

IF223 - Algorithmique Distribuée

2021-2022

Rohan Fossé - rohan.fosse@labri.fr

Quelques infos

2

- Rohan Fossé, rohan.fosse@labri.fr
- Site web: www.labri.fr/perso/rfosse
- « Distributed Computing Fundamentals, Simulations, and Advanced Topics » H. Attiya, J. Welch. Wiley, 2nd edition
- La plupart des slides : Une version modifiée des slides d'Alessia Milani

Systemes Distribués

- ❑ Les systèmes distribués sont devenus omniprésents:
 - ❑ Ressources partagées
 - ❑ Communications
 - ❑ Accroître les performances
 - ❑ Vitesse
 - ❑ Tolérance aux pannes

Incertitude dans les systèmes distribués

- ❑ L'incertitude provient essentiellement:
 - ❑ des vitesses de processeur différentes
 - ❑ des délais de communication variables
 - ❑ Défaillances (partielles)
 - ❑ flux d'entrée multiples et comportement interactif

Raisonnement sur les systèmes distribués

- ❑ L'incertitude fait qu'il est difficile d'être sûr que le système est correct.
- ❑ Pour remédier à cette difficulté :
 - ❑ identifier les problèmes fondamentaux
 - ❑ énoncer les problèmes avec précision
 - ❑ concevoir des algorithmes pour résoudre ces problèmes
 - ❑ prouver l'exactitude des algorithmes
 - ❑ analyser la complexité des algorithmes (e.g temps, espace, messages)

Domaines d'application

- ❑ Ces domaines ont fourni des problèmes classiques de l'informatique distribuée :
 - ❑ systèmes d'exploitation
 - ❑ systèmes de bases de données (distribuées)
 - ❑ les réseaux de communication
 - ❑ architectures multiprocesseurs
- ❑ Des domaines d'application plus récents :
 - ❑ cloud computing (l'informatique en nuage en français)
 - ❑ informatique mobile, ...

Résumé du cours : Principes fondamentaux

- ❑ Présentation de deux modèles de communication de base :
 - ❑ passage de messages
 - ❑ mémoire partagée
- ❑ et deux modèles de synchronisation de base :
 - ❑ synchrone
 - ❑ asynchrone

Résumé du cours : Principes fondamentaux

- ❑ Aborder les problèmes et questions classiques :
 - ❑ exclusion mutuelle
 - ❑ algorithmes de graphes
 - ❑ élection du leader
 - ❑ consensus tolérant aux pannes

ALGORITHMES DISTRIBUÉS : MODÈLE DE MÉMOIRE PARTAGÉE



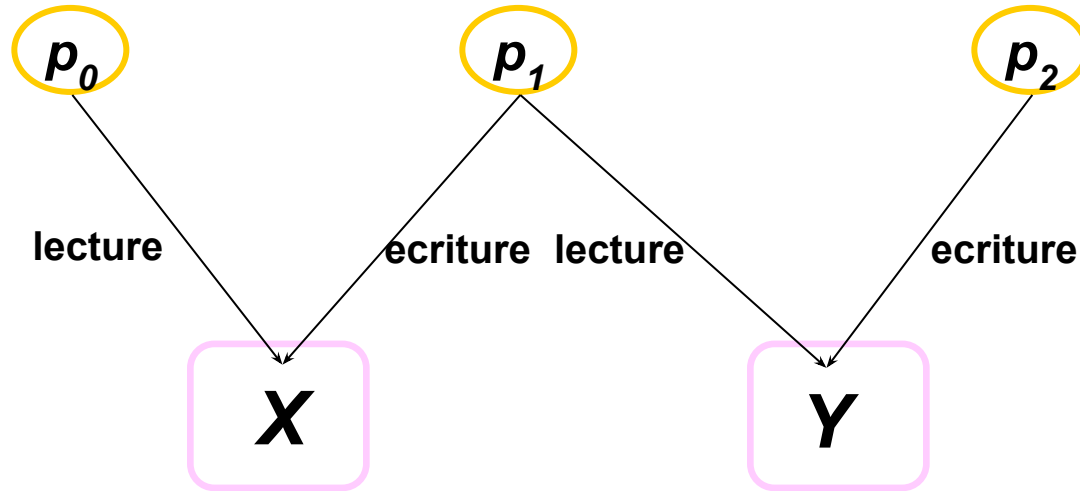
Modèle de mémoire partagée

10

- Les processeurs communiquent via un ensemble de variables partagées
- Chaque variable partagée a un type, définissant un ensemble d'opérations qui peuvent être effectuées de façon *atomique*.

Exemple de modèle de mémoire partagée

11



Modélisation des processeurs

- Le processeur est une machine à états incluant l'état local du processeur
- Le vecteur d'états du processeur, un par processeur, est une *configuration* du système.

Étape de calcul dans le modèle de mémoire partagée

13

- ❑ Quand le processeur p_i fait une étape:
 - ❑ l'état de p_i dans l'ancienne configuration spécifie quelle variable partagée doit être accédée et avec quelle opération
 - ❑ l'opération est effectuée: la valeur de la variable partagée dans la nouvelle configuration change selon la sémantique de l'opération.
 - ❑ L'état de p_i dans la nouvelle configuration change en fonction de son ancien état et du résultat de l'opération.

Exécution du calcul dans le modèle de mémoire partagée

- ❑ Le format est le suivant:
 - ❑ config, étape, config, étape, config, ...
- ❑ dans la première configuration: chaque processeur est dans l'état initial
- ❑ Pour chaque consécutive (config, step, config), la nouvelle config est la même que l'ancienne sauf que :
 - ❑ changement d'état du processeur spécifié selon la fonction de transition

Admissibilité

- ❑ La définition de l'exécution donne quelques conditions "syntaxiques" de base.
 - ❑ conditions de sécurité habituelles: rien de "mauvais" ne se produit.
- ❑ Parfois, nous voulons imposer des contraintes supplémentaires
- ❑ Les exécutions satisfaisant les contraintes supplémentaires sont **admissibles**. Ce sont les exécutions qui doivent résoudre le problème d'intérêt.
 - ❑ La définition du terme "**admissible**" peut changer d'un contexte à l'autre, en fonction des détails de ce que nous modélisons.

ALGORITHMES DISTRIBUÉS EN MÉMOIRE PARTAGÉE : UN EXEMPLE



Un problème classique : Renommage

17

- n entités informatiques (processus) avec des noms uniques dans un grand espace nom : nombre de noms possibles $M \gg n$



Anne



Bob



Claire

...



Paul

- N ressources, chacune pouvant être accédée par au plus un processus à la fois : $n \leq N \ll M$



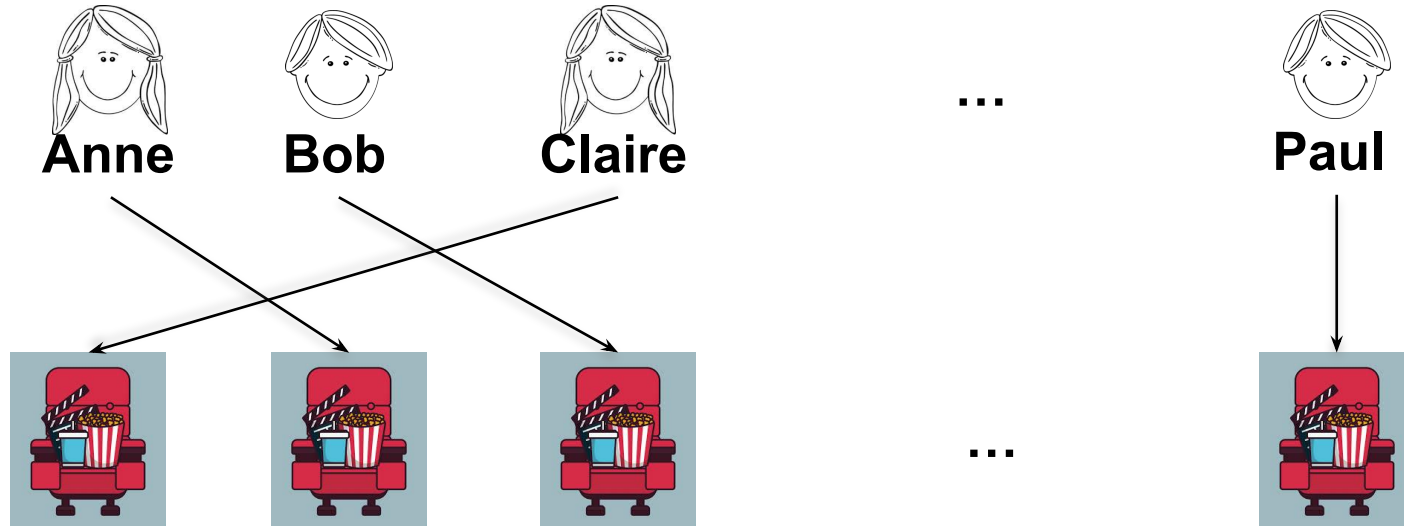
...



Un problème classique

18

- ❑ Affecter des ressources aux processus de telle sorte que
 - ❑ Sécurité : Deux processus n'obtiennent pas la même ressource
 - ❑ Vivacité : Chaque processus obtient une ressource



Un algorithme distribué : pour deux processus ($n=2$, $N=3$)

19



Anne

Shared memory



Paul

Écrire son nom

Lire l'autre nom (s'il y en a)

1. Pas d'autres noms
2. Mon nom < son nom
(ordre alphabétique)
3. Sinon



1



2

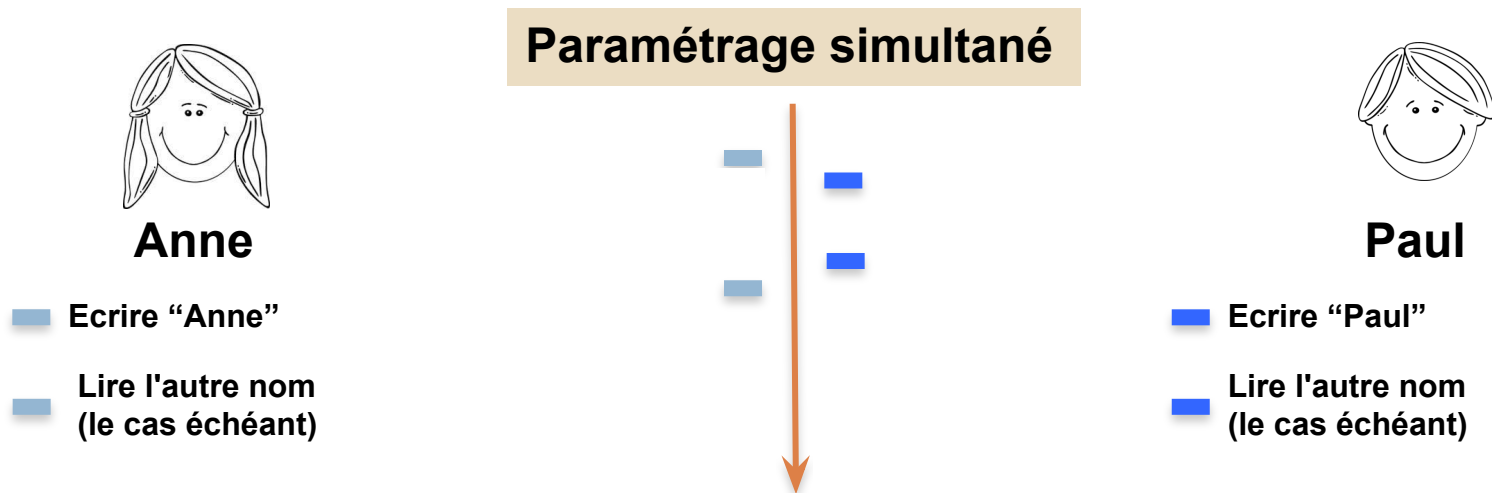


3

Concurrence



20

Anne et Paul exécutent leur algorithme en parallèle.



Modèle de synchronisation : Système asynchrone

21

1. Les délais entre les étapes de calcul d'Anne  sont imprévisibles (de même pour Paul .
2. Aucune hypothèse sur la vitesse relative d'Anne et de Paul





Anne

-  Ecrire « Anne »
-  Lire l'autre nom
(le cas échéant)



Paul

-  Ecrire « Paul »
-  Lire l'autre nom
(le cas échéant)

Concurrence et asynchronisme

22



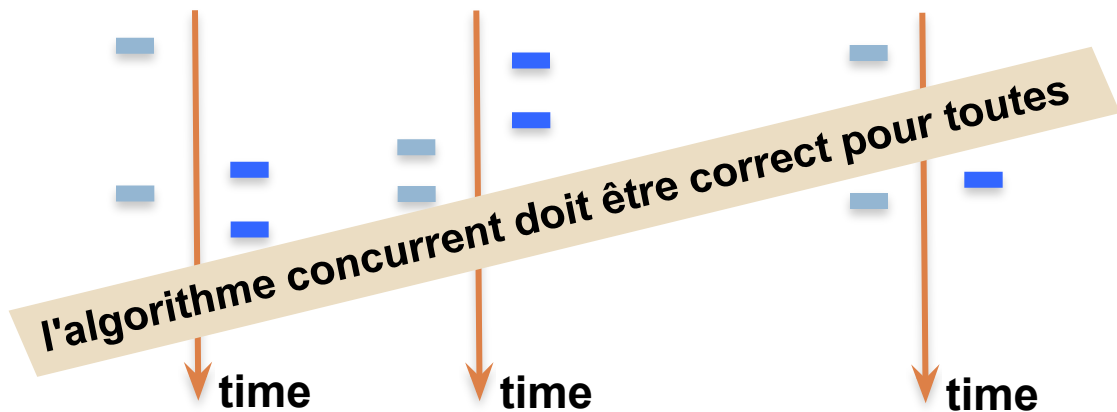
Anne



Beaucoup d'imbrications possibles



Paul



Connaissance partielle

23

Est-ce que Paul
a lu "Anne" ?



Anne

- Ecrire « Anne »
- Lire l'autre nom (le cas échéant)

Shared memory



Des scénarios indiscernables

Scénario 1

-
- Ecrire Paul
- Lire Anne
-

Scénario 2

-
- Ecrire Paul
- Ne lire aucun autre
-

Tolérance aux pannes

24

Crash failure : un processus peut arrêter d'exécuter son algorithme.



Anne

■ Ecrire « Anne »

■ Lire l'autre nom
(le cas échéant)

**Anne ne peut pas compter sur le fait
que Paul fasse quelque chose si elle
veut toujours obtenir un siège
(idem pour Paul)**



Paul

ALGORITHMES DISTRIBUÉS EN MÉMOIRE PARTAGÉE : LE PROBLÈME DE L'EXCLUSION MUTUELLE (DIJKSTRA 1965)

RÉFÉRENCES BIBLIOGRAPHIQUE :

- [1] MAURICE HERLIHY, NIR SHAVIT. *THE ART OF MULTIPROCESSOR PROGRAMMING*. MORGAN KAUFMANN, 2012. MOTIVATING EXAMPLE
- [2] GADI TAUBENFELD. *SYNCHRONIZATION ALGORITHMS AND CONCURRENT PROGRAMMING*. PEARSON, 2006. CHAPTER 2

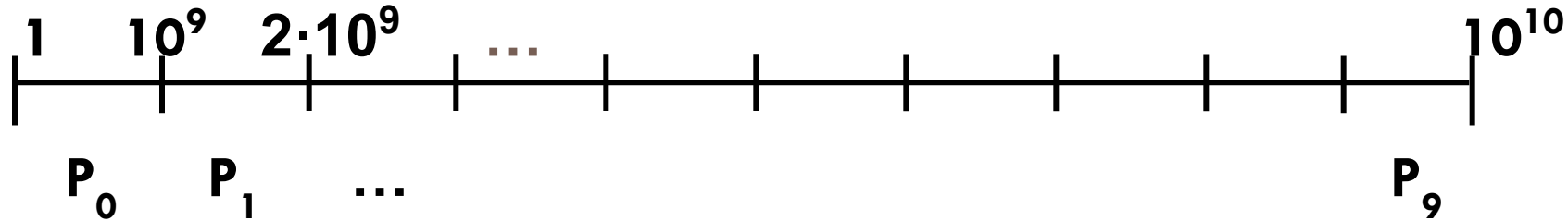
Un exemple motivant : Test de primalité parallèle

26

- ❑ Challenge
 - ❑ Afficher les nombres premiers de 1 à 10^{10}
- ❑ Étant donné
 - ❑ Multiprocesseur à dix processeurs
 - ❑ Un processus par processeur

Equilibrage du travail

27



- Répartir le travail de manière égale
- Chaque processus teste une gamme de 10^9

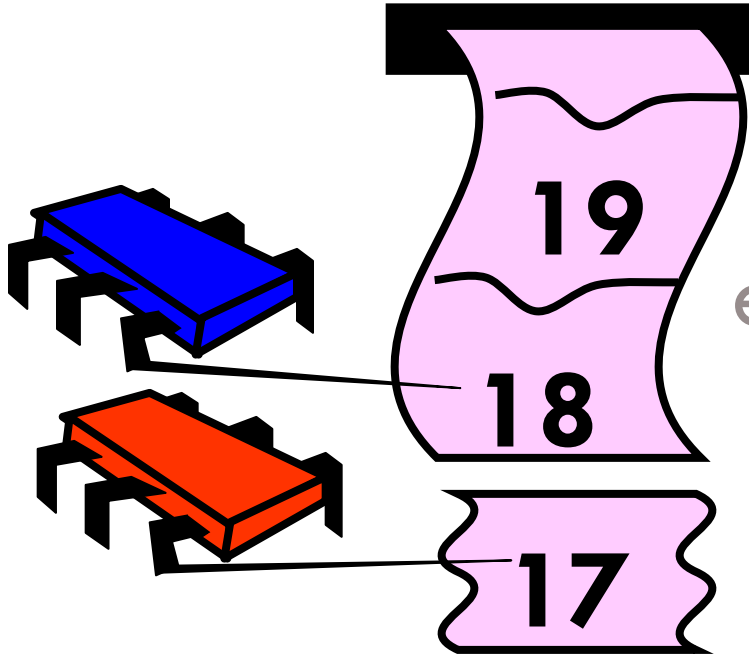
Problèmes

28

- ❑ Les plages supérieures ont moins de nombres premiers
- ❑ Pourtant, les grands nombres sont plus difficiles à tester
- ❑ Charges de travail des processeurs
 - ❑ Non égale
 - ❑ Dure à prédire
- ❑ Besoin d'un équilibrage dynamique du travail

Compteur partagé

29



each process takes a
number

Implémentation de la procédure pour le processus *i*

30

```
1 int counter = new Counter(1);
2
3 void primePrint {
4     long j = 0;
5     while (j < 1010) {
6         j = counter.getAndIncrement();
7         if (isPrime(j))
8             print(j);
9     }
10 }
11
```

Implémentation de la procédure pour le processus i

31

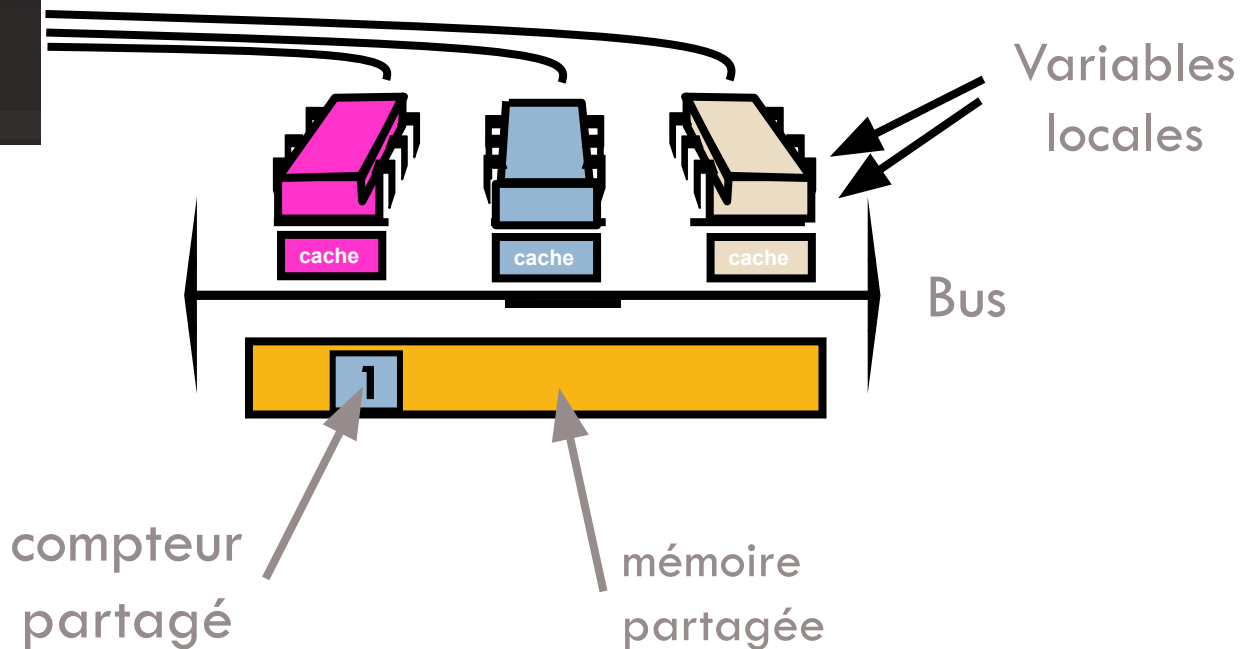
```
1 int counter = new Counter(1); ← Compteur partagé
2
3 void primePrint {
4     long j = 0;
5     while (j < 1010) {
6         j = counter.getAndIncrement();
7         if (isPrime(j))
8             print(j);
9     }
10 }
11
```

Où résident les choses

32

```
1 void primePrint {  
2   int i = ThlectureID.get(); // IDs in {0..9}  
3   for (j = i*109+1, j<(i+1)*109; j++) {  
4     if (isPrime(j))  
5       print(j);  
6   }  
7 }
```

code



Implémentation de la procédure pour le processus i

33

```
1 int counter = new Counter(1);
2
3 void primePrint {
4     long j = 0;
5     while (j < 1010) {
6         j = counter.getAndIncrement();
7         if (isPrime(j))
8             print(j);
9     }
10 }
11
```

on s'arrête lorsque chaque
valeur est prise

Implémentation de la procédure pour le processus i

34

```
1 int counter = new Counter(1);
2
3 void primePrint {
4     long j = 0;
5     while (j < 1010) {
6         j = counter.getAndIncrement();
7         if (isPrime(j))
8             print(j);
9     }
10 }
11
```

← **Incrémenter et renvoyer
chaque nouvelle valeur**

Implémentation concurrente

35

```
1 public class Counter {  
2     private long value;  
3  
4     public long getAndIncrement() {  
5         temp = value;  
6         value = temp + 1;  
7         return temp;  
8     }  
9 }  
10
```

Implémentation concurrente

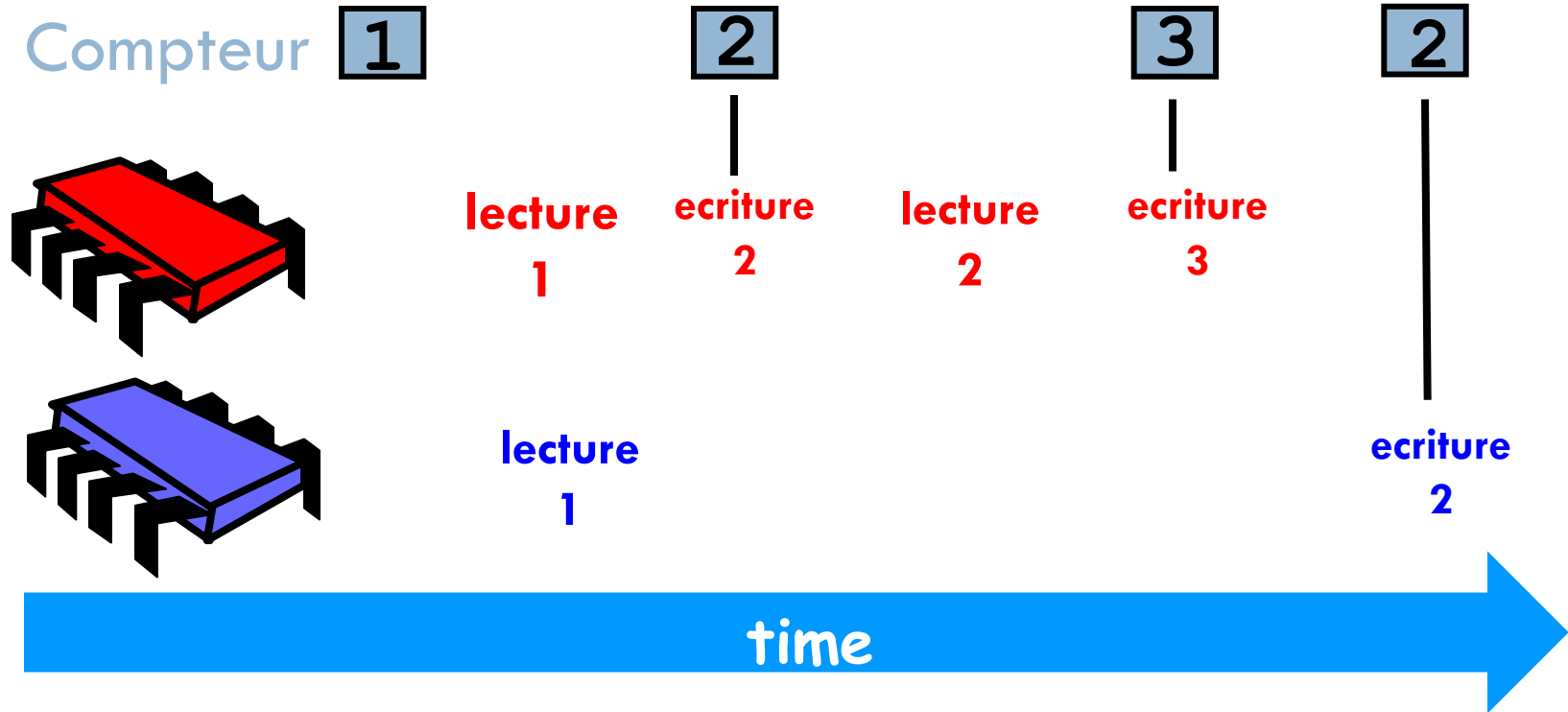
36

```
1 public class Counter {  
2     private long value;  
3  
4     public long getAndIncrement() {  
5         return value++;  
6     }  
7 }  
8
```

**OK for single process,
not for concurrent processes**

Not so good...

37



Challenge

38

```
1 public class Counter {  
2     private long value;  
3  
4     public long getAndIncrement() {  
5         temp = value;  
6         value = temp + 1;  
7         return temp;  
8     }  
9 }  
10
```

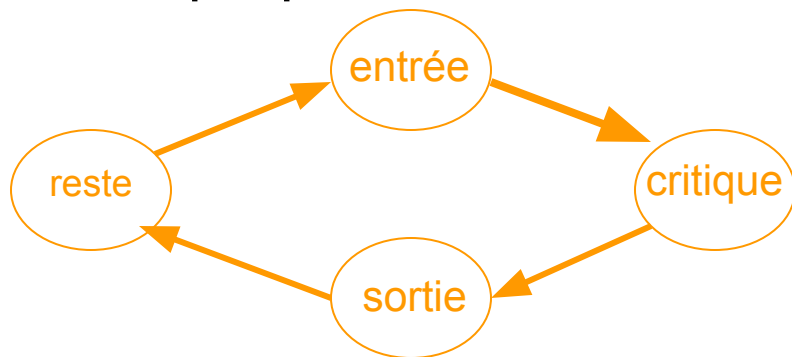
**Rendre ces étapes atomiques
(indivisibles)**

Le problème consistant à garantir qu'une seule personne à la fois peut exécuter un bloc de code particulier est appelé exclusion mutuelle.

Problème d'exclusion mutuelle (Mutex)

39

- Le code de chaque processeur est divisé en quatre sections :



- **entrée:** se synchronisent avec d'autres pour assurer un accès mutuellement exclusif à la...
- **critique:** utilise une ressource ; une fois terminé, entrer dans la...
- **sortie:** nettoie ; quand c'est fait, entrez dans le...
- **reste:** pas intéressé par l'utilisation de la ressource


Algorithmes d'exclusion mutuelle

40

- ❑ Un *algorithme d'exclusion mutuelle* spécifie un code pour les sections d'entrée et de sortie afin de garantir :
 - ❑ **exclusion mutuelle**: un processeur au maximum se trouve dans sa section critique à tout moment, et
 - ❑ une sorte de condition de "**vivacité**" ou de "**progrès**". Il y a trois conditions communément considérées...

Conditions de vivacité des mutex

41

- 
- **pas de blocage (*deadlock-freedom*)**: si un processeur se trouve dans sa section d'entrée à un moment donné, un autre processeur se trouvera plus tard dans sa section critique.
 - **pas de verrouillage (*starvation-freedom*)**: si un processeur se trouve dans sa section d'entrée à un moment donné, ce même processeur se trouve plus tard dans sa section critique
 - **attente bornée**: pas de verrouillage + pendant qu'un processeur est dans sa section d'entrée, les autres processeurs n'entrent pas dans la section critique plus d'un certain nombre de fois.
 - Ces conditions sont de plus en plus fortes.

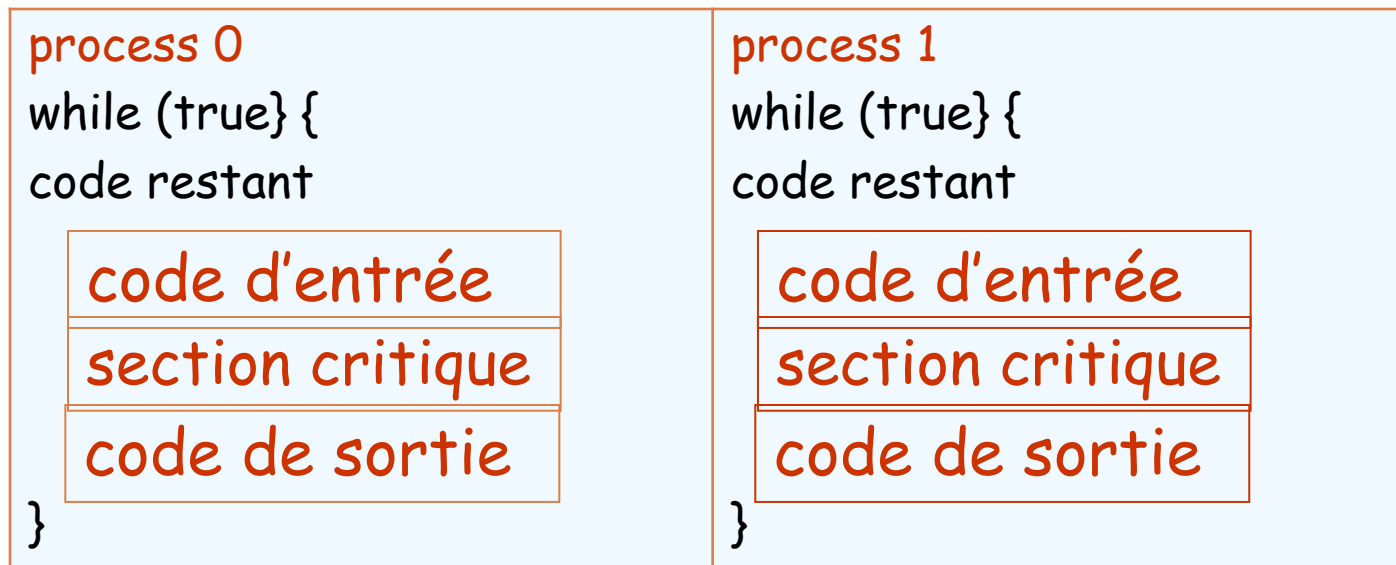
Résoudre l'exclusion mutuelle : Hypothèses

42

- Rien n'est supposé sur le code restant, sauf qu'il ne peut pas influencer le comportement des autres processus.
- Les objets partagés apparaissant dans un code d'entrée ou de sortie ne peuvent pas être mentionnés dans un code de reste ou une section critique.
- Un processus ne peut pas échouer (c'est-à-dire s'arrêter) lors de l'exécution du code d'entrée, de la section critique et du code de sortie.
- Un processus ne peut effectuer qu'un nombre fini d'étapes dans sa section critique et son code de sortie.
- Les processus individuels sont séquentiels et asynchrones.

Convention

43

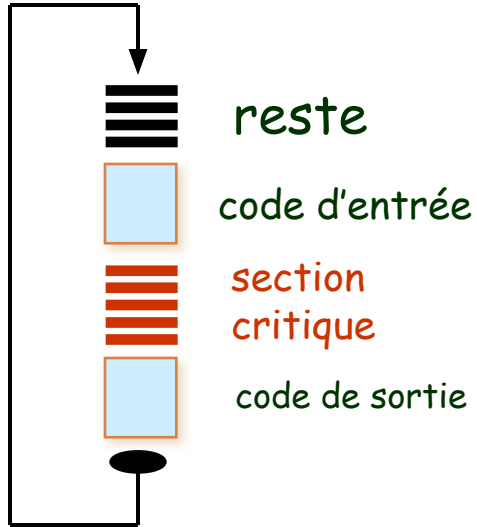


- Pour simplifier la présentation, seuls les codes d'entrée et de sortie sont décrits.

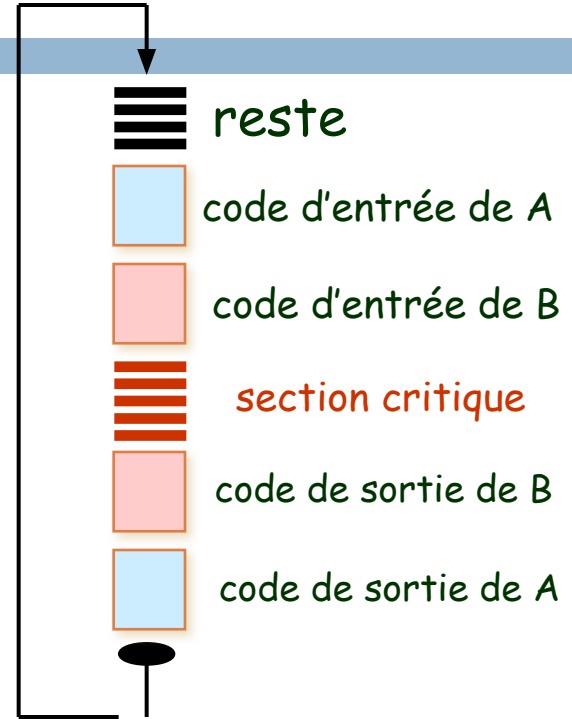
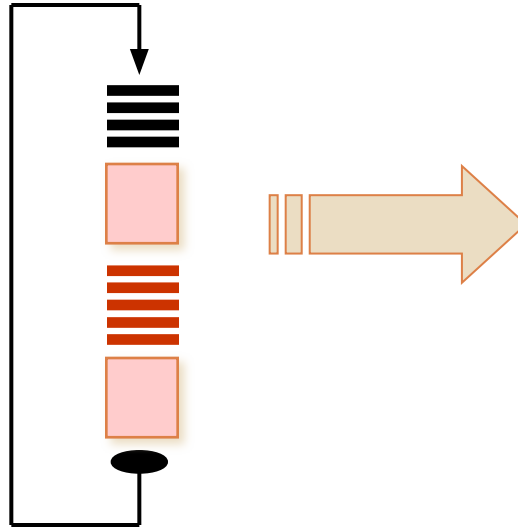
Question: vrai ou faux ?

44

Algorithme A



Algorithme B



45

Algorithmes pour deux processus

Algorithme de Peterson (1981)

46

process 0

$b[0] := \text{true}$

$\text{turn} := 0$

await($b[1]=\text{false}$ or $\text{turn} = 1$)

section critique

$b[0] := \text{false}$

process 1

$b[1] := \text{true}$

$\text{turn} := 1$

await ($b[0]=\text{false}$ or $\text{turn} = 0$)

section critique

$b[1] := \text{false}$

await(x)= attendre tant
que la condition x n'est
pas vérifiée

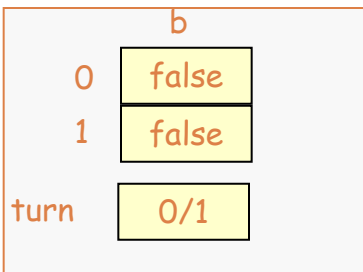
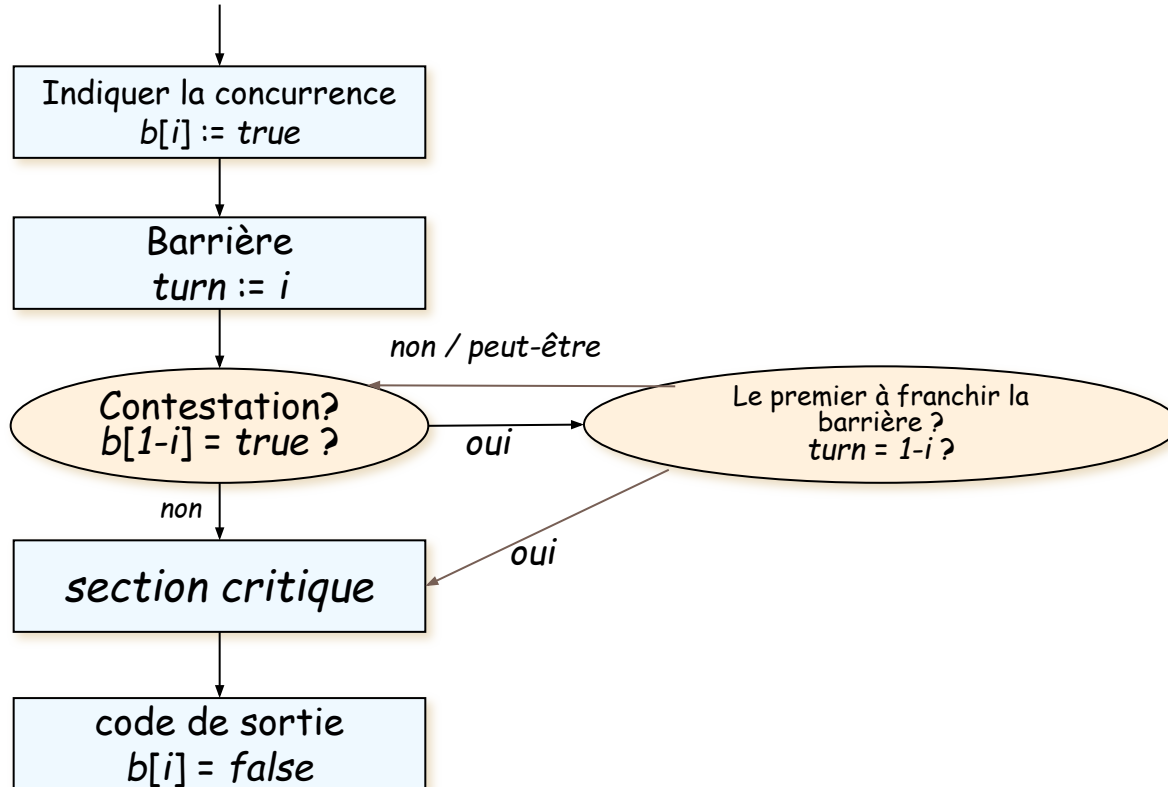


Schéma de l'algorithme de Peterson

47



Propriétés de la solution de Peterson

48

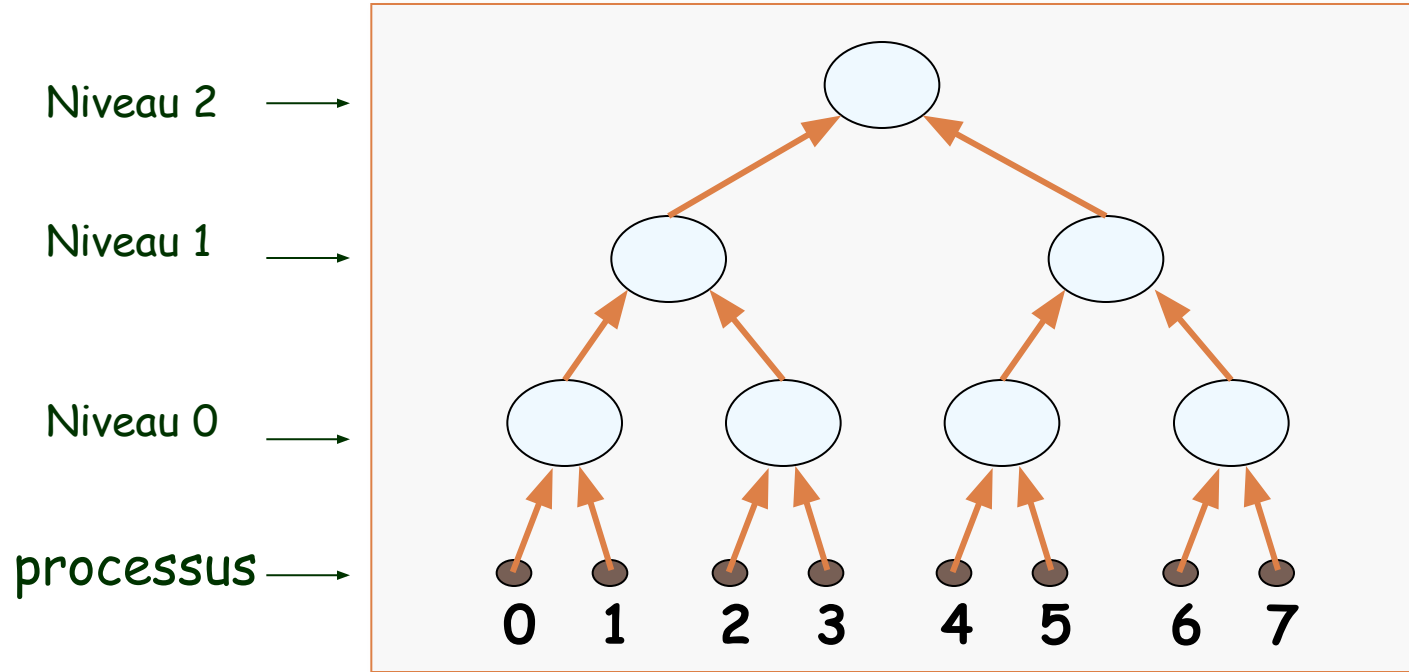
- Satisfait l'**exclusion mutuelle** et la *starvation-freedom (pas de verrouillage)*
- Les accès à la mémoire sont considérés comme atomiques
- Solution pour deux processus seulement

Algorithmes pour plusieurs processus

Comment peut-on utiliser un
algorithme à deux processus pour
construire un algorithme pour de
nombreux processus ?

Algorithmes de tournois

50



L'algorithme du tournoi basé sur l'algorithme de Peterson

51

- ❑ Solution pour n processus
- ❑ Pour simplifier, on suppose que le nombre de processus n est une puissance de deux.
- ❑ Les processus sont numérotés de 0 à $n-1$
- ❑ À chaque niveau de l'arbre, les nœuds sont numérotés de gauche à droite en commençant par 0.
 - ❑ Chaque nœud de l'arbre est identifié de manière unique par son **niveau** et son **numéro** de nœud.

Un algorithme de tournoi basé sur l'algorithme de Peterson : code du processus i , $i \in \{0, \dots, n-1\}$.

52

```
1 node := i
2 for level = 0 to log2(n) - 1 do {
3     id := node mod 2
4     node := node/2
5     b[level, 2node+id] := true
6     turn[level, node] := id
7     await (b[level, 2node+1-id] = false or turn[level, node] = 1-id[level])
8 }
9 critical section
10 for level = log2(n)-1 to 0 do {
11     node := i/2level+1
12     b[level, node] := false
13 }
14
```

Propriétés de l'algorithme de tournoi basé sur l'algorithme de Peterson

53

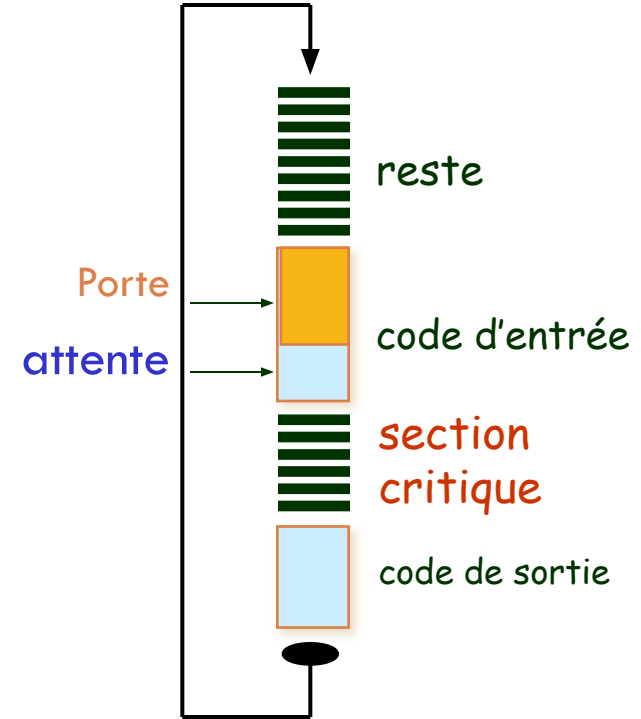
- Satisfait l'exclusion mutuelle et starvation-freedom
- Les accès à la mémoire sont considérés comme atomiques
- Solution pour n processus

Algorithmes d'exclusion mutuelle FIFO

Exclusion mutuelle FIFO

55

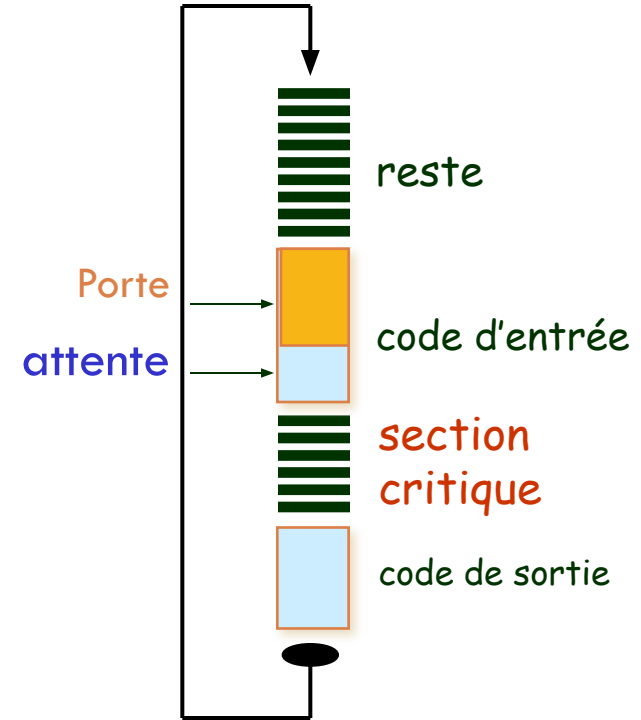
- Le code d'entrée se compose de deux parties:
 1. **Porte** : le code avant la déclaration d'attente.
 2. **Déclaration d'attente**: une boucle qui comprend une ou plusieurs déclarations
- La porte est **sans attente** : son exécution ne nécessite qu'un nombre limité d'étapes (elle se termine toujours).



Le problème de l'exclusion mutuelle

56

- ❑ **Processus d'attente:** un processus qui attend l'instruction d'attente dans son code d'entrée.
- ❑ **First-in-First-Out (FIFO):** Un processus en attente pourra entrer dans sa section critique avant que chacun des autres processus puisse entrer dans sa section critique 1 fois.
 - ❑ une instance d'attente bornée
 - ❑ aucun début de processus ne peut dépasser un processus en attente de lecture.



Implémentation 1

code du processus i , $i \in \{1, \dots, n\}$

57

```
1 number[i] = 1 + max {number[j] | (1 ≤ j ≤ n)}
2 for j = 1 to n {
3     await (number[j] = 0) ∨ (number[j] > number[i])
4 }
5 critical section
6 number[i] = 0
7
```

	1	2	3	4		n
number	0	0	0	0	0	0

Donner une exécution de cet algorithme pour montrer qu'il ne garantit pas la propriété d'absence d'inter-blocages (deadlock-freedom)

Implémentation 1

code du processus i , $i \in \{1, \dots, n\}$

58

```
1 number[i] = 1 + max {number[j] | (1 ≤ j ≤ n)}  
2 for j = 1 to n {  
3     await (number[j] = 0) ∨ (number[j] > number[i])  
4 }  
5 critical section  
6 number[i] = 0  
7
```

Et si on remplace >
avec ≥ ?

	1	2	3	4		n
number	0	0	0	0	0	0

L'implémentation est-elle correcte si on remplace >
avec ≥ ? Justifier la réponse

Implémentation 2

code du processus i , $i \in \{1, \dots, n\}$

59

```
1 number[i] = 1 + max {number[j] | (1 ≤ j ≤ n)}
2 for j = 1 to n {
3     await (number[j] = 0) ∨ ((number[j], j) ≥ (number[i], i))
4 // lexicographical order
5 }
6 critical section
7 number[i] = 0
```

	1	2	3	4	—	n
number	0	0	0	0	0	0

Donner une exécution de cet algorithme pour montrer qu'il ne garantit pas l'exclusion mutuelle