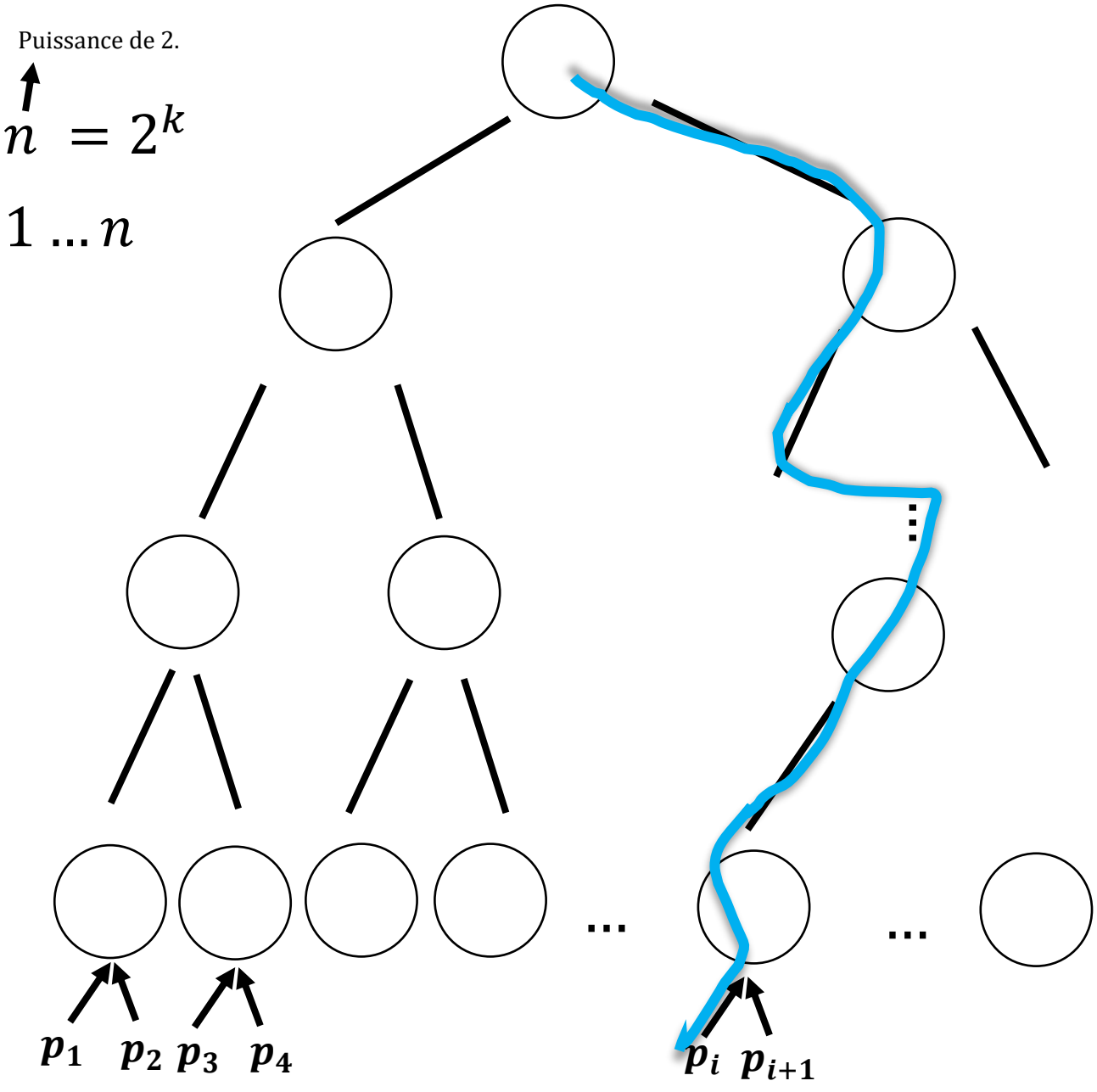
n	0
n	1
n - 1	2
n - 2	
. . .	
1 au plus	CS

Famine

- ***On admet que les gens en section critique vont sortir et revenir·***
- *Tous processus qui va sortir de 0 va forcément a un moment arrivé à la section critique, mais on peut pas savoir à l'avance combien de temps il va mettre· Les gens qui sont parti dans un certain ordre, vont pas forcement arriver à la section critique dans le même ordre·*
- *On peut montrer que quelqu'un à une place X peut être dépasser par quelqu'un arrivé après lui·*

- *On peut donc avoir des gens qui font beaucoup de tours avant que d'autre puisse avancer. C'est moins pire que la famine mais ce n'est pas une propriété désirable. One propriété intéressante peut être celle de FIFO, ici on voit qu'il y'a pas cette ordre.*
- *On a vu semaine dernier ce qui s'exécute avant la SC (Section Critique) : demande du lock → ensuite libération du lock.*

- *Si on avait des lock Peterson avec 2 processus. On peut étendre ca à n processus : l'idée est de considérer un arbre binaire. On suppose pour simplifier que n est une puissance de 2.*



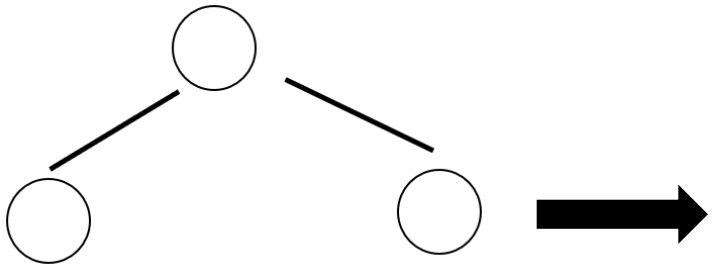
- Avant SC : prendre tous les lock qui vont du chemin à la racine.
- Apres SC : libérer tous les lock qu'on à pris.

Cette algorithme :

- ✓ Exclusion
- ✓ Pas de blocage
- ✓ Pas de famine

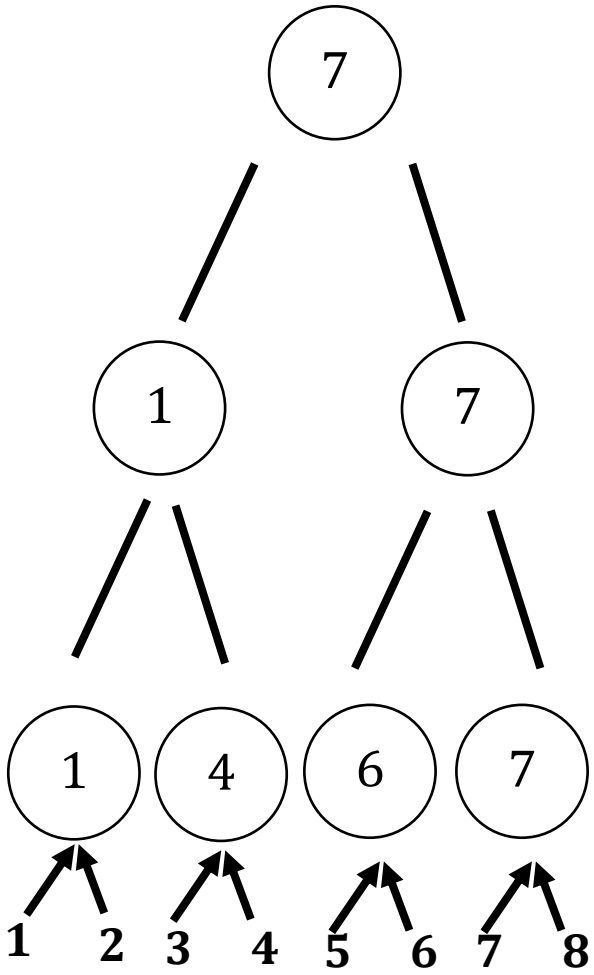
Propriété de l'équiter

- Les processus vont arriver à la section critique dans l'ordre OU des processus peuvent dépasser arbitrairement d'autres ? Il y a une façon de borné le nombre de fois qu'un processus entre en SC, avant que l'autre rentre ? Il y a des processus qui vont pouvoir dépasser les autres ?
- Si quelqu'un bloque quelque-part, un va dépasser devant lui. Ok. Et les autres ? Rien ne les empêche de monter, d'arriver en section critique et revenir.
- Supposons que quelqu'un arrive en haut, il revient en bas, c'est sûr qu'il devra attendre son binôme, mais rien n'engage qu'il va pas dépasser d'autres processus.
- Les binômes qui se batte un contre l'autre : les couples p_i et p_{i+1} .
- Dans le Top de l'arbre :

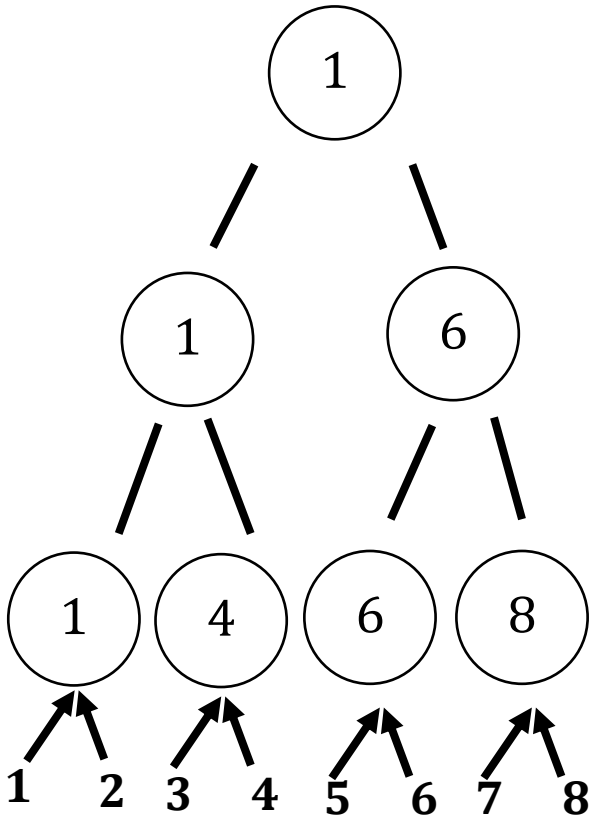


A ce stade on a les 2 premières gagnants. Si c'est celui de gauche qui passe, c'est sur que ensuite celui de droite va passer.

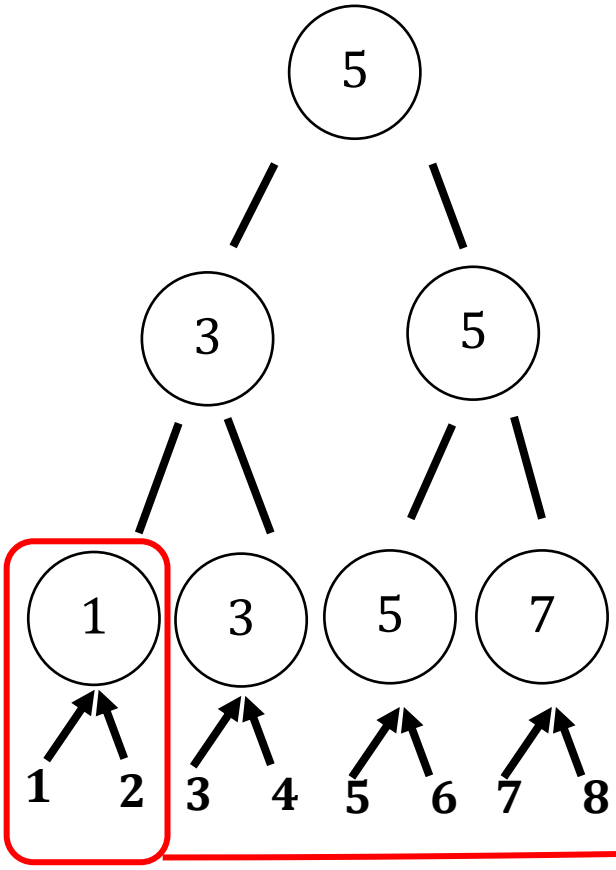
Essayons d'écrire un scénario ou y'a un processus qui va dépasser un autre



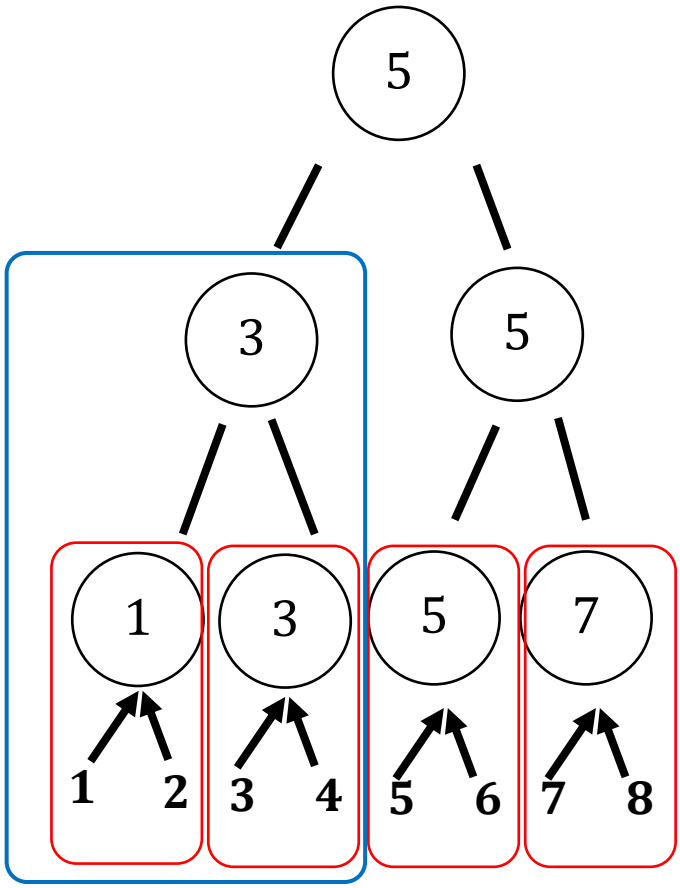
L'orsque 7 sort : 1 rentre, 6 est le victime.



Ensuite vont rentrer en section critique : 6, 4, 8, 5



Entre 1 et 2 il y a l'équiter : si c'est vrai pour tous l'arbre on pourrai le montrer par induction.



Pour chaque arbre l'ordre qui existe pour la racine de l'arbre est préservé.

Si c'est vrai pour chaque sous-arbre, ça va être préserver pour l'arbre tout-entier.

Bakery algorithm (Lampart)

- *L'algorithme qu'on va voir maintenant est assez simple, c'est quelque chose qu'on peut implémenter aussi de manière distribué, ca va aussi assurer l'équiter.*
- *L'orsque on entre en boulangerie on reçois un ticket. Celui qui va accéder au guichet (la section critique), c'est celui qui a le numéro le plus petit.*
- *Donc, intuitivement, on regarde les tickets des autres, et on prend un ticket plus petit.*
- *C'est un algorithme qui est directement pour n processus.*

- *On va commencer par une version un peut FAUSSE :*

```
Flag = new boolean[n] // Pour dire si je suis intéresser
Label = new int[n] // Les tickets avec les numéros

for(i = 0 ; i < n ; i++) {
      Flag[i] := false ;
      Label[i] := 0 ;
}
```

Proc P_i

```
Flag[i] := true // La ressource m'intéresse
// Prendre un numéro plus grand que celui à la fin de la queue
Label[i] := max(Label[0], … , Label[n-1]) + 1 ;

// Tant que il y a quelqu'un diffèrent de moi qui est intéresser et qui a un numéro
// plus petit que moi : j'attend
while( ∃ k ≠ i (Flag[k]) ∧ (Label[k] < Label[i]) ) do {}
       k différent de moi (i)   Egale true
crit_i ;
Flag[i] := false // Je ne suis plus intéresser
```

Il y a un problème dans cet version :

Si 2 processus arrive en même temps, alors il obtiennes le même numéro ⇒ pas d'exclusion mutuelle !

Donc, lorsque il y a une égalité faut la casser. Si 2 processus on le même Label faut leur ordonner d'une certain manière, par exemple : le numéro de processus.

Donc, on va changer dans le code :

$$\text{while}(\exists \underbrace{k \neq i}_{\substack{\text{k différent} \\ \text{de moi (i)}}} \underbrace{(\text{Flag}[k])}_{\text{Egale true}} \wedge ((\text{Label}[k], \mathbf{k}) < \textbf{\textit{lex}}(\text{Label}[i], \mathbf{i})) \text{ do } \{\}$$

- **lex : Ordre lexicographique.**

- **Exclusion mutuelle ? Oui.** Une fois qu'un processus est en SC, sont Label et sont Flag ne change pas. Pour que quelqu'un d'autre entre, faut qu'il y aura quelqu'un avec un Label plus petit, or on fait une vérification de l'ordre lexicographique, donc c'est OK).
- **Blocage ? Non.** Ou peut y avoir un risque se bloquer ? dans la boucle. C'est le cas ? non, car il y a toujours un processus avec le numéro le plus petit. Donc, absence de blocages .

- **Famine ? Non.** Un processus peut rester tout sa vie entrain d'attendre ?
  - Là on va aussi s'intéresser au sujet de FIFO : supposons qu'on a 2 processus. Un arrive avant l'autre, ils vont passer dans l'ordre ?

- *Le premier qui demande un Label, reçoie un Label·*
- *Le deuxième reçoie un autre label (ou, au pire, le même Label), mais si il y a un qui est arriver avant l'autre : il aura un label strictement plus petit que l'autre Label·*
  Donc : on a l'ordre FIFO !

la propriété de FIFO + absence de blocage
⇒ absence de famine

Pas de famine ⇏ FIFO

Implémentation

Les Label ont un domaine infini, or pour l'implémentation le fait que ca soit non-borné, ça peut être un problème· Donc, grâce à la propriété de FIFO on pourra se débrouiller avec un nombre fini de numérateur·

Sémaphores

On peut exprimer tous les algorithmes de la façon suivante :

$$lock(l)$$
$$unlock(l)$$

Donc, une façon d'assurer l'exclusion mutuelle :

$$lock(l)$$
$$crit$$
$$unlock(l)$$

Dans nos algorithmes, on suppose que les opérations se font de manière atomique. Notre dernier algorithme peut marcher même si c'est pas le cas.

Une façon de raisonner est de dire que les 2 opérations $\boldsymbol{lock(l)}$ et $\boldsymbol{unclock(l)}$ sont fait par le même processus. Maintenant, il y a d'autres mécanismes qui peuvent être utiles. C'est donc la qu'on va voir les sémaphores (ca va ressembler à ces **lock**).

Un sémaphore c'est un compteur, on a le droit de le décrémenter un certain nombre de fois. **Le sémaphore binaire** c'est un peu comme un **lock**.

Donc, soit j'attend une certaine condition, alors je peux décrémenter, sinon - j'attends.

Je peut aussi signaler que le sémaphore est devenu positif (incrémenter le compteur, réveiller tous ce qui sont en attente).

- ***Sémaphore positif :*** *on décrémente, on avance*
- ***Sémaphore négatif :*** *on entre*

Celui qui va rendre le sémaphore positif, il va réveiller le prochain dans la fil d'attente, et alors il y a un qui peut passer.

Un sémaphore S

On a un sémaphore S*, c'est un entier (on peut le voir comme une variable de nombre entier) et maintenant, on a que 2 opérations possibles :*

- **Wait(S)** *Notation historique :* **P(S)**
- **Signal(S)** *Notation historique :* **V(S)**

S variable entière (compteur)

Wait(S) : si $s > 0$
alors $s := s - 1$
sinon *exécution suspendu*

Signal(S) : **si** un processus P est suspendu (= entrain d'attendre), après un Wait(S), alors le réveiller (donc il entre en section critique)
sinon $s := s + 1$

- On suppose que les processus qui sont en attente ne sont pas ordonnée, mais on peut très bien imaginer des version ou les processus attende d'une manière ou une autre.
- **S** est jamais négatif (toujours positif) ⟶ si on aurai pu décrémenter / incrémenter autan qu'on veut, il n'aurai servi à rien.

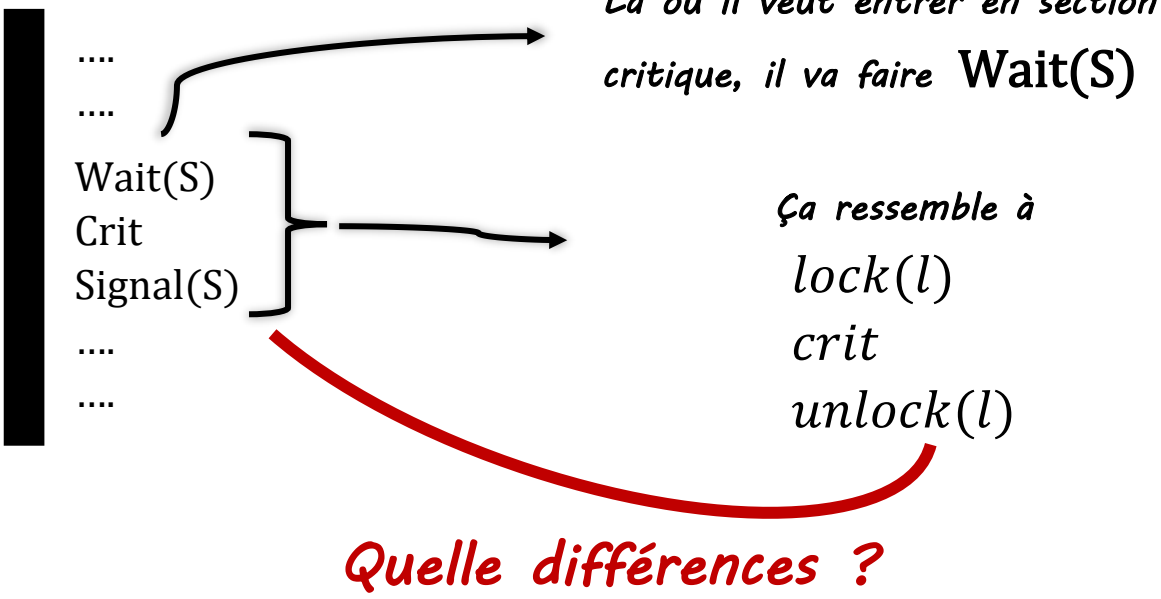
- Le cas binaire : un sémaphore S qui est égal à 0 ou 1.
- On initialise toujours le sémaphore par quelque chose, généralement on initialise S par le nombre maximal qu'on veut qu'il atteint.
- Les processus qui sont en attente : quand je fait **Signal(S)**, un processus qui est en attente va se réveiller. (On considère pas ca à l'implémentation pour le moment)

- Les sémaphores : quelque chose qui devra nous être fourni par une certain libraire.
- On utilise les sémaphores pour les problèmes d'exclusion mutuelle, producteur-consommateur, etc.

Exclusion mutuelle (pour n processus)

S : sémaphore binaire
S := 1 ; // Initialisation à 1

Proc P_i

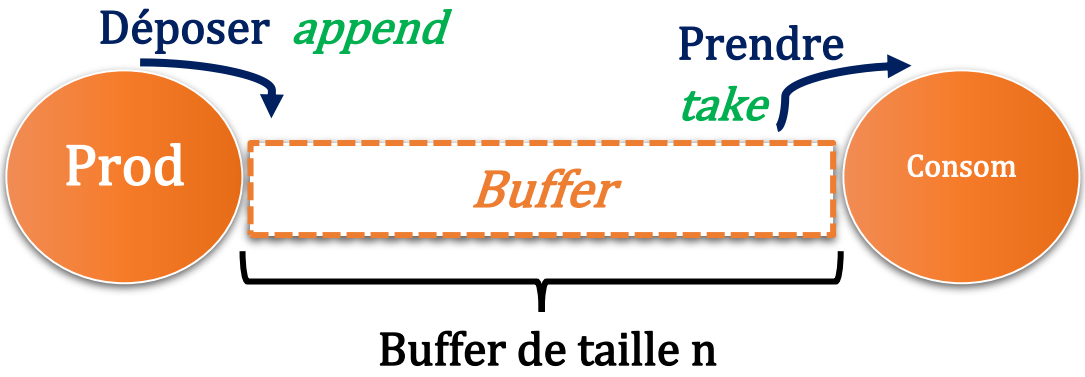


Quelle différences ?

- Les **lock** : même processus.
- Sémaphores : ça peut être par des processus différents. Il y a pas la restriction que celui qui a fait le **wait**, va faire aussi **signal**.

- S peut être vu comme une variable protéger.

Producteur - consommateur



- On va utiliser un sémaphore, mais ca va pas forcement être un sémaphore binaire.
- Opérations abstraites :
  - **produce**
  - **append**
  - **take**
  - **consomme**
- Par exemple : le producteur va produire quelque chose et ensuite il va faire **append** pour l'ajouter au buffer.

La version la plus simple

// n : Le nb d'élements dans le buffer. Dans cette version : pas de limite sur ce nombre.
n : semaphore
n := 0

Producteur

repeat
produce
append
sigal(n)
for ever

Consommateur

```
repeat
    wait(n)
    take ;
    consomme
for ever
```

Maintenant, supposons qu'on veut aussi l'exclusion mutuelle.

Faudrait que **append** *et* **take** *devienne conflictuelle. Il faut donc ajouter un sémaphore* *binaire* *pour protéger ces actions. Donc, on aura 2 types de sémaphores :*

// n : Le nb d'élements dans le buffer. Dans cette version : pas de limite sur ce nombre.
n : semaphore
n := 0
S : semaphore binaire
S := 1

Producteur

```
repeat
    produce
    wait(S)
    append
    signal(S)
    sigal(n)
for ever
```

Consommateur

```
repeat
    wait(n)
    wait(S)
    take
    signal(S)
    consomme
for ever
```

On a vu dans ce CM des mécanismes de bas niveau. Par la suite, on va voir les moniteurs et on va parler d'implémentation.

- **Moniteur** : mécanisme de plus haut niveau.