Théorie et pratique de la concurrence

# CM8 Exclusion mutuelle

## Algorithme de Peterson

**Lundi 15·11·2021**

Suite à l'algorithme de Dekker, aujourd'hui on va voir des algorithmes supplémentaires, notamment pour n processus, et pas que 2·

Une première tentative (qui ne va pas être bonne au départ, on va améliorer par la suite), est de partir du point de départ de la dernier fois : on va avoir une variable **Flag** pour dire qu'on est intéressé·

## Algorithme 1

On va déclarer un tableau de booléen qu'on va appeler Flag et on va dire qu'il est de taille 2 (2 processus)· Ensuite, on va voir la généralisation pour n processus·

Flag = new boolean[2] // Indices de 0 à 1. Donc, i peut être égale à 1 ou 0.
Flag[0] = Flag[1] = false

Proc $P_i$

Flag[i] := true // La ressource m'intéresse
while(Flag[1-i]) do {} // Attente tant que l'autre est intéresser
$crit_i$ ;
Flag[i] := false // Je ne suis plus intéresser

**On lance 2 procédures comme ça en parallèle.**

1. **Ca empêche que les 2 rentre en section critique en même temp ? Oui.** Au moment ou je m'apprête à entrer, mon Flag est à 1, je teste (dans le while) que l'autre est pas intéresser. L'autre, il est soit au début, soit à la fin de sa procédure. Si j'ai déjà « bloquer la porte » en disant que je suis intéressé, l'autre ne pourra pas franchir le while.

2. **Famine ? Blocage ? Oui.**

   - **Famine ? :** Si quelqu'un veut rentrer, un jour il rentrera ?
   - **Blocage ? :** L'orsque on parle de blocage, on suppose que personne ne meure dans la section critique. Si il rentre – il sort.
   - Blocage : Si tous les 2 arrives et dise qu'il sont intéresser.
   - Si y'a blocage, y'a famine aussi, forcément.

## Algorithme 2 (toujours avec 2 processus).

L'orsque je vais dire que je suis intéressé, je dis : « je suis intéressé mais je vais laisser passer l'autre ». Pas de « Flag » maintenant, mais on va avoir une variable globale qu'on va appeler « la victime » (= le processus qui va rester en arrière pour laisser passer les autres).

int victim

Proc $P_i$

```
victim := i // c'est moi la victime
while(victim = i) do {} // Tant que la victime c'est moi, j'attend
crit_i ;
```

**Exclusion mutuelle ? Oui.** Car l'orsque on dépasse le while, l'autre est forcément victime,

**Problème ?** Celui qui sort de section critique, risque de ne pqs se déclarer à nouveau victime (car, par exemple, le thread est terminer) et ainsi l'autre processus reste bloquer.

# Algorithme de Peterson (pour 2 processus)

Un algorithme assez simple qui combine les 2 algorithmes prédécèdent.

```
Flag = new bool[2] // Initialisé à false
int victim

Proc Pi
    Flag[i] := true // La ressource m'intéresse
    victim := i // Je suis aussi la victim
    // Tant que l'autre est intéresser et victime est égal à i, j'attends
    while(Flag[1-i] ∧ victim = i ) do {}
    crit_i ;
    Flag[i] := false /* Je ne suis plus intéresser (donc même si l'autre reste la victime, on empêche pas l'autre d'entrer en section critique) */
```

**Exclusion mutuelle ? Oui.** Y'a toujours un qui va affecter victime en dernier.

> **Faut résonner de manière : « Quelle opération va se passer avant / peut se passer avant l'autre ».**

**Blocage ? Non.** Même si l'autre reste la victime, on empêche pas l'autre d'entrer en section critique

**Famine ? Non.** Est-ce que quelqu'un peut revenir à chaque fois demander la section critique et voir à chaque fois que quelqu'un passe devant ?

## Donc,

**Les propriétés de cet algorithme :**

1. Exclusion mutuelle
2. Absence de blocage
3. Absence de famine

# - Algorithme plus simple que l'algo de Dekker

## Rappelles de nos 2 hypothèses

1. Les lectures et écritures sont atomique, ce qui n'est pas le cas dans la réalité.

2. On ne meure pas en section critique

## Comment généraliser à n processus ?

L'idée est de choisir un certain ordre entre les processus. Si y'a que 2, c'est soit un, soit l'autre. Mais, si y'a plusieurs, faut décider qui va d'abord rester attendre et ainsi on fait passer les n-1 autres processus, et ensuite : parmi les n-1 processus, qui va rester attendre et ainsi va laisser passer les n-2 processus... ? ... Jusqu'à la fin il ne reste plus que 2 processus.

Les processus qui sont derrière (les victimes) peuvent passer que lorsque ceux devant-eu, on tous passer.

# Algo de Peterson pour n processus

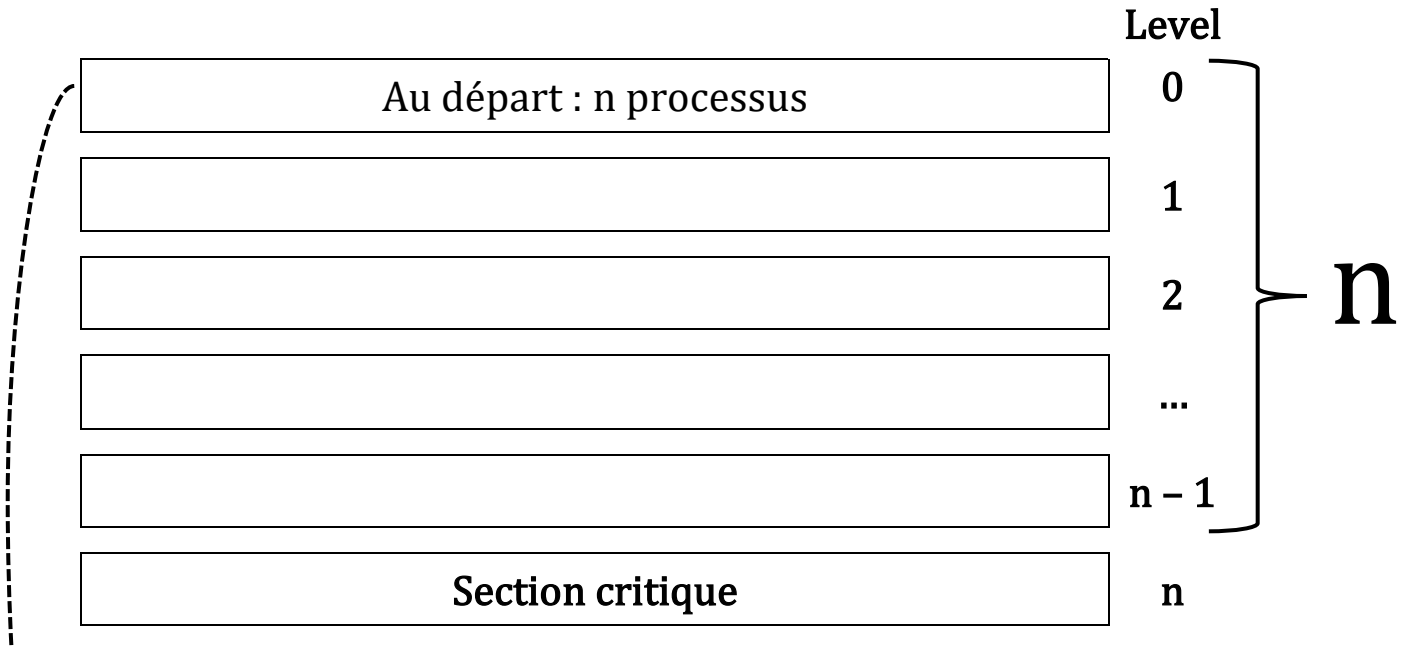
*On va avoir un tableau qui s'appelle* **Level**...

```
Level = new int[n] // Les processus sont de 0 jusqu'à n-1
Victim = new int[n]
for(i = 0 ; i < n ; i++) {
        Level[i] = 0 // Au départ ils sont tous au niveau 0
}
```

Proc $P_i$

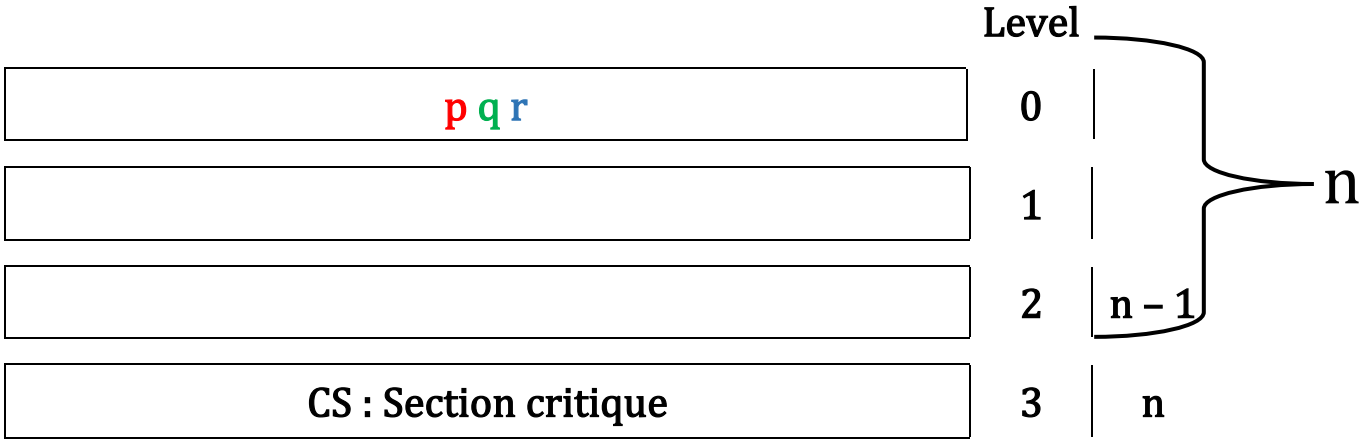
*Pour pouvoir entrer faudra qu'il passe au level 1, ensuite level 2, ensuite level 3... Si il est choisi comme victime il reste au même level, et il pourra bouger (avancer) que a une certaine condition.*

```
for(l = 0 ; l < n ; l++)   // l pour level
    Level[i] := l
    Victim[l] := i   // Je suis victime
    while(∃ k ≠ i : Level[k] ≥ l ∧ Victim[l] = i) do {}
crit_i ;
Level := 0 // Je reviens au niveau 0
```



Mon niveau c'est $l$ et je me met tous de suite en victime (je laisse passer les autres). Donc, celui qui arrive en dernier – c'est lui la victime. Pour chaque niveau, je reste dans ce niveau, tant qu'il existe un certain processus $k$ (différent de moi) qui est a un niveau supérieur a moi et c'est moi la victime.

## Exemple




- On suppose que **p** commence
  - (1) **p** d'abord est level 1 et victime
  - (2) **q** arrive en level 1 et devient victime. **p** est plus victime : il a quelqu'un avec lui au même niveau mais lui il est pas victime. **q** se dit : « il y a quelqu'un avec moi et je suis victime, donc j'attends ».
  - (3) **p** arrive au level 2
  - (4) **r** arrive en level 1, donc c'est lui qui devient maintenant la victime. **q** maintenant peut aussi arriver au level 2 et alors **p** rentre en section critique.
  - (5) L'orsque **p** sort de section critique, il passe au level 0. **q** rentre en section critique.

*Tlm peuvent arriver au level 1*

| | |
|---|---|
| n | 0 |
| n | 1 |
| n - 1 | 2 |
| . . . | |
| n- $l$ - 1 | $l$ |
| 2 au plus | n - 1 |
| 1 au plus | CS |

- *Donc,* ***on a l'exclusion mutuelle.***
- *Blocage ?* ***Non.***
- ***Famine ?*** *est-ce-que quelqu'un risque de voir passer les gens devant lui alors qu'il attend à l'infini ? Quelqu'un risque de ne jamais rentrer ? Quelqu'un peut attendre éternellement bloqué ?*

  ***Non,*** *car « bloqué » veut dire qu'il y a toujours des gens devant lui et qu'il est toujours victime.*

  *Plus précisément, si tous ce qui sont devant li rentre, finalement il n'y aura personne devant lui.*

  *Mais est-ce-que quelqu'un peut nous dépasser ?*
  *Pas de famine, mais pas de garanti d'équiter car les processus peuvent se doubler de manière arbitraire (si un system s'exécute plus rapidement que l'autre).*