

SYNCHRONISATION DES PROCESSUS

SE

Madame Khaoula ElBedoui-Maktouf

2^{ème} année Ingénieur Informatique

Plan

SE

- I. Communication interprocessus**
- II. Section critique**
- III. Solutions avec attente active**
- IV. Solutions avec attente passive**
- V. Problèmes classiques de communication interprocessus**

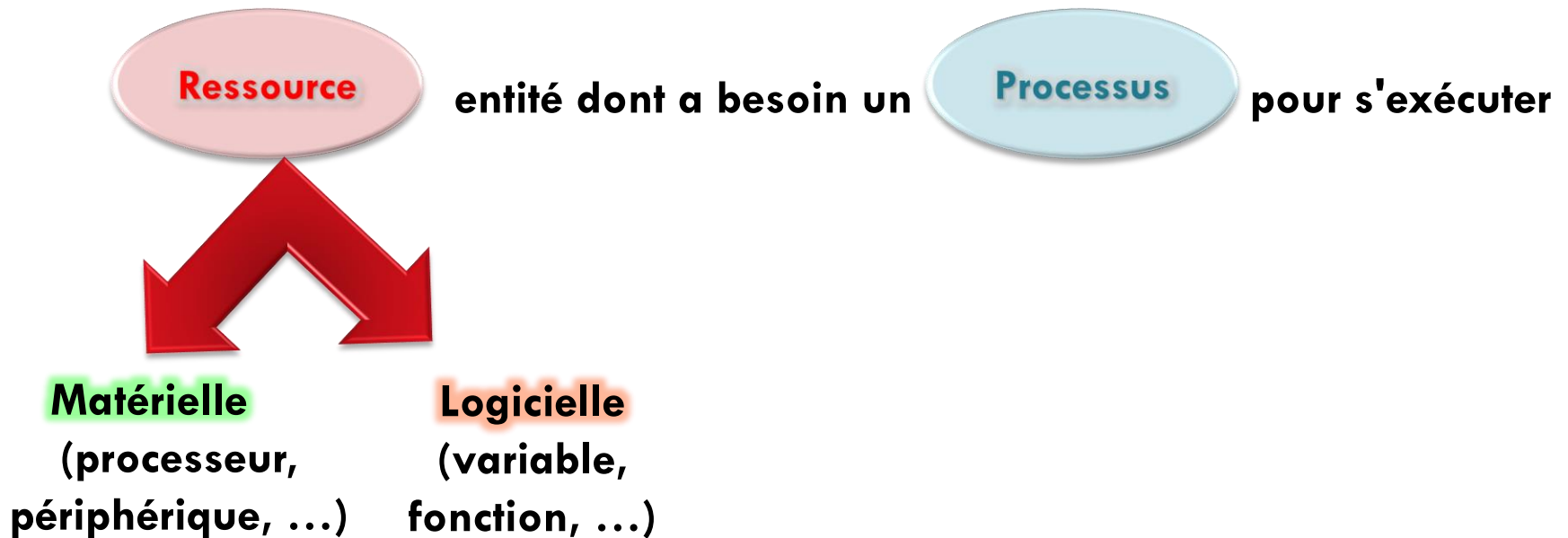
Communication interprocessus

1. Ressources et processus



Communication interprocessus

1. Ressources et processus



Communication interprocessus

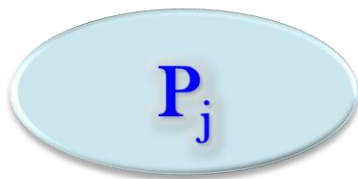
1. Ressources et processus



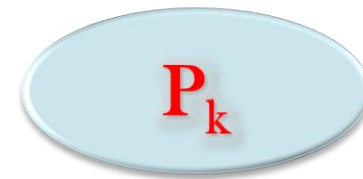
Communication interprocessus

2. Cas de coopération

❖ Utiliser des outils de Synchronisation



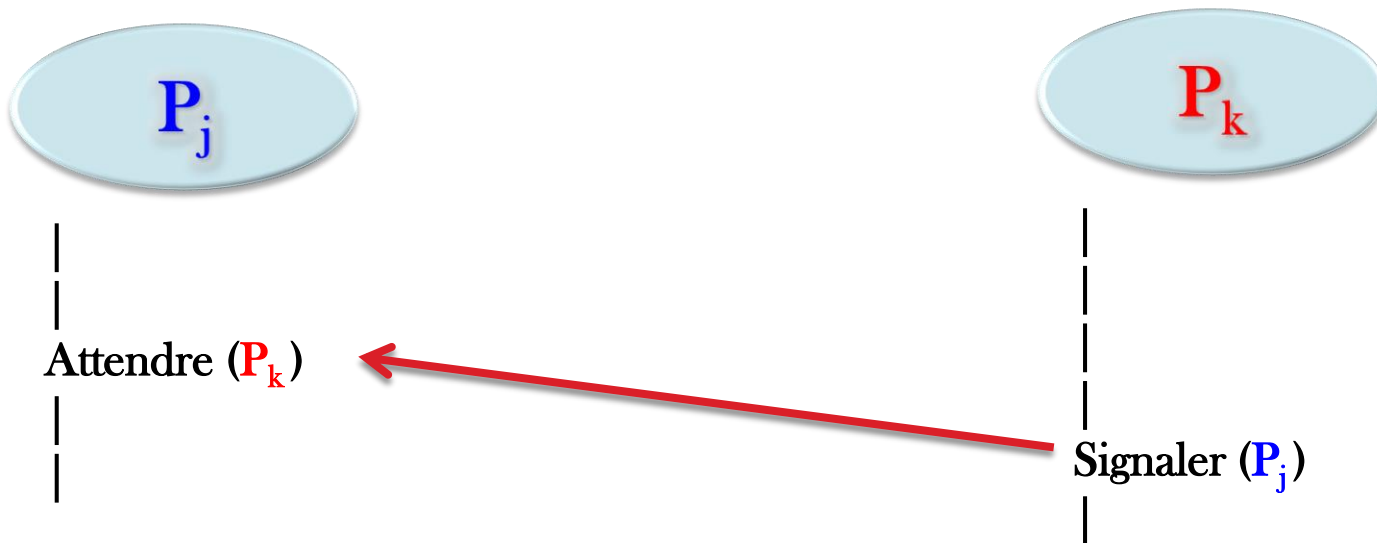
Attendre (P_k)



Communication interprocessus

2. Cas de coopération

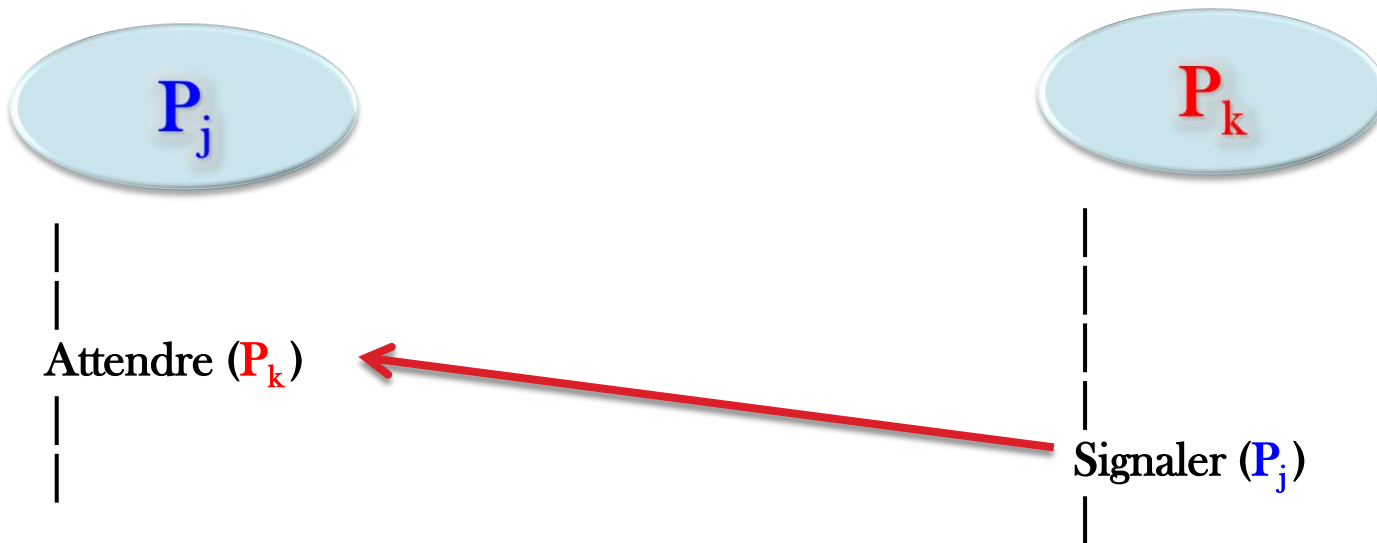
❖ Utiliser des outils de Synchronisation



Communication interprocessus

2. Cas de coopération

❖ Utiliser des outils de Synchronisation

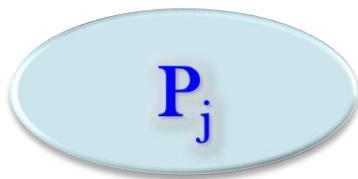


Exp. &&

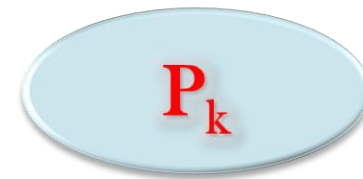
Communication interprocessus

2. Cas de coopération

❖ Utiliser des outils de Communication



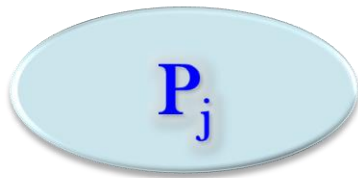
Envoyer (P_k , msg)



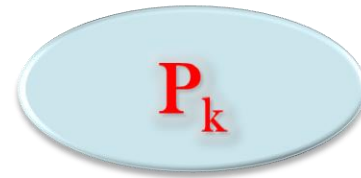
Communication interprocessus

2. Cas de coopération

❖ Utiliser des outils de Communication



Envoyer (P_k , msg)

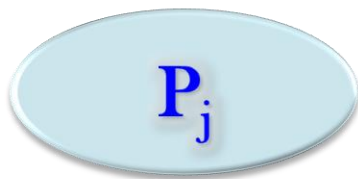


Recevoir (P_j , msg)

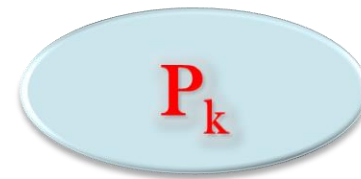
Communication interprocessus

2. Cas de coopération

❖ Utiliser des outils de Communication



Envoyer (P_k , msg)



Recevoir (P_j , msg)

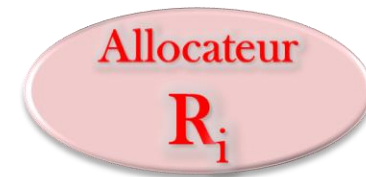
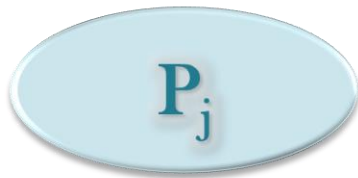


Exp. Pipe : `cmd1 | cmd2`

Communication interprocessus

3. Cas de concurrence

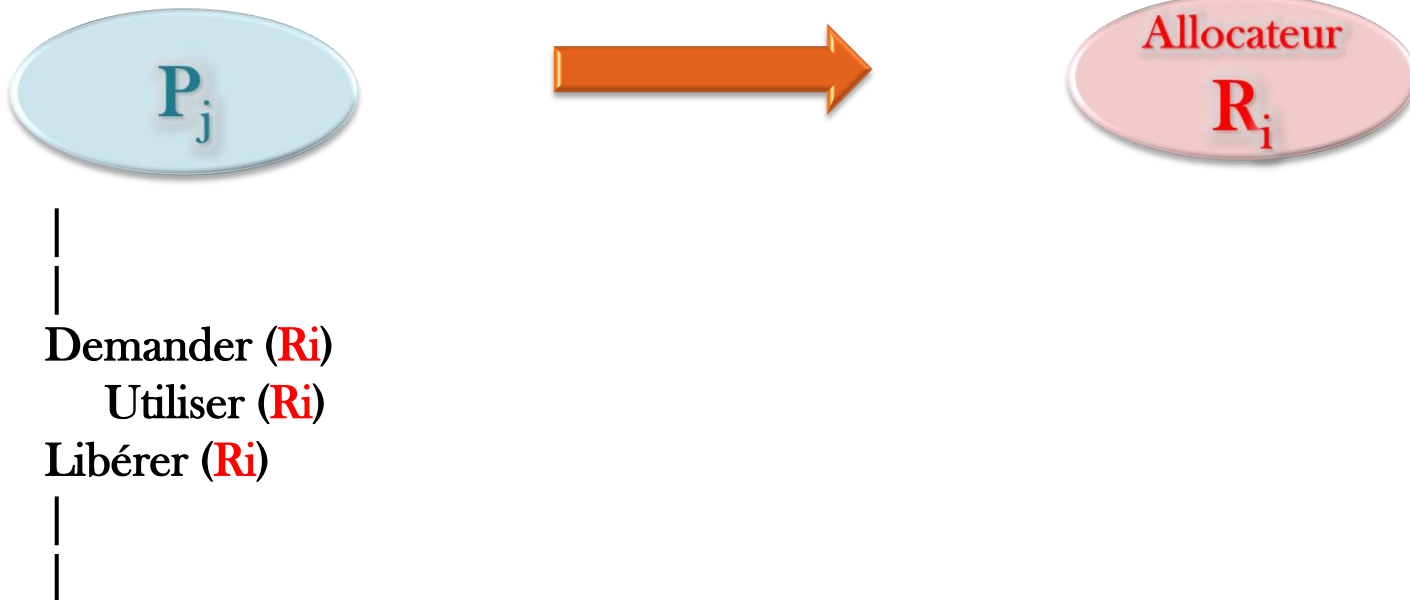
❖ Ressource gérée par un allocateur



Communication interprocessus

3. Cas de concurrence

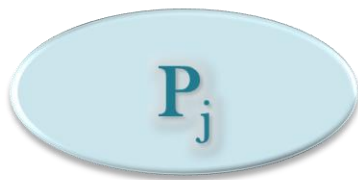
❖ Ressource gérée par un allocateur



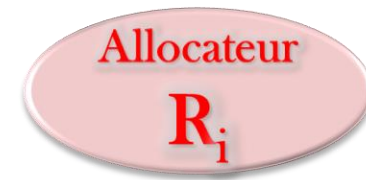
Communication interprocessus

3. Cas de concurrence

❖ Ressource gérée par un allocateur



|
|
Demander (R_i)
Utiliser (R_i)
Libérer (R_i)
|
|

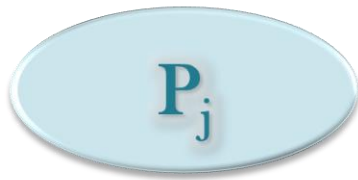


|
|
Si R_i occupée alors
Bloquer (P_j)
Sinon Honorer (P_j)
|
|

Communication interprocessus

3. Cas de concurrence

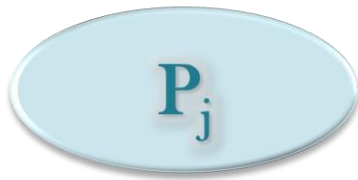
❖ Ressource NON gérée par un allocateur



Communication interprocessus

3. Cas de concurrence

❖ Ressource NON gérée par un allocateur



Les processus appliquent entre eux une règle
exp. Tour de rôle, FIFO, ...

Communication interprocessus

4. Problème de concurrence

Gestion d'un compte bancaire

Val : entier

Ajout (C: entier)

$Val = Val + C$

Retrait (d: entier)

Si ($Val < d$)

Ecrire (" impossible ")

Sinon

$Val = Val - d$

Communication interprocessus

4. Problème de concurrence

Gestion d'un compte bancaire

Val : entier

Ajout (C: entier)

1

$Val = Val + C$

Retrait (d: entier)

2

Si ($Val < d$)

3

Ecrire (" impossible ")

4

Sinon

$Val = Val - d$

Communication interprocessus

4. Problème de concurrence

Gestion d'un compte bancaire

Val = 10

Machine monoprocesseur
Ordonnancement préemptif

Retrait (9)

Retrait (8)

Retrait (d: entier)

2 Si (Val < d)

3 Ecrire (" impossible ")

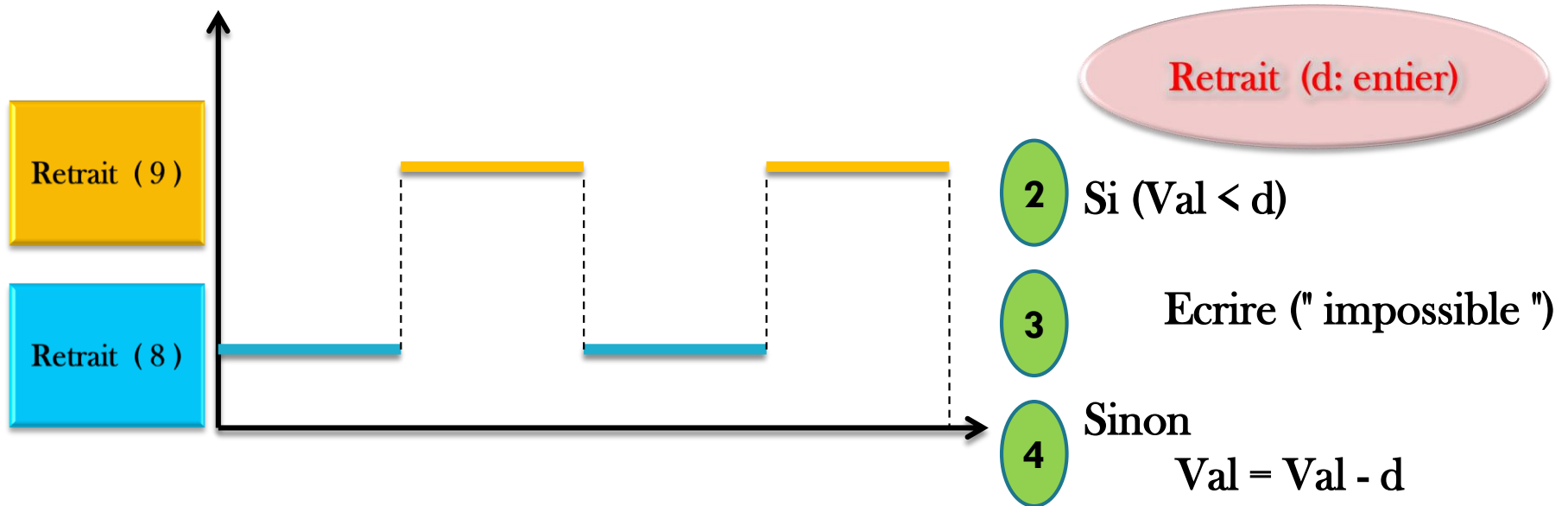
4 Sinon
Val = Val - d

Communication interprocessus

4. Problème de concurrence

Gestion d'un compte bancaire

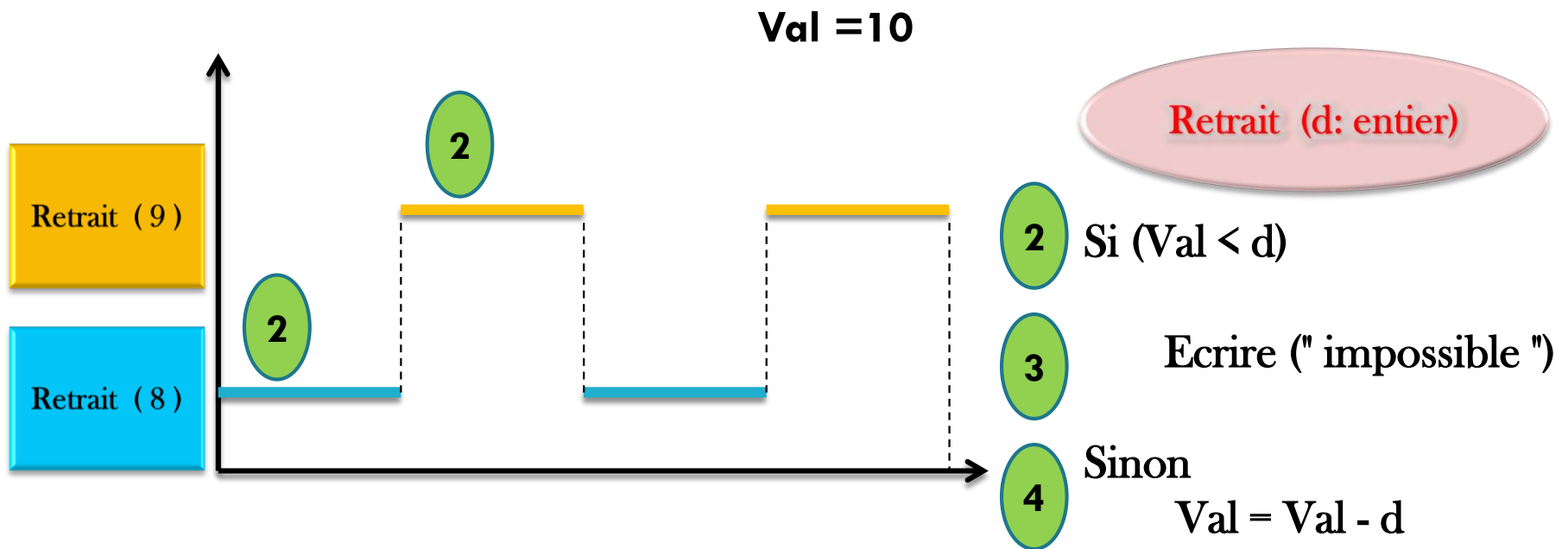
Val = 10



Communication interprocessus

4. Problème de concurrence

Gestion d'un compte bancaire

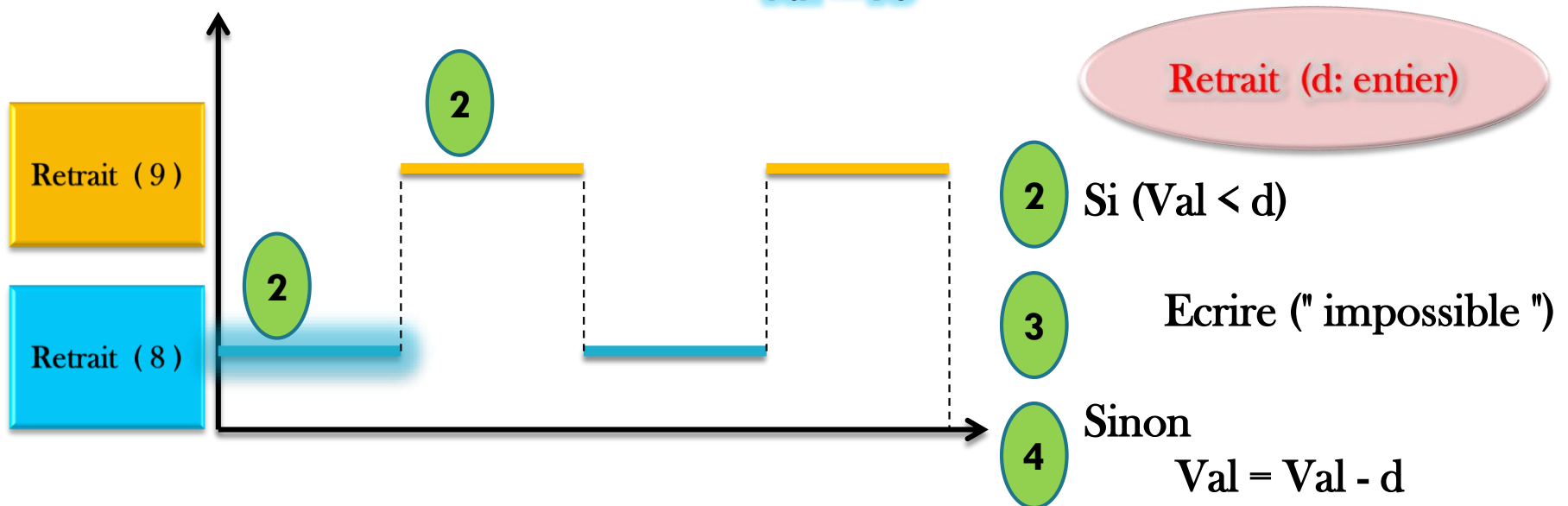


Communication interprocessus

4. Problème de concurrence

Gestion d'un compte bancaire

Val = 10



Communication interprocessus

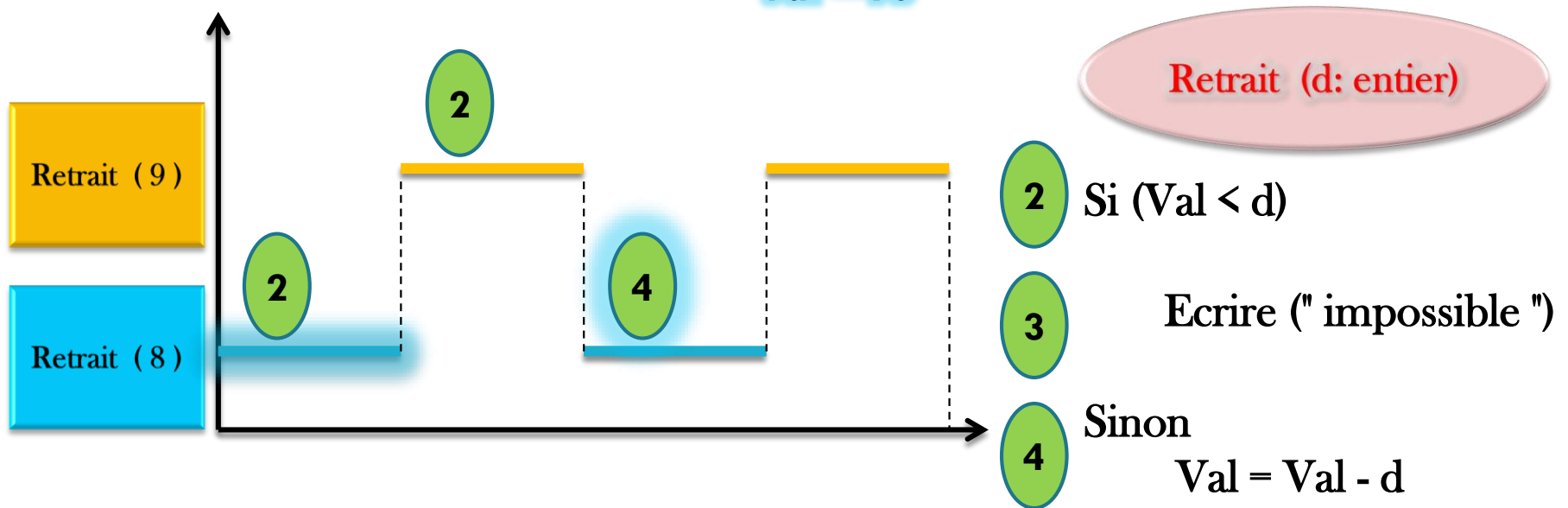
SE

Chap 6. Synchronisation

4. Problème de concurrence

Gestion d'un compte bancaire

Val = 10



Communication interprocessus

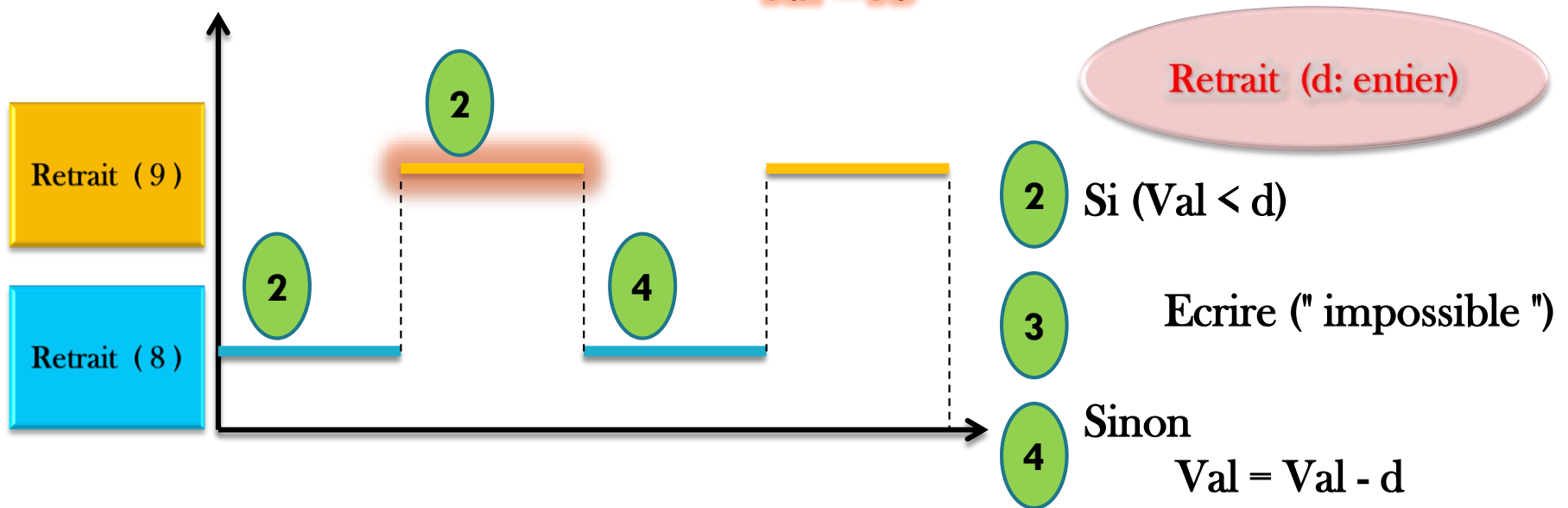
SE

Chap 6. Synchronisation

4. Problème de concurrence

Gestion d'un compte bancaire

Val = 10



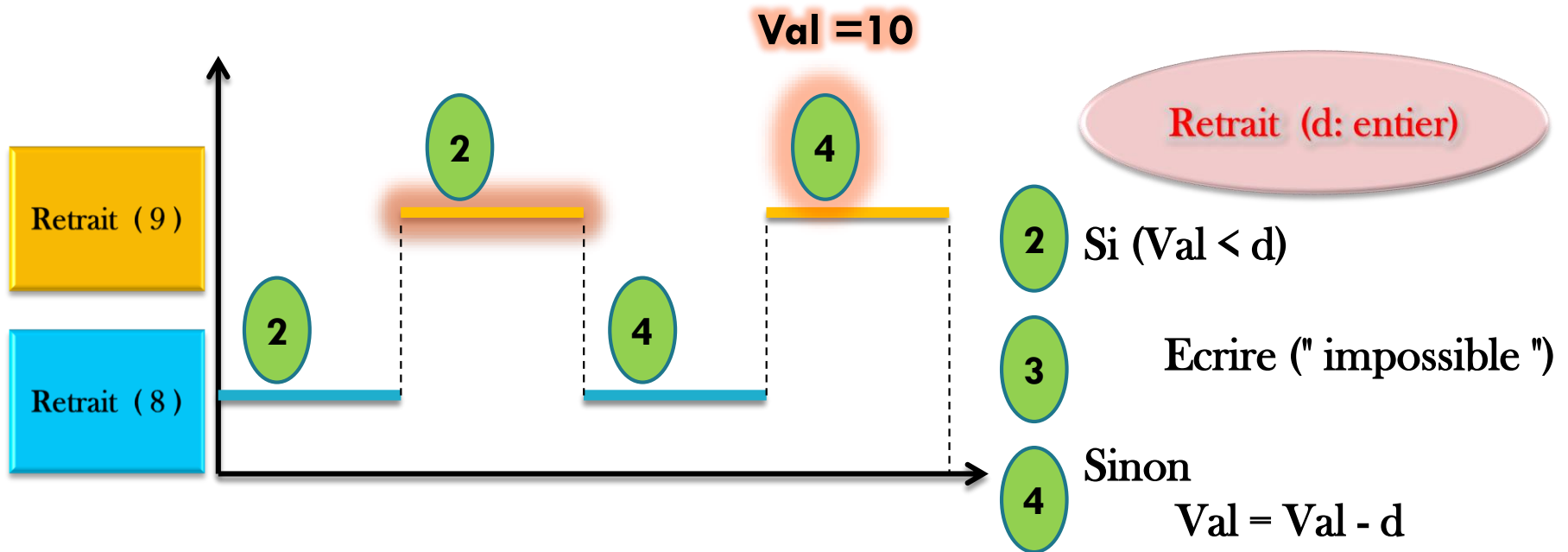
Communication interprocessus

SE

Chap 6. Synchronisation

4. Problème de concurrence

Gestion d'un compte bancaire



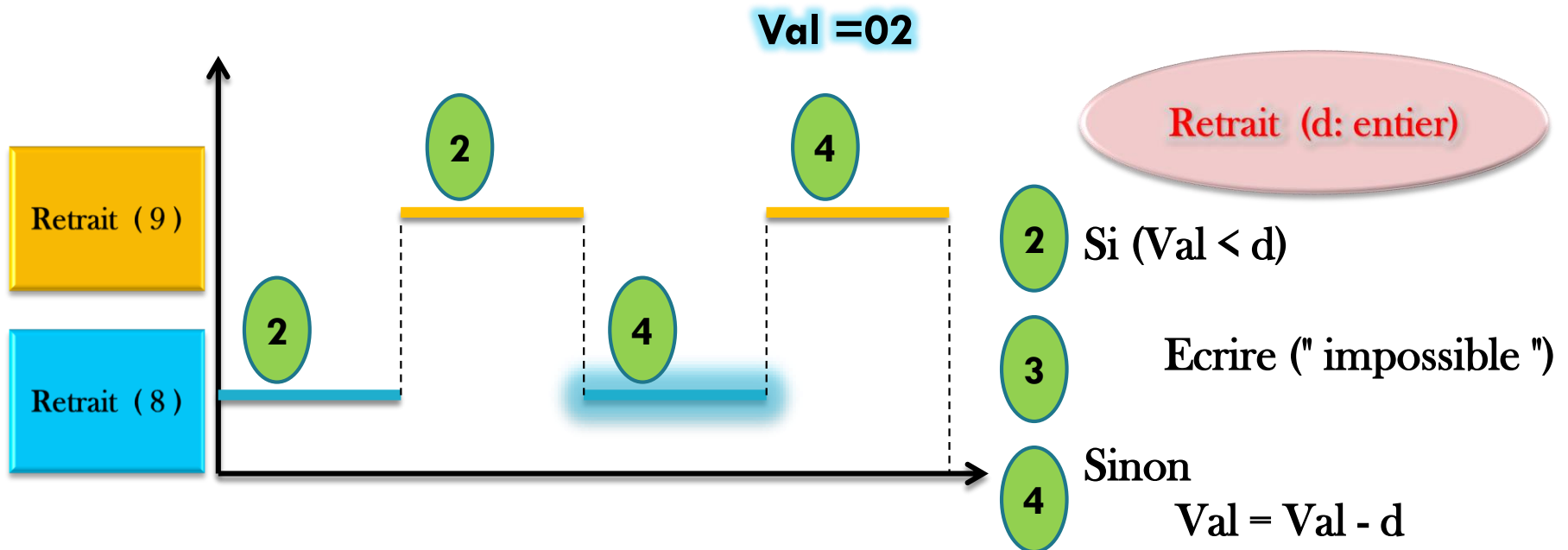
Communication interprocessus

SE

Chap 6. Synchronisation

4. Problème de concurrence

Gestion d'un compte bancaire



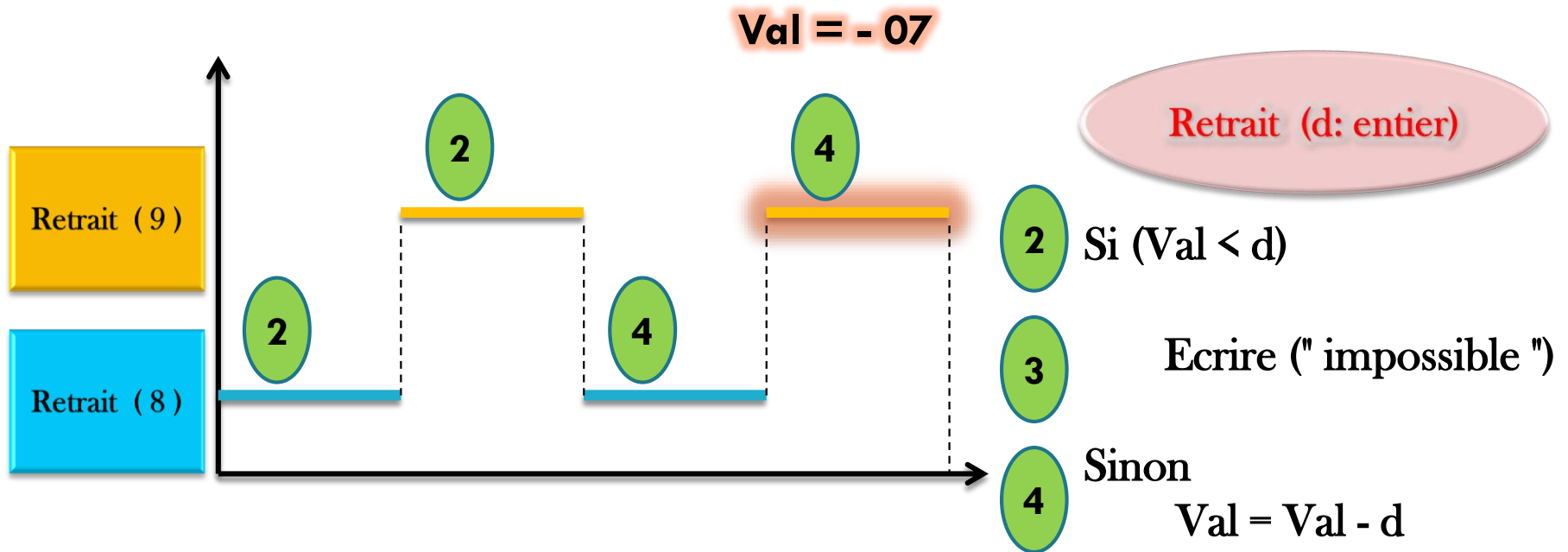
Communication interprocessus

SE

Chap 6. Synchronisation

4. Problème de concurrence

Gestion d'un compte bancaire



Communication interprocessus

SE

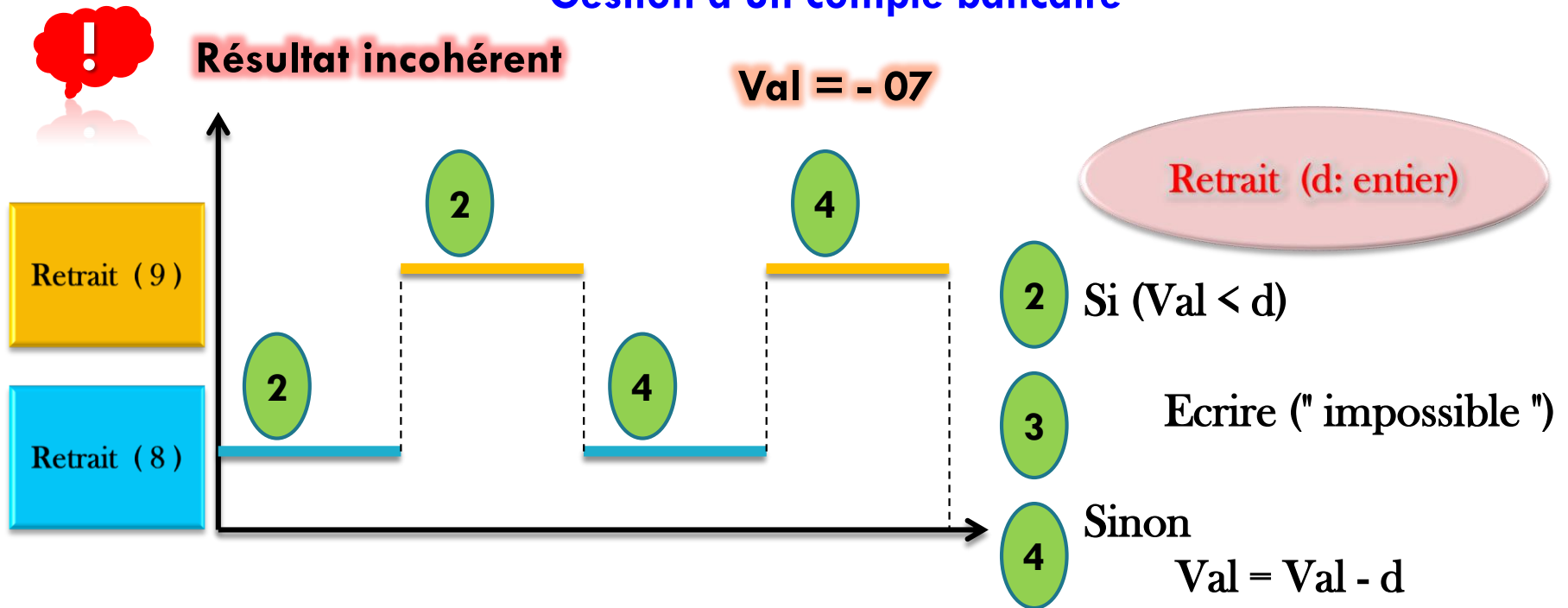
Chap 6. Synchronisation

4. Problème de concurrence

Gestion d'un compte bancaire

Résultat incohérent

Val = - 07



Communication interprocessus

4. Problème de concurrence

Gestion d'un compte bancaire



Résultat incohérent

Ce problème est **un problème d'accès concurrent** à une ressource partagée (Val)

Retrait (d: entier)

2

Si ($Val < d$)

3

Ecrire (" impossible ")

4

Sinon

$Val = Val - d$

Communication interprocessus

4. Problème de concurrence

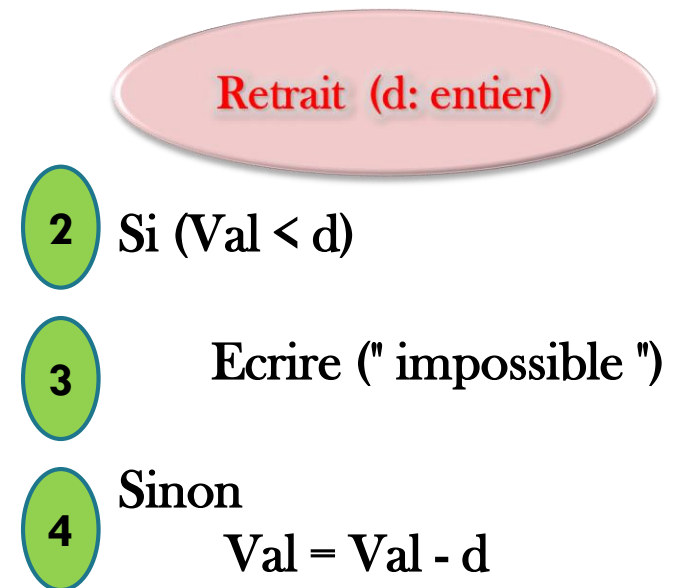
Gestion d'un compte bancaire



Résultat incohérent

Ce problème est **un problème d'accès concurrent** à une ressource partagée (Val)

La solution consiste à interdire la modification de la variable partagée par plus qu'un processus



Communication interprocessus

4. Problème de concurrence

Gestion d'un compte bancaire



Résultat incohérent

Ce problème est **un problème d'accès concurrent** à une ressource partagée (Val)

La solution consiste à interdire la modification de la variable partagée par plus qu'un processus

Il faut définir une section du programme qui ne doit pas être interrompue : **Section critique**

Retrait (d: entier)

2

Si ($Val < d$)

3

Ecrire (" impossible ")

4

Sinon

$Val = Val - d$

Communication interprocessus

4. Problème de concurrence

Gestion d'un compte bancaire

un problème d'accès concurrent

Retrait (d: entier)

La solution

Section critique

2

Si ($Val < d$)

3

Ecrire (" impossible ")

4

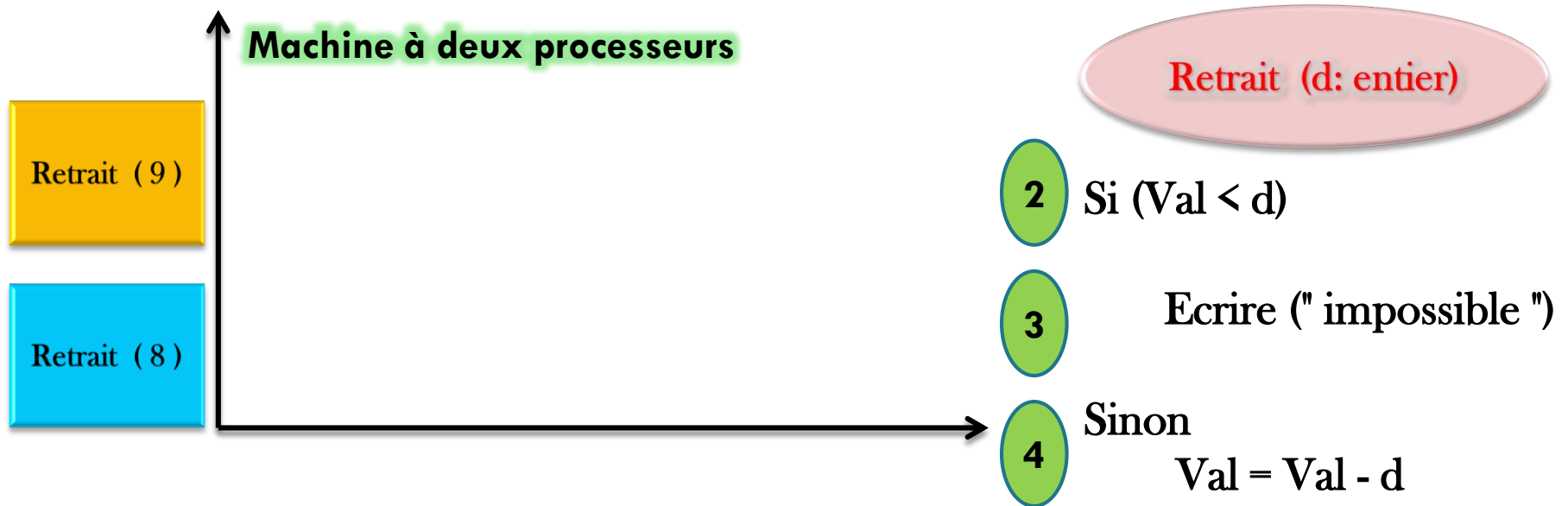
Sinon

$Val = Val - d$

Communication interprocessus

4. Problème de concurrence

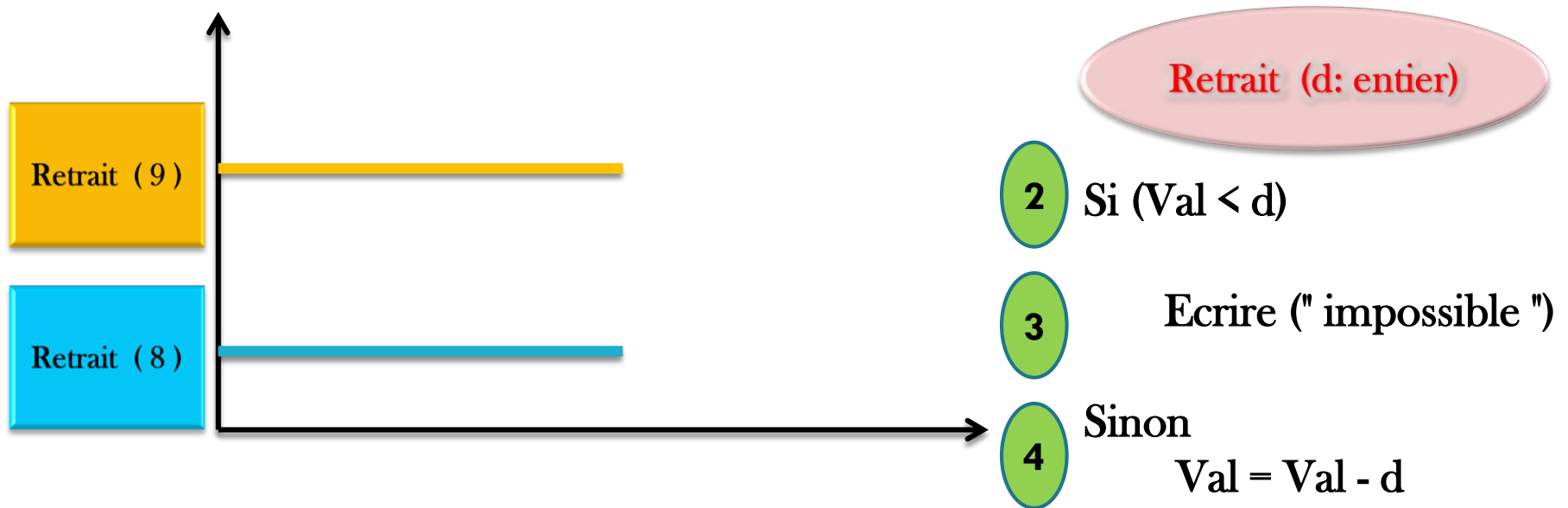
Gestion d'un compte bancaire



Communication interprocessus

4. Problème de concurrence

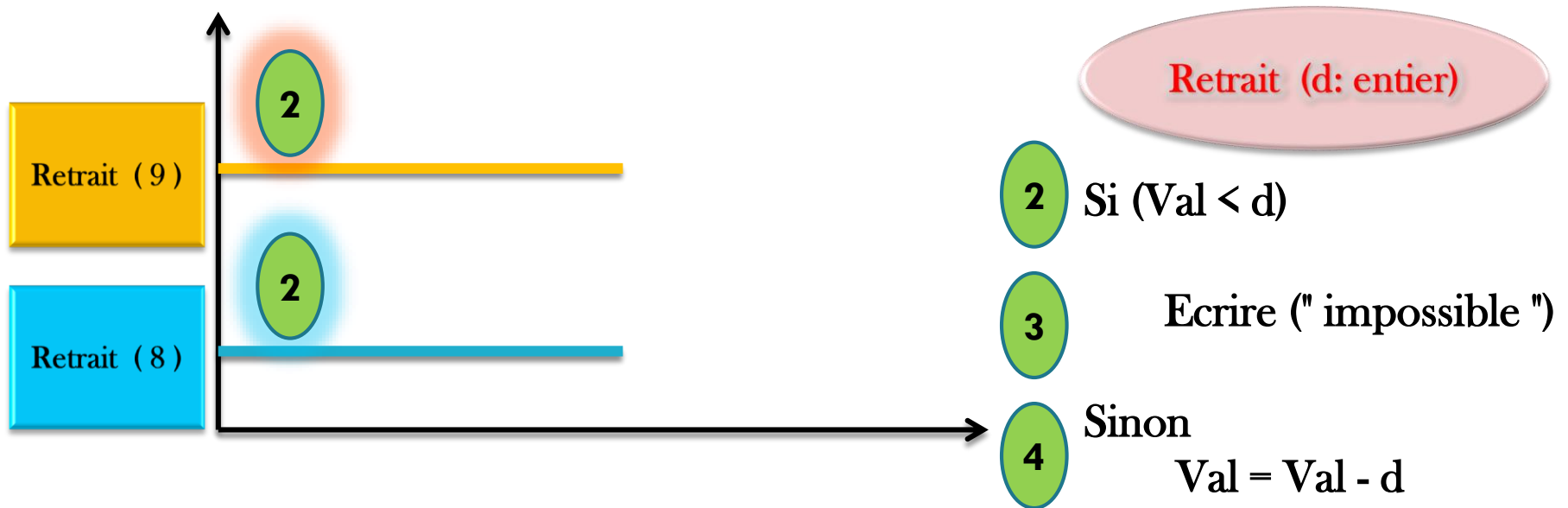
Gestion d'un compte bancaire



Communication interprocessus

4. Problème de concurrence

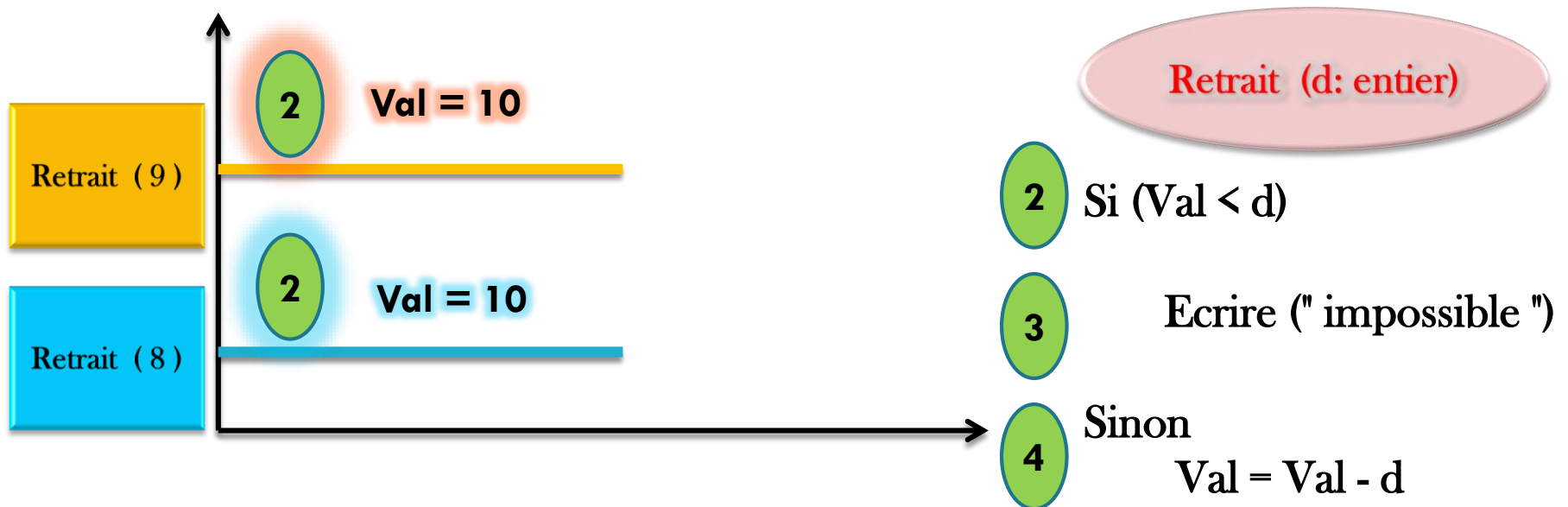
Gestion d'un compte bancaire



Communication interprocessus

4. Problème de concurrence

Gestion d'un compte bancaire



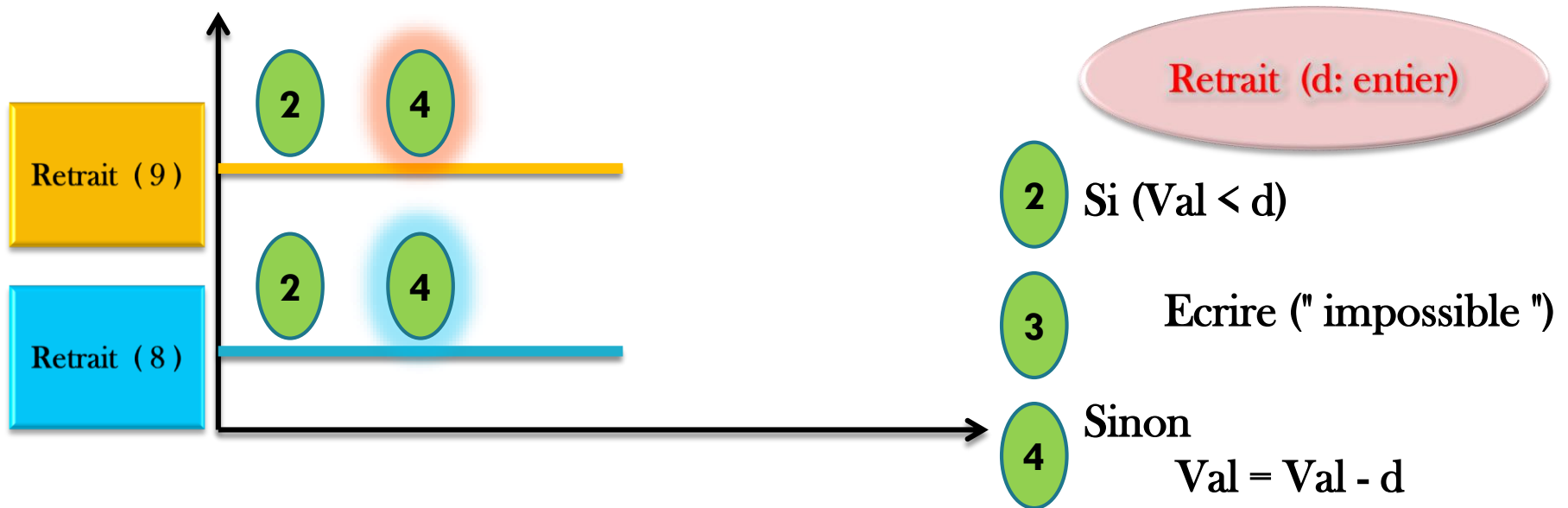
Communication interprocessus

SE

Chap 6. Synchronisation

4. Problème de concurrence

Gestion d'un compte bancaire



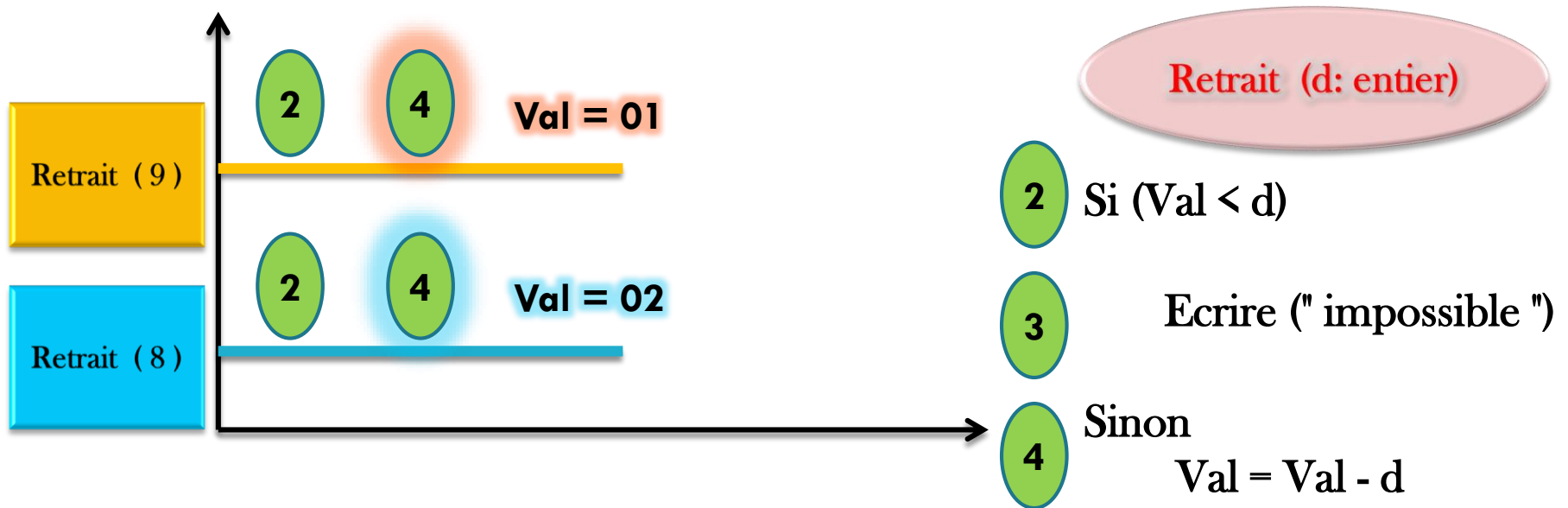
Communication interprocessus

SE

Chap 6. Synchronisation

4. Problème de concurrence

Gestion d'un compte bancaire



Communication interprocessus

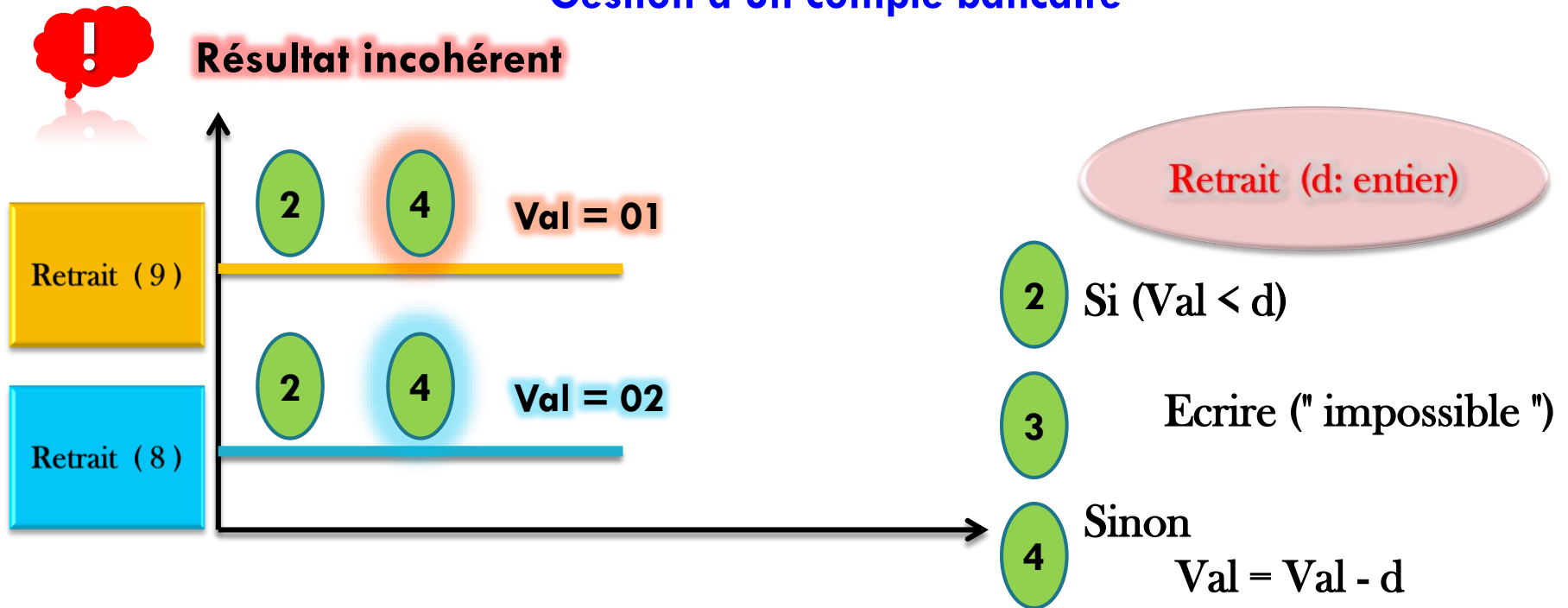
SE

Chap 6. Synchronisation

4. Problème de concurrence

Gestion d'un compte bancaire

Résultat incohérent



Communication interprocessus

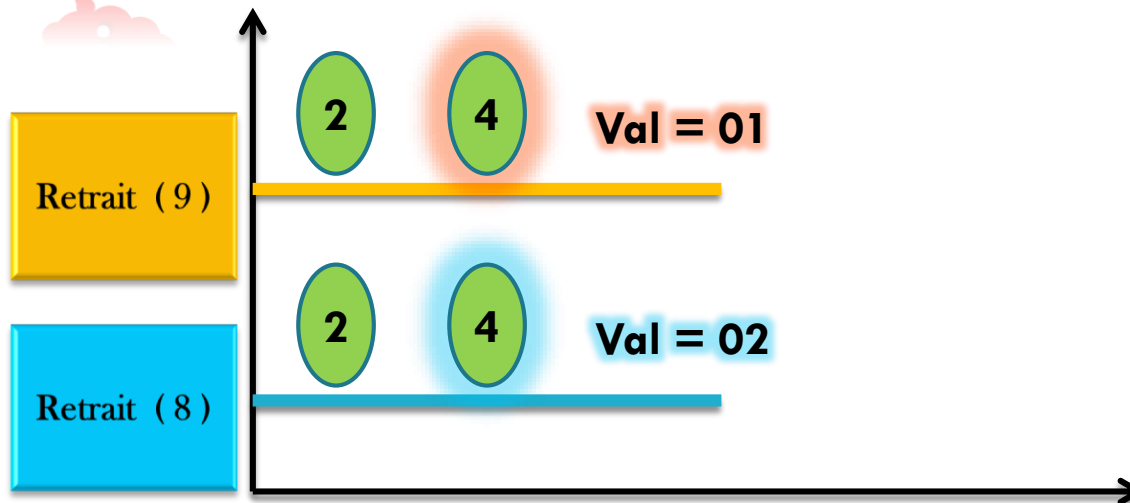
SE

Chap 6. Synchronisation

4. Problème de concurrence

Gestion d'un compte bancaire

 **Résultat incohérent**



Problème d'accès simultané à une ressource partagée (Val)

La solution : les Sections critiques doivent être exécutées en **exclusion mutuelle**

Section critique

1. Définition et objectif

La partie du code utilisant la ressource partagée est dite **section critique**

Section critique

1. Définition et objectif

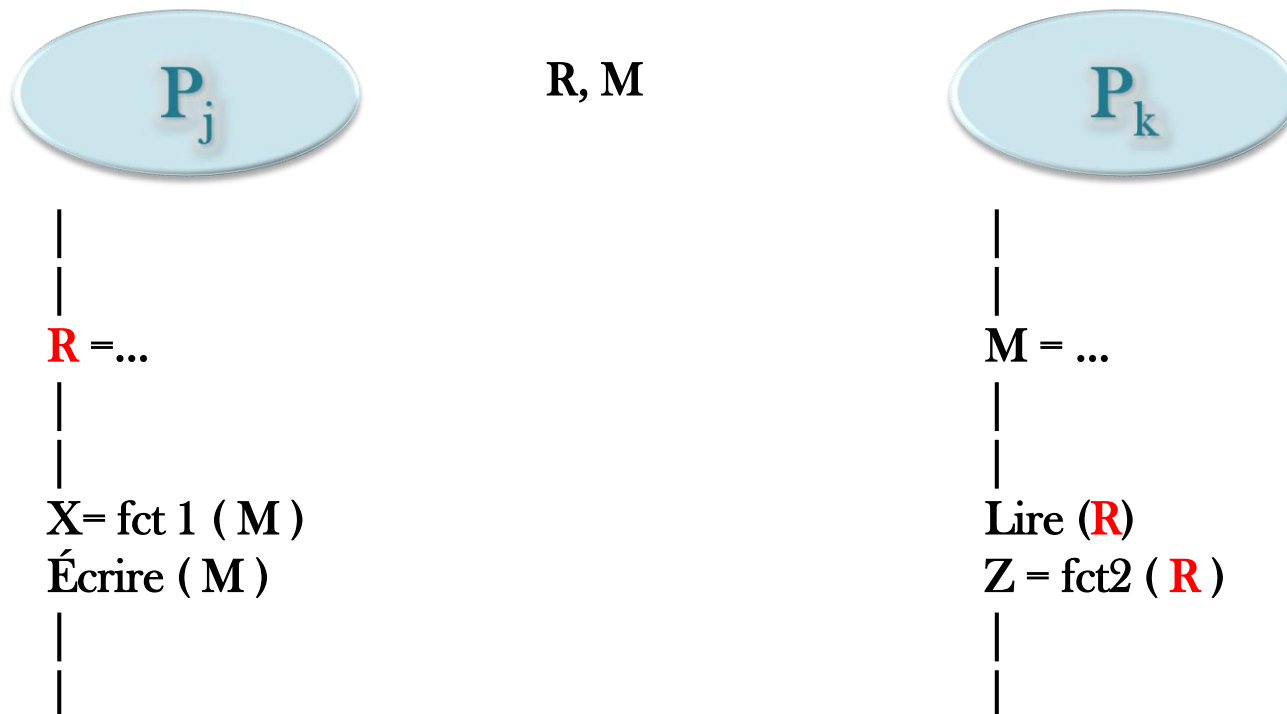
La partie du code utilisant la ressource partagée est dite **section critique**



Section critique

1. Définition et objectif

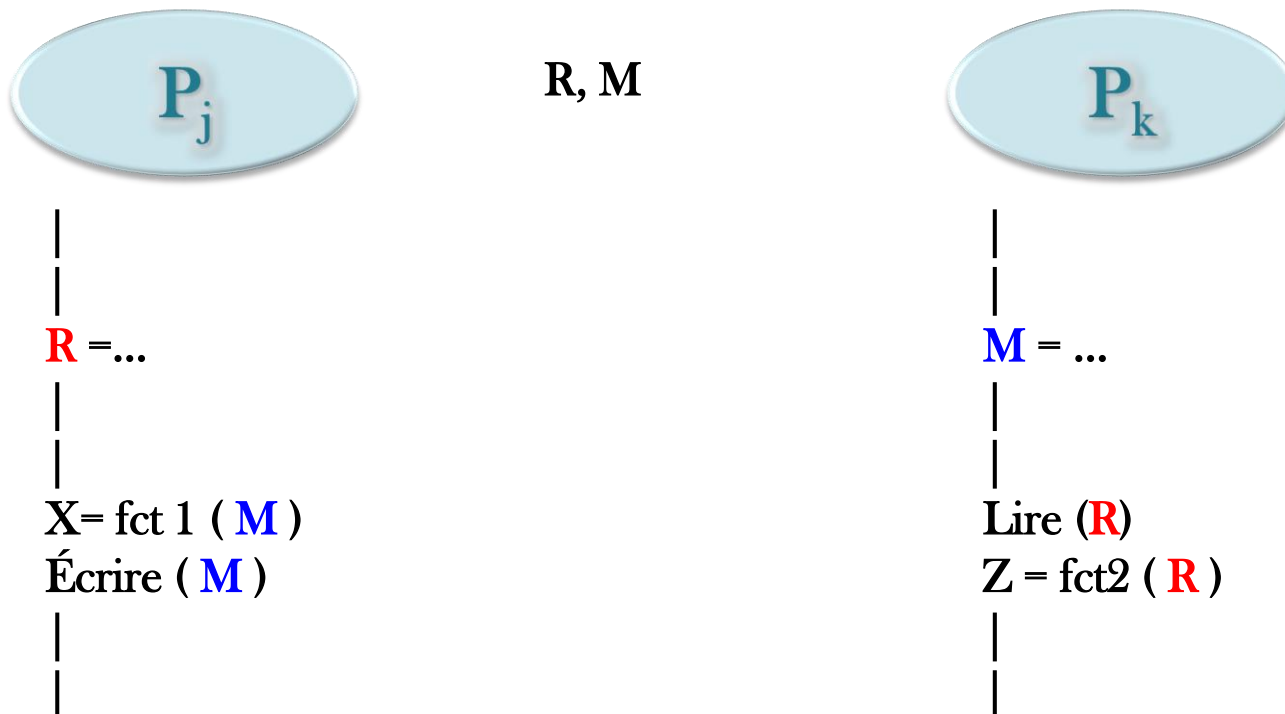
La partie du code utilisant la ressource partagée est dite **section critique**



Section critique

1. Définition et objectif

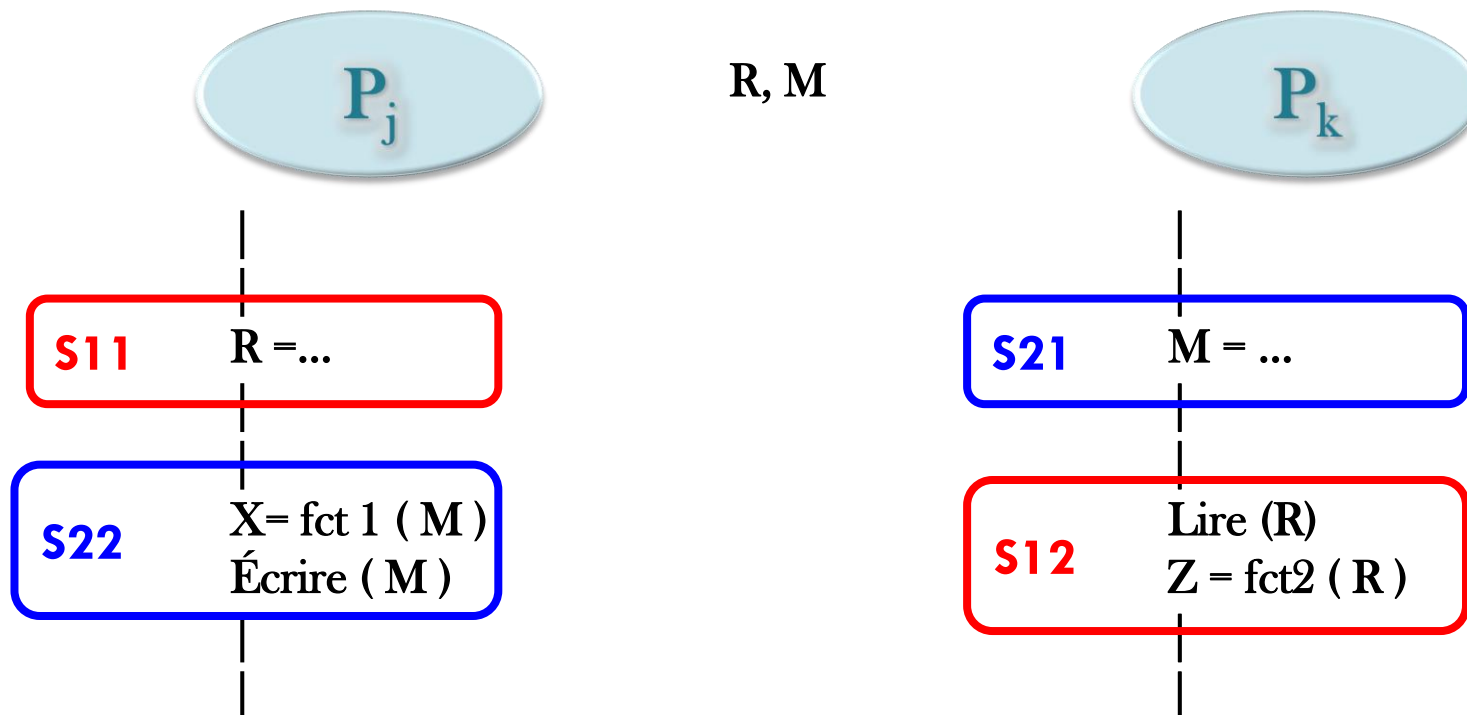
La partie du code utilisant la ressource partagée est dite **section critique**



Section critique

1. Définition et objectif

La partie du code utilisant la ressource partagée est dite **section critique**



Section critique

1. Définition et objectif

Toute solution au problème **section critique** est de la forme

```
|  
|  
Entree_SC( )  
    Section Critique  
Sortie_SC( )  
|  
|
```

Section critique

1. Définition et objectif

Toute solution au problème **section critique** est de la forme

```
|  
|  
| Entree_SC( )  
|   Section Critique  
| Sortie_SC( )  
|  
|
```



Comment doivent êtres ces deux fonctions d'allocation et de libération ?

Section critique

2. Propriétés fondamentales

Toute solution au problème **section critique** doit assurer les 5 PF suivantes

Section critique

2. Propriétés fondamentales

Toute solution au problème **section critique** doit assurer les 5 PF suivantes

1) **Unicité (exclusion mutuelle)**

Un et un seul processus au sein de sa section critique à un instant donné

Section critique

2. Propriétés fondamentales

Toute solution au problème **section critique** doit assurer les 5 PF suivantes

1) **Unicité (exclusion mutuelle)**

2) **Équité**

Pas de privilège entre les processus demandeurs de la section critique

Section critique

2. Propriétés fondamentales

Toute solution au problème **section critique** doit assurer les 5 PF suivantes

- 1) **Unicité (exclusion mutuelle)**
- 2) **Équité**
- 3) **Pas de famine**

Aucun processus ne doit attendre à l'infinie sa section critique

Section critique

2. Propriétés fondamentales

Toute solution au problème **section critique** doit assurer les 5 PF suivantes

- 1) **Unicité (exclusion mutuelle)**
- 2) **Équité**
- 3) **Pas de famine**
- 4) **Pas d'interblocage**

Si un processus est bloqué en dehors de sa section critique alors ce blocage ne doit pas empêcher les autres d'utiliser la section critique

Section critique

2. Propriétés fondamentales

Toute solution au problème **section critique** doit assurer les 5 PF suivantes

- 1) **Unicité (exclusion mutuelle)**
- 2) **Equité**
- 3) **Pas de famine**
- 4) **Pas d'interblocage**
- 5) **Indépendance de l'architecture**

La solution doit être valable quelque soit le nombre de processeurs et l'architecture

Section critique

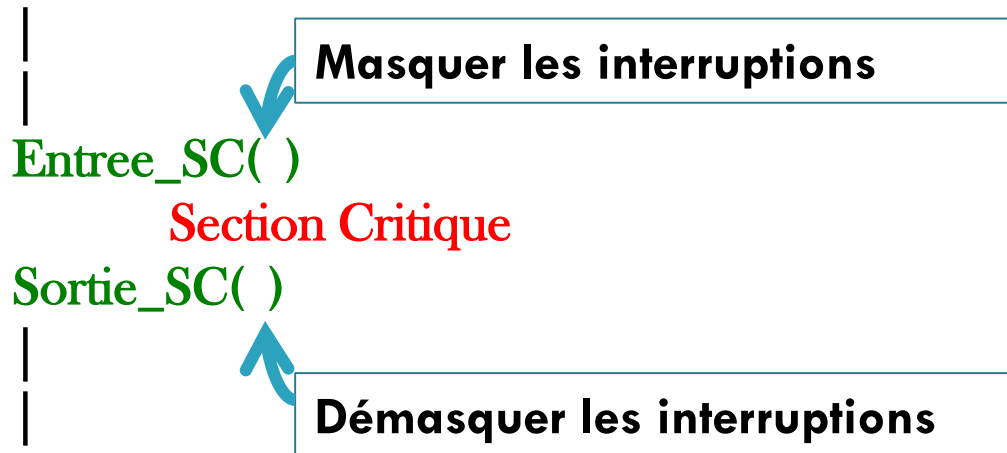
2. Propriétés fondamentales

Toute solution au problème **section critique** doit assurer les 5 PF suivantes

- 1) **Unicité (exclusion mutuelle)**
- 2) **Equité**
- 3) **Pas de famine**
- 4) **Pas d'interblocage**
- 5) **Indépendance de l'architecture**

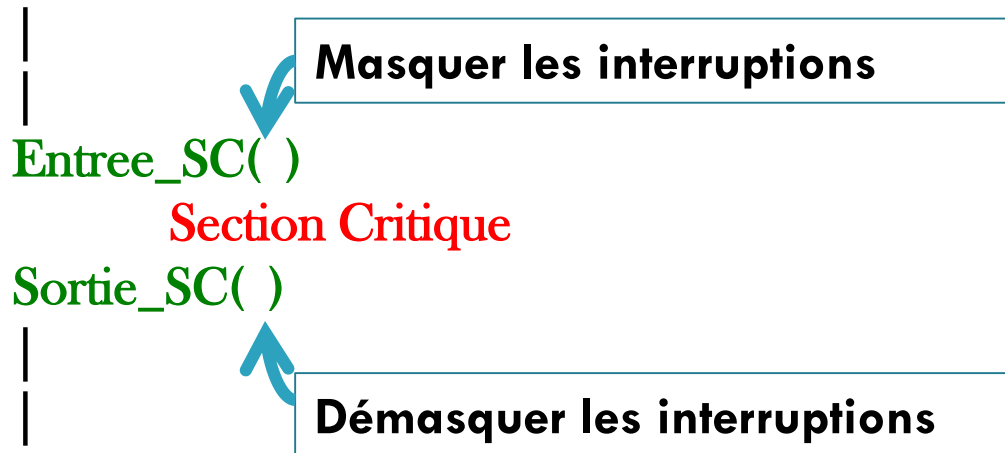
Solutions avec attente active

1. Solution Matérielle 1 : Désactivation des interruptions



Solutions avec attente active

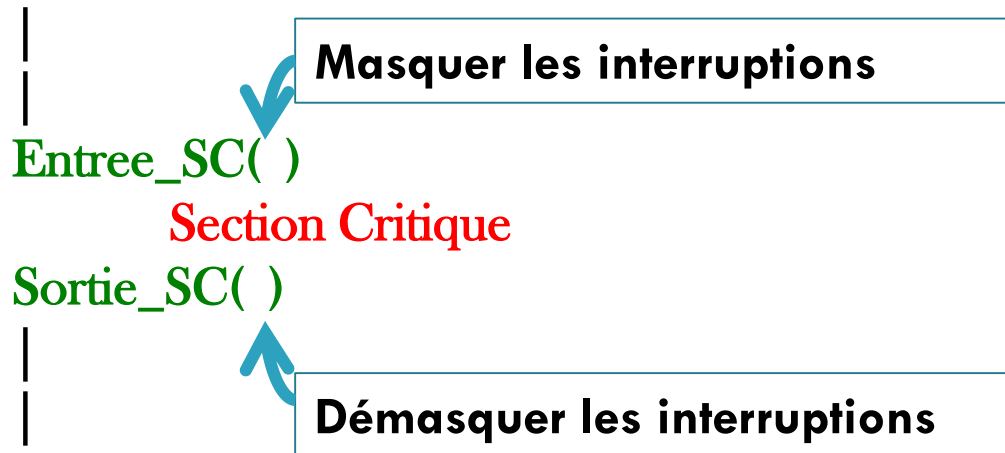
1. Solution Matérielle 1 : Désactivation des interruptions



La section critique est exécutée sans préemption

Solutions avec attente active

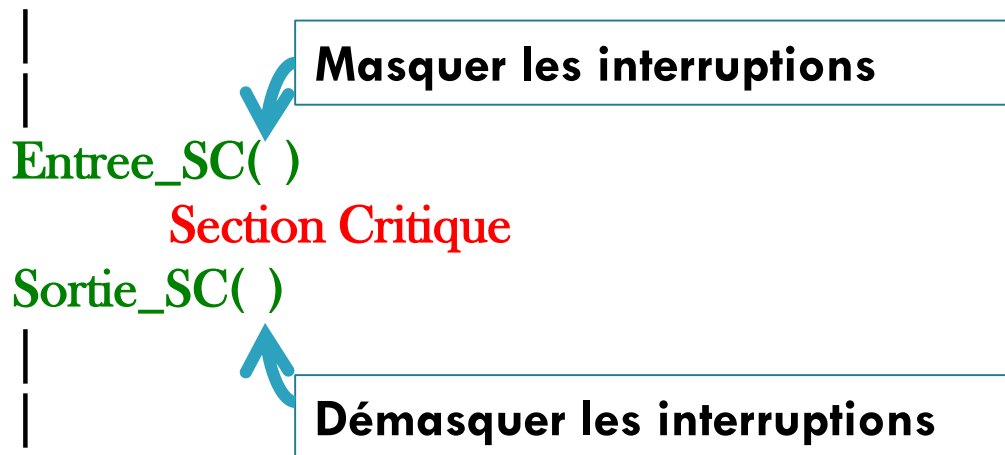
1. Solution Matérielle 1 : Désactivation des interruptions



Il est **dangereux** de donner ce droit à un utilisateur

Solutions avec attente active

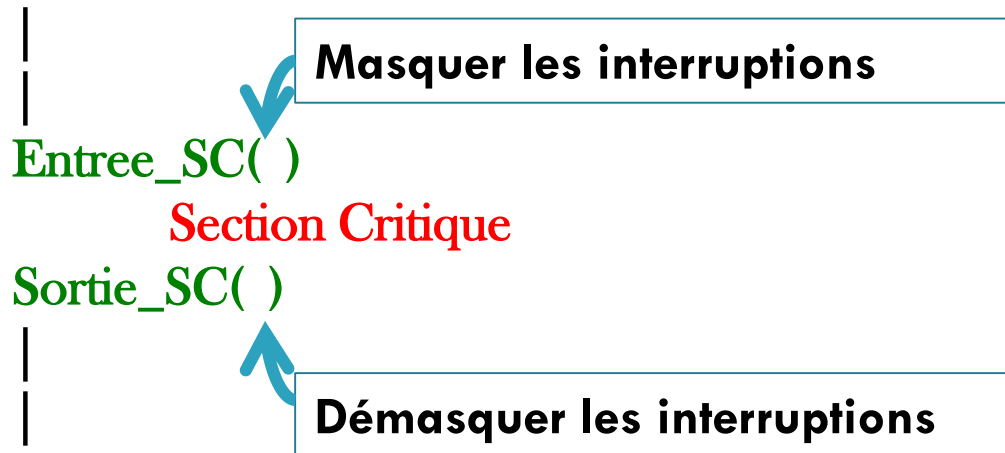
1. Solution Matérielle 1 : Désactivation des interruptions



Inefficace dans le cas d'un système multiprocesseurs

Solutions avec attente active

1. Solution Matérielle 1 : Désactivation des interruptions



Cette solution est utilisée par le SE quand il exécute des tâches sensibles

Solutions avec attente active

2. Solution Matérielle 2 : Test and Set Lock (TSL)

TSL (tester et définir le verrou) est une instruction assembleur

Solutions avec attente active

2. Solution Matérielle 2 : Test and Set Lock (TSL)

TSL (tester et définir le verrou) est une instruction assembleur

TSL Reg, Flag

- ❖ **Reg** : est un registre
- ❖ **Flag** : adresse mémoire partagée par tous les processus dite drapeau

Solutions avec attente active

2. Solution Matérielle 2 : Test and Set Lock (TSL)

TSL (tester et définir le verrou) est une instruction assembleur

TSL Reg, Flag

TSL charge le contenu du Flag dans Reg et en même temps mets le Flag à 1

Solutions avec attente active

2. Solution Matérielle 2 : Test and Set Lock (TSL)

TSL (tester et définir le verrou) est une instruction assembleur

TSL Reg, Flag

TSL charge le contenu du Flag dans Reg et en même temps mets le Flag à 1

TSL est atomique et verrouille le bus mémoire pendant son exécution

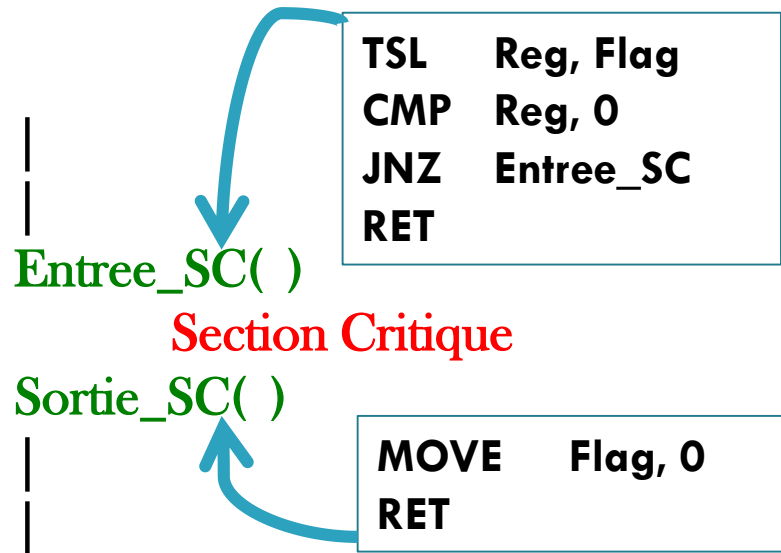
Solutions avec attente active

2. Solution Matérielle 2 : Test and Set Lock (TSL)

```
|  
|  
Entree_SC( )  
    Section Critique  
Sortie_SC( )  
|  
|
```


Solutions avec attente active

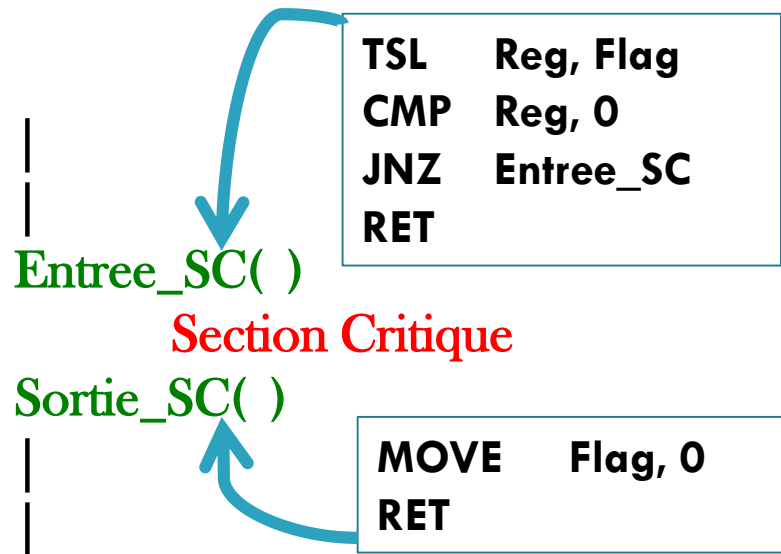
2. Solution Matérielle 2 : Test and Set Lock (TSL)



La valeur initiale de Flag est 0

Solutions avec attente active

2. Solution Matérielle 2 : Test and Set Lock (TSL)



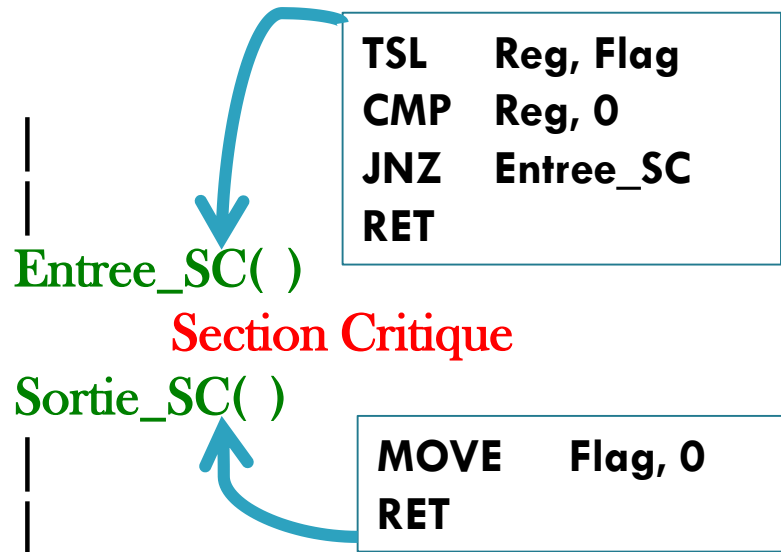
La solution est **efficace** et valable même dans le cas multiprocesseurs



La valeur initiale de Flag est 0

Solutions avec attente active

2. Solution Matérielle 2 : Test and Set Lock (TSL)



Le seul **inconvénient** de cette solution est **l'attente active** (consommation du temps processeur pour rien)



La valeur initiale de Flag est 0

Solutions avec attente active

3. Solution Logicielle 1 : Variable Verrou

Principe : utiliser une variable verrou qui joue le rôle de Flag.

Cette variable est :

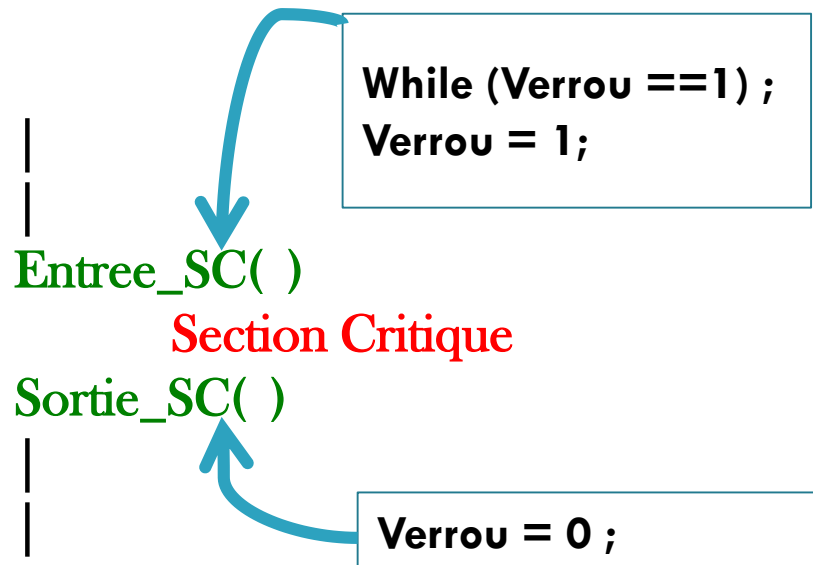
- ❖ mise à 1 par le processus entrant en SC
- ❖ remise à 0 lorsque le processus quitte la SC



La valeur initiale de Verrou est 0

Solutions avec attente active

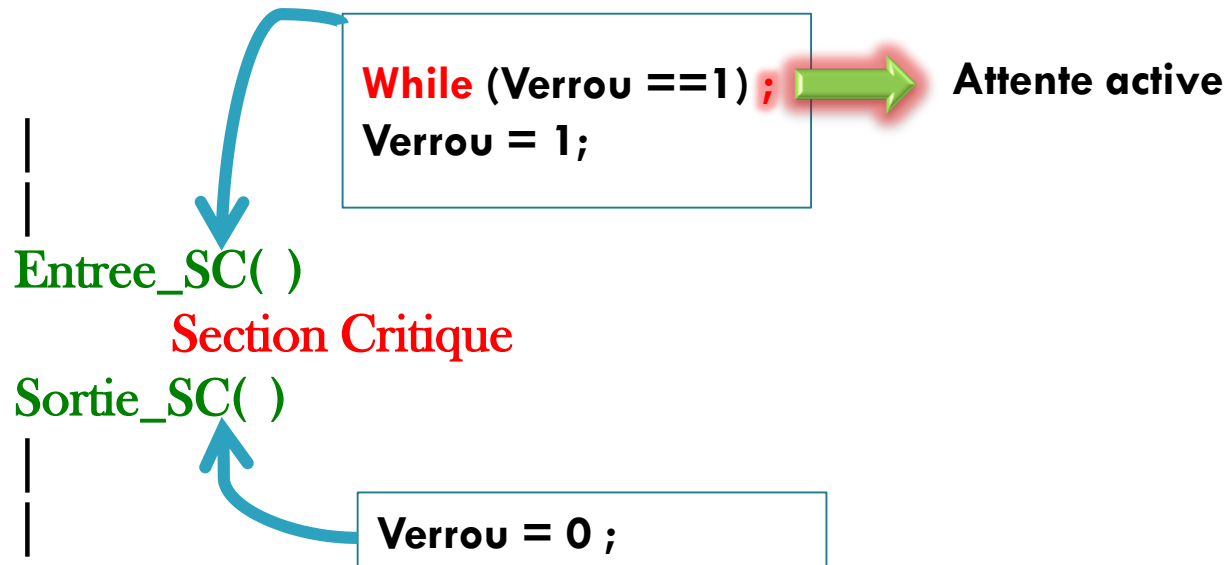
3. Solution Logicielle 1 : Variable Verrou



La valeur initiale de Verrou est 0

Solutions avec attente active

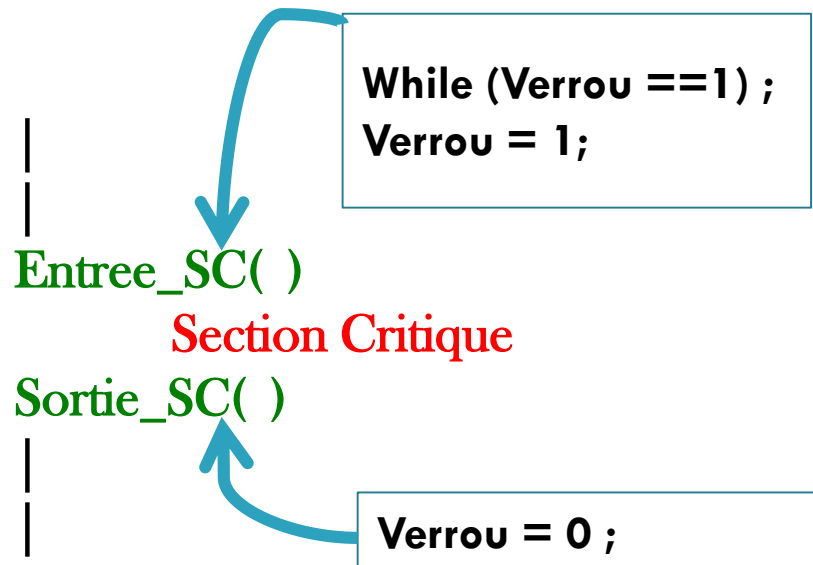
3. Solution Logicielle 1 : Variable Verrou



La valeur initiale de Verrou est 0

Solutions avec attente active

3. Solution Logicielle 1 : Variable Verrou



Fausse Solution puisqu'elle n'assure **pas l'unicité** (La variable `Verrou` est à son tour une variable partagée)



La valeur initiale de `Verrou` est 0

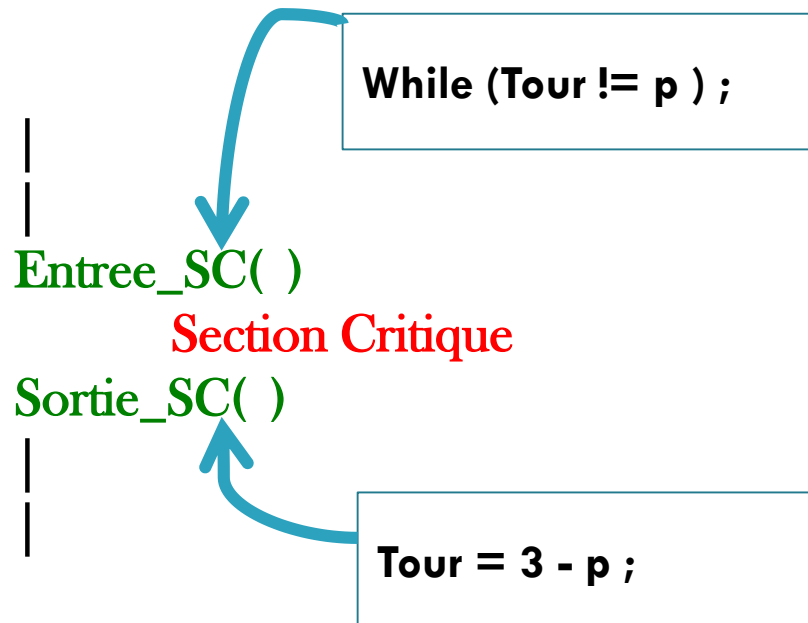
Solutions avec attente active

4. Solution Logicielle 2 : Variable d'alternance

Principe : éviter le problème de concurrence dans la solution précédente et définir une variable de tour de rôle.

Solutions avec attente active

4. Solution Logicielle 2 : Variable d'alternance



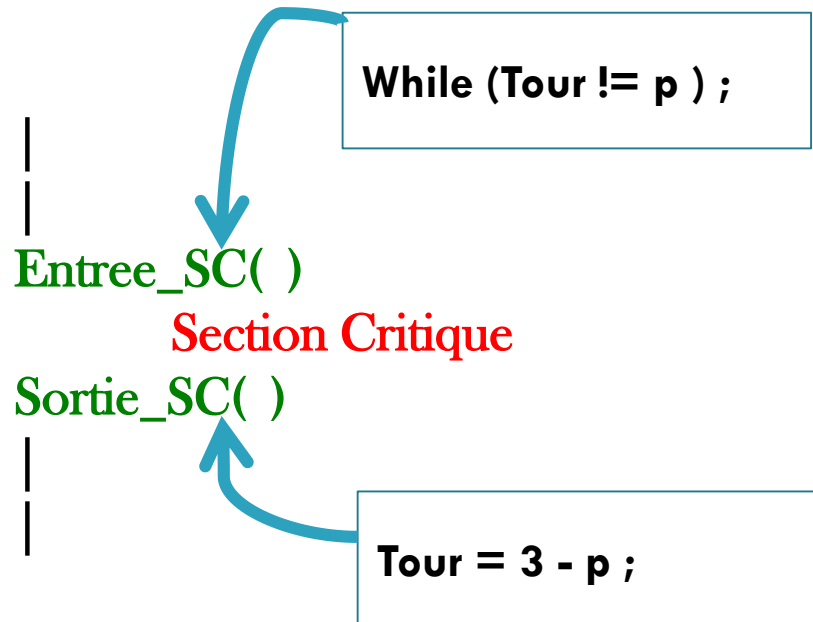
Pour un processus p et le nombre de processus est **deux**



La valeur initiale de Tour est ?

Solutions avec attente active

4. Solution Logicielle 2 : Variable d'alternance



Fausse Solution puisque :

- ❖ Si $Tour = 1 \rightarrow$ privilège de processus 1 et donc **pas d'équité**
- ❖ **Risque de famine** : si le tour est au processus 1 et celui-ci ne demande pas la SC



La valeur initiale de Tour est ?

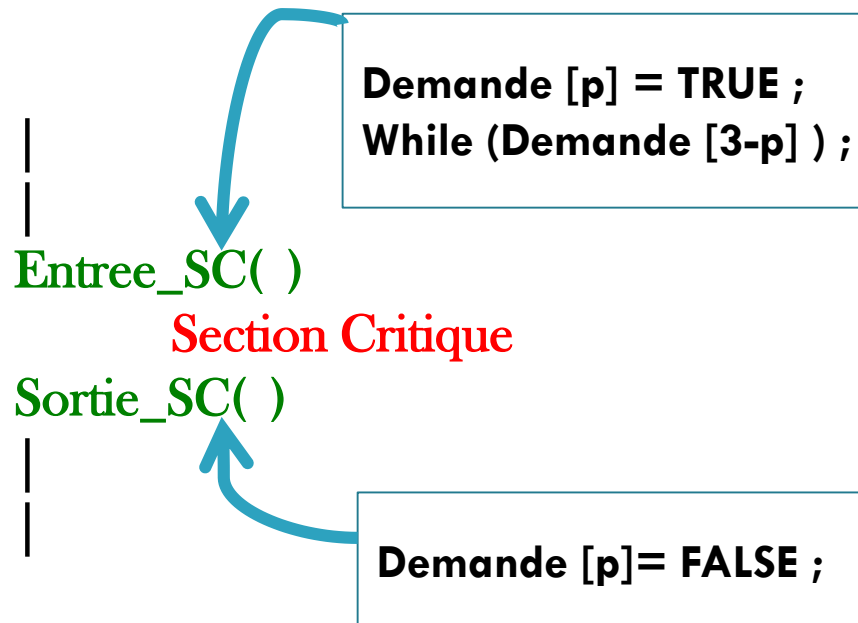
Solutions avec attente active

5. Solution Logicielle 3 : Tableau de booléens

Principe : la demande de processus d'entrée en SC doit être explicitée.

Solutions avec attente active

5. Solution Logicielle 3 : Tableau de booléens



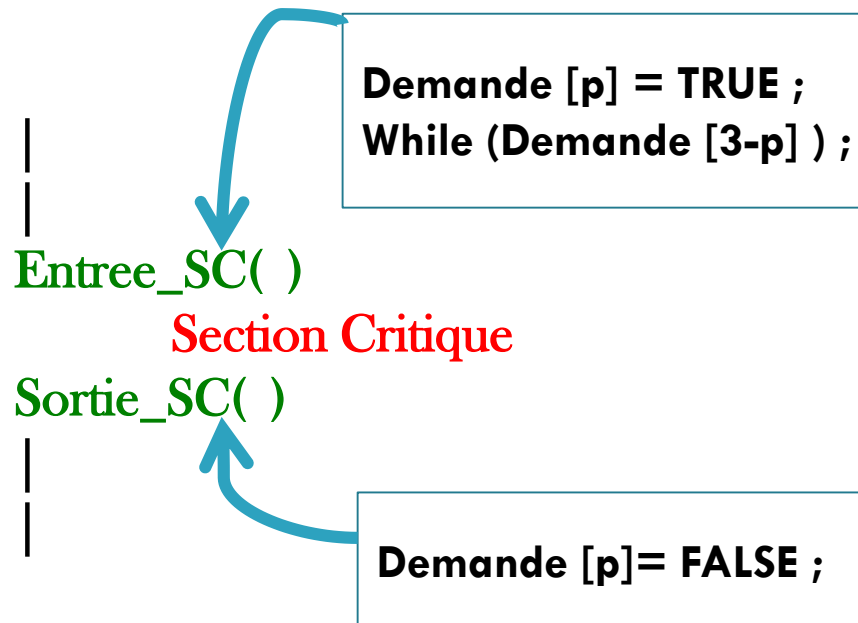
Pour un processus p et le nombre de processus est **deux**



La valeur initiale de `Demande []` est FALSE

Solutions avec attente active

5. Solution Logicielle 3 : Tableau de booléens



Fausse Solution puisqu'il y a un risque **d'interblocage** (si les demandes sont à TRUE)



La valeur initiale de Demande [] est FALSE

Solutions avec attente active

6. Solution de Peterson (1981)

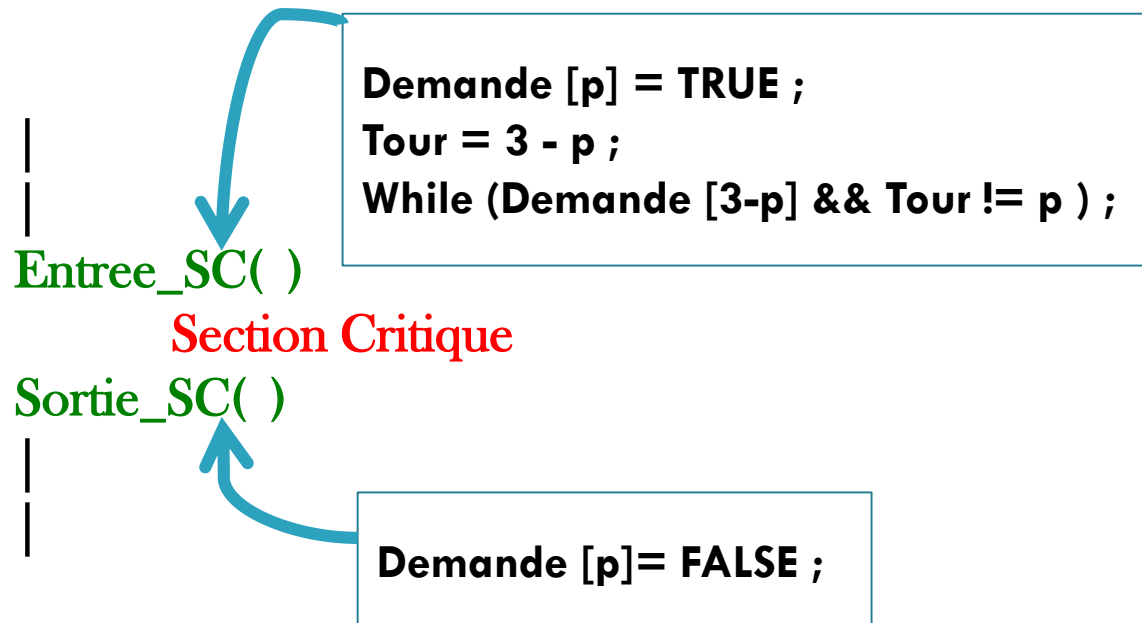
Principe : combine le **tour de rôle** et l'explicitation de la **demande** d'accès en SC.



Gary Lynn Peterson
Professeur de mathématiques
Département de mathématiques
Université de James Madison
Harrisonburg, Virginia

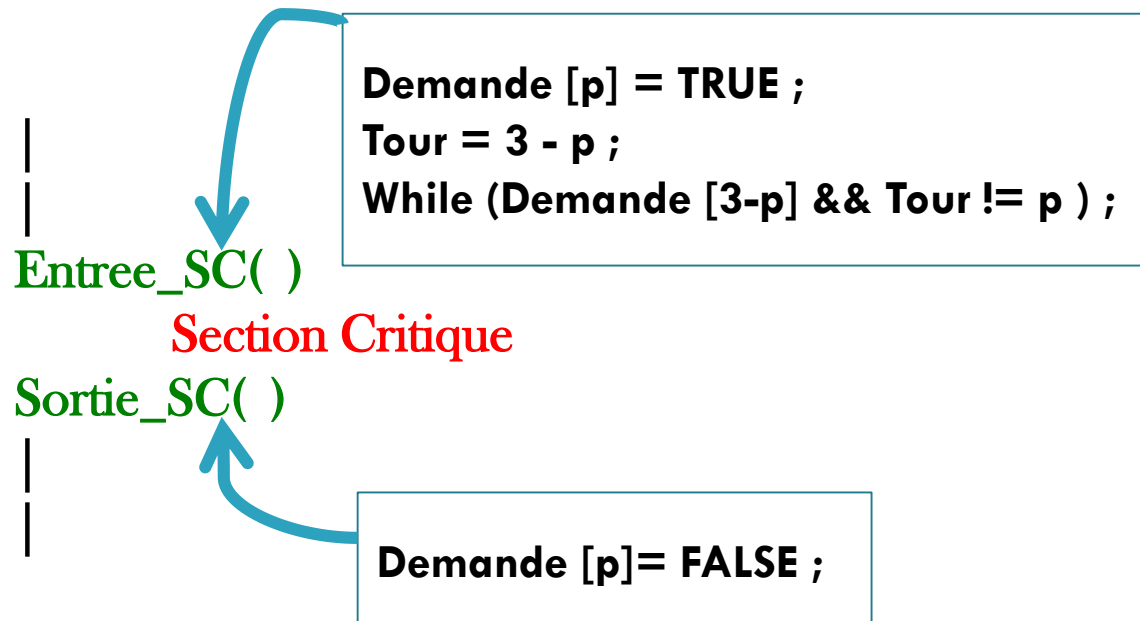
Solutions avec attente active

6. Solution de Peterson (1981)



Solutions avec attente active

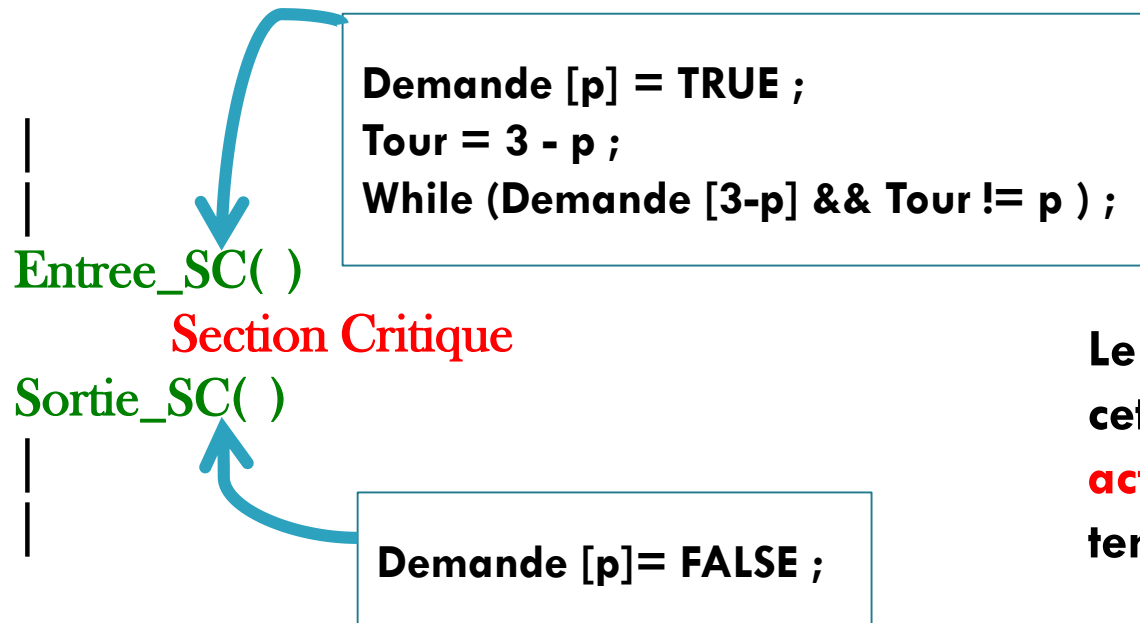
6. Solution de Peterson (1981)



La valeur initiale de **Demande** [] est FALSE
La variable **Tour** eut être initialisée à 1 ou 2

Solutions avec attente active

6. Solution de Peterson (1981)



Le seul **inconvenient** de cette solution est **l'attente active** (consommation du temps processeur pour rien)



La valeur initiale de **Demande []** est FALSE
La variable **Tour** peut être initialisée à 1 ou 2

Solutions avec attente passive

1. Principe

Le processus qui attend l'accès en SC doit être **bloqué**

Solutions avec attente passive

1. Principe

Le processus qui attend l'accès en SC doit être **bloqué**

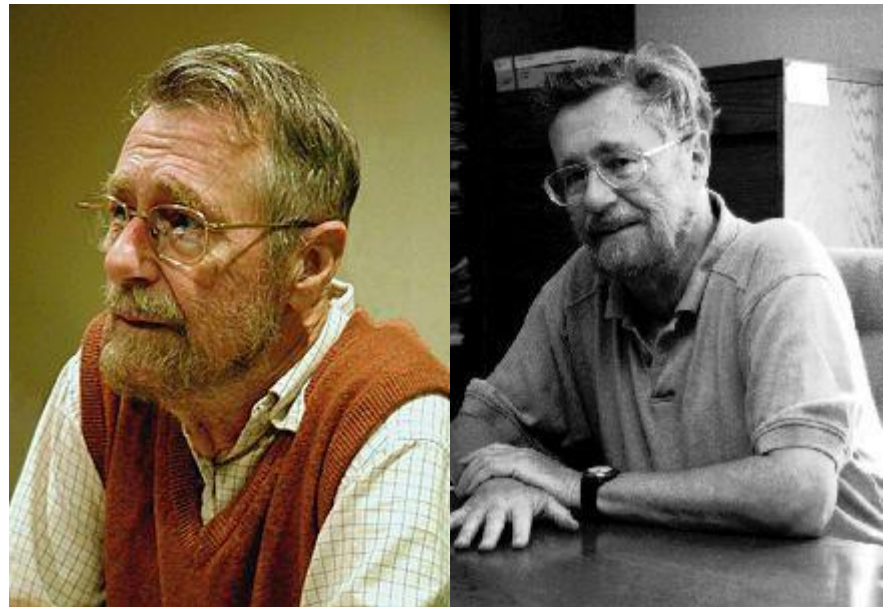
➤ L'ordonnanceur ne doit pas accorder le processeur aux processus en attente de leur section critique

Solutions avec attente passive

2. Sémaphores

Un sémaphore est une solution proposée en 1965 par Dr. Edsger Wybe Dijkstra
(1930-2002)

mathématicien et informaticien
néerlandais



Solutions avec attente passive

2. Sémaphores

Un sémaphore est une structure (enregistrement)

Struct semaphore

```
{    int val ;           // compteur d'accès  
    fa;                //file d'attente des processus bloqués  
};
```

Solutions avec attente passive

2. Sémaphores

Un sémaphore est une structure (enregistrement)

Struct semaphore

```
{    int val ;           // compteur d'accès

    fa;                 //file d'attente des processus bloqués

};

sem_init (semaphore S, int compt)           // Initialisation

{
    S.val = compt;

    S.fa = NULL;

}
```

Solutions avec attente passive

2. Sémaphores

Un sémaphore est une structure (enregistrement)

Struct semaphore

```
{    int val ;           // compteur d'accès  
    fa;                 //file d'attente des processus bloqués  
};
```

Deux primitives (**atomiques**) sont possibles pour manipuler un sémaphores

P Peut-on passer?

V Vas y !

Solutions avec attente passive

2. Sémaphores

Un sémaphore est une structure (enregistrement)

Struct semaphore

```
{    int val ;           // compteur d'accès  
    fa;                //file d'attente des processus bloqués  
};
```

Deux primitives (**atomiques**) sont possibles pour manipuler un sémaphores

P (semaphore S)

```
{    S.val --;  
    if (S.val < 0)  
        // ajouter le processus appelant à S.fa  
}
```

V (semaphore S)

Solutions avec attente passive

2. Sémaphores

Un sémaphore est une structure (enregistrement)

Struct semaphore

```
{    int val ;           // compteur d'accès

    fa;                 //file d'attente des processus bloqués

};
```

Deux primitives (**atomiques**) sont possibles pour manipuler un sémaphore

P (semaphore S)

```
{    S.val --;
    if (S.val < 0)
        // ajouter le processus appelant à S.fa
}
```

V (semaphore S)

```
{    S.val ++;
    if (S.val <= 0)
        // retirer un processus de S.fa et le débloquent
}
```

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P1



//P1 demande une imprimante

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

```
struct semaphore S;
```

```
Sem_init( S, 3);
```

P1

```
P(S) : S.val=2
```



```
//P1 demande une imprimante
```

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P1

P(S) : S.val=2



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P1

P(S) : S.val=2

P2



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P1

P(S) : S. val=2

P2

P(S) : S. val=1



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P1

P(S) : S. val=2

P2

P(S) : S. val=1



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P1

P(S) : S. val=2

P2

P(S) : S. val=1

P3



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P1

P(S) : S. val=2

P2

P(S) : S. val=1

P3

P(S) : S. val=0



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P1

P(S) : S. val=2

P2

P(S) : S. val=1

P3

P(S) : S. val=0



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P1 P(S) : S. val=2

P2 P(S) : S. val=1

P3 P(S) : S. val=0

P4



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P1 P(S) : S. val=2

P2 P(S) : S. val=1

P3 P(S) : S. val=0

P4 P(S) : S. val=-1



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P1 P(S) : S. val=2

P2 P(S) : S. val=1

P3 P(S) : S. val=0

P4 P(S) : S. val=-1



P4 bloqué et mis dans S.fa

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P1 P(S) : S. val=2

P2 P(S) : S. val=1

P3 P(S) : S. val=0

P4 P(S) : S. val=-1

P5



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P1 P(S) : S. val=2

P2 P(S) : S. val=1

P3 P(S) : S. val=0

P4 P(S) : S. val=-1

P5 P(S) : S. val=-2



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

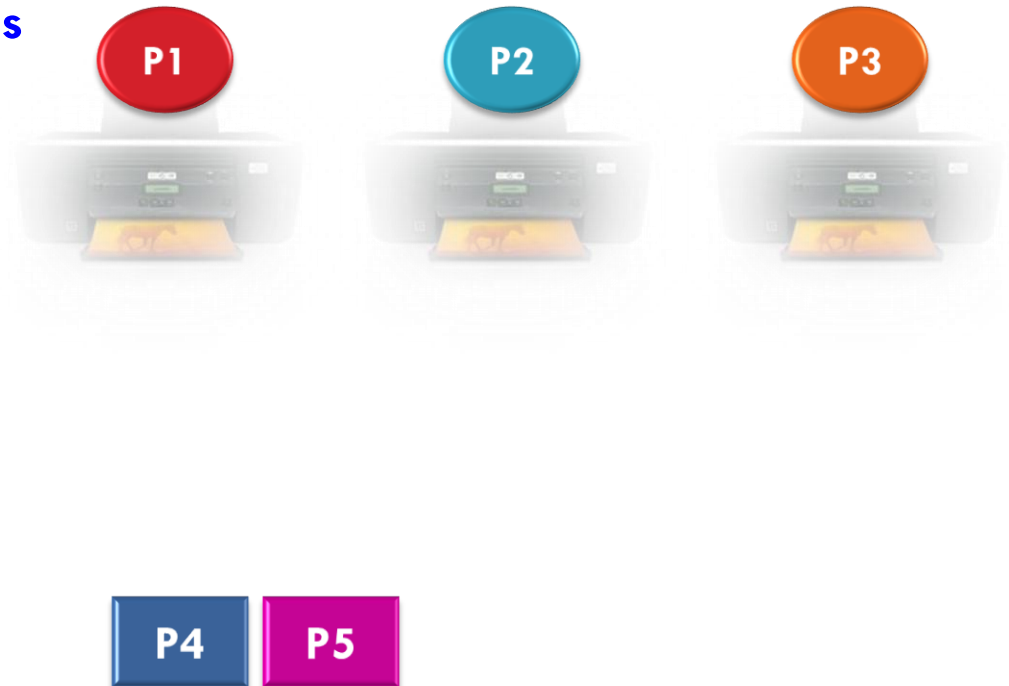
P1 P(S) : S. val=2

P2 P(S) : S. val=1

P3 P(S) : S. val=0

P4 P(S) : S. val=-1

P5 P(S) : S. val=-2



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P1 $P(S) : S.val=2$

P2 $P(S) : S.val=1$

P3 $P(S) : S.val=0$

P4 $P(S) : S.val=-1$

P5 $P(S) : S.val=-2$



La valeur de sémaphore est:

positive = nombre
d'exemplaires disponibles

P4

P5

négative = nombre de
processus en attente de la
libération de ressource

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P2



//P2 termine

P4

P5

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P2

V(S) : S.Val=-1

//P2 termine



P4

P5

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P2

V(S) : S.Val=-1

// libération



P4

P5

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P2

V(S) : S.Val=-1

// libération et déblocage de P4



P5

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P2

V(S) : S.Val=-1

P3



//P3 termine

P5

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P2

V(S) : S.Val=-1

P3

V(S) : S.Val =0

//P3 termine



P5

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P2

V(S) : S.Val=-1

P3

V(S) : S.Val =0

//Libération



P5

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P2 $V(S) : S.Val = -1$

P3 $V(S) : S.Val = 0$



//Libération et déblocage de P5

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);



V(S) : S.Val=-1



V(S) : S.Val = 0



//P1 termine



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);



V(S) : S.Val=-1



V(S) : S.Val = 0



V(S) : S.Val=1

//P1 termine



Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P2

V(S) : S.Val=-1

P3

V(S) : S.Val =0

P1

V(S) : S.Val=1



//Libération

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P2

V(S) : S.Val=-1

P3

V(S) : S.Val = 0

P1

V(S) : S.Val=1



//Libération

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P2

V(S) : S.Val=-1

P3

V(S) : S.Val = 0

P1

V(S) : S.Val=1

P4

V(S) : S.Val=2



//P4 termine, Libération

Solutions avec attente passive

2. Sémaphores

Cas de 3 imprimantes partagées

struct semaphore S;

Sem_init(S, 3);

P2

V(S) : S.Val=-1

P3

V(S) : S.Val =0

P1

V(S) : S.Val=1

P4

V(S) : S.Val=2

P5

V(S) : S.Val=3



//P5 termine, Libération

Solutions avec attente passive

2. Sémaphores

Cas d'une seule ressource partagée

struct semaphore S;

Sem_init(S, ?);

Solutions avec attente passive

2. Sémaphores

Cas d'une seule ressource partagée

```
struct semaphore S;
```

```
Sem_init( S, 1 );
```

Ce sémaphore est dit **binaire**. Il assure l'**exclusion mutuelle**

Solutions avec attente passive

2. Sémaphores

Cas d'une seule ressource partagée

struct semaphore S;

Sem_init(S, **1**);

|
|
Entree_SC()

Section Critique

Sortie_SC()
|
|

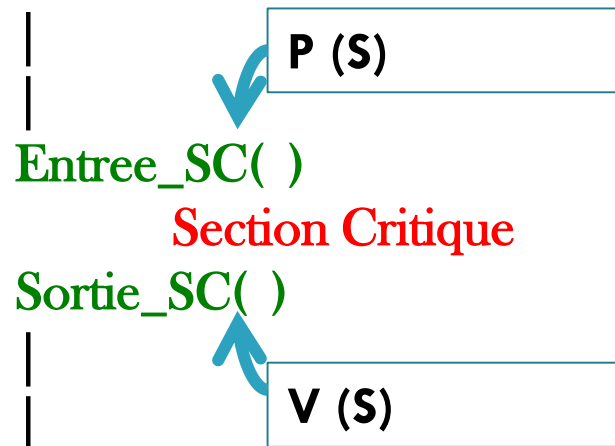
Solutions avec attente passive

2. Sémaphores

Cas d'une seule ressource partagée

struct semaphore S;

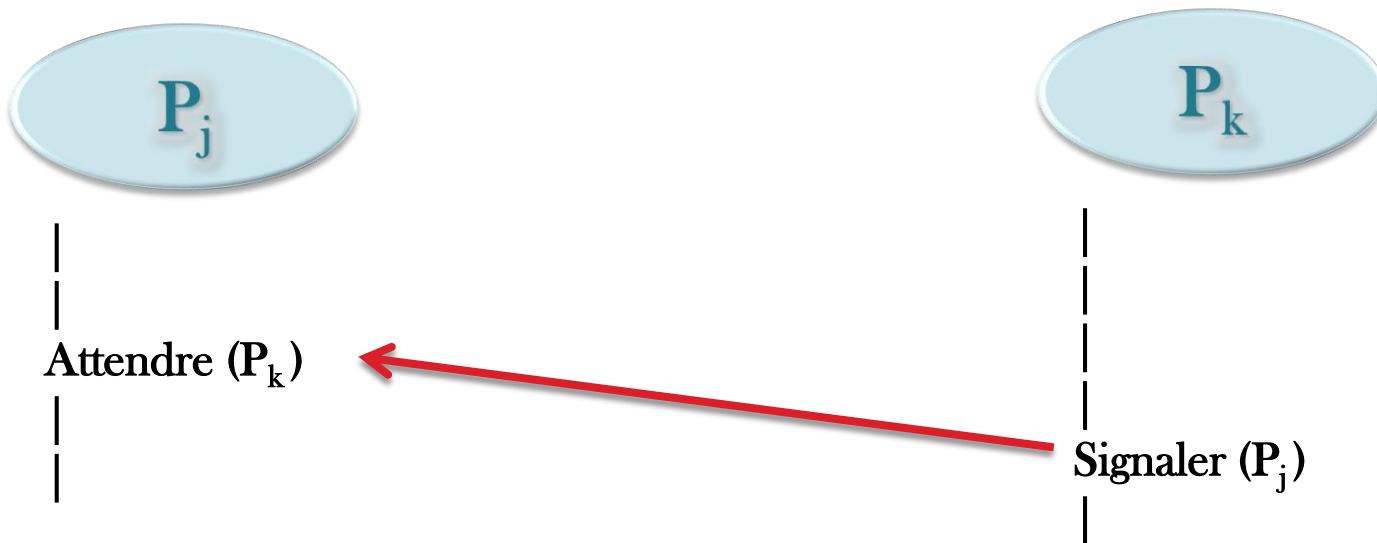
Sem_init(S, **1**);



Solutions avec attente passive

2. Sémaphores

Cas d'une synchronisation



Solutions avec attente passive

2. Sémaphores

Cas d'une synchronisation

struct semaphore S;

Sem_init(S, **0**);

P1	P2
1er travail	P(S)//attente de P1
V(S)//réveille P2	2eme travail

Solutions avec attente passive

2. Sémaphores

- ▣ **Sémaphore de comptage :**

Initialisé au nombre d'exemplaires de la ressource partagée

- ▣ **Sémaphore binaire :**

sert pour l'exclusion mutuelle. Il est initialisé à 1 et il a deux valeurs positives possibles 0 et 1

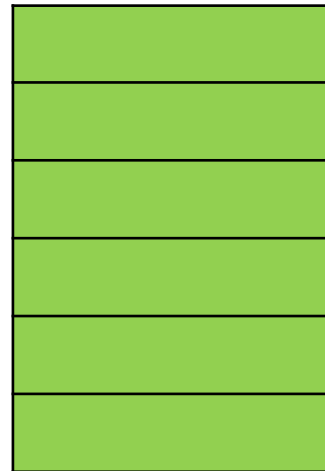
- ▣ **Sémaphore de synchronisation :**

Initialisé à 0

Solutions avec attente passive

2. Sémaphores

Exemple : **Producteur-Consommateur**



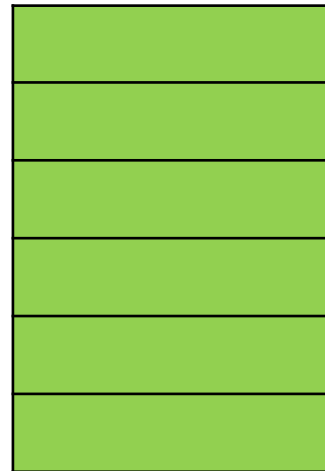
**Tampon
de taille N**



Solutions avec attente passive

2. Sémaphores

Exemple : **Producteur-Consommateur**



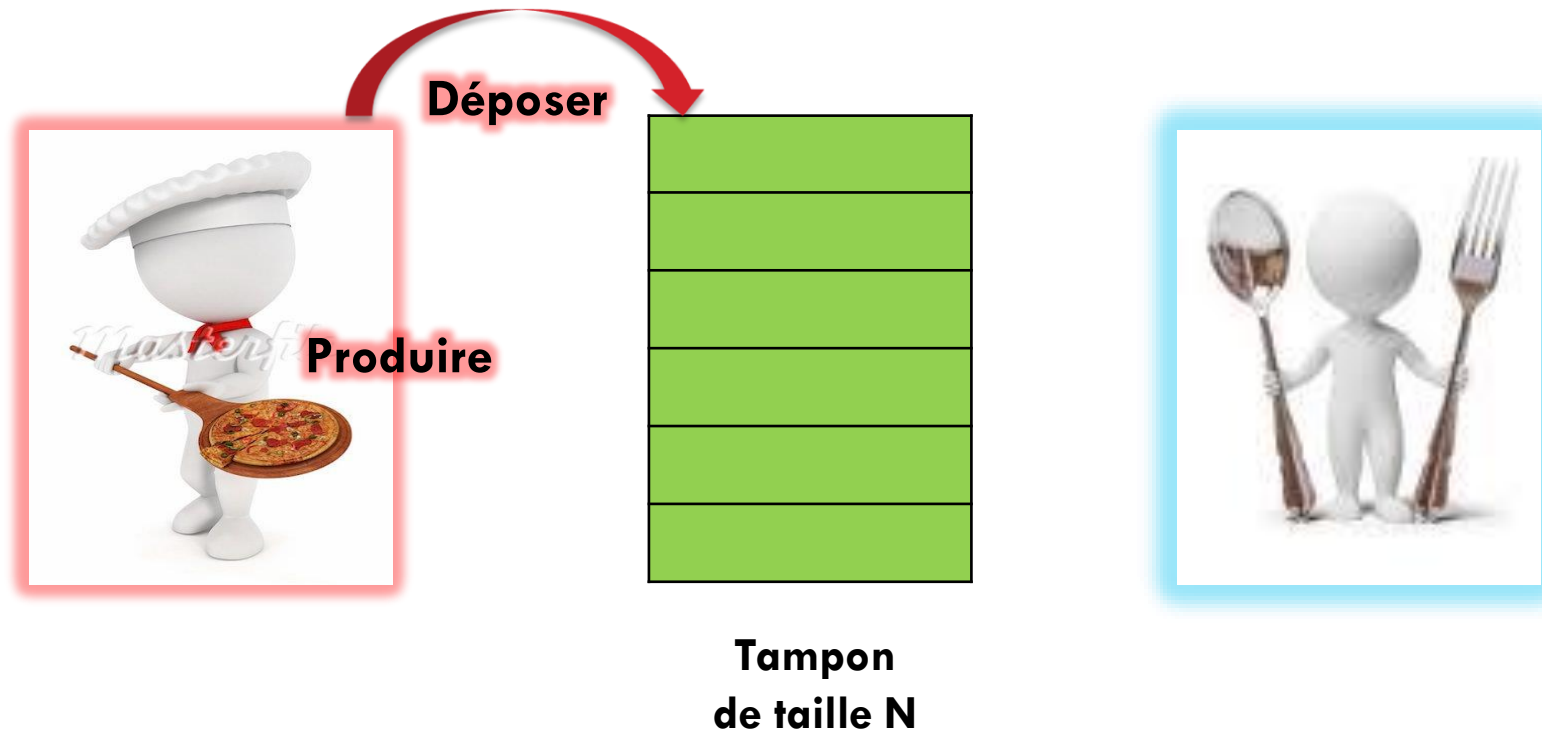
**Tampon
de taille N**



Solutions avec attente passive

2. Sémaphores

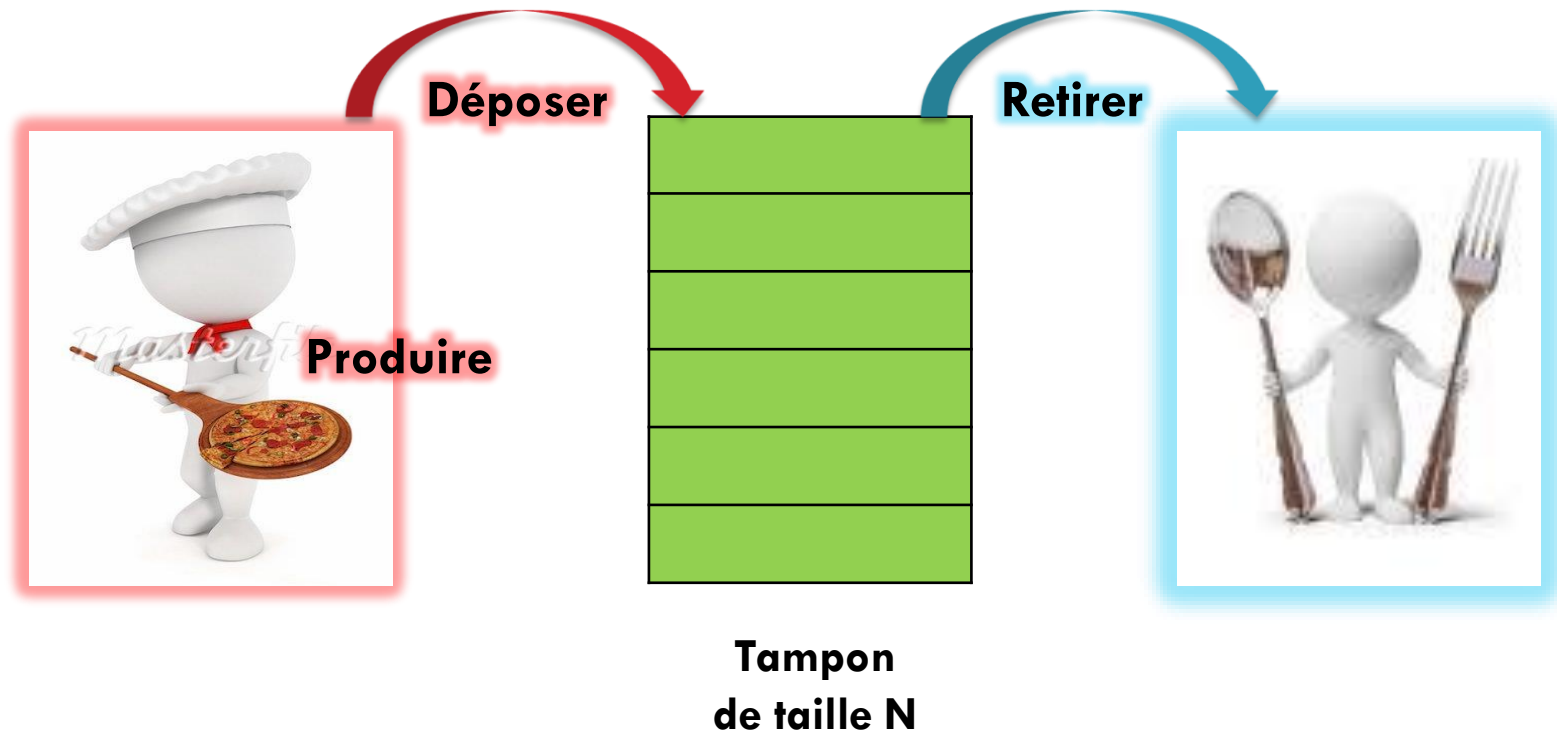
Exemple : **Producteur-Consommateur**



Solutions avec attente passive

2. Sémaphores

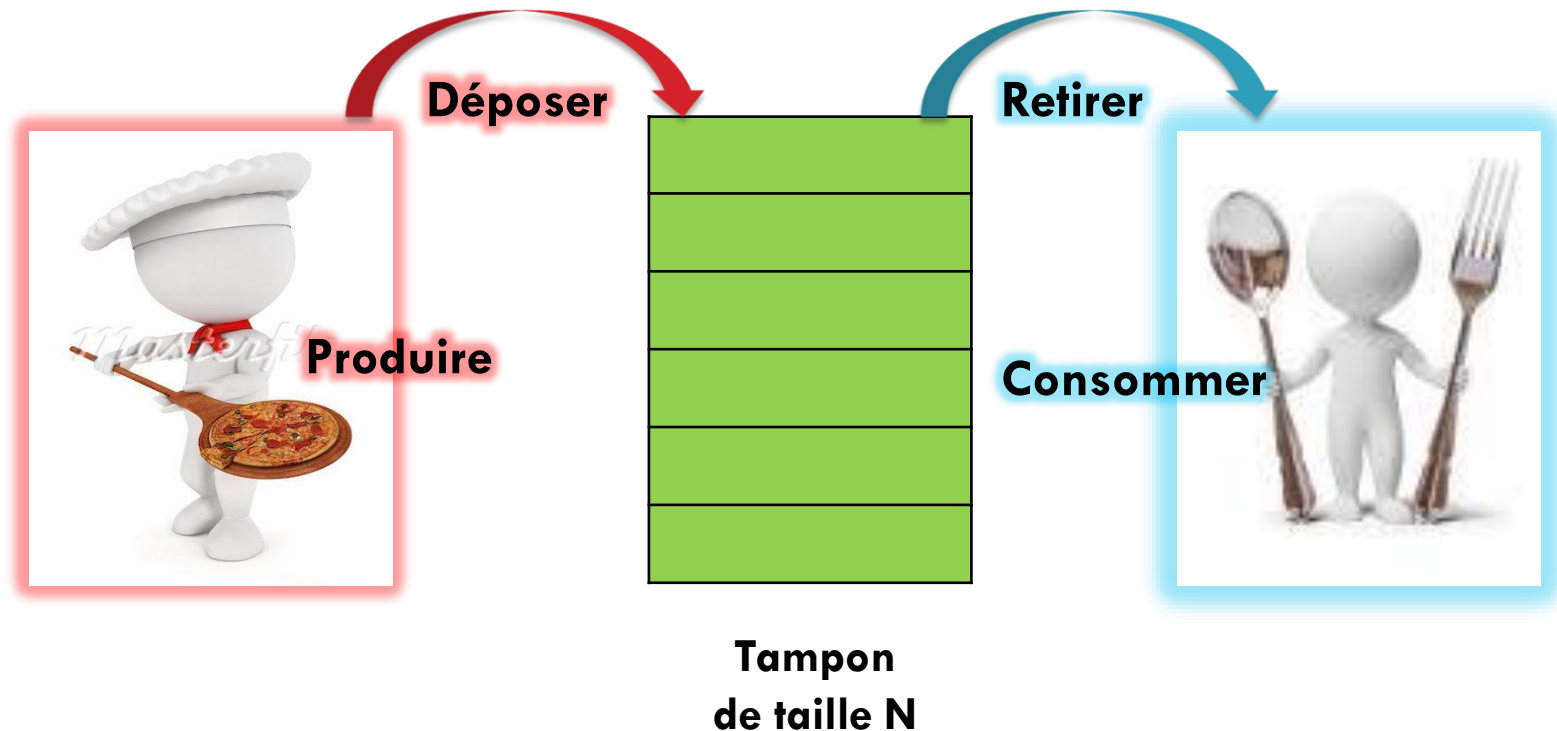
Exemple : **Producteur-Consommateur**



Solutions avec attente passive

2. Sémaphores

Exemple : **Producteur-Consommateur**



Solutions avec attente passive

2. Sémaphores

Exemple : **Producteur-Consommateur**



```
void producteur ( )  
{  
    int objet;  
    while (1)  
    {  
        objet = produire ( ) ;  
        Deposer (objet) ;  
    }  
}
```



Tampon
de taille N

```
void consommateur ( )  
{  
    int objet;  
    while (1)  
    {  
        objet = Retirer( ) ;  
        Consommer ( objet) ;  
    }  
}
```



Solutions avec attente passive

2. Sémaphores

Exemple : **Producteur-Consommateur**



```
void producteur ( )
```

```
{
```

```
    int objet;
```

```
    while (1)
```

```
    {
```

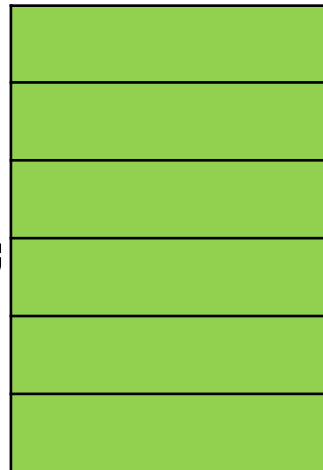
```
        objet = produire ( ) ;
```

```
        P(vide);
```

```
        Deposer (objet) ;
```

```
    }
```

```
}
```



```
void consommateur ( )
```

```
{
```

```
    int objet;
```

```
    while (1)
```

```
    {
```

```
        objet = Retirer( ) ;
```

```
        V(vide);
```

```
        Consommer ( objet) ;
```

```
    }
```

```
}
```



```
semaphore vide = N;
```

Le producteur ne peut pas déposer un objet si le tampon ne contient pas de place vide : il doit alors tester l'existence de place vide, s'il n'y pas il doit être bloqué

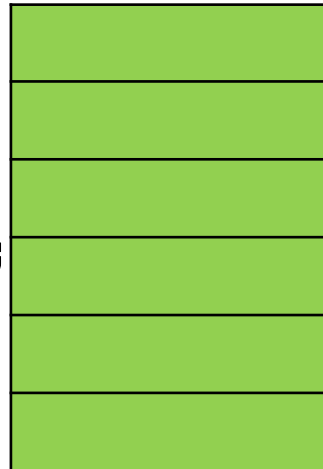
Solutions avec attente passive

2. Sémaphores

Exemple : **Producteur-Consommateur**



```
void producteur ( )  
{  
    int objet;  
    while (1)  
    {  
        objet = produire ( ) ;  
        P(vide);  
        Deposer (objet) ;  
        V(plein);  
    }  
}
```



Le consommateur ne peut pas retirer un objet si le tampon ne contient pas d'objet: il doit alors tester l'existence de place pleine, s'il n'y pas il doit être bloqué



```
void consommateur ( )  
{  
    int objet;  
    while (1)  
    {  
        P(plein);  
        objet = Retirer( ) ;  
        V(vide);  
        Consommer ( objet) ;  
    }  
}
```


Solutions avec attente passive

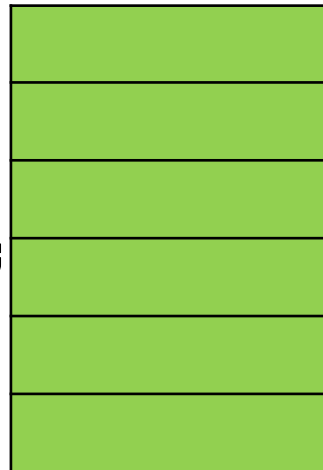
2. Sémaphores

Exemple : **Producteur-Consommateur**



```
void producteur ( )
{
    int objet;
    while (1)
    {
        objet = produire ( ) ;
        P(vide);
        P(mutex);
        Deposer (objet) ;
        V(mutex);
        V(plein);
    }
}
```

L'accès au tampon doit se faire en exclusion mutuelle puisqu'il est la ressource critique et donc il faut utiliser un sémaphore binaire pour l'allouer et pour le libérer



```
void consommateur ( )
{
    int objet;
    while (1)
    {
        P(plein);
        P(mutex);
        objet = Retirer ( ) ;
        V(mutex);
        V(vide);
        Consommer ( objet) ;
    }
}
```



Solutions avec attente passive

2. Sémaphores

```
# define N 100 // taille du tampon
```

```
semaphore mutex = 1; // assure l'exclusion mutuelle sur le tampon (deposer/retirer)
```

```
semaphore vide = N; // nombre de places vides
```

```
semaphore plein = 0; // nombre de places pleines
```

```
void producteur ( )
```

```
{
```

```
    int objet;
```

```
    while (1)
```

```
    {
```

```
        objet = produire ( ) ;
```

```
        P(vide);
```

```
        P(mutex);
```

```
        Deposer (objet) ;
```

```
        V(mutex);
```

```
        V(plein);
```

```
    }
```

```
}
```

```
void consommateur ( )
```

```
{
```

```
    int objet;
```

```
    while (1)
```

```
    {
```

```
        P(plein);
```

```
        P(mutex);
```

```
        objet = Retirer( ) ;
```

```
        V(mutex);
```

```
        V(vide);
```

```
        Consommer ( objet) ;
```

```
    }
```

```
}
```

Solutions avec attente passive

2. Sémaphores

Mais si les deux P sont inversés ?

```
void producteur ( )
{
    int objet;
    while (1)
    {
        objet = produire ( ) ;
        P(vide);
        P(mutex);
        Deposer (objet) ;
        V(mutex);
        V(plein);
    }
}
```

```
void consommateur ( )
{
    int objet;
    while (1)
    {
        P(plein);
        P(mutex);
        objet = Retirer ( ) ;
        V(mutex);
        V(vide);
        Consommer ( objet) ;
    }
}
```

Solutions avec attente passive

2. Sémaphores

Alors ?

```
void producteur ( )
{
    int objet;
    while (1)
    {
        objet = produire ( ) ;
        P(mutex);
        P(vide);
        Deposer (objet ) ;
        V(mutex);
        V(plein);
    }
}
```

```
void consommateur ( )
{
    int objet;
    while (1)
    {
        P(plein);
        P(mutex);
        objet = Retirer( ) ;
        V(mutex);
        V(vide);
        Consommer ( objet) ;
    }
}
```

Solutions avec attente passive

2. Sémaphores

Si le tampon ne contient aucune place vide alors le producteur se bloque après l'allocation de la SC et le consommateur se bloque au niveau de l'allocation de la SC et ne peut plus retirer des objets du tampon

```
void producteur ( )
{
    int objet;
    while (1)
    {
        objet = produire ( ) ;
        P(mutex);
        P(vide);
        Deposer (objet ) ;
        V(mutex);
        V(plein);
    }
}
```

Le producteur se bloque à ce niveau

```
void consommateur ( )
{
    int objet;
    while (1)
    {
        P(plein);
        P(mutex);
        objet = Retirer ( ) ;
        V(mutex);
        V(vide);
        Consommer ( objet ) ;
    }
}
```

Le consommateur se bloque à ce niveau

Solutions avec attente passive

2. Sémaphores

C'est le problème d'interblocage

```
void producteur ( )
{
    int objet;
    while (1)
    {
        objet = produire ( ) ;
        P(mutex);
        P(vide);
        Deposer (objet ) ;
        V(mutex);
        V(plein);
    }
}
```

```
void consommateur ( )
{
    int objet;
    while (1)
    {
        P(plein);
        P(mutex);
        objet = Retirer( ) ;
        V(mutex);
        V(vide);
        Consommer ( objet) ;
    }
}
```

Solutions avec attente passive

3. Moniteur

- ❖ Pour surmonter les erreurs de programmation qui peuvent apparaître avec les sémaphores, on utilise le moniteur comme solution de synchronisation de haut niveau (Hansen et Hoare 1973)



Solutions avec attente passive

3. Moniteur

- ❖ Pour surmonter les erreurs de programmation qui peuvent apparaître avec les sémaphores, on utilise le moniteur comme solution de synchronisation de haut niveau (Hansen et Hoare 1973)
- ❖ Le moniteur est un ensemble de **procédures** et de **variables** regroupées dans un module ayant comme propriété :
 - ❖ **Un et un seul** processus peut être actif dans le moniteur à un instant donné

Solutions avec attente passive

3. Moniteur

- ❖ Pour surmonter les erreurs de programmation qui peuvent apparaître avec les sémaphores, on utilise le moniteur comme solution de synchronisation de haut niveau (Hansen et Hoare 1973)
- ❖ Le moniteur est un ensemble de **procédures** et de **variables** regroupées dans un module ayant comme propriété :
 - ❖ **Un et un seul** processus peut être actif dans le moniteur à un instant donné
- C'est le compilateur qui assure l'exclusion mutuelle

Solutions avec attente passive

3. Moniteur

- ❖ Seules les **procédures** d'un moniteur peuvent accéder à ses **variables**
- ❖ **Variable de condition** : permet la synchronisation et est manipulée par :

Solutions avec attente passive

3. Moniteur

- ❖ Seules les **procédures** d'un moniteur peuvent accéder à ses **variables**
- ❖ **Variable de condition** : permet la synchronisation et est manipulée par :
 - ▣ **Wait (C):**
 - Bloque le processus appelant sur la variable condition C
 - ▣ **Signal (C):**
 - Débloque un processus dans la file d'attente de la variable C. Cependant le processus réveillant et le processus réveillé ne peuvent pas être actifs en même temps dans le moniteur
 - L'ordonnanceur doit en choisir un (réveillant)

Solutions avec attente passive

3. Moniteur

Monitor **ProducteurConsommateur**

```
{ condition Plein, Vide; // variables de condition
```

```
int compteur = 0;
```

```
procedure depot (int objet)
```

```
{ if (compteur == N ) wait(Plein);
```

```
  deposer (objet) ;
```

```
  compteur ++;
```

```
  if (compteur == 1) signal(Vide); }
```

```
procedure retrait (int objet)
```

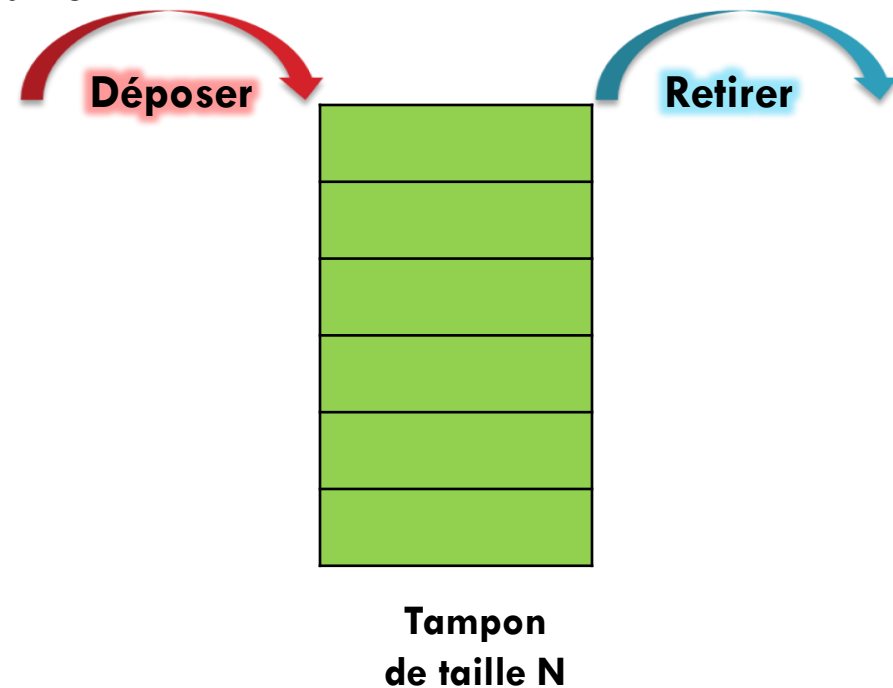
```
{ if (compteur == 0 ) wait(Vide);
```

```
  objet = retirer ( ) ;
```

```
  compteur = compteur - -;
```

```
  if (compteur == N-1) signal(Plein); }
```

```
} //fin de moniteur
```



Solutions avec attente passive

3. Moniteur

```
producteur ( )
{
    int objet;
    while (1)
    {
        objet = produire ( ) ;
        ProducteurConsommateur.depot(objet);
    }
}

consommateur
{
    int objet ;
    while (1)
    {
        objet = ProducteurConsommateur.retrait(objet);
        consommer ( objet) ;
    }
}
```



Problèmes classiques

SE

Chap 6. Synchronisation

1. Lecteur/rédacteur



Base de données

Problèmes classiques

1. Lecteur/rédacteur

Problème de cohérence des données :
Il faut assurer l'exclusion mutuelle entre le lecteur et le rédacteur



Base de données

Problèmes classiques

1. Lecteur/rédacteur

Plusieurs lecteurs peuvent accéder en même temps à la base.



À un instant donné, un seul écrivain peut exister seul dans la base.



Base de données

Problèmes classiques

SE

Chap 6. Synchronisation

1. Lecteur/rédacteur

```
void lecteur ()  
{ while(1)  
  {
```

`lire_BD();`

`traitement();`

`}`

`}`



Base de données

```
void redacteur()  
{
```

```
  while(1)  
  {
```

`creerDonnees();`

`ecrire_BD();`

`}`

`}`



Problèmes classiques

1. Lecteur/rédacteur

```
void lecteur ()  
{ while(1)
```

C'est le premier
lecteur qui doit
allouer la base
de données

```
    NbL = NbL + 1;  
    if (NbL == 1) P(db);
```

```
    lire_BD();
```

```
    NbL = NbL - 1;  
    if (NbL == 0) V(db);
```

```
    traitement();
```

```
  }  
}
```



C'est le dernier
lecteur qui doit
libérer la base
de données



```
void redacteur()  
{
```

```
    while(1)  
    {
```

```
        creerDonnees()
```

```
        ecrire_BD();
```


```
    }  
}
```



```
int NbL = 0 ; // nombre de lecteurs  
semaphore db = 1; // un sémaphore pour l'accès à la base
```

Problèmes classiques

1. Lecteur/rédacteur



```
void lecteur ()
{
    while(1)
    {
        NbL = NbL + 1;
        if (NbL == 1) P(db);

        lire_BD();


        NbL = NbL - 1;
        if (NbL == 0) V(db);

        traitement();
    }
}
```



L'exclusion mutuelle
doit être assurée avec
le rédacteur sur la
ressource critique Base
de données

```
void redacteur()
{
    while(1)
    {
        creerDonnees()
        P(db);
        ecrire_BD();
        V(db);
    }
}
```




```
int NbL = 0 ; // nombre de lecteurs
semaphore db = 1; // un sémaphore pour l'accès à la base
```

Problèmes classiques

1. Lecteur/rédacteur

La variable NbL est à son tour une ressource critique dont l'accès doit être protégé



```
void lecteur ()
{
    while(1)
    {
        P(mutex);
        NbL = NbL + 1;
        if (NbL == 1) P(db);
        V(mutex);
        lire_BD();

        NbL = NbL - 1;
        if (NbL == 0) V(db);

        traitement();
    }
}
```



```
void redacteur()
{
    while(1)
    {
        creerDonnees()
        P(db);
        ecrire_BD();
        V(db);
    }
}
```




semaphore mutex = 1; // un sémaphore pour l'accès à la variable globale partagée NbL

Problèmes classiques

1. Lecteur/rédacteur


La variable NbL est à son tour une ressource critique dont l'accès doit être protégé



```
void lecteur ()
{
    while(1)
    {
        P(mutex);
        NbL = NbL + 1;
        if (NbL == 1) P(db);
        V(mutex);
        lire_BD();
        P(mutex);
        NbL = NbL - 1;
        if (NbL == 0) V(db);
        V(mutex);
        traitement();
    }
}
```



```
void redacteur()
{
    while(1)
    {
        creerDonnees()
        P(db);
        ecrire_BD();
        V(db);
    }
}
```



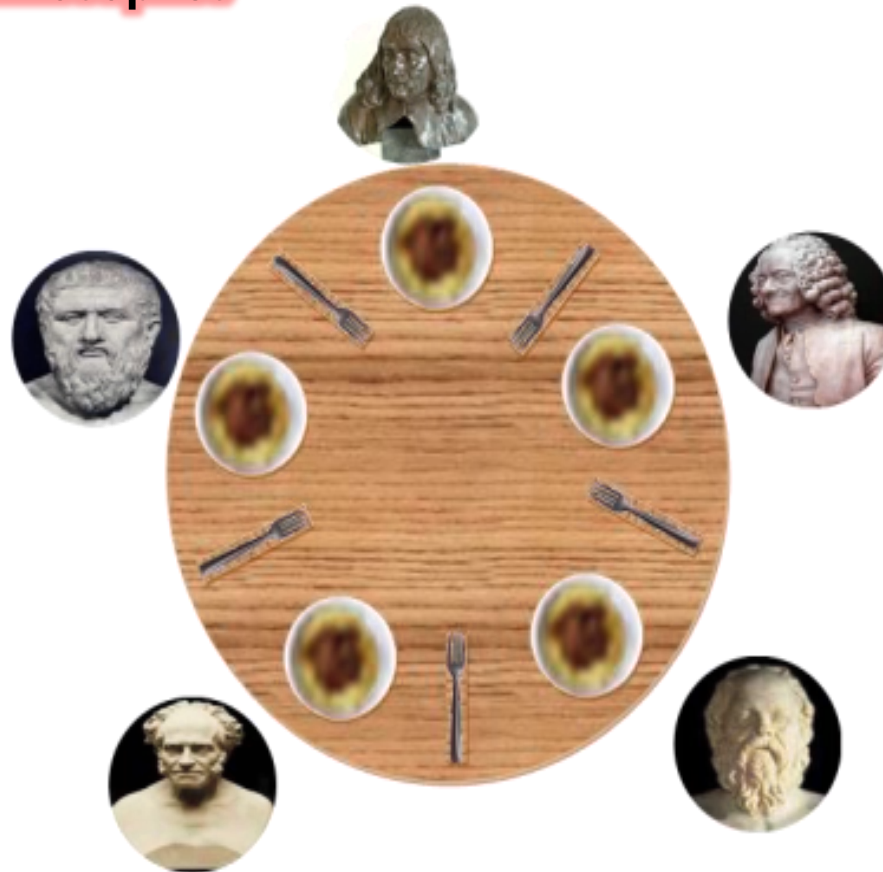
semaphore mutex = 1; // un sémaphore pour l'accès à la variable globale partagée NbL

Problèmes classiques

SE

Chap 6. Synchronisation

2. Repas des philosophes



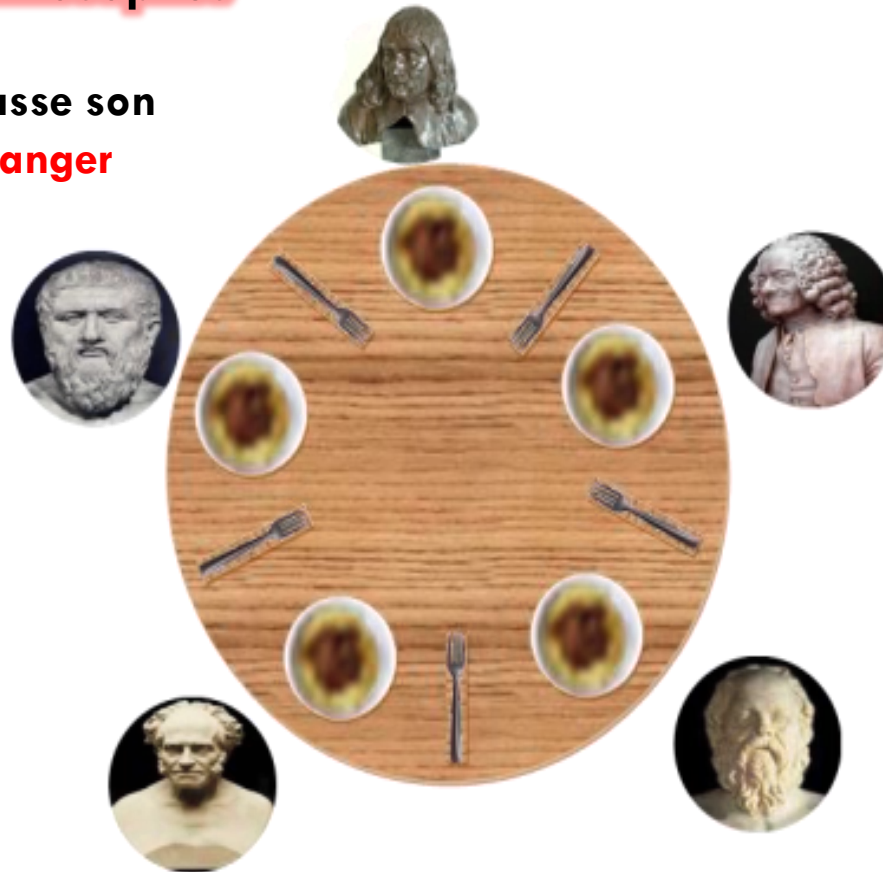
Problèmes classiques

SE

Chap 6. Synchronisation

2. Repas des philosophes

Chaque philosophe passe son temps à **penser** et à **manger**

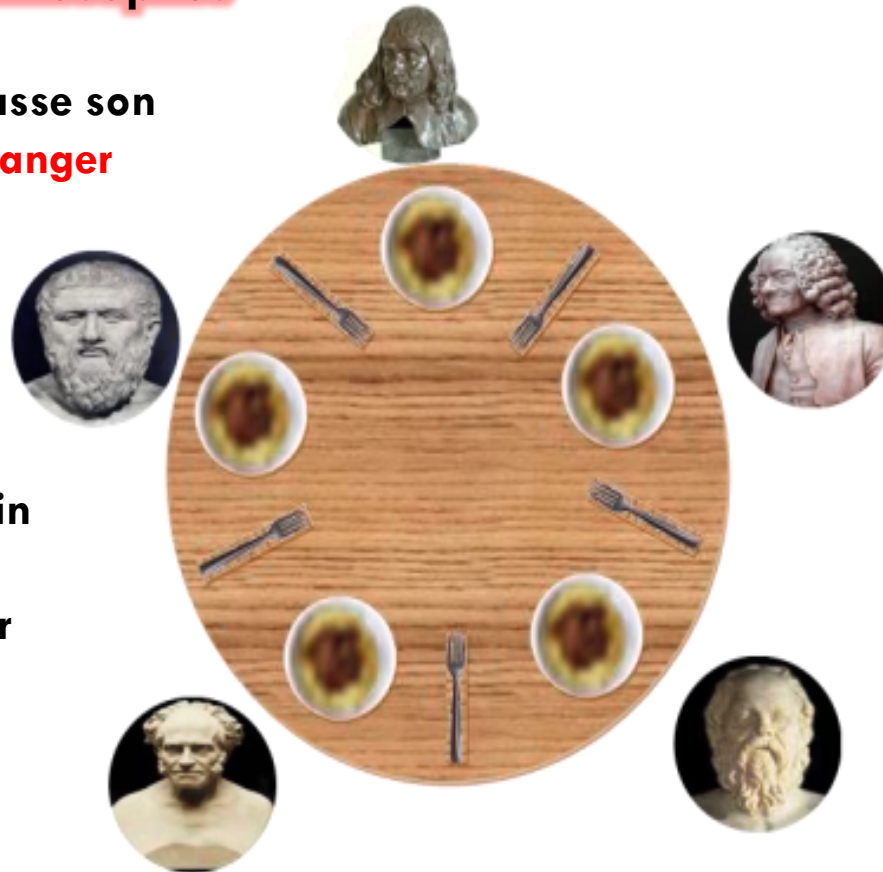


Problèmes classiques

2. Repas des philosophes

Chaque philosophe passe son temps à **penser** et à **manger**

Un philosophe a besoin de **deux fourchettes** (gauche et droite) pour **manger** son plat.



Problèmes classiques

2. Repas des philosophes

```
#define N 5 // nombre de philosophes
```

```
#define GAUCHE (i+N-1)%N
```

```
#define DROITE (i+1)%N
```

```
#define PENSE 0 // le philosophe pense
```

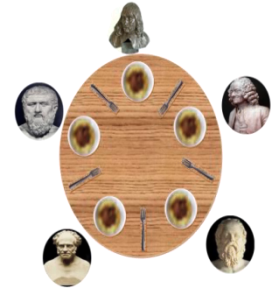
```
#define FAIM 1 // le philosophe a faim
```

```
#define MANGE 2 // le philosophe mange
```

```
int state[N]; // tableau pour suivre les états des philosophes
```

```
semaphore mutex = 1; // pour l'exclusion mutuelle sur les fourchettes
```

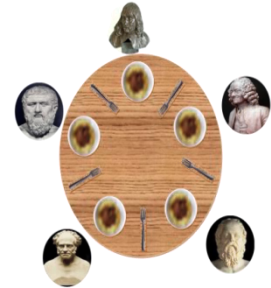
```
semaphore s[N] = 0; // un sémaphore par philosophe
```



Problèmes classiques

2. Repas des philosophes

```
void philosophe (int i) // i est le numéro du philosophe
{
    while(1)
    {
        penser();
        prendre_fourchettes(i); // il prend deux fourchette ou se bloque
        manger();
        poser_fourchettes(i); // il pose les deux fourchettes sur la table
    }
}
```



Problèmes classiques

2. Repas des philosophes

```
void prendre_fourchettes (int i)
```

```
{
```

```
    P(mutex); // entrer en section critique
```

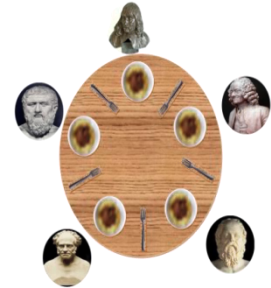
```
    state[i] = FAIM; // signaler que le philosophe a faim
```

```
    test(i); // il tente de prendre deux fourchettes
```

```
    V(mutex); // quitter la section critique
```

```
    P(s[i]); // il se bloque s'il n'a pas pu prendre les deux fourchettes
```

```
}
```



Problèmes classiques

2. Repas des philosophes

```
void poser_fourchettes (int i)
```

```
{
```

```
    P(mutex); // entrer en section critique
```

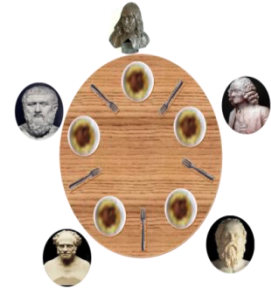
```
    state[i] = PENSE; // signaler que le philosophe a fini de manger
```

```
    test(GAUCHE); // vérifier si le voisin de gauche peut manger
```

```
    test(DROITE); // vérifier si le voisin de droite peut manger
```

```
    V(mutex); // quitter la section critique
```

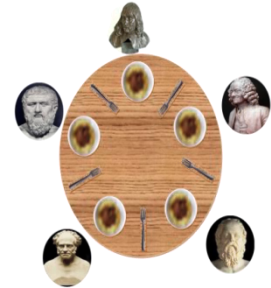
```
}
```



Problèmes classiques

2. Repas des philosophes

```
void test (int i)
{
    if (state[i] == FAIM && state[GAUCHE] != MANGE &&
        state[DROITE] != MANGE )
    {
        state[i] = MANGE;
        V(s[i]);
    }
}
```



FIN
LIA

SE

Madame Khaoula ElBedoui-Maktouf
2^{ème} année Ingénieur Informatique