

# Systèmes concurrents et distribués

Pierre Leone

[pierre.leone@unige.ch](mailto:pierre.leone@unige.ch)

# Plan du cours

- Programmation concurrente
  - Exclusion mutuelle
  - Synchronisation
  - synchronisation wait-free
- Programmation distribuées
  - Sockets/RMI en Java
  - ...

# Plan du cours II

But:

- Considérer les problèmes classiques de programmation concurrente/distribuée
- Comprendre les contraintes liées aux applications concurrentes
- Illustrer les problèmes/solutions avec Java
- Discuter le langage Java le plus complètement possible

# Java

Pourquoi programmer en Java?

- C'est le seul langage pour lequel il existe un modèle formel de programmation
- Tout le monde parmi vous sait programmer en Java (séquentiel).

# Java pour illustrer

Le paquetage `java.util.concurrent` met à disposition des outils 'prêts à l'emploi'.

On va s'intéresser à l'implémentation de ces objets, le cours utilise Java pour illustrer les concepts, mais ne se limite pas à passer en revue les outils offerts par les librairies Java

# Références

*Concurrent and distributed computing in Java, Vijay K. Garg*

*Concurrent and Real-Time programming in Java, Andy Wellings*

*Java Concurrency in practice, Brian Goetz*

*The art of multiprocessor programming, Maurice Herlihy, Nir Shavit*

En français:

*Programmation concurrente et temps réel avec Java, Luigi Zaffalon*

# Introduction

Un programme **concurrent** est un programme qui peut s'exécuter en parallèle.

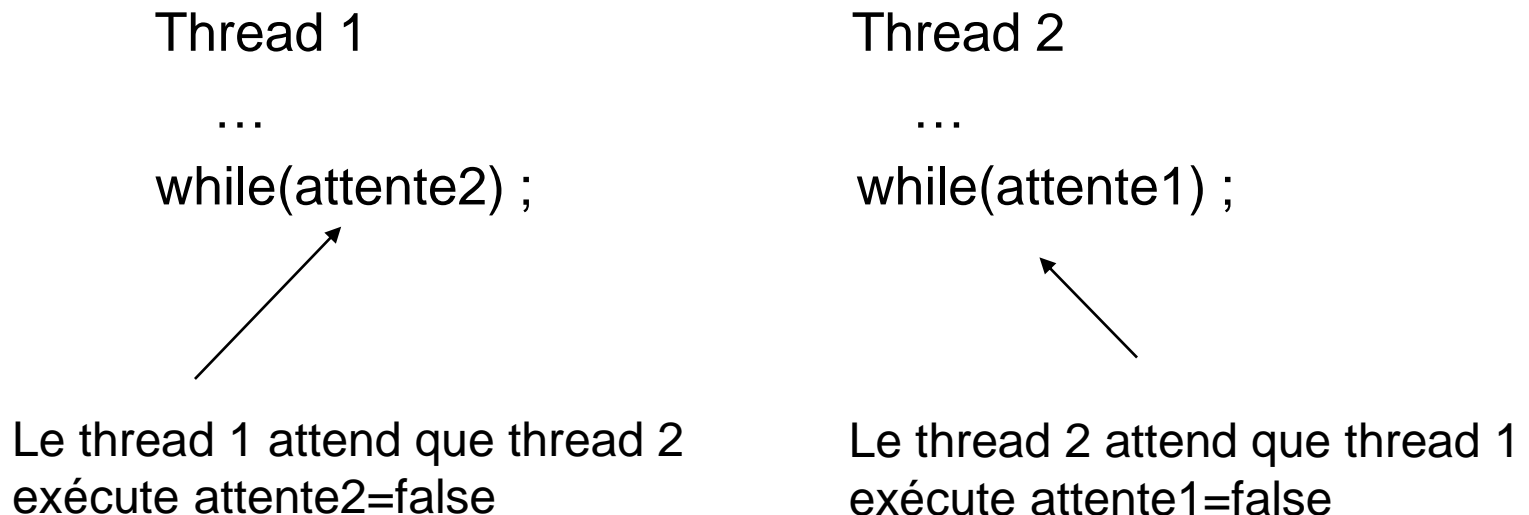
Un **processus** est un programme séquentiel qui compose un programme concurrent.

Des processus qui partagent une zone mémoire pour les données/programme sont appelés des **threads**.

# Problèmes classiques en programmation concurrente

Inter blocage (deadlock):

- Survient lorsque les entités d'un programme attendent une action d'une autre entité pour continuer





# Problèmes classiques en programmation concurrente

## Interférences:

- Plusieurs entités concurrentes modifient une donnée en même temps, la donnée peut finalement être corrompue

Par exemple la donnée est une date de naissance jj/mm/aaaa

Thread1 exécute jj = 1, mm = 12, aaaa = 1982

Thread2 exécute jj = 20, mm = 1, aaaa = 1946

Après exécution le résultat est jj = 1, mm = 1, aaaa=1946

# Problèmes classiques en programmation concurrente

Insuffisance de ressources (starvation):

- Une entité n'arrive pas accéder une ressource, cette dernière étant perpétuellement utilisée par d'autres entités.

# Propriétés désirées

## Sûreté (safety):

- un événement indésirable ne doit pas se produire pendant l'exécution, en particulier pas d'interférences entre les différentes entités.

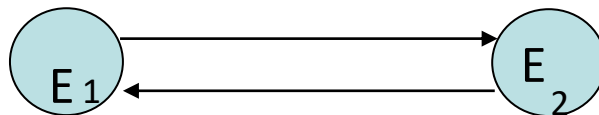
## Vivacité (*liveness*) :

- un événement souhaité arrivera nécessairement, en d'autres termes l'exécution du programme ne résultent pas en interblocages ou insuffisance de ressources

# Inter blocage

Un programme doit satisfaire quatre conditions nécessaires pour qu'un inter blocage soit possible

- **Exclusion mutuelle:** Une ressource non partageable simultanément doit être accédée, une telle ressource doit être partagée séquentiellement.
- **Hold and wait:** Les entités impliquées dans l'inter blocage doivent posséder l'accès à une ressource non partageable simultanément et attendre l'accès à une autre ressource non partageable.
- **Pas de préemption:** Les ressources doivent impérativement être explicitement libérées par l'entité qui dispose de l'accès.
- **Attente circulaire:** Les entités forment une chaîne dans laquelle chacune dispose de l'accès à une ressource et attend pour accéder la ressource dont dispose la prochaine entité dans la chaîne.



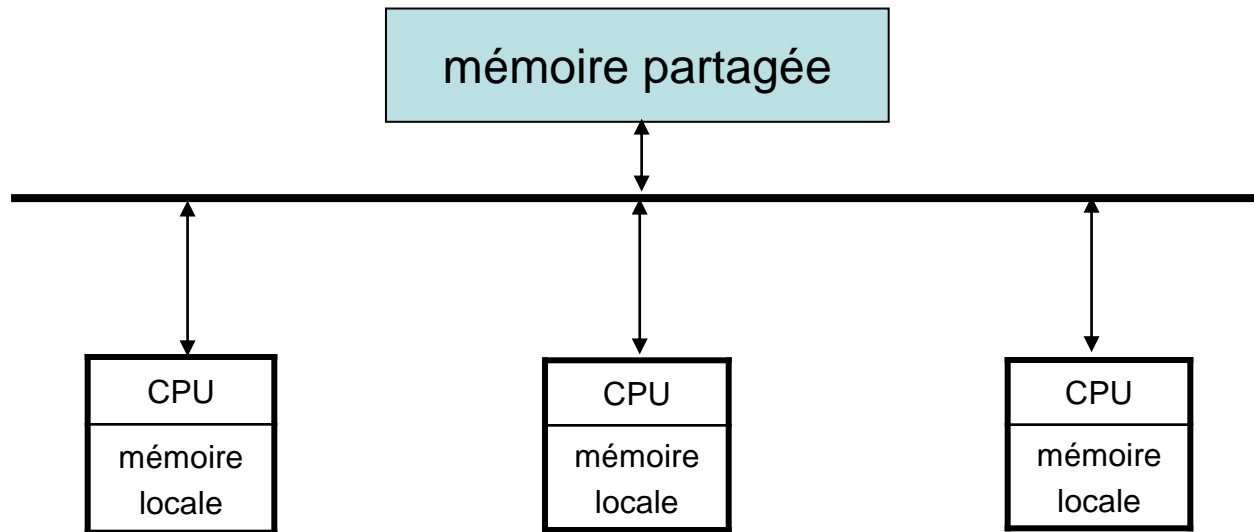
# Les solutions

Il existe trois solutions pour prévenir les inter blocages

- S'assurer qu'au moins une des quatre conditions nécessaires ne peut pas se produire. Par exemple en interdisant une entité d'attendre l'accès à une ressource si elle est en possède déjà une (**deadlock avoidance**).
- Utiliser un algorithme d'allocation des ressources qui gère dynamiquement les accès en évitant les inter blocages (**deadlock avoidance algorithm**).
- Utiliser un système de détection des inter blocages qui effectue les actions nécessaires au déblocage. (Par exemple en retirant l'accès à un processus et en donnant l'accès à un autre par préemption).

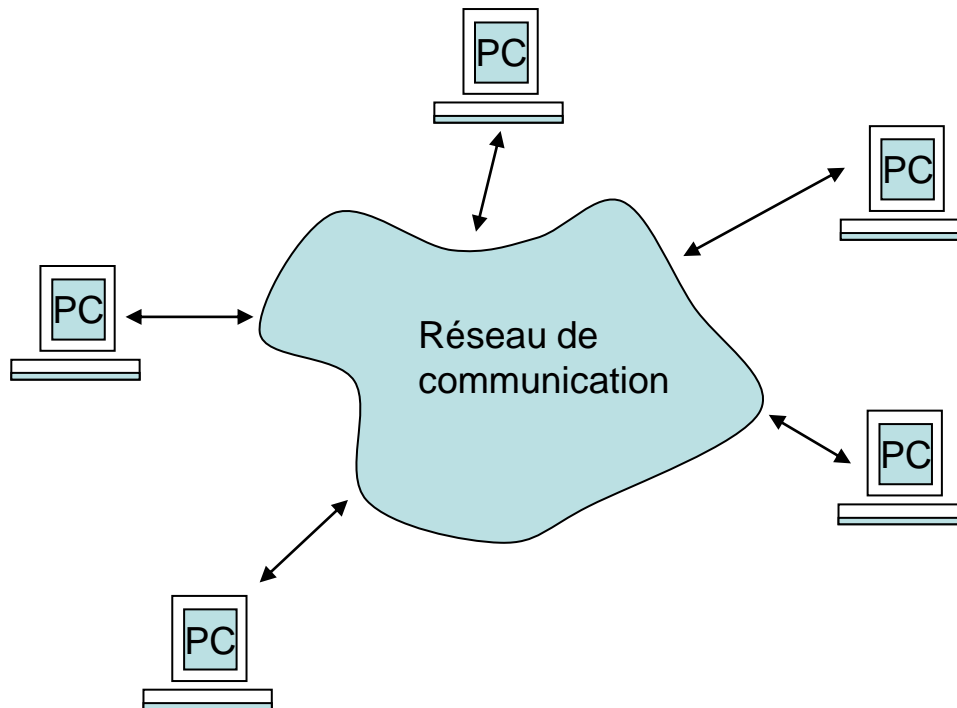
# Généralités

On réserve généralement le terme **programmation parallèle** aux programmes qui s'exécutent vraiment en parallèle et communiquent en utilisant une mémoire partagée. Un programme concurrent n'est pas nécessairement parallèle (systèmes multitâches).



# Généralités

Un **système distribué** est composé de systèmes qui communiquent entre eux par un réseau de communication en utilisant des **messages**.



# Programmation concurrente en Java

Le langage java permet de composer un programme en différents **threads**, qui sont exécutés par la même machine virtuelle java (JVM) et donc peuvent partager les mêmes ressources.

Le modèle concurrent de java est basé sur la notion d'objets actifs qui s'exécutent concurremment. Ces objets encapsulent un thread. Pour le programmeur cela revient à utiliser des instances de la classe Thread.



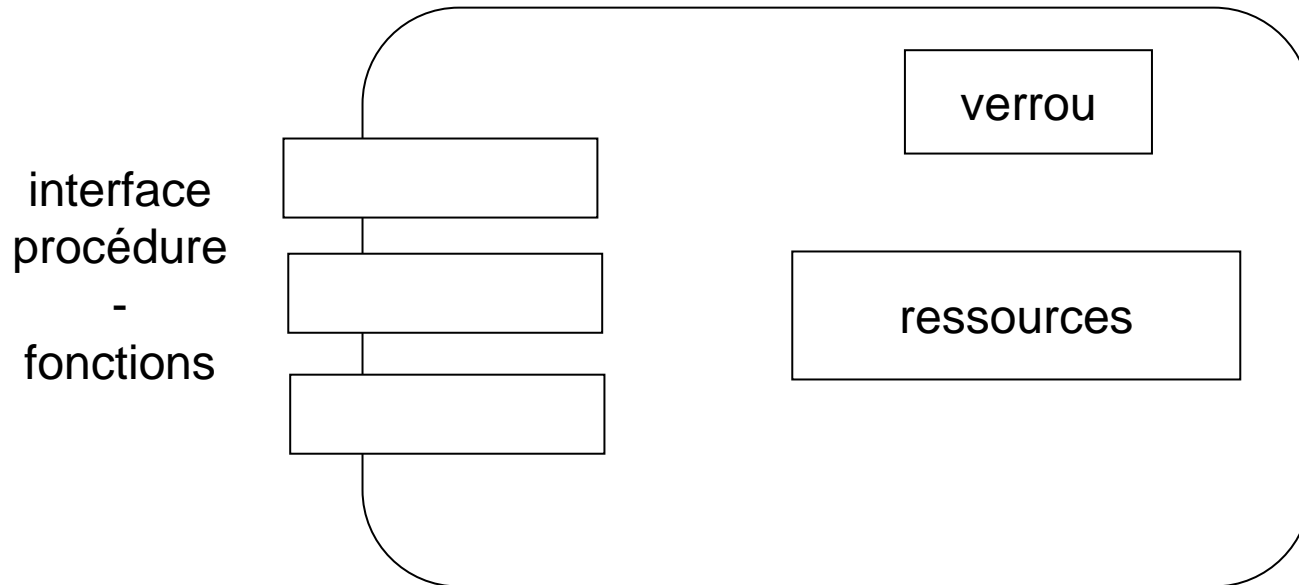
# Programmation concurrente en java

Les mécanismes de communication et de synchronisation proposés par Java sont inspirés de la notion de **moniteur**.

Un moniteur encapsule une/des ressources partagées (généralement des variables) et propose des procédures/fonctions qui permettent de manipuler ces ressources.

Les moniteurs sont des extensions des objets car les procédures/fonctions s'exécutent de manière **atomique**, c'est-à-dire que les différentes exécutions ne peuvent pas interférer. En effet, les exécutions sont **mutuellement exclusives**: A chaque moniteur est associé un verrou (lock) que chaque thread/processus doit acquérir avant de d'exécuter une procédure/fonction et restituer après l'exécution.

# Un moniteur

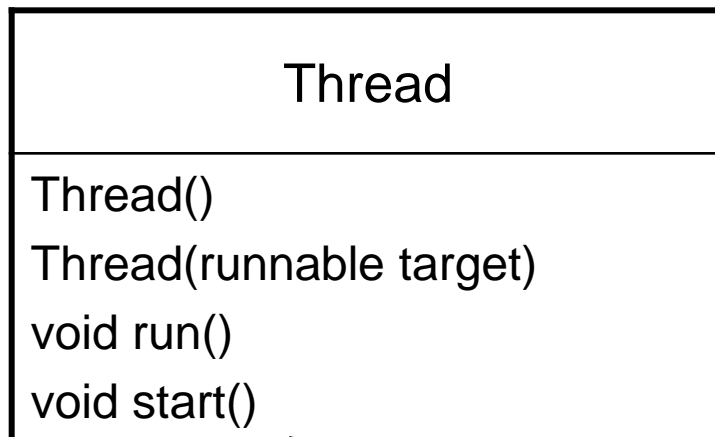


L'exclusion mutuelle est assurée pour l'accès aux ressources encapsulées, une liste des threads en attente est aussi associées au moniteur.

# Hello World

En Java un thread est une instance d'une classe.

On peut définir une telle classe en la dérivant de la classe Thread (java.lang.Thread).



effectue les opérations nécessaires pour que le thread soit pris en compte par la JVM, puis exécute la méthode run()

# Hello world

```
public class HelloWorldThread extends Thread
```

```
{
```

```
    public void run()
```

```
{
```

```
        System.out.println("Hello, world");
```

```
}
```

point d'entrée du  
programme



```
    public static void main(String[] args)
```

```
{
```

```
        HelloWorldThread t = new HelloWorldThread();
```

```
        t.start();
```

```
}
```

```
}
```

création d'un objet de  
la classe

exécution du thread



# plusieurs threads

```
class Ecrit extends Thread
{
    public Ecrit(String texts, int nb)
    { this.texte = texts;
      this.nb = nb;
    }
    public void run()
    { for(int i=0; i<nb; i++)
      System.out.print(texte);
    }
    private String texte;
    private int nb;
}
```

} variables locales, une copie par thread

# Plusieurs threads

```
public class TstThread1
{ public static void main(String args[ ])
  { Ecrit e1= new Ecrit("bonjour ", 10);
    Ecrit e2 = new Ecrit("bonsoir ", 12);
    Ecrit e3 = new Ecrit("\n ",5);

    e1.start();
    e2.start();
    e3.start();
  }
}
```

point d'entrée du programme

création des objets

exécution des threads

# Premier exemple d'exécution

```
K:\Cours\systemesdistribues\prgJava\chap2>  
K:\Cours\systemesdistribues\prgJava\chap2>  
K:\Cours\systemesdistribues\prgJava\chap2>  
K:\Cours\systemesdistribues\prgJava\chap2>  
K:\Cours\systemesdistribues\prgJava\chap2>javac Ecrit.java  
  
K:\Cours\systemesdistribues\prgJava\chap2>javac TstThread1.java  
  
K:\Cours\systemesdistribues\prgJava\chap2>java TstThread1  
bonjour bonjour bonjour bonjour bonjour bonjour bonjour bonjour bonjour bonjour  
bonsoir bonsoir bonsoir bonsoir bonsoir bonsoir bonsoir bonsoir bonsoir bonsoir  
bonsoir bonsoir  
  
K:\Cours\systemesdistribues\prgJava\chap2>
```

# Premier exemple d'exécution

On constate que les threads se sont exécutés dans l'ordre dans lequel ils ont été créés. Les threads sont toujours exécutés dans le même ordre (Pas une règle: dépendent de l'environnement).

Un thread peut explicitement interrompre son exécution pendant une période de temps  $t$  (millisecondes) en exécutant la méthode *sleep(t)*.

```
public void static mySleep(int time)
{ try {
    Thread.sleep( time );
} catch{ InterruptedException e) {} // le bloc try est imposé par sleep(t)
}
```



# Utilisation de Thread.sleep(t)

```
public void run ()
{ try
  { for( int i=0 ; i<nb ; i++)
    { System.out.print(texte);
      sleep(attente); // on étend la classe Thread
    }
  } catch (InterruptedException e) {} // nécessaire pour sleep()
}
```

# Attente aléatoire

```
import java.util.Random;

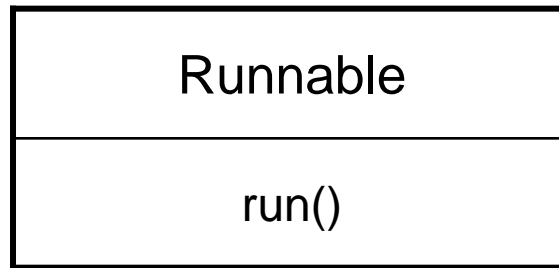
try {
    Thread.sleep(r.nextInt(10));
} catch (InterruptedException ie) {}
```

# Remarques

- La méthode *main()* qui correspond au programme principal est le **thread principal**.
- La méthode *sleep()* permet de donner la main à un autre thread (y compris le thread principal).
- On peut invoquer seulement la méthode *run()*, le programme s'exécute mais dans un seul thread.
- La méthode *start()* ne peut être appelée qu'une seule fois pour un objet donné.

# Interface Runnable

Il est possible de créer un thread en implémentant l'interface *Runnable*, laquelle comporte une seule méthode *run()*.



# Interface Runnable et classe Thread

En fait, la classe *Thread* implémente l'interface *Runnable*

```
package java.lang;
```

```
public class Thread extends Object implements Runnable
{ public Thread();
  public Thread(String name);
  public Thread(Runnable target);
  public Thread(Runnable target, String name);
  public Thread(Runnale target, String name, long stackSize);

  public static Thread currentThread();
  public void run();
  public void start();
  .....
}
```

# Exemple

**class** Ecrit **implements** Runnable

{ **public** Ecrit (String texte, **int** nb, **long** attente)

{ this.texte = texte;

  this.nb = nb;

  this.attente = attente;

}

**public void** run()

{ **try**

{ **for** (int i=0; i<nb; i++)

{ system.out.print(texte);

  Thread.sleep(attente);

} } **catch** (InterruptedException e) {}

}

**private** String texte; **private** int nb; **private long** attente;

}

}  
définition de la  
méthode run()

# Exemple (suite)

On peut maintenant créer une instance de la classe Ecrit

```
Ecrit e1 = new Ecrit("bonjour ", 10,5);
```

Ensuite on crée le thread

```
Thread t1 = new Thread(e1);
```

Que l'on peut exécuter

```
t1.start();
```

# Définition de la méthode *start()*

Dans la classe Ecrit, il est possible de définir la méthode `start()` comme dans la classe Thread

```
public void start()  
{ Thread t = new thread(this);  
  t.start();  
}
```



# Interruption d'un thread

Java dispose d'un mécanisme permettant à un thread d'en interrompre un autre.

La méthode `Thread.interrupt()` positionne un indicateur qui indique au thread qu'une requête 'interrupt' à été déposée.

Un thread peut connaître l'état de l'indicateur à l'aide de la méthode statique *`interrupted()`*.

Thread 1

`Thread2.interrupt();`

Thread 2

....

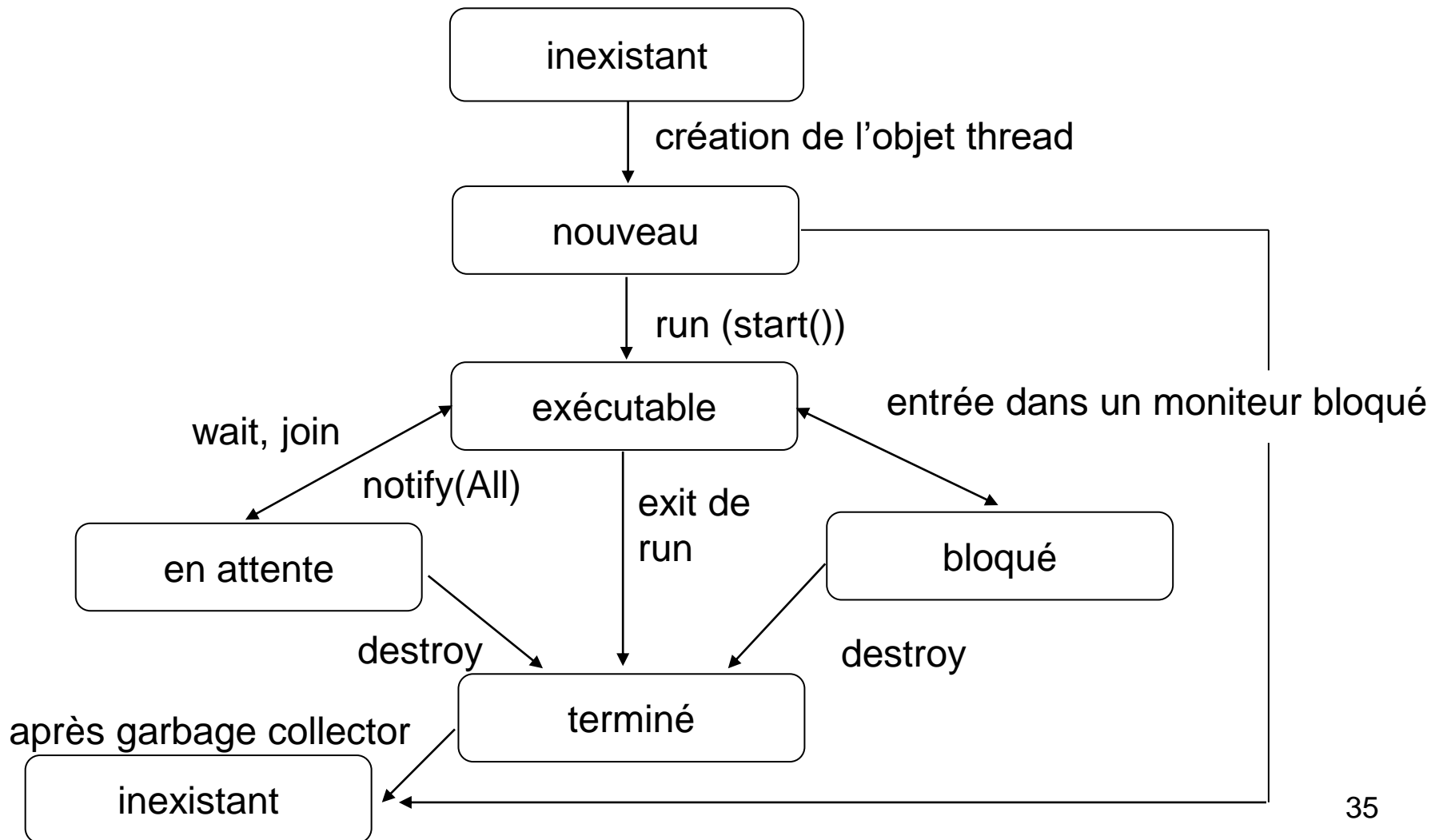
`if(interrupted())`

`return;        // fin du thread`

# Remarques

- La prise en compte du signal 'interrupt' par le thread reste sous sa propre responsabilité.
- La méthode `Tread.interrupt()` n'a pas d'effet immédiat sur le thread.
- Ce dernier doit périodiquement scruter l'indicateur en exécutant la méthode *interrupted()*.
- La méthode *interrupted()* positionne l'indicateur à *false* après lecture.
- La méthode *isInterrupted()* s'exécute comme la méthode *interrupted()* mais ne modifie pas l'indicateur après lecture.
- Les méthodes *wait()*, *sleep()*, *join()* testent l'état de l'indicateur et lèvent l'exception *InterruptedException* si l'indicateur est positionné à *true*.

# Les différents états d'un thread



# Ordonnancement des threads

A chaque thread est associée une priorité qui peut être  
MAX\_PRIORITY, MIN\_PRIORITY, NORM\_PRIORITY (par défaut)

Les méthodes de la classe Thread

```
public final void setPriority(int newPriority)
```

```
public final int getPriority()
```

permettent de modifier/lire la priorité d'un thread.

Lorsque l'ordonnanceur peut donner la main (p. ex. fin du time slice) à un thread il choisit celui de plus haute priorité, si plusieurs threads sont candidats le choix dépend de la JVM.

Si un thread plus prioritaire que le courant devient exécutable, l'ordonnanceur lui donne la main, le courant passant dans l'état exécutable.

# Un exemple avec *join()*

La méthode *join()* permet à un thread d'attendre qu'un autre thread ait fini son exécution.

Pour illustrer comment on peut utiliser *join()*, on considère le calcul des nombre de Fibonacci  $F_n$   $n \geq 0$ . Ils sont définis récursivement par:

$$F_0 = F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \quad n \geq 2$$

L'idée est de procéder comme on le ferait avec un programme récursif. C'est-à-dire, pour calculer  $F_n$  on exécute deux threads, l'un qui calcule  $F_{n-1}$  et l'autre  $F_{n-2}$ . Une fois que les deux threads ont terminés leur exécution, on calcule .

# Fibonacci

```
Public class Fibonacci extends Threads {  
    int n; int result;  
    public Fibonacci( int n) {  
        this.n = n;  
    }  
    public void run() {  
        if ((n==0) || (n==1)) result = 1;  
        else {  
            Fibonacci f1 = new Fibonacci(n-1);  
            Fibonacci f2 = new Fibonacci(n-2);  
            f1.start();  
            f2.start();  
            try{  
                f1.join();  
                f2.join();  
            } catch (InterruptedException e) {};  
            result = f1.getResult() + f2.getResult();  
        }  
    }  
}
```

# Fibonacci (suite)

```
public int getResult() {  
    return result;  
}
```

```
public static void main(String[] args) {  
    Fibonacci f1 = new Fibonacci(Integer.parseInt(args[0]));  
    f1.start();  
    try{  
        f1.join();  
    } catch (InterruptedException e) {};  
    System.out.println(" La réponse est " + f1.getResult());  
}  
}
```

# Exclusion mutuelle



# Problème de l'exclusion mutuelle

Lorsque plusieurs threads accèdent des variables partagées, les accès à ces variables doivent être synchronisés.

Thread 1

```
....  
x = x + 1  
....
```

Thread 2

```
....  
x = x + 1  
....
```

x est une variable partagée

# Exclusion mutuelle

Les instructions  $x = x + 1$  se décomposent en plusieurs sous-instructions:

1. lecture de  $x$
2. addition ( $x+1$ )
3. écriture de  $x$

il existe un scénario qui ne modifie pas la variable  $x$  deux fois

Thread 1

lecture  $x$

addition

écriture de  $x$

Thread 2

lecture de  $x$

addition

écriture de  $x$

# Atomicité

Cet exemple d'exécution montre que la variable  $x$  ne peut pas être utilisée comme compteur par exemple.

Pour résoudre le problème l'instruction  $x=x+1$  doit s'exécuter de manière **atomique**, c'est-à-dire que les deux accès à la variable  $x$  (l/e) doivent s'effectuer de manière **indivisible**.

Une portion de code qui doit s'exécuter de manière atomique est une **section critique (SC)**.

Une solution consiste à inclure l'instruction dans une section critique qui est protégée par un verrou.

# Section critique

On protège donc les instructions  $x=x+1$  par un protocole d'entrée/sortie de la section critique

Thread 1

Entrée en SC

$x = x + 1$

Libère la SC

Thread 2

Entrée en SC

$x = x + 1$

Libère la SC

# verrou

un verrou peut-être vu comme un objet Java dont l'interface est:

```
public interface Lock      // voir l'interface Runnable pour l'utilisation
{
    public void requestCS(int pid); // requête pour entrer en SC
    public void releaseCS(int pid); // indique que le thread quitte la SC
}
```

La méthode *requestCS(int pid)* est bloquante, c'est-à-dire que si un processus se trouve en section critique au moment de son exécution le processus appelant est bloqué.

Les méthodes doivent assurer que jamais plus d'un processus se trouve en SC (safety, sûreté) et qu'un processus qui désire entrer en SC le fera (liveness, vivacité).

# Verrou

## Thread1

```
...  
requestCS(1);  
...  
code de la section critique  
...  
releaseCS(1);
```

} Un thread qui effectue requestCS avant que Thread1 ait exécuté releaseCS sera bloqué.

# Exceptions

Lorsqu'un thread à obtenu un verrou, il est important pour le bon fonctionnement du programme qu'il le libère quoi qu'il arrive, en particulier si une exception est levée pendant l'exécution du code correspondant à la section critique. Pour cela, on écrit:

```
mutex.requestCS();  
try {  
    ... le corps de la section critique ...  
} finally {  
    mutex.releaseCS(); // est exécutée qu'une exception soit levée  
}                      // ou pas...
```

# Programme test

```
import java.util.Random;

public class MyThread extends Thread {
    int myId; Lock lock; Random r= new Random();

    public MyThread(int id, Lock lock) {
        myId = id;           // chaque processus possède une identité propre
        this.lock = lock;    // les processus utilisent le même verrou pour accéder la SC
    }

    void nonCriticalSection() {
        System.out.println(myId + " n'est pas en SC ");
        mySleep(r.nextInt(1000));
    }

    void CriticalSection() {
        System.out.println(myId + " est en SC ");
        mySleep(r.nextInt(1000));
    }
}
```



# Programme test

```
public void run() {  
    while(true) {  
        lock.requestCS(myId);  
        // section critique  
        lock.releaseCS(myId);  
        // section non critique  
    }  
}  
  
public static void main(String [] args) throws Exception {  
    MyThreadt [];  
    int N = Integer.parseInt(args[0]);  
    t = new myThread[n];  
    Lock = lock new .....(N); compléter avec un algorithme mutex  
    for(int i=0; i<N; i++){  
        t[i]=new MyThread(i,lock);  
        t[i].start();  
    }  
}
```

# Première tentative – 2 processus

```
class Attempt1 implements Lock {  
    private boolean openDoor = true;  
    public void requestCS(int i) {  
        while (!openDoor) ; // attente active  
        openDoor = false; // verouille l'accès à la SC  
    }  
  
    public void releaseCS(int i) {  
        openDoor = true; // libère l'accès à la SC  
    }  
}
```

# Première tentative

Cette première tentative n'est pas correcte car si le processus perd la main après qu'il ait testé `openDoor = true` et avant qu'il ait exécuté `openDoor = false`, un processus peut entrer en section critique et finalement les deux processus se retrouveront simultanément en SC.

Le problème ici est que le test de la valeur de `openDoor` (lecture) et sa modification (écriture) ne sont pas exécutés de manière atomique.

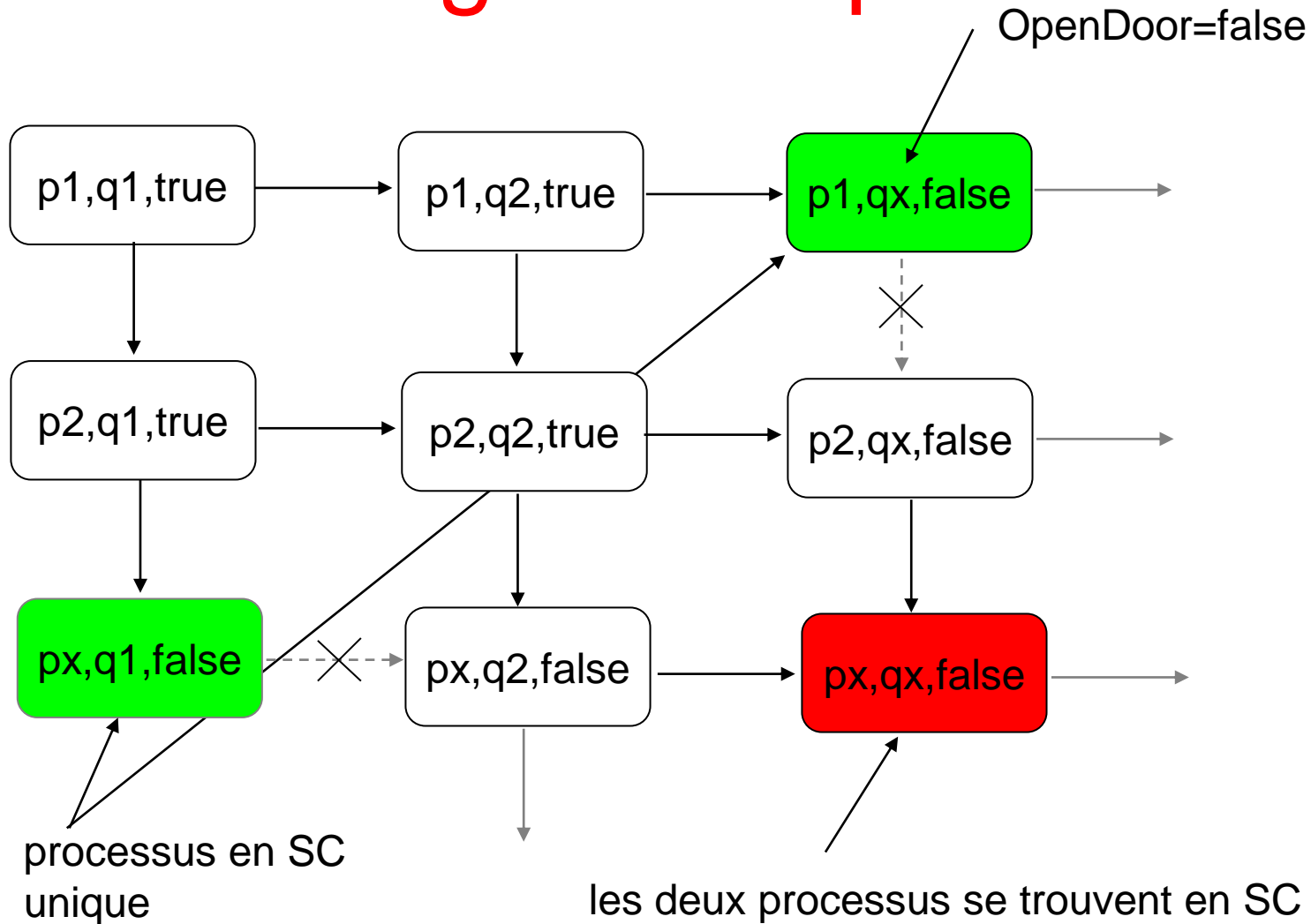
# Diagramme

```
public void requestCS(int i) {  
1   while (!openDoor) ;  
2   openDoor = false;  
}  
  
public void releaseCS(int i) {  
3   openDoor = true; // libère l'accès à la SC  
}
```

On numérote les lignes à exécuter et on utilise les lettres p et q pour désigner les deux processus. C'est-à-dire p2 indique que la prochaine instruction à exécuter par le processus p est celle qui se trouve en ligne 2.

L'état du programme est défini par le 3-tuple (pi,qj,openDoor)

# Diagramme partiel



# Deuxième tentative – 2 processus

On essaye de résoudre le problème en introduisant deux nouvelles variables partagées (une par processus) wantCS[0] et wantCS[1] que les processus utilisent pour signifier qu'ils désirent entrer en SC.

```
class Attempt2 implements Lock {  
    private boolean wantCS[] = {false, false};  
    public void requestCS(int i) {  
        wantCS[i] = true;    // réserve l'accès à la SC  
        while (wantCS[1-i]); // attente active si le second processus en SC  
    }  
    public void releaseCS(int i) {  
        wantCS[i] = false;  
    }  
}
```

# Deuxième tentative

Cette deuxième tentative assure bien **l'exclusion mutuelle**. En effet, si les deux processus se trouvent en section critique simultanément alors on a  $\text{wantCS}[0] = \text{wantCS}[1] = \text{true}$ .

Pour que le thread  $i$  entre en SC il faut qu'il teste  $\text{wantCS}[1-i] = \text{false}$  sinon il est bloqué dans la boucle **while**, c'est-à-dire que le thread  $1-i$  n'ait pas encore exécuté  $\text{wantCS}[1-i] = \text{true}$ . Ce processus va donc être bloqué par le test dans la boucle **while** (car  $\text{wantCS}[i] = \text{true}$ ).

On suppose implicitement que l'exécution de  $\text{wantCS}[i] = \text{true}$  par le processus  $i$  est vue immédiatement par le processus  $1-i$  (entrelacement des instructions).

# Deuxième tentative

Le problème avec cette deuxième tentative est que les deux processus peuvent positionner  $\text{wantCS}[i] = \text{wantCS}[1-i] = \text{true}$  et se trouvent simultanément bloqués par la boucle **while**.

On a possibilité d'inter blocage.

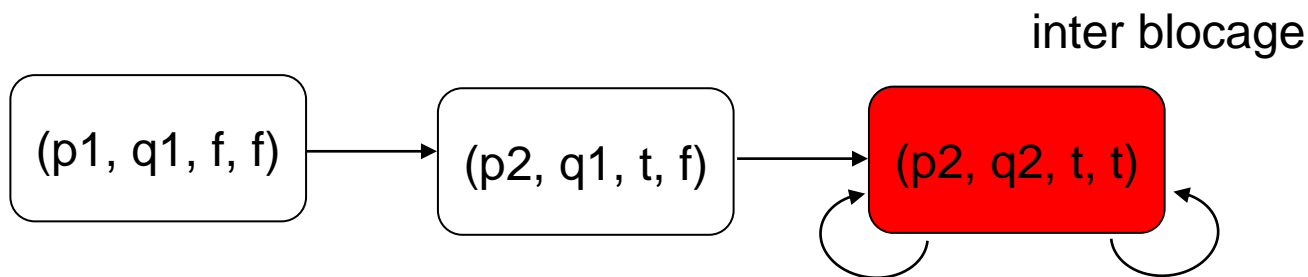
Pour dessiner le diagramme, l'état de l'algorithme est déterminé par le processus actif  $p$  ( $i=0$ ) et  $q$  ( $i=1$ ), l'état  $\text{wantp}$  et  $\text{want q}$



# Diagramme

```
class Attempt2 implements Lock {  
    boolean wantCS[] = {false, false};  
    public void requestCS(int i) {  
1        wantCS[i] = true;    // réserve l'accès à la SC  
2        while (wantCS[1-i]); // attente active si le second processus en SC  
    }  
    public void releaseCS(int i) {  
3        wantCS[i] = false;  
    }  
}
```

L'algorithme peut donner lieu à un inter blocage.



# Troisième tentative

Pour éviter l'inter blocage on peut " inverser" la stratégie et utiliser une variable booléenne pour favoriser l'accès à l'autre processus.

```
class Attempt3 implements Lock {  
    private int turn = 0;  
    public void requestCS(int i) {  
        while(turn==1-i); // attente active, ce n'est pas notre tour  
    }  
  
    public void releaseCS(int i) {  
        turn = 1-i; // on sort de SC et on libère pour l'autre processus  
    }  
}
```

# Troisième tentative

Avec cette solution les processus sont obligés d'alterner leurs accès en section critique. En effet, si le processus  $i$  quitte la section critique il peut y retourner seulement si le processus  $1-i$  y accède et lui permet l'accès.

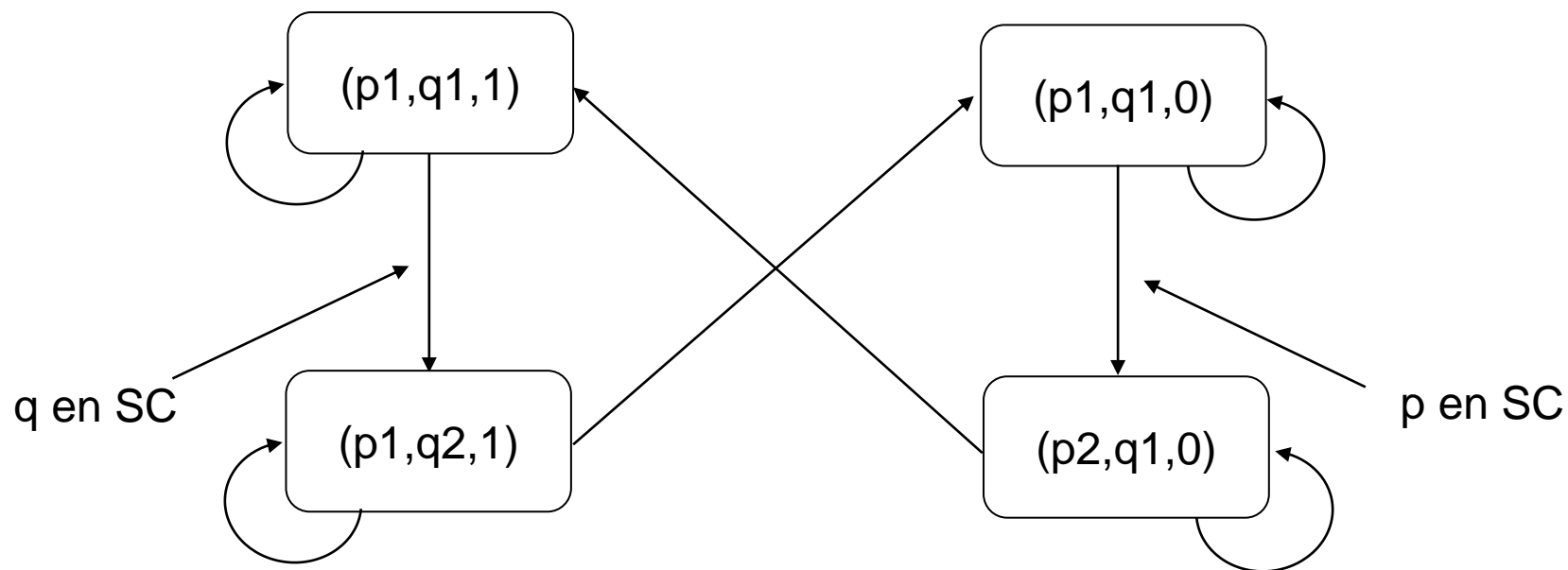
Si le processus  $1-i$  n'accède pas la section critique, alors le processus  $i$  ne peut plus jamais l'accéder.

# Diagramme

```
class Attempt3 implements Lock {  
    private int turn = 0;  
    public void requestCS(int i) {  
1        while(turn==1-i); // attente active, ce n'est pas notre tour  
    }  
  
    public void releaseCS(int i) {  
2        turn = 1-i; // on sort de SC et on libère pour l'autre processus  
    }
```

On a deux processus p ( $i=0$ ) et q ( $i=1$ ) et une variable partagée turn, (px,py,turn) définit l'état courant de l'algorithme

# Diagramme



L'algorithme assure l'exclusion mutuelle, l'état  $(p2, q2, \dots)$  n'apparaît pas dans le diagramme.

# Algorithme de Peterson

*G.L. Peterson Myths about the mutual exclusion problem, Information processing letters, 12(3):115-116, 1981.*

```
class PetersonAlgoritm implements Lock {  
    private boolean wantCS[] = {false, false};  
    private int turn = 1;  
    public void requestCS(int i) {  
        int j = 1-i;  
        wantCS[i] = true;  
        turn = j;  
        while (wantCS[j] && (turn == j)); // les deux processus ne peuvent plus  
    }                                     // se trouver simultanément en SC,  
    public void releaseCS(int i) {      // l'alternance n'est plus nécessaire  
        wantCS[i] = false;  
    }  
}
```

écriture de wantCS  
écriture de turn  
lecture de wantCS et turn

# Algorithme de Peterson

L'algorithme assure l'exclusion mutuelle:

On suppose le processus p ( $i=0$ ) se trouve en SC, c'est-à-dire p doit lire  $wantCS[1]=false$  ou  $turn=0$

**Cas 1:** p lit  $wantCS[1]=false$ .

avant de rentrer en SC le processus q ( $i=1$ ) doit positionner  $wantCS[1]=true$ .

p lit  $wantCS[1]=false \longrightarrow$  q écrit  $wantCS[1]=true$  **hypothèse**

la flèche indique une relation de précédence.

q écrit  $wantCS[1]=true \longrightarrow$  q écrit  $turn=0$  ordre du programme (p.o)  
(intra-processus)

# Algorithme de Peterson

On obtient la séquence suivante:

p écrit turn=1  $\xrightarrow{\text{p.o.}}$  p lit wantCS[1]=false  $\xrightarrow{\text{hyp.}}$  q écrit wantCS[1]=true  
 $\xrightarrow{\text{p.o.}}$  q écrit turn=0  $\xrightarrow{\text{p.o.}}$  q lit turn

**alors q lit turn = 0**

p écrit wantCS[0]=true  $\xrightarrow{\text{p.o.}}$  p lit wantCS[1]=false  $\xrightarrow{\text{par hypothèse}}$   
q écrit wantCS[1]=true  $\longrightarrow$  q lit wantCS[0]

**alors q lit wantCS[0]=true**

**le processus q est bloqué par la boucle while**

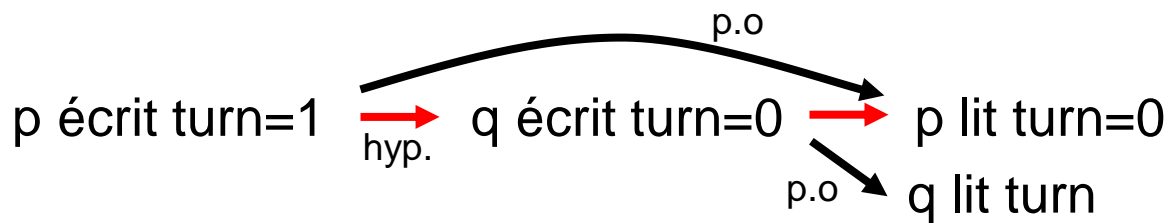
q ne peut pas entrer en SC



# Algorithme de Peterson

**Cas 2:** p lit turn=0

Dans ce cas, q doit positionner turn=0 après que p exécute turn=1 et avant que p lise turn



**q doit nécessairement lire turn=0** (première condition d'attente)

On insiste sur le fait qu'en reconstruisant la séquence des événements on doit être cohérent avec la séquence induite par l'ordre du programmes (p.o.)

# Algorithme de Peterson

Le processus p écrit  $\text{wantCS}[0]=\text{true}$  avant d'écrire  $\text{turn}=1$

p écrit  $\text{wantCS}[0]=\text{true}$   $\xrightarrow{\text{p.o.}}$  p écrit  $\text{turn}=1$   $\xrightarrow{\text{hyp.}}$  q écrit  $\text{turn}=0$   $\xrightarrow{\text{p.o.}}$   
q lit  $\text{wantCS}[0]$

**q doit nécessairement lire  $\text{wantCS}[0]=\text{true}$**  (deuxième condition d'attente)

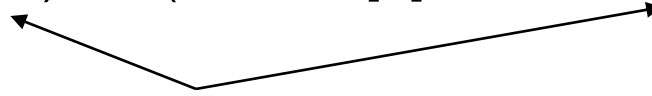
finalement q ne peut pas entrer en SC.

# Algorithme de Peterson

Aucune exécution du protocole ne peut générer un **inter-blocage**.

En effet, la condition pour que les deux processus soient en attente simultanément est que:

$(\text{wantCS}[1]=\text{true} \ \&\& \ \text{turn}=1) \ \&\& \ (\text{wantCS}[0]=\text{true} \ \&\& \ \text{turn}=0)$



De plus, un processus accède toujours à la section critique (**insuffisance de ressources, starvation**). En effet, supposons p en SC et q bloqué. Lorsque p quitte la section critique il positionne  $\text{wantCS}[0]=\text{false}$ .

Si p désire entrer en SC avant que q ait remarqué que  $\text{wantCS}[0]=\text{false}$ , il exécute  $\text{wantCS}[0]=\text{true}$  (bloquant) et  $\text{turn}=1$  qui donne l'accès au processus q.

# Algorithme de Lamport

Le nom original de l'algorithme est : Lamport's bakery algorithm car l'idée est que chaque processus qui exécute le protocole d'entrée en SC se voit attribuer un numéro et le processus qui possède le numéro le plus petit se voit attribuer l'accès en SC

Une difficulté est qu'on ne peut pas assurer que chaque processus reçoit un numéro unique (les processus sont identifiés).

Le protocole d'entrée est divisé en deux:

1. Chaque processus choisit un numéro plus grand que les numéros déjà attribués.
2. Chaque processus teste s'il peut entrer en SC
  - a) S'assure qu'aucun processus est en phase 1.
  - b) S'assure qu'il possède le plus petit numéro, en cas d'égalité les identificateurs de processus font la différence

# Lamport's Bakery algorithm

*L. Lamport, A new solution of Dijkstra's concurrent programming problem, Comm. of the ACM, 17(7), 1974.*

**class Bakery implements Lock {**

**int** N;

**volatile boolean** [] choosing; // processus en phase 1

**volatile int** [] number; // gestion de la file d'attente

**public Bakery(int numProc) {**

N = numProc;

choosing = **new int** [N];

**for** (**int** j = 0; j < N; j++) {

choosing[j] = **false**;

number[j] = 0;

}

}

initialisation du verrou  
pas de processus en phase 1.  
numéro 0 attribué à tous les processus

# Lamport's bakery algorithm

```
public requestCS(int i) {  
    choosing[i] = true;  
    for (int j = 0; j < N; j++)  
        if (number[i] < number[j])  
            number[i] = number[j];  
    number[i]++;    // on choisi le plus grand numéro  
    choosing[i]=false  
    for (int j = 0; j < N; j++) {  
        while (choosing[j]); // attente proc. en phase 1.  
        while ((number[j] != 0) && ((number[j] < number[i]) ||  
            ((number[j]==number[i] && j < i))) ; // attente active  
    }  
}
```

Phase 1. choix du numéro

$(\text{number}[j], j) < (\text{number}[i], i)$

# Lamport's bakery algorithm

```
public void releaseCS(int i) { // protocole de sortie de SC
    number[i] = 0;
}
```

La relation d'ordre introduite est:

$(\text{number}[i], i) < (\text{number}[j], j)$  si  $\text{number}[i] < \text{number}[j]$   
ou  $(\text{number}[i] == \text{number}[j] \ \&\& \ i < j)$

c'est une relation d'ordre totale si on associe a chaque processus un numéro différent.

# Preuve de l'algorithme

## 1<sup>ère</sup> assertion:

Si un processus  $P_i$  se trouve en SC et un autre processus  $P_k$  a déjà choisi un numéro alors

$$(\text{number}[i], i) < (\text{number}[k], k)$$

En effet, pour que le processus  $P_i$  se trouve en SC il doit avoir lu  $\text{number}[k]=0$  ou  $(\text{number}[i], i) < (\text{number}[k], k)$



# Preuve de l'algorithme

**Cas 1:**  $P_i$  lit  $\text{number}[k]=0$ . Alors, lors de la lecture  $P_k$  n'a pas encore choisi un numéro.

- **Cas 1.1:**  $P_k$  n'exécute pas la phase 1. du protocole d'entrée, alors  $P_k$  va lire  $\text{number}[i]$  et choisir  $\text{number}[k] > \text{number}[i]$ .
- **Cas 1.2:**  $P_k$  exécute la phase 1. du protocole d'entrée. L'entrée de  $P_k$  en phase 1. doit être postérieure au test par  $P_i$  de  $\text{choosing}[k]$ . Alors  $P_k$  va lire  $\text{number}[i]$  et choisir  $\text{number}[k] > \text{number}[i]$

# Preuve de l'algorithme

**Cas 2:**  $P_i$  lit  $(\text{number}[i], i) < (\text{number}[k], k)$

Lors de l'entrée par le processus  $i$  en SC on a bien  $(\text{number}[i], i) < (\text{number}[k], k)$ .

En SC  $P_i$  ne modifie pas la valeur de  $\text{number}[i]$ .  $P_k$  modifie  $\text{number}[k]$  pour entrer en SC, mais cette valeur peut seulement croître.

# preuve de l'algorithme

**2<sup>ème</sup> Assertion:** Si  $P_i$  est en SC alors  $\text{number}[i] > 0$ .

En effet,  $P_i$  exécute  $\text{number}[i]++$  avant d'entrer en SC.

On montre que deux processus  $P_i$  et  $P_k$  ne peuvent pas se trouver simultanément en SC. En effet, on a  $\text{number}[i] > 0$  et  $\text{number}[k] > 0$  par la deuxième assertion.

On doit donc avoir

$(\text{number}[i], i) < (\text{number}[k], k)$  et

$(\text{number}[k], k) < (\text{number}[i], i)$

mais l'ordre est total, c'est une contradiction, l'algorithme assure donc l'exclusion mutuelle.

# Preuve de l'algorithme

## Insuffisance des ressources (starvation):

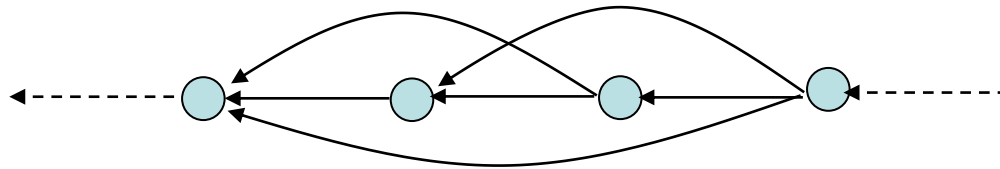
Un processus qui désire entrer en SC possède un numéro, et un nombre fini de processus se trouvent avec un numéro inférieur. Le processus aura donc accès à la SC en un temps fini.

## Remarques:

1. Les variables sont toujours modifiées par un seul processus,  $P_i$  modifie `number[i]` et `choosing[i]`
2. Le nombre d'opérations est proportionnel au nombre de processus  $O(N)$ .
3. Les numéros attribués aux processus ne sont pas bornés.

# Estampilles temporelles

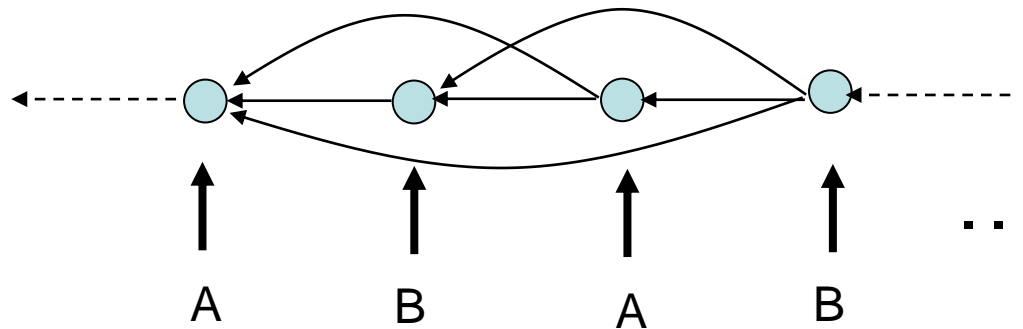
L'algorithme de Lamport utilise des compteurs *number[]* qui déterminent un ordre d'arrivée des processus. Schématiquement cet ordre se représente comme:



où chaque nœud représente une valeur de *number[]* (estampille temporelle, timestamp) et une flèche qui part d'un nœud *u* vers un nœud *v* indique que *u* est un est 'plus grand' que *v*, *u* se produit après. L'algorithme de Lamport utilise une infinité d'estampilles temporelles. Dans certaines situation un dépassement de capacité peut se produire.

# Estampilles temporelles

On considère deux processus A et B qui choisissent alternativement un nombre à chaque fois supérieur au nombre choisi par l'autre processus



On a possibilité de dépassement de capacité car le graphe de précedence utilisé est infini.

Pour résoudre ce problème on utilise un graphe de précedence fini.

# Estampilles temporelles

Pour résoudre ce problème difficile (**utiliser un nombre fini d'estampilles**), il faut d'abord modifier l'algorithme de Lamport de telle manière que la notion de précédence ne soit pas un ordre total. On peut remplacer la boucle d'attente

```
for (int j = 0; j < N; j++) {  
    while (choosing[j]); // attente proc. en phase 1.  
    while ((number[j] != 0) && ((number[j] < number[i]) ||  
        ((number[j]==number[i]) && j < i))) ; // attente active  
}
```

par (pseudo-code)

```
while(choosing[k] || (number[k],k)<<(number[i],i));
```

# Estampilles temporelles

de plus, on remplace le précédent code pour déterminer le numéro

```
for (int j = 0; j < N; j++)  
    if (number[i] > number[j])  
        number[i] = number[j];  
number[i]++;
```

par (pseudo-code)

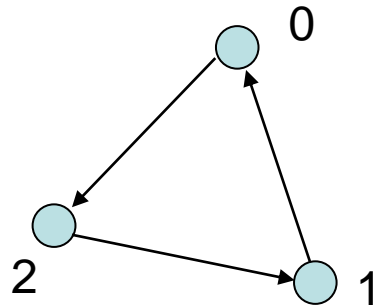
*number[i] = max + 1(number[0], ..., number[N - 1]);*

Dans cette version number[] n'est plus un nombre entier, c'est un nœud d'un graphe ....



# Estampilles temporelles

On considère le graphe de précedence suivant:

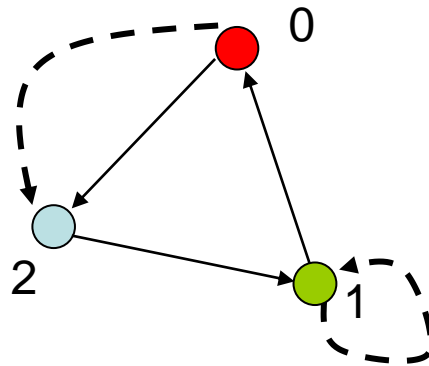


dans lequel 0 est plus petit que 1, 1 est plus petit que 2 et 2 est plus petit que 0.

**Si le nombre de processus est 2, on peut utiliser ce graphe pour générer les estampilles temporelles.**

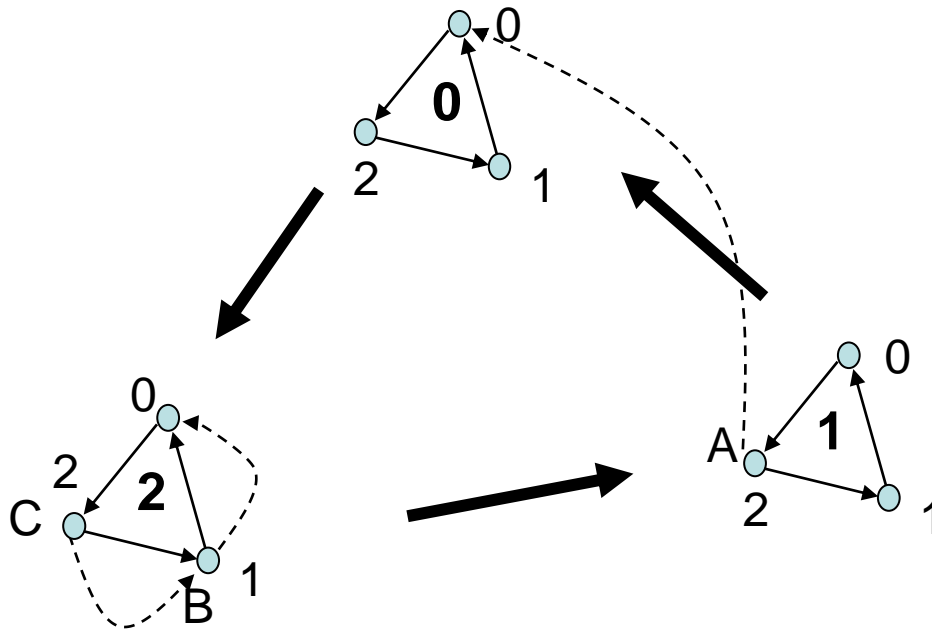
# Estampilles temporelles

En effet, supposons que le processus **A** possède l'estampille 0 et le processus **B** l'estampille 1, le numéro de **B** est plus grand.



# Estampilles temporelles

On peut généraliser le procédé pour N processus. Pour trois processus A, B, C qui se trouvent respectivement en 12, 21 et 22. B passe en 20 pour être le plus grand, puis C en 21. A passe en 00 pour être le plus grand,....



# Instructions atomiques

La difficulté principale pour assurer l'exclusion mutuelle vient du fait qu'un processus doit

1. Lire la valeur d'une variable pour tester que la SC est accessible
2. Réserver l'accès à la SC en modifiant la valeur d'une variable

et que le processus peut perdre le contrôle pendant l'intervalle de temps nécessaire à ces opérations (lecture et écriture).

En Java le mot clé **synchronized** permet de définir une routine qui ne peut être exécutée que par un seul processus simultanément.

# TestAndSet

```
public class TestAndSet {  
    int myValue = -1;
```

```
    public synchronized int testAndSet(int newValue) {  
        int oldValue = myValue;  
        myValue = newValue;  
        return oldValue;  
    }  
}
```



Section Critique,  
exclusion mutuelle,  
exécution atomique

A un instant donné, un seul processus peut exécuter la fonction testAndSet. L'environnement gère un verrou qui est attribué au processus qui exécute la fonction. Si le verrou est déjà attribué, un processus appelant est bloqué.

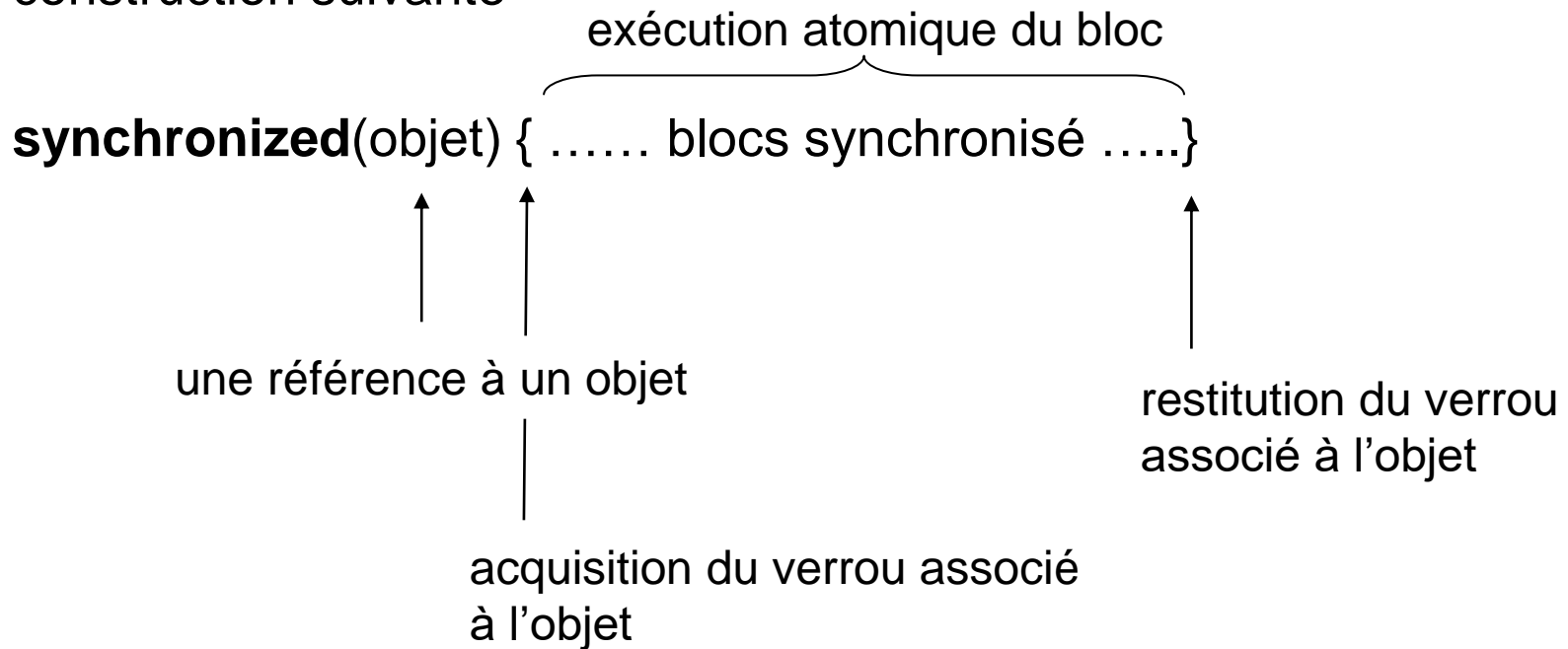
# Exclusion mutuelle avec testAndset

En utilisant testAndset on peut implémenter simplement un objet Lock

```
class HWMutex implements lock {  
    TestAndSet lockFlag;  
  
    public void requestCS(int i) { // protocole d'entrée en SC  
        while (lockFlag.testAndSet(1) == 1);  
    }  
  
    public void releaseCS(int i) { // protocole de sortie de SC  
        lockFlag.testAndSet(0);  
    }  
}
```

# Remarques

On a défini une méthode synchronisée, c'est un cas particulier de la construction suivante



# Remarques

L'utilisation de section de code synchronisée étant bloquante il faut limiter l'utilisation de ce mécanisme aux seules portions de code où c'est nécessaire. Sinon, les applications perdent en efficacité.

Java ne spécifie pas quel processus est sélectionné lorsqu'il y a contention pour un verrou. Il n'y a pas de garantie de **vivacité**.

Les blocs synchronisés en java sont dits **réentrant**, c'est-à-dire que si un processus dispose du verrou sur un objet et qu'il redemande un verrou sur le même objet, il l'obtient. Ce qui veut dire que l'environnement gère l'acquisition d'un verrou et mémorise le processus qui dispose du verrou. Un processus peut donc posséder plusieurs fois un verrou, chaque opération *unlock* annule un seul *lock*.

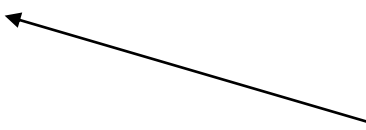
les blocs non réentrants peuvent être la source d'inter blocages.



# Remarques

```
public class Widget {  
    public synchronized void doSomething() { ....}  
}  
}
```

```
public class LogginWidget extends Widget {  
    public synchronized void doSomething {  
        System.out.println(.....);  
        super.doSomething();  
    }  
}
```



cet appel serait la source d'un  
inter blocage si le verrou n'était  
par réentrant.

# Remarques

Un verrou est associé à chaque objet. Pour implémenter une sémaphore binaire on peut donc utiliser n'importe quel objet dérivé de la class *Object*.

```
Object obj = new Objetc();
```

Ensuite, on synchronise le code de la section critique

```
synchronized (obj) {  
    ....  
    section critique  
    ....  
}
```

# Remarques

Si une méthode synchronisée est **static** le verrou est associé à la classe. Il y a un seul verrou commun à tous les objets instances de la classe.

Lorsque l'exécution d'une méthode se termine normalement ou pas, une opération *unlock()* est automatiquement générée.

# Exclusion mutuelle en pratique

En pratique pour garantir l'exclusion mutuelle on utilise une queue FIFO.

Une solution simple pour implémenter une telle queue dans un environnement concurrent est de synchroniser les méthodes de gestion de la queue.

```
public class queue {  
    private int head = 0, tail = 0;  
    Item[QSIZE] items;  
    public synchronized void enq(Item x) {  
        while (this.tail - this.head == QSIZE) // on attend si la queue est pleine  
            this.wait();  
        this.items[this.tail++] = x; // on insère l'élément  
        this.notifyAll(); // on informe les processus bloqués après deq  
    }
```

on acquiert le verrou avant d'accéder,  
on le libère après

on incrémente après évaluation

# Queue bloquante

Utiliser des méthodes synchronisées permet d'assurer le bon fonctionnement du programme.

Il y a des situations où ce n'est pas obligatoire, par exemple si un seul processus appelle *enq()* et un seul processus appelle *deq()*.

Synchroniser les processus réduit les performances d'un algorithme.

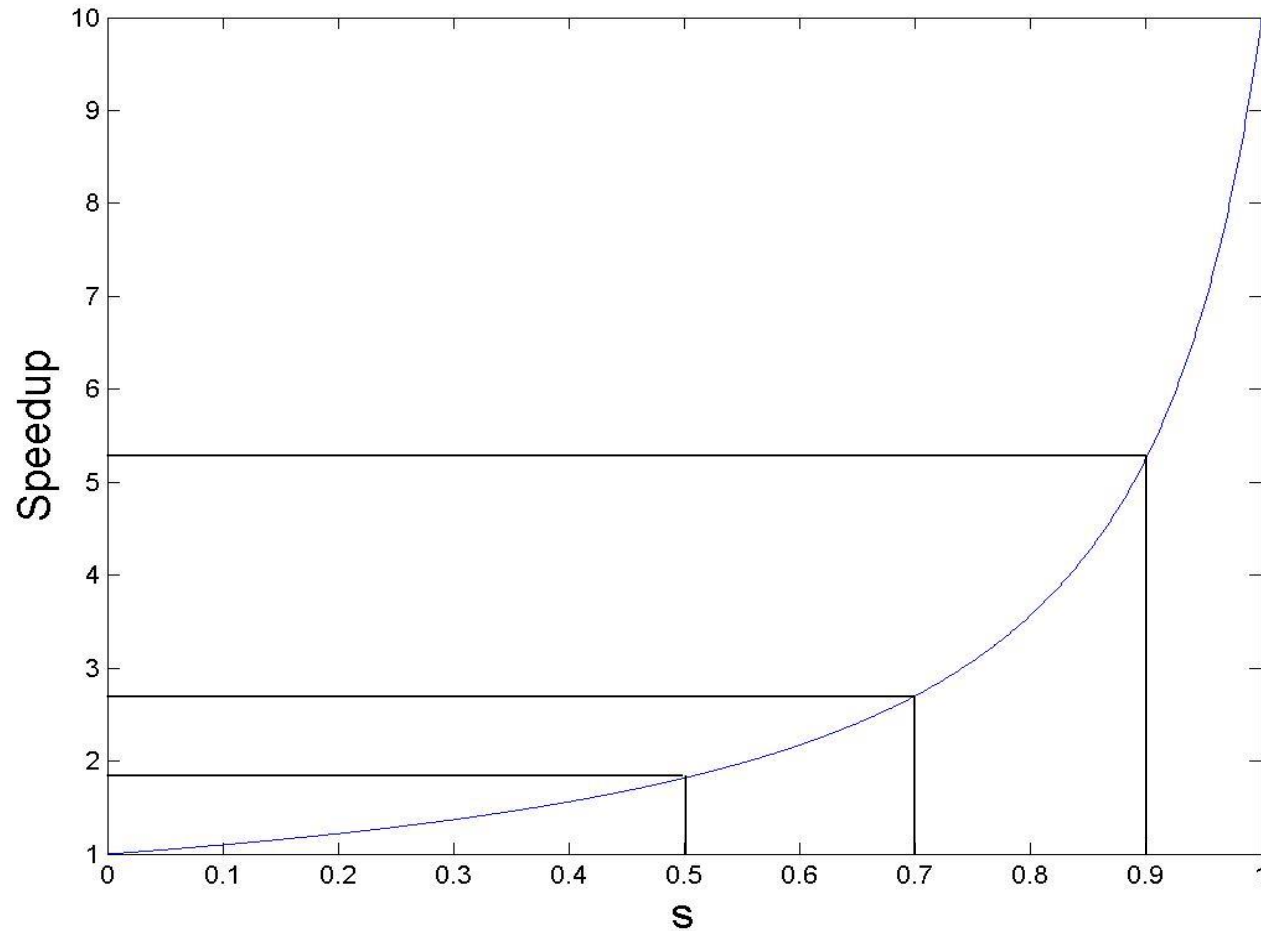
**Rappel:** La loi d'Amdahl

Soit  $s$  la proportion de programme parallélisable, et  $N$  processus à disposition

$$speedup = \frac{1}{1 - s + \frac{s}{N}}$$

# Queue bloquante

loi d'Amdhal 10 processeurs



# Queue bloquante

On observe que le speedup diminue très rapidement en fonction de la proportion de code parallèle. Les méthodes *synchronisées* ne sont pas exécutables en parallèle et pénalisent les performances du système.

Des objets complexes sont implémentés en utilisant d'autres mécanismes qui leur permettent d'être accédés par plusieurs processus simultanément sans mécanisme de blocage (voir plus tard, lock-free algorithm).

Dans un système réel les processus sont susceptibles d'être interrompus, par exemple pour le traitement d'une exception. C'est particulièrement gênant si le processus interrompu exécute une méthode synchronisée.

Néanmoins, il est important d'utiliser les mécanismes de synchronisations pour s'assurer du bon fonctionnement du programme.

# Preuves des algorithmes concurrents

Pour les algorithmes présentés on a développé des preuves qui sont basées sur l'analyse des différentes exécutions possibles (cas1, cas2, ...). Ce type d'analyse peut être formalisé en utilisant les diagrammes d'états et de transitions. On a deux difficultés avec cette approche:

1. Il est difficile de se persuader que l'on a pas 'oubliée' de traiter une situation, la construction du diagramme est problématique dès que le nombre d'état devient grand. Les exemples ont principalement montrés des fragments de diagrammes qui montrent que les algorithmes ne fonctionnent pas (cas particulier). Cette approche est problématique pour montrer des propriétés qui ne doivent jamais se produire (safety).



# Preuves des algorithmes concurrents

2. La méthode est difficilement 'mécanisable', par exemple si on veut développer un programme qui vérifie automatiquement qu'une propriété donnée est toujours vérifiées.

De plus, trouver les erreurs dans les programmes concurrents par essai-erreur-correction (test) est souvent difficile voir impossible:

1. Les erreurs peuvent se produire seulement dans des conditions très particulières (liées à la course entre les données, data race) qui peuvent se produire tous les mois, les années,...
2. Quelque fois les mécanismes d'observations ne permettent plus aux exécutions fautives d'exister...
3. Comment peut-on décider que plusieurs processus sont inter bloqués? (il faut définir une période de temps)

# Preuve des algorithmes concurrents

Pour toutes ces raisons, et d'autres, des méthodes formelles ont été développées et sont essentielles pour la validation des programmes concurrents. Des programmes tels que SPIN sont des outils d'aide à la validation.

Ces méthodes font l'objet d'autres cours (par exemple D.Buchs, concurrency and distribution).

Une méthode intermédiaire pour démontrer formellement des propriétés des programmes concurrents est d'utiliser des invariants.

# Invariants

On utilise exclusivement la logique des propositions.

On a un ensemble de propositions atomiques  $\{p, q, r, \dots\}$

Les opérateurs,  $\neg$  (non),  $\vee$  (ou),  $\wedge$  (et),  $\Rightarrow$  (implication),  $\Leftrightarrow$  equivalence.

Les propositions atomiques sont des expressions constituées des variables et des pointeurs de programmes.

# Invariants

Par exemple, pour la deuxième tentative

```
boolean wantCS[] = {false, false};
```

1 **section non critique**

```
    public void requestCS(int i) {
```

2 wantCS[i] = **true**;

3 **while** (wantCS[1-i]); }

4 **section critique**

```
    public void releaseCS(int i) {
```

5 wantCS[i] = **false**;

wantCS[0], wantCS[1], sont des propositions atomiques

wantCS[1] ^ p4 (le processus p exécute la ligne 4 et wantCS[1]=true, p  
le processus numéro 0 et q le processus numéro 1)

Pour la troisième tentative, on a turn=1 est une proposition atomique.

# Invariant - définition

## Définition:

Une proposition est invariante si elle est toujours vraie, c'est-à-dire dans tous les états possibles du programme.

Pour prouver que la deuxième tentative pour résoudre l'exclusion mutuelle est correcte, on doit prouver que:

$$\neg(p4 \wedge q4)$$

est toujours vraie (c'est un invariant). Une telle preuve se fait par induction

1. On prouve qu'elle est vraie initialement
2. On suppose l'assertion vraie dans tous les états jusqu'à l'état courant et on montre qu'elle est vraie dans tous les états suivants possibles.

# Preuve de l'exclusion mutuelle avec les invariants

**Lemme 1:**  $p3 \vee p4 \vee p5 \Rightarrow \text{wantCS}[0]$  est un invariant

- a. Dans l'état initial on a  $\text{wantCS}[0]=\text{false}$ , alors la proposition est vraie.
- b. (induction) Aucune exécution du processus  $q$  ne peut modifier la validité de la proposition car elle dépend que de l'état courant de  $p$  et de  $\text{wantCS}[0]$  qui est modifiée que par  $p$ .

La proposition ne peut être fausse que lorsque  $p$  exécute  $p3$ ,  $p4$  ou  $p5$  ( $1 \Rightarrow 0$ ).

Exécuter  $p3$  ou  $p4$  ne modifie pas la validité de l'assertion car dans les deux cas  $p3 \vee p4 \vee p5$  et  $\text{wantCS}[0]$  ne sont pas modifiés

Lorsque  $p5$  est exécutée alors  $p3 \vee p4 \vee p5 = \text{false}$  et l'assertion est toujours valide ( $0 \Rightarrow x$  est vraie).

Lorsque  $p2$  est exécutée alors  $p3 \vee p4 \vee p5 = \text{true}$  (le compteur de programme devient  $p3$ , mais on a bien  $\text{wantCS}[0]=\text{true}$ ).

# Preuve de l'exclusion mutuelle avec les invariants

En fait, seuls p2 et p5 peuvent modifier la validité de la proposition.

**Lemme 2:**  $\text{wantCS}[0] \Rightarrow p3 \vee p4 \vee p5$  est un invariant.

En effet, cette proposition peut être fausse que si  $\text{wantCS}[0]=\text{true}$ .

- a. Dans l'état initial  $\text{wantCS}[0]=\text{false}$ , la proposition est donc vraie
- b. Les seules exécutions qui peuvent modifier la validité de la proposition sont p2 et p5. Après p2 on a  $\text{wantCS}[0] = \text{true}$  mais le compteur de programme se trouve en p3, la proposition est valide. Après p5  $\text{wantCS}[0]=\text{false}$ ,...

# Preuve de l'exclusion mutuelle avec les invariants

**Lemme 3:**  $p3 \vee p4 \vee p5 \Leftrightarrow \text{wantCS}[0]$  et  $q3 \vee q4 \vee q5 \Leftrightarrow \text{wantCS}[1]$

Les deux lemmes précédents montrent l'équivalence et il est clair que les mêmes preuves s'appliquent au processus q.

**Théorème:**  $\neg(p4 \wedge q4)$  est invariant.

a. L'assertion est trivialement vraie initialement.

b. Seules deux exécutions doivent être vérifiées. Si p3 est exécutée avec succès lorsque q4 est vraie, et q3 est exécutée avec succès lorsque p4 est vraie (symétrique).

pour que p3 soit exécutée avec succès, il faut que  $\text{wantCS}[1] = \text{false}$ .  
D'après le lemme précédent q4 ne peut pas être vraie, c'est une contradiction et p3 ne peut pas s'exécuter avec succès.



# Remarques

Les logiques temporelles (LTL, CTL,...) permettent de généraliser l'approche basée sur les invariants.

Il est difficile de prouver des conditions de vivacité (liveness), c'est-à-dire qu'un événement souhaité se produira nécessairement avec les invariants. En logique temporelle on a un opérateur dédié.

On a toujours supposé que considérer toutes les différentes exécutions possibles revient à entrelacer (interleaving) les différentes instructions à exécuter, c'est-à-dire

p1 p2 q1 p3 p4 q2 ,...

p1 p2 p3 p4 p5 q1 q2, ...

etc.

# Synchronisation

# Mécanismes de synchronisations

Les mécanismes présentés dans la première partie réalisent l'exclusion mutuelle et utilisent des boucles d'attente actives.

Des mécanismes de synchronisations permettent

1. de réaliser l'exclusion mutuelle
2. de coordonner l'exécution des threads

sans utiliser des boucles d'attentes actives.

# Sémaphore booléenne

Une sémaphore booléenne est un objet qui permet de réaliser l'exclusion mutuelle et qui dispose de deux champs de données

1. une valeur booléenne
2. une file d'attente de processus bloqués

et deux méthodes P() et V().

Lorsque P() est exécutée on a

1. si valeur=true, alors valeur=false
2. si valeur=false, le processus appelant est bloqué

Lorsque V() est exécutée on a

1. valeur=true, si la file d'attente des processus n'est pas libre on libère un processus

# Sémaphore booléenne

```
public class Binarysemaphore {  
    private boolean value;  
    BinarySemaphore(boolean initValue) {  
        value = initValue;  
    }  
    public synchronized P() { // protocole d'entrée en SC  
        while (value == false)  
            try { this.wait() } catch (InterruptedException e) {}  
        value = false;  
    }  
    public synchronized V() { // protocole de sortie de SC  
        value = true;  
        notify();                // libère un processus en attente  
    }  
}
```

# SC avec sémaphore

Pour assurer l'exclusion mutuelle on utilise le code suivant

```
BinarySemaphore mutex = new BinarySemaphore(true);
```

```
mutex.P();
```

```
section critique
```

```
mutex.V();
```

# Remarques

Les méthodes `wait()` et `notify()` sont des méthodes de la classe `Object`.

Ces méthodes peuvent être exécutées seulement depuis des méthodes qui disposent du verrou sur l'objet, c'est-à-dire qui sont **synchronisées**. Dans le cas contraire une exception *`IllegalMonitorStateException`* est levée.

La méthode `wait()` bloque le processus appelant et libère le verrou sur l'objet. Attention, si le processus dispose de plusieurs verrous, seul le verrou associé à l'objet auquel appartient la méthode depuis laquelle `wait()` est exécutée est libéré. **Possibilité d'inter blocage.**

# Remarques

La méthode `notify()` active un processus en attente, le langage Java ne spécifie pas lequel.

Il existe une spécification temps-réel de java (RTSJ) qui spécifie l'ordre dans lequel les processus doivent être libéré.

Lorsque `notify()` est exécutée, le verrou sur l'objet n'est pas libéré (le thread qui s'exécute continue de s'exécuter).

La méthode `notifyAll()` permet d'activer tous les processus en attente, ne libère pas le verrou et les processus activés doivent essayer d'acquérir le verrou avant de continuer.

`notify()` et `notifyAll()` n'ont pas d'effet si aucun processus n'est en attente.



# Remarques

Un processus peut être réactivé en utilisant `Thread.interrupt()`

Lorsqu'un processus est réactivé, il devient exécutable mais ne s'exécute pas nécessairement immédiatement. C'est-à-dire que la condition qui à permis sa réactivation n'est pas forcément vérifiée lorsqu'il s'exécute. Dans notre implémentation des sémaphores:

**while** (value == **false**)

**try** { this.wait } **catch** (InterruptedException e) {}

Lorsque le processus est réactivé `value==false` est possible.

Pour cette raison, le code suivant **ne fonctionne pas**

**if** (value == **false**)

**try** { this.wait } **catch** (InterruptedException e) {}

# Remarques

les méthodes `Thread.sleep()` et `Thread.yield()` permettent à un thread de forcer l'ordonnanceur à changer le thread actif. Néanmoins,

1. un thread ne perd pas les verrous dont il dispose.
2. ces opérations n'ont pas d'effets sur la synchronisation; les données sauvegardées dans les registres ne sont pas copiées en mémoire principale; les données ne sont pas lues depuis la mémoire principale après l'exécution.

# Retour sur *wait()*

Il y a trois versions:

*wait()*, *wait(long millisecs)*, *wait(long millisecs, int nanosecs)*.

Si la valeur des paramètres est nulle, les appels sont équivalents à *wait()*.

La méthode *wait()* peut lever une exception `InterruptedException`

Soit un thread **t** qui exécute une méthode *wait()* dans l'objet **m** et **n** est le nombre de *lock()* de **t** sur **m**.

- si **n** = 0 (pas de *lock()* ) une exception **`IllegalMonitorStateException`** est levée
- si l'argument nanosecs n'est pas dans l'intervalle 0-999999, ou l'argument millisecs est négatif l'exception **`IllegalArgumentException`** est levée

# Retour sur *wait()*

- Si le thread **t** est **Interrupted** une exception **InterruptedException** est levée

Sinon la séquence suivante se produit

- Le thread **t** est ajouté à l'ensemble des threads en attente de l'objet **m** et exécute **n** opérations *unlock()* (libère le verrou) sur l'objet **m**.
- Le thread est bloqué jusqu'à être retiré de l'ensemble des processus en attente dans **m**. Ce qui se produit si:
  - *notify()* est exécuté dans **m** et **t** est sélectionné.
  - *notifyAll()* est exécuté dans **m**.
  - *t.interrupt()* est exécutée.
  - Le temps d'attente spécifié est écoulé.

# Retour sur *wait()*

- Le thread **t** exécute **n** *lock()* sur **m** (une fois activé).
- Si **t** à été retiré de l'ensemble des processus en attente après exécution de *t.interrupt()*, l'exception **InterruptedException** est levée.

# Retour sur *wait()*

La spécification Java laisse aux implémentations de JVM la possibilité de générer une action interne pour retirer un processus en attente sans qu'aucune des conditions citées précédemment ne se produisent. Cette pratique n'est pas recommandée, néanmoins il est spécifié de placer le *wait()* dans une boucle *while* et de tester une condition logique si nécessaire.

# Retour sur *notify()*

De même, pour *notify()*, on a

- si  $n = 0$  l'exception **IllegalMonitorStateException** est levée (le thread  $t$  ne possède pas le verrou sur  $m$ )
- si  $n > 0$  et que l'ensemble des processus en attente n'est pas vide, un processus est sélectionné et devient exécutable (*notify()*) ou idem pour tous les threads si *notifyAll()*.

# Retour sur interrupt()

Lorsqu'un thread **t** exécute **u.interrupt()** (**u** et **t** pas forcément différents) le drapeau d'état du processus **u** est positionné.

Si **u** est en attente dans un objet **m** il est retiré de l'ensemble des processus en attente. Après avoir obtenu le verrou sur l'objet, il lève l'exception **InterruptedException**.



# Remarques

Lorsqu'un thread a exécuté avec succès *semaphore.P()*, il est important qu'il exécute *semaphore.V()* en quittant la section critique pour permettre à un autre thread d'y accéder. Pour s'assurer que le thread ne bloque pas d'autres threads **si une exception est levée**, on accède la section critique comme cela:

```
semaphore.P();
```

```
try {
```

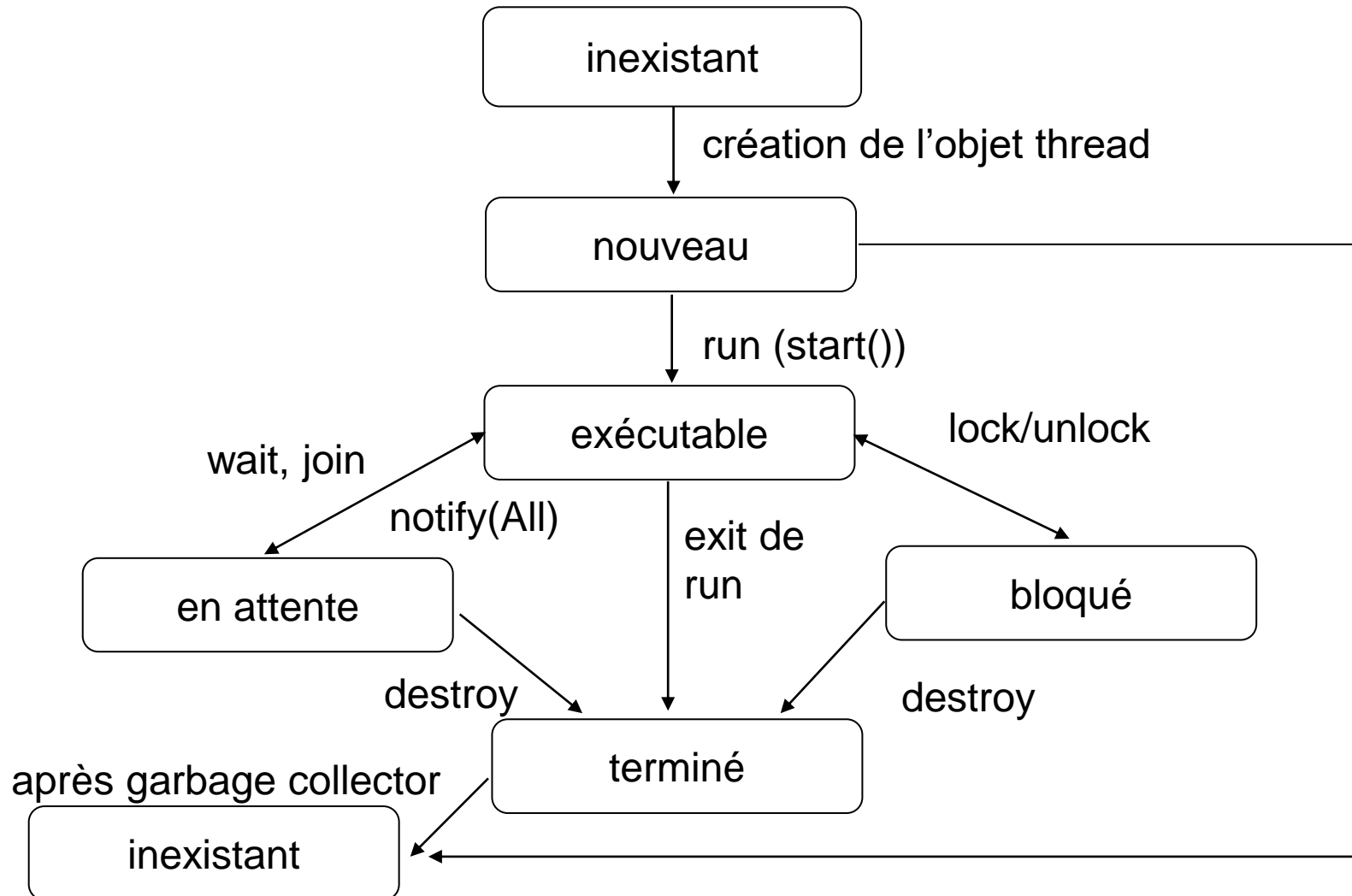
```
    ... section critique...
```

```
} finally {
```

```
    semaphore.V(); // cette instruction est toujours exécutée, qu'une
```

```
}                // exception ait été levée ou non
```

# Les différents états d'un thread



# Remarques


Les méthodes wait(), notify(), notifyAll() permettent de synchroniser les processus.

Il est aussi possible pour un thread d'attendre sur la fin (thread dans l'état terminé) d'un autre thread en utilisant la méthode join().

```
public class Exemple extends Thread { ....}
```

```
public static void main(....) {  
    Exemple objet = new Exemple(....);  
    objet.start()  
    try {  
        objet.join();  
    } catch (InterruptedException e){};  
}
```

# Sémaphore entière

```
public class CountingSemaphore {  
    private int value;  
    public CountingSemaphore(int initValue) {  
        value = initValue;  
    }  
    public synchronized void P() {  
        while (value == 0)    
            try { this.wait(); } catch (InterruptedException e) {}  
        value--;  
    }  
    public synchronized void V() {  
        value++;  
        if (value == 1) notify();  
    }  
}
```

Le thread libéré ne s'exécute pas nécessairement

# Invariants pour les sémaphores

On considère seulement les sémaphores entières, une sémaphore booléenne étant un cas particulier de sémaphore entière.

- Une sémaphore  $s$  est initialisée avec la valeur `initValue`.
- La valeur de la sémaphore  $s.value$  satisfait toujours

$$s.value \geq 0$$

- Si on note  $\text{comp\_P}(s)$  et  $\text{comp\_V}(s)$  le nombre d'exécution complète des méthodes  $P()$  et  $V()$  de la sémaphore  $s$ , on a toujours

$$s.value = \text{initValue} - \text{comp\_P}(s) + \text{comp\_V}(s)$$

# Preuve de SC

On suppose qu'une section critique est accédée par tous les processus selon le protocole déjà vu:

s.P();

section critique

s.V();

et que s est initialisée avec  $\text{initValue} = 1$ .

Supposons que deux processus se trouvent en section critique simultanément. Comme  $\text{s.P()} \longrightarrow \text{section critique} \longrightarrow \text{s.V()}$ , on a  $-\text{comp\_P()} + \text{comp\_V()} \leq -2$ .

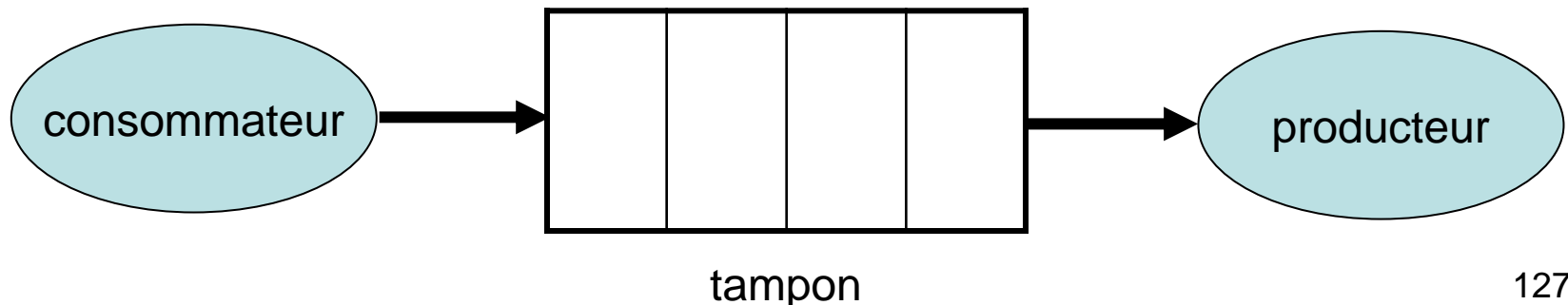
On a alors une contradiction avec les invariants associés à la sémaphore cars  $\text{cars.value} = \text{initValue} - \text{comp\_P()} + \text{comp\_V()} \leq -1$

# Le problème du producteur-consommateur

Le problème du producteur-consommateur modélise des problèmes concrets dans lesquels on peut identifier deux types de comportements

1. Un processus (producteur) qui détermine les tâches à exécuter
2. Un processus (consommateur) qui exécute les tâches.

Les deux processus communiquent en déposant/retirant des objets d'un tampon de taille finie.



# Le problème du producteur-consommateur

une solution de ce problème consiste à définir un thread producteur et un thread consommateur.

La programmation concurrente s'impose pour résoudre ce problème car:

1. Le problème s'exprime naturellement avec des threads.
2. Les programmes relatifs au producteur/consommateur peuvent être indépendants. Le producteur n'a pas besoin de connaître comment est implémenté le consommateur.
3. Avec la programmation concurrente on ne doit pas considérer explicitement les problèmes de vitesses différentes des processus.



# Le problème du producteur-consommateur

On trouve ce genre de problèmes, par exemple:

1. Accès à une ressource séquentielle, telle qu'une imprimante ou des périphériques d'entrée/sortie.
2. Un programme qui scan un disque à la recherche de fichiers et les indexe pour les retrouver plus rapidement ensuite. Le producteur scan le disque, le consommateur les indexe.
3. Un browser web produit des informations qui doivent être transmises par le consommateur via une ligne de communication
4. Le clavier produit des données qui doivent être transmises à la tâche concernée.
5. Un jeu produit des images qui doivent être affichées (consommateur)
6. ....

# Le problème du producteur-consommateur

Les contraintes de synchronisation pour ce problème sont:

- Le producteur ne doit pas ajouter un élément si le buffer est plein.
- Le consommateur ne doit pas retirer un élément si le tampon est vide.

On parle de **synchronisation conditionnelle**, un processus doit attendre qu'une condition soit satisfaite pour continuer de s'exécuter.

# Code java – le tampon avec sémaphores

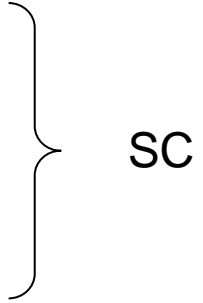
```
class BoundedBuffer {  
    final int size = 10; // taille du tampon  
    double[] buffer = new double[size]; // tampon  
    int inBuf =0, outBuf = 0; // pointeurs  
    BinarySemaphore mutex = new BinarySemaphore(true);  
    CountingSemaphore isEmpty = new CountingSemaphore(0);  
    CountingSemaphore isFull = new CountingSemaphore(size);  
    ...
```

pour l'accès en  
section critique

pour la synchronisation conditionnelle

# Code Java – le tampon avec sémaphores

```
public void deposit (double value) {  
    isFull.P();           //attente si le tampon est plein  
    mutex.P();            // accès exclusif au tampon  
    buffer[inBuf] = value; // on place la valeur  
    inBuf = (inBuf + 1) % size;  
    mutex.V();            // on quitte la section critique  
    isEmpty.V();          // notification au consommateur  
}
```



SC

...

# Code Java – le tampon avec sémaphores

```
public double fetch() {  
    double value;  
    isEmpty.P();    // attente si buffer vide  
    mutex.P();      // accès exclusif au tampon  
    value = buffer[outBuf];  
    outBuf = (outBuf + 1) % size;  
    mutex.V();      // on quitte la section critique  
    isFull.V();     // notification au producteur  
    return value;  
}  
}
```

# Utilisation du tampon fini

```
import java.util.Random;
class Producer implements Runnable {
    BoundedBuffer b = null;
    public Producer(BoundedBuffer initb) {
        b = initb;
    }
    public void run() {
        double item;
        Random r = new Random();
        while(true) {
            item = r.nextDouble();
            System.out.println("valeur produite " + item);
            b.deposit(item);
            sleep(50); // cette instruction ne fonctionne pas comme cela....
        }
    }
}
```

# Utilisation du tampon fini

```
class Consumer implements Runnable {  
    BoundedBuffer b = null;  
    public Consumer(BoundedBuffer initb) {  
        b = initb;  
    }  
    public void run() {  
        double item;  
        while (true) {  
            item = b.fetch();  
            System.out.println("valeur à consommer " + item);  
            sleep(50);    // cette instruction ne fonctionne pas comme cela...  
        }  
    }  
}
```

# Utilisation du tampon fini

```
class ProducerConsumer {  
    public static void main(String [] args) {  
        BoundedBuffer buffer = new BoundedBuffer();  
        Producer producer = new Producer(buffer);  
        Consumer consumer = new Consumer(buffer);  
        new Thread(producer).start();  
        new Thread(consumer).start();  
    }  
}
```



# Preuve de la synchronisation

On considère les méthodes *deposit(...)* et *fetch()*.

On a déjà montré que l'utilisation de la sémaphore *mutex* assure l'exclusion mutuelle lors des accès au tampon.

*deposit(..)*

...

mutex.P();

buffer[inBuf] = value;

inBuf = (inBuf + 1) % size;

mutex.V();

*fetch()*

...

mutex.P();

value = buffer[outBuf];

outBuf = (outBuf + 1) % size;

mutex.V();

# Preuve de la synchronisation

On montre que la sémaphore *isEmpty* assure que l'on a jamais **underflow**, c'est-à-dire que le consommateur n'accède pas à plus de données que le producteur a déposé.

On sait par invariance que

$$\text{isEmpty.value} = \text{initvalue} - \text{comp\_P}(\text{isEmpty}) + \text{comp\_V}(\text{isEmpty}) \geq 0$$

ou encore

$$\text{comp\_V}(\text{isEmpty}) \geq \text{comp\_P}(\text{isEmpty})$$

La méthode *deposit(..)* se termine par *isEmpty.V()* et la méthode *fetch()* commence par *isEmpty.P()*. L'inégalité ci-dessus nous indique que le nombre d'exécutions de *deposit()* (terminée) est toujours plus grand ou égal au nombre d'exécutions de *fetch()* terminée ou en cours. Il ne peut donc pas y avoir d'underflow.

# Preuve de la synchronisation

On montre que la sémaphore *isFull* assure que l'on a jamais **overflow**, c'est-à-dire que le producteur ne dépose pas plus de données que peut contenir le tampon (size).

On sait par invariance que

$$\text{isFull.value} = \text{size} - \text{comp\_P}(\text{isFull}) + \text{comp\_V}(\text{isFull}) \geq 0,$$

ou encore

$$\text{size} \geq \text{comp\_P}(\text{isFull}) - \text{comp\_V}(\text{isFull}) .$$

La méthode *deposit()* commence par *isFull.P()* et la méthode *fetch()* se termine par *isFull.V()*.

Donc le nombre d'exécutions de *deposit* en cours ou terminées moins le nombre d'exécutions de *fetch()* est toujours inférieur à la taille du tampon, il ne peut donc pas y avoir d'overflow.

# Remarque sur la synchronisation

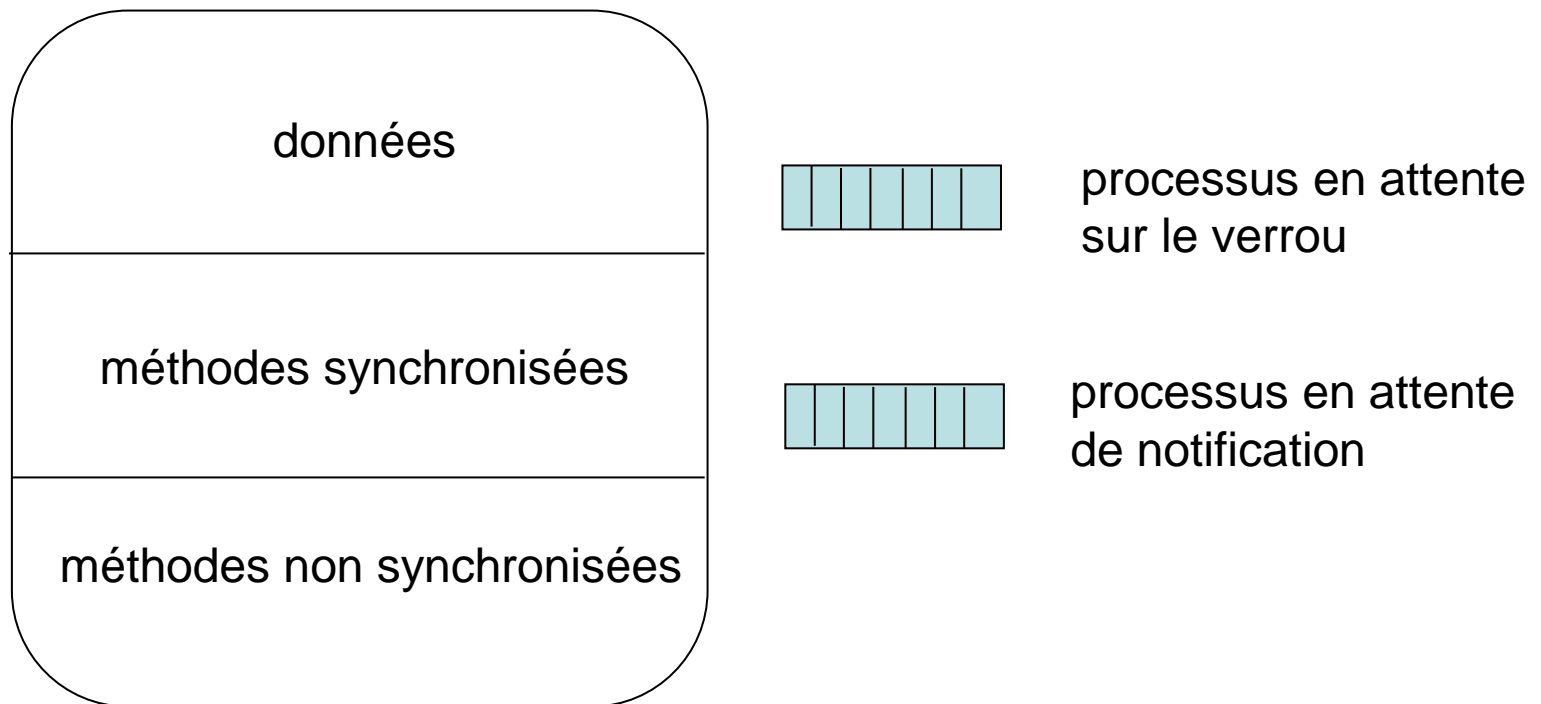
L'implémentation proposée pour le tampon de taille finie utilise une section critique pour accéder le tampon (le tableau buffer).

Si on suppose qu'il n'y a qu'un seul producteur et un seul consommateur cette section critique est superflue.

En effet, la section critique est utile seulement lorsqu'on accède une variable commune. Dans notre cas, on accède une même position du tableau seulement si  $\text{inBuf} = \text{outBuf}$ , ce qui correspond aux situations où le tableau est vide ou plein et des accès simultanés par le producteur et le consommateur sont donc impossibles.

Cette propriété se démontre formellement en utilisant les invariants des sémaphores.

# Représentation d'un moniteur Java



# Implémentation d'un tampon fini avec les moniteurs Java

```
Class BoundedBufferMonitor {
```

```
    final int sizeBuf = 10;
```

```
    double[] buffer = new double[sizeBuf];
```

```
    int inBuf = 0, outBuf = 0, count = 0;
```

```
    public synchronized void deposit(double value) {
```

```
        while (count == sizeBuf) // le tampon est plein
```

```
            myWait(this); // ne fonctionne pas comme cela
```

```
        buffer[inBuf] = value;
```

```
        inBuf = (inBuf + 1) % sizeBuf;
```

```
        count++;
```

```
        if (count == 1) notify(); // libère un éventuel thread en attente
```

```
    }
```

assure l'exclusion mutuelle  
correspond à acquérir le verrou

le thread est placé dans la file  
d'attente

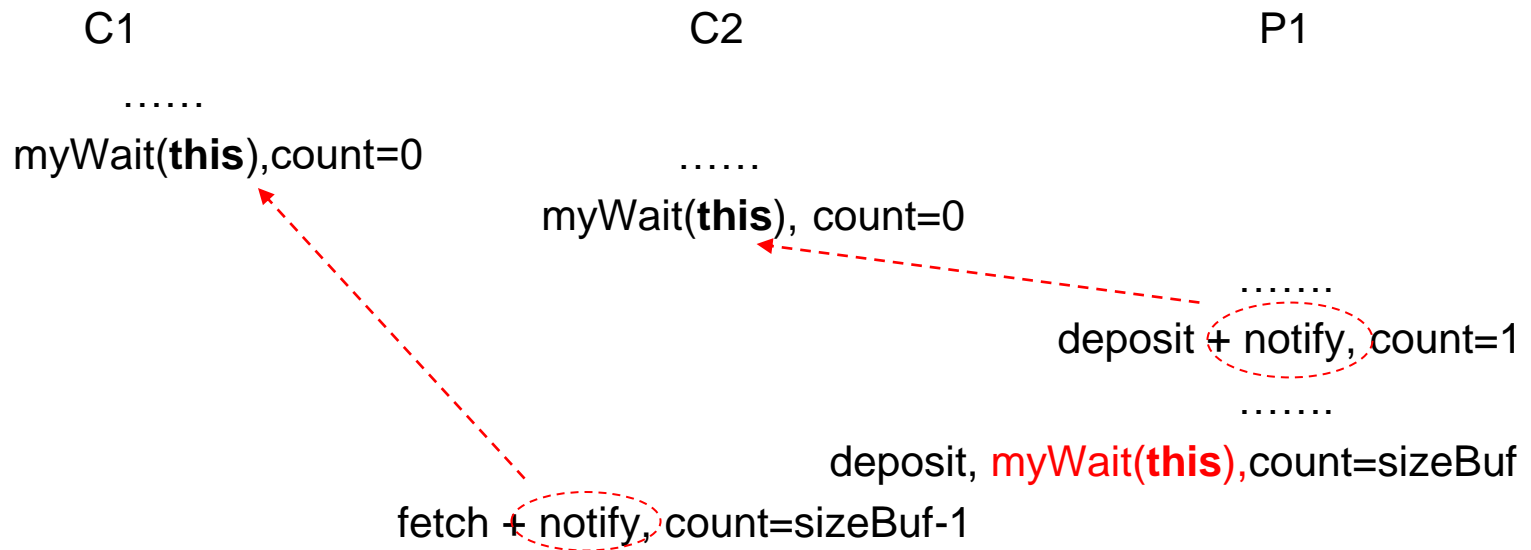
# Implémentation d'un tampon fini avec les moniteurs Java

```
public synchronized double fetch();  
    double value;  
    while (count == 0) // le tampon est vide  
        myWait(this);  
    value = buffer[outBuf];  
    outBuf = (outBuf + 1) % sizeBuf;  
    count--;  
    if (count == sizeBuf - 1)  
        notify(); // libère un éventuel thread en attente  
    return value;  
}  
}
```

# Une exécution particulière

L'implémentation proposée du tampon fini est correcte seulement dans le cas d'un consommateur et un producteur.

En effet, supposons que l'on a deux consommateurs C1 et C2 et un producteur P1 et considérons l'exécution suivante.



seuls C1 et C2 sont exécutables, ils vident le tampon, exécutent `myWait()` et tous les processus sont en attente.



[illegible]

# Solutions

Le problème survient parce que la notification n'est pas sélective. Un processus consommateur est notifié alors que le *notify()* s'adressait au producteur.

Une première solution à ce problème est d'utiliser *notifyAll()* à la place de *notify()*.

Cette solution a le désavantage d'entraîner beaucoup de changement de contextes, certains inutiles.

Une solution plus élégante consiste à implémenter des **variables de conditions** qui permettent à un processus d'attendre qu'une condition particulière soit réalisée (par exemple, `bufferNoFull`, `bufferNotEmpty`).

# Les variables de conditions

Si on considère l'implémentation d'un tampon dans le problème du producteur-consommateur, lorsque le producteur est bloqué, il attend sur la condition `isFull==false` (`bufferNotFull`) alors que le consommateur attend sur la condition `isEmpty==false` (`bufferNotEmpty`).

Dans l'implémentation du tampon en Java, les appels à *notify()* ne sont pas sélectifs et tous les processus deviennent exécutables, qu'ils soient producteurs ou consommateurs.

# multi-producteur multi-consommateur

On définit deux objets pour la synchronisation

```
private Object conveyD = null, conveyF = null;
```

Le tampon est géré comme dans l'exemple précédent avec inBuf et outBuf et un compteur.

On utilise deux compteurs supplémentaires spaces et elements qui permettent de compter le nombre de producteur et consommateur en attente respectivement.

Initialisation: spaces = sizeBuf, elements = 0

# multi-producteur

```
public void deposit(double value) {  
    synchronized(conveyD) {  
        spaces--;  
        if (spaces < 0){ // si négatif, compte le nombre de producteur en attente  
            try {  
                conveyD.wait(); // tampon plein, attente  
            } catch (InterruptedException e) {}  
        }  
        buffer[inBuf++] = value;  
    }  
    synchronized(conveyF) {  
        elements++;  
        if (elements <= 0) conveyF.notify(); // notify consommateur  
    }  
}
```

# multi-consommateur

```
public double fetch() {  
    double value;  
    synchronized(conveyF) {  
        elements--; // si négatif compte le nombre de consommateur en attente  
        if (elements < 0) {  
            try {  
                conveyF.wait();  
            } catch (InterruptedException e) {}  
        }  
        value = buffer[outBuf++];  
    }  
    synchronized(conveyD) {  
        spaces++;  
        if (spaces <= 0) conveyD.notify(); // notifie un producteur  
    }  
}
```

# Le problème des lecteurs-rédacteurs

Ce problème classique est une extension du problème de la section critique. En effet, plusieurs processus désirent accéder une même ressource mais les processus sont divisés en deux classes:

1. Les lecteurs, qui peuvent accéder la ressource de manière concurrente.
2. Les rédacteurs, qui doivent accéder la ressources comme une ressource critique.

En d'autres termes,

les lecteurs doivent exclure l'accès à la ressource aux seuls rédacteurs.

Les rédacteurs doivent exclure l'accès à la ressource aux lecteurs et aux rédacteurs.

# Le problème des lecteurs-rédacteurs

Ce problème est une abstraction des accès à une base de données partagée.

Les utilisateurs qui ne modifient pas la base peuvent l'accéder de manière concurrente.

Les modifications à la base de donnée doivent se faire de manière atomique pour éviter des problèmes de cohérence.

Le protocole de base pour accéder la ressource en lecture est d'appeler les routines *startRead()* et *endRead()*.

De même pour accéder la ressource en écriture il faut appeler les routines *startWrite()* et *endWrite()*.



# Le problème des lecteurs-rédacteurs

Plusieurs lecteurs peuvent accéder la ressource, il faut compter le nombre de lecteurs *numReaders*.

1. Pour savoir quand un rédacteur peut accéder la ressource.
2. Le premier lecteur doit empêcher l'accès à un rédacteur.
3. Le dernier lecteur à quitter la ressource doit le signaler à d'éventuels rédacteurs en attente.

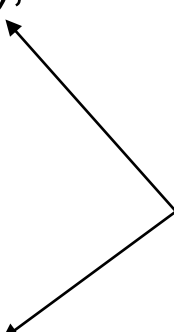
On utilise une variable entière *numReaders* pour compter les lecteurs.

Une sémaphore binaire pour restreindre l'accès à la ressource.

Une sémaphore binaire pour manipuler les données critiques du protocole. C'est-à-dire pour manipuler *numReaders*, cette sémaphore va aussi bloquer les lecteurs lorsque la ressource est accédée par un rédacteur.

# Le problème des lecteurs-rédacteurs

```
class ReaderWriter {  
    int numReaders = 0;  
    BinarySemaphore mutex = new BinarySemaphore(true);  
    BinarySemaphore wlock = new BinarySemaphore(true);  
    public void startRead() {  
        mutex.P(); // accès aux variables critiques du protocole  
        numReaders++;  
        if (numReaders == 1) wlock.P(); // réserve l'accès à la ressource  
        mutex.V();  
    }  
    public void endRead() {  
        mutex.P();  
        numReaders--;  
        if (numReaders == 0) wlock.V(); // libère l'accès à la ressource  
        mutex.V();  
    }  
}
```



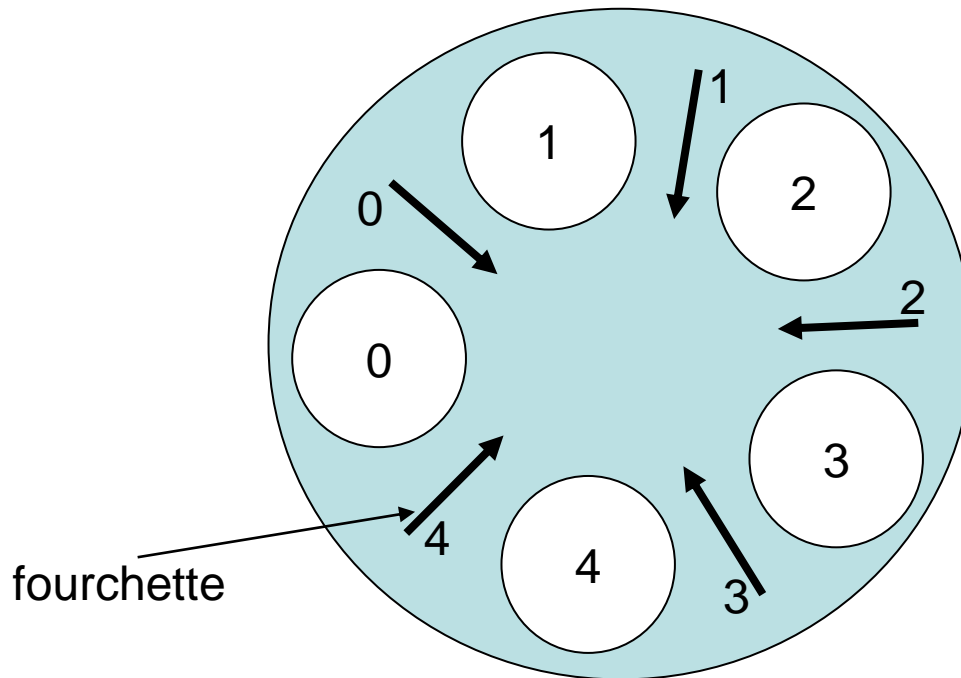
# Le problème des lecteurs-rédacteurs

```
public void startWrite() {  
    wlock.P();  
}
```

```
public void endWrite() {  
    wlock.V();  
}  
}
```

# Le problème des philosophes

Dans ce problème on considère 5 philosophes dont les seules activités sont manger et penser.



# Le problème des philosophes

Avant de commencer à manger, un philosophe doit acquérir deux fourchettes (ressources critiques).

Avant de commencer à penser, un philosophe libère les fourchettes

répéter

penser

pré-protocole // acquérir deux fourchettes

manger

post-protocole // rendre les fourchettes

# Le problème des philosophes

Chaque philosophe est associé à un thread

Les fourchettes sont associées à des ressources et on doit implémenter l'interface

```
interface Ressource {  
    public void acquire(int i);  
    public void release(int i);  
}
```

# Le problème des philosophes

L'implémentation de l'interface Ressource doit satisfaire:

- Un philosophe mange seulement s'il possède deux fourchettes
- Deux philosophes ne possèdent pas la même fourchette simultanément (exclusion mutuelle)
- Pas de d'inter-blocage (freedom from deadlock).
- Pas d'insuffisance de ressource (freedom from starvation)

# Programme de test

```
class Philosopher implements Runnable {  
    int id = 0;  
    Ressource r = null;  
    public Philosopher(int initId, Ressource initr) {  
        id = initId;  
        r = initr;  
    }  
}
```



# Programme de test (suite)

```
public void run() {  
    while(true) {  
        try {  
            System.out.println(" Philosophe " + id + " pense");  
            Thread.sleep(30);  
            System.out.println(" Philosophe " + id + " a faim");  
            r.acquire(id); // acquisition des ressources critiques  
            System.out.println(" Philosophe " + id + " mange");  
            Thread.sleep(40);  
            r.release(id); // libère les ressources  
        } catch (InterruptedException e) { return;}  
    }  
}
```

# 1<sup>ère</sup> tentative

On associe à chaque fourchette (ressource) un sémaphore binaire.  
Le pré-protocole exécuté par le philosophe  $i$  s'écrit:

```
fork[i].P();  
fork[(i + 1) % 5].P();
```

Le post-protocole

```
fork[i].V();  
fork[(i + 1) % 5].V();
```

# 1<sup>ère</sup> tentative

```
class DiningPhilosopher implements Ressource {  
    int n = 0;  
    BinarySemaphore[] fork = null;  
    public DiningPhilosopher(int initN) {  
        n = initN; // nombre de philosophes  
        fork = new BinarySemaphore[n];  
        for (int i = 0; i < n; i++) {  
            fork[i] = new BinarySemaphore(true); // les ressources sont  
                                                    // initialement accessibles  
        }  
    }  
}
```

# 1<sup>ère</sup> tentative (suite)

```
public void acquire(int i) {  
    fork[i].P();  
    fork[(i + 1) % n].P();  
}  
public void release(int i) {  
    fork[i].V();  
    fork[(i + 1) % n].V();  
}  
public static void main(String [] args) {  
    DiningPhilosopher dp = new DiningPhilosopher(5);  
    for(int i = 0; i < 5; i++)  
        new Thread(new Philosopher(i,dp)).start();  
}  
}
```

# Analyse

On montre que deux philosophes ne possèdent jamais la même fourchette

En effet, considérons la fourchette  $i$ .  $\text{fork}[i]$  est un sémaphore binaire et les invariants associés s'écrivent:

$$\text{fork}[i] \geq 0$$

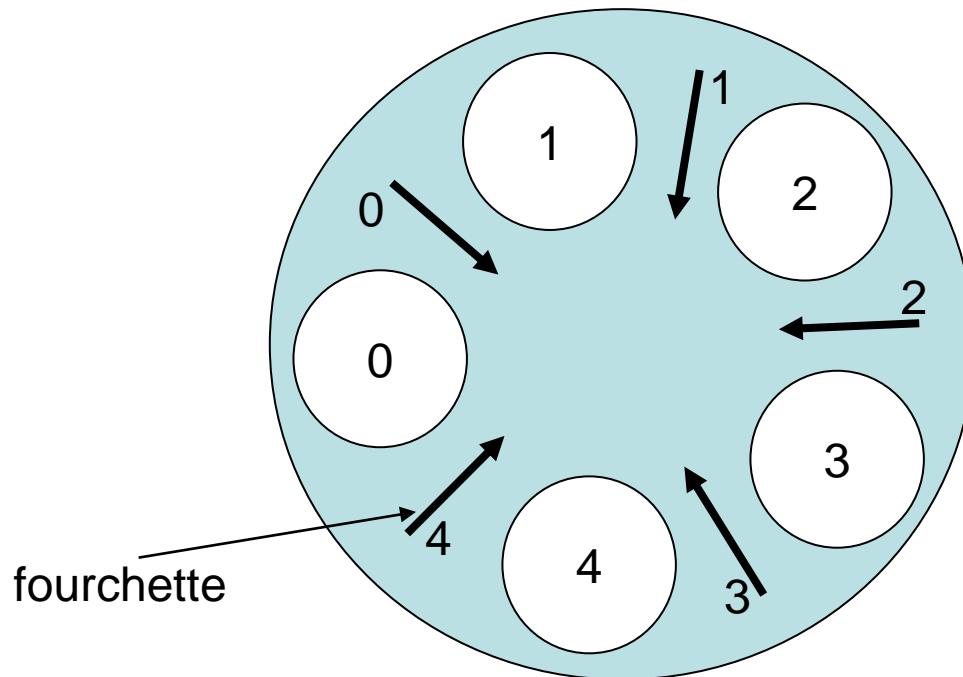
$$\text{fork}[i] = 1 - \# \text{fork}[i].P() + \# \text{fork}[i].V()$$

$$\underbrace{\# \text{fork}[i].P() - \# \text{fork}[i].V()} \leq 1$$

le nombre de philosophes qui disposent de la fourchette

# Analyse 1<sup>ère</sup> tentative (suite)

Malheureusement, il existe une exécution qui conduit à un inter-blocage: Tous les philosophes exécutent `fork[i].P()` séquentiellement.



## 2<sup>ème</sup> tentative

Une solution consiste à empêcher que tous les philosophes puissent exécuter le pré-protocole simultanément. On utilise une sémaphore entière *room* pour limiter le nombre de philosophe à  $n - 1$ .

Le pré-protocole exécuté par le philosophe  $i$  s'écrit:

```
room.P(); CountingSemaphore room = new CountingSemaphore(n-1);  
fork[i].P();  
fork[(i + 1) % 5].P();
```

Le post-protocole

```
fork[i].V();  
fork[(i + 1) % 5].V();  
room.V();
```

# Analyse de la deuxième tentative

La preuve de l'exclusion mutuelle est la même que pour la 1<sup>ère</sup> tentative.

On montre que la deuxième tentative **ne conduit jamais à une absence de ressources (free from starvation)**. Pour cela, on doit supposer que *les processus bloqués dans la file d'attente associée à la sémaphore room sont libérés dans l'ordre d'arrivée*.

Pour les autres sémaphores *fork[i]* on a pas besoin d'une telle hypothèse.



# Analyse de la deuxième tentative

**1<sup>er</sup> cas:** Le philosophe  $i$  est bloqué après l'appel à  $fork[i].P()$ .

Le philosophe  $i-1 \pmod n$  a donc exécuté avec succès  $fork[i].P()$ ,  
comme l'ordre du programme est tel que le philosophe  $i-1$  exécute  
 $fork[i-1].P()$  avant d'exécuter  $fork[i].P()$ ,

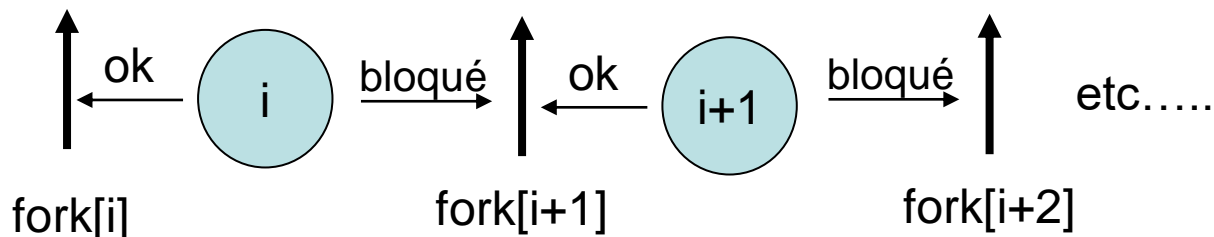
$$fork[i-1].P() \longrightarrow fork[i].P()$$

le processus associé au philosophe  $i-1$  mange... Lorsqu'il aura fini il  
libérera le processus  $i$  qui pourra exécuter  $fork[i+1].P()$ .

# Analyse de la deuxième tentative

**2<sup>ème</sup> cas:** Le processus associé au philosophe  $i$  est bloqué après l'appel à  $fork[i+1].P()$ .

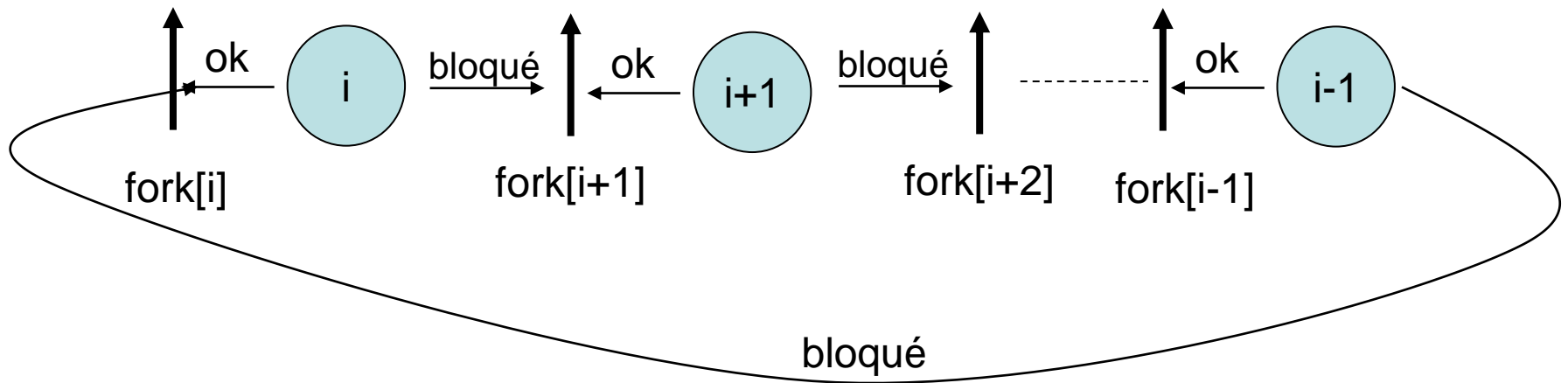
On suppose que le processus  $i+1$  a exécuté  $fork[i].P()$  avec succès et qu'il ne libère jamais cette sémaphore, il y a insuffisance de ressources pour le processus  $i$ . Alors ce processus doit être bloqué sur  $fork[i+2].P()$  sinon il mange et exécutera  $fork[i+1].V()$  libérant le processus  $i$ .



# Analyse de la deuxième tentative

## 2<sup>ème</sup> cas (suite):

Par induction on montre que tous les processus  $j$  doivent être bloqués sur  $fork[j+1].P()$ . C'est une contradiction car dans cette situation tous les processus ont exécutés  $room.P()$  ce qui est impossible



# Analyse de la deuxième tentative

**3<sup>ème</sup> cas:** Le processus associé au philosophe  $i$  est bloqué après l'appel à  $room.P()$ .

On a supposé que cette sémaphore libère les processus selon une stratégie FIFO.

Il doit se trouver  $n-1$  processus dans la section de programme entre  $room.P()$  et  $room.V()$ . L'étude des deux cas précédents montre que ces processus ne peuvent pas rester bloqués dans cette section de code. Alors au moins un processus va exécuter  $room.V()$ , ce qui libérera le processus  $i$ .

# symétrie-asymétrie

Un algorithme distribué est symétrique si au début de l'exécution tous les processus sont dans le même état (variables internes) et que toutes les variables partagées ont la même valeur.

Si on suppose que le système ne dispose pas de mémoire partagée commune ou ne peuvent pas communiquer avec un processus central alors on peut montrer **qu'il n'existe pas de solution au problème des philosophes qui soit symétrique et déterministe.**

Dans la deuxième tentative, la sémaphore *room* est partagée de manière centralisée entre les processus.

# Asymétrie

Une solution asymétrique au problème de philosophes est la suivante.  
Tous les philosophes exécutent l'algorithme précédent sauf le processus  $n-1$  qui exécute

```
public void acquire() {
```

```
    fork[0].P();
```

```
    fork[n-1].P();
```

```
}
```

```
public void release() {
```

```
    fork[0].V();
```

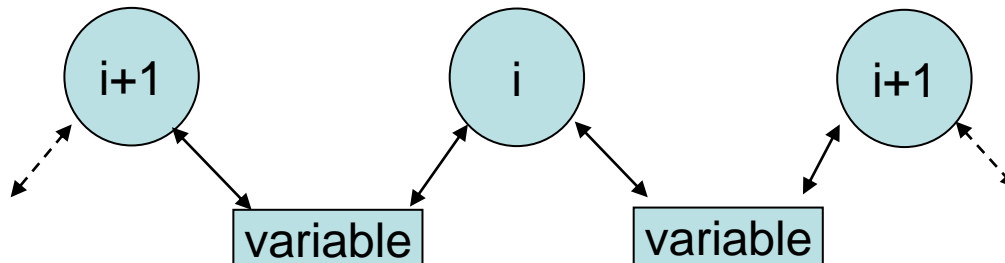
```
    fork[n-1].V();
```



# Impossibilité pour le problème des philosophes

On s'intéresse à l'impossibilité de résoudre le problème des philosophes de manière complètement distribuée (sans processus ou variables communes à tous les philosophes) et symétrique.

Comme le système est complètement distribué les philosophes communiquent uniquement avec leurs voisins, par exemple en partageant une variable commune. Cette variable permet aux philosophes de se mettre d'accord sur lequel va disposer de la fourchette commune.



# Impossibilité pour le problème des philosophes

Les philosophes disposent aussi de variables internes.

L'algorithme appliqué par chacun des philosophe est déterministe.

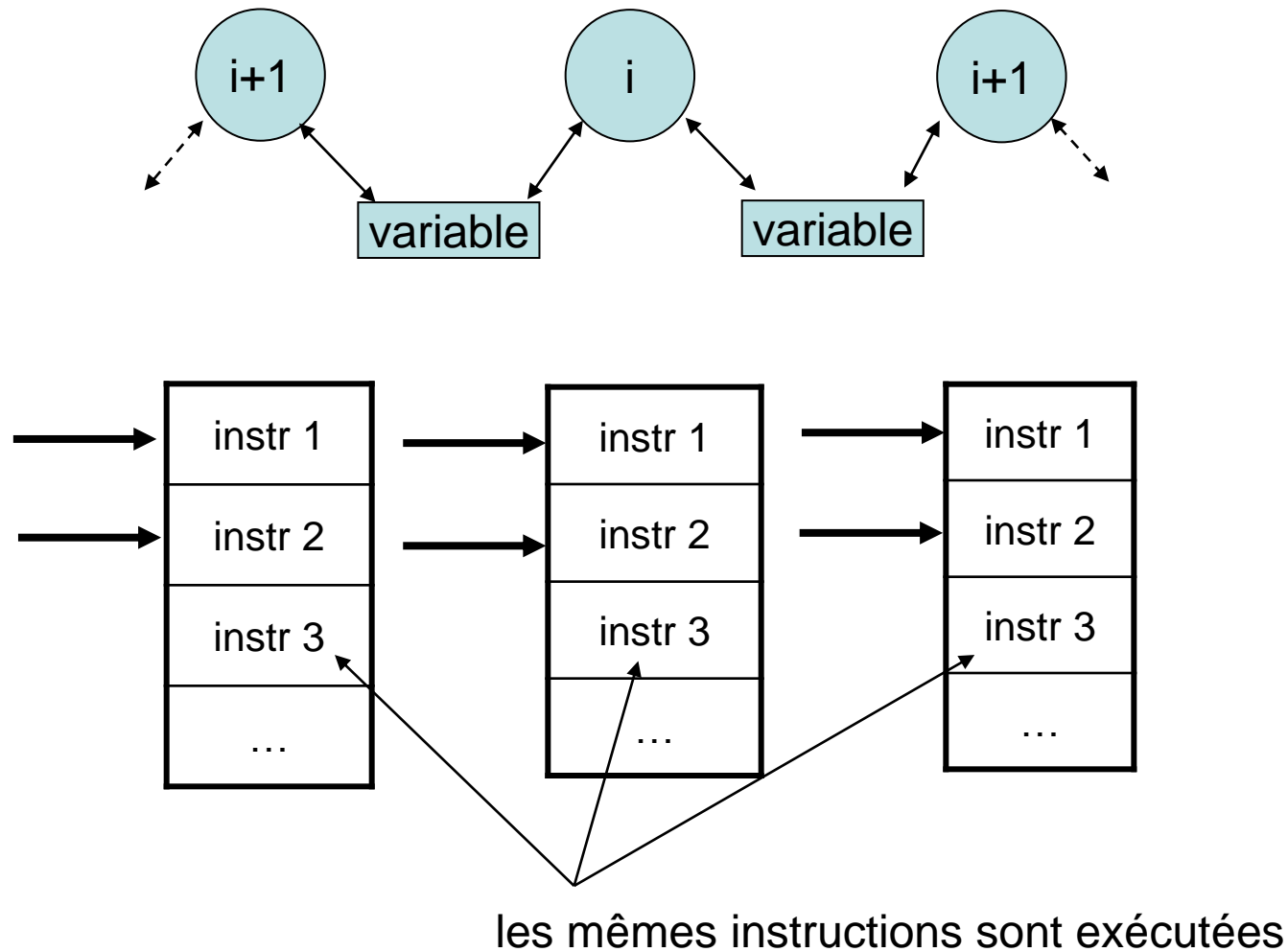
La solution au problème est symétrique, c'est-à-dire que toutes les variables (internes et partagées) sont initialisées de manière identique et les programmes exécutés par les philosophes sont les mêmes.

**1<sup>ère</sup> étape:** On montre qu'il existe une séquence d'activation des processus qui fait que les processus sont toujours symétriques.

Supposons que les philosophes soient activés dans l'ordre 1, 2, ..., N et que chaque fois qu'un philosophe est activé il exécute une unique instruction atomique



# Impossibilité pour le problème des philosophes



# Impossibilité pour le problème des philosophes

On a trois cas à traiter

1. l'instruction accède une variable interne
2. l'instruction accède la variable partagée avec le philosophe de droite
3. l'instruction accède la variable partagée avec le philosophe de gauche

On suppose qu'avant l'activation des processus 1, 2, ..., N l'état des processus est symétrique et on montre qu'après que tous les processus ont été activés ils sont dans un état symétrique.

Dans le premier cas, l'opération étant interne aux processus, les processus étant tous dans le même état avant l'exécution de l'instruction, les processus **restent tous dans le même état après l'exécution.**

# Impossibilité pour le problème des philosophes

Dans le deuxième cas, les philosophes accèdent la variable en lecture ou en écriture. Les variable partagées ont toutes la même valeur avant l'exécution par hypothèse.

Donc si c'est une lecture, après exécution de l'instruction les philosophes ont **tous lu la même valeur**.

Si c'est une écriture, les philosophes écrivent la valeur d'une variable interne dans la variable partagée, comme la valeur de cette variable interne est la même pour tous les processus, **la valeur écrite dans la variable partagée est la même pour tous** les philosophes.

Le troisième cas se traite de la même manière.

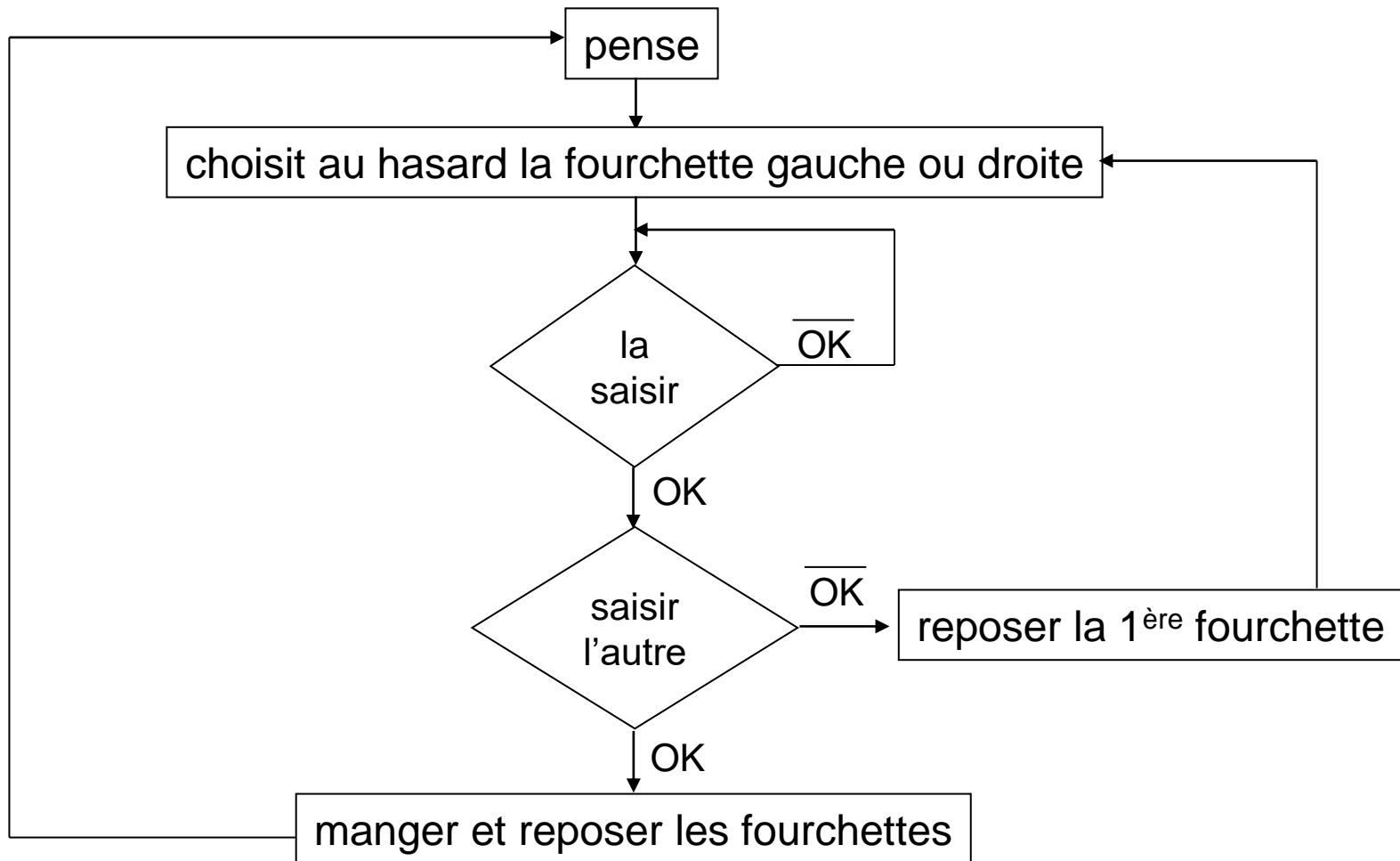
# Impossibilité pour le problème des philosophes

Supposons qu'il existe une exécution d'une instruction après laquelle un philosophe possède ses deux fourchettes. Par symétrie, après l'exécution de la même instruction par le prochain philosophe, ce dernier est en possession de ses deux fourchettes. On poursuit le raisonnement et on montre qu'à la fin de l'activation de tous les philosophes, ils sont tous en possession de deux fourchettes ce qui est impossible.

**On a donc montré qu'il n'existe pas d'algorithme symétrique, complètement distribué et déterministe pour résoudre le problème des philosophes.**

**Il existe des algorithmes probabilistes.**

# Algorithme probabiliste



# Lemme 1

## **Lemme 1:**

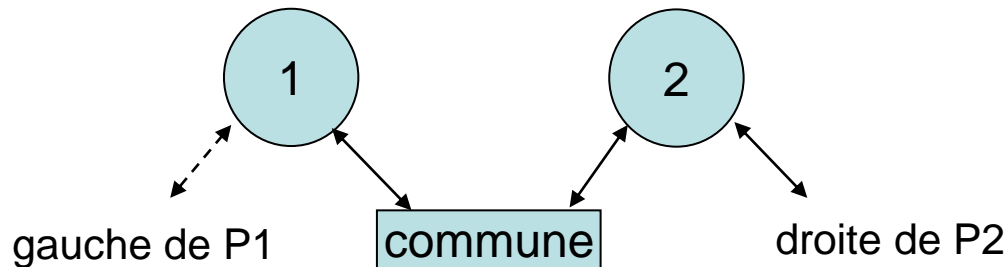
Les penseurs ne peuvent pas être interbloqués. Ils choisissent donc un nombre infini de fois une fourchette (et donc saisissent un nombre infini de fois chacune des fourchettes)

La condition Hold and Wait de la p. 15 du cours n'est pas satisfaite. Les penseurs ne peuvent donc pas être interbloqués. Ils doivent procéder au tirage aléatoire d'une fourchette (et saisir la fourchette correspondante) une infinité de fois.

# Lemme 2

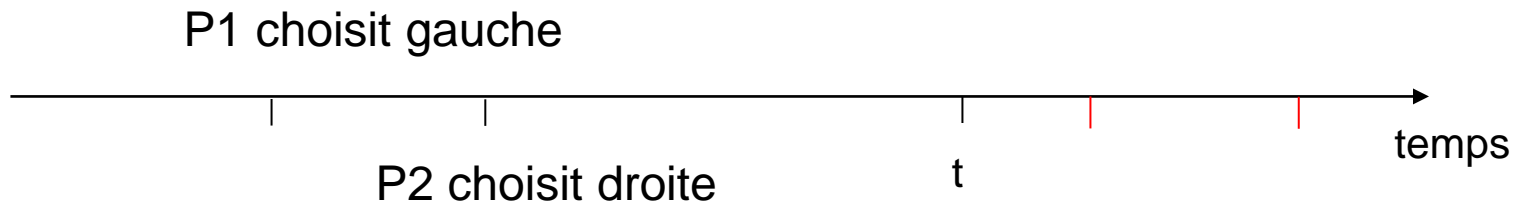
## Lemme 2:

Dans la situation ci-dessous ou P1 saisit la fourchette gauche et P2 la fourchette droite alors P1 ou P2 va manger avec une probabilité positive.



# Lemme 2

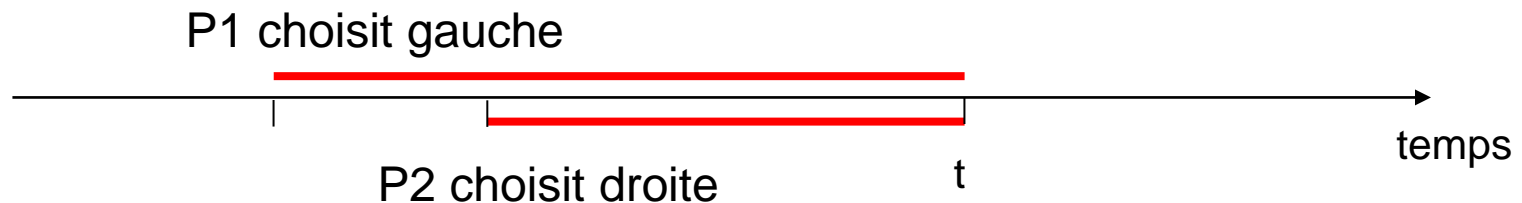
Par symétrie on peut supposer que la configuration ci-dessous se produit.





# Lemme 2

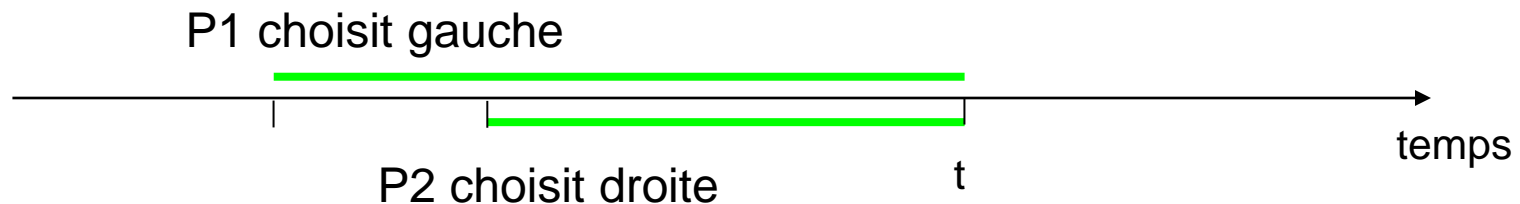
Cas 1: Ni P1 ni P2 essayent de saisir la fourchette commune après le tirage aléatoire et avant  $t$ .



Après  $t$  le premier de P1 ou P2 qui essaye de saisir la fourchette commune va l'obtenir.

# Lemme 2

**Cas 2:** P1 et P2 ont essayé de saisir la fourchette commune après leur dernier tirage aléatoire et avant  $t$ .

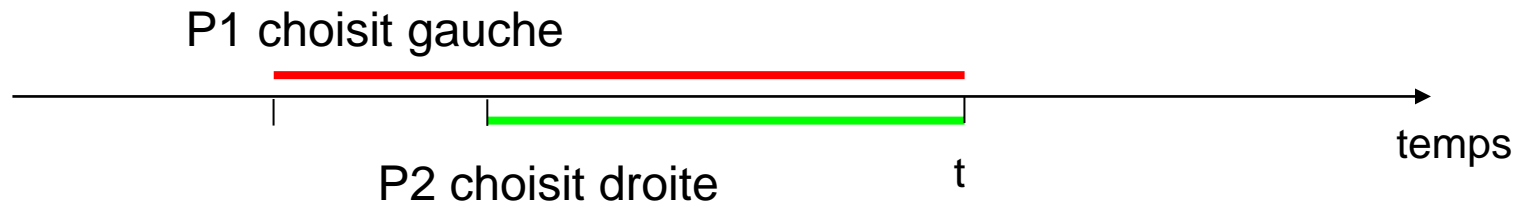


Si le premier à essayer de saisir la fourchette commune la trouve sur la table, il la saisit et commence à manger.

Sinon, il (P1) repose les fourchettes et effectue un nouveau tirage aléatoire mais après  $t$ . Donc l'autre (P2) philosophe va trouver la fourchette commune libre et va commencer à manger.

# Lemme 2

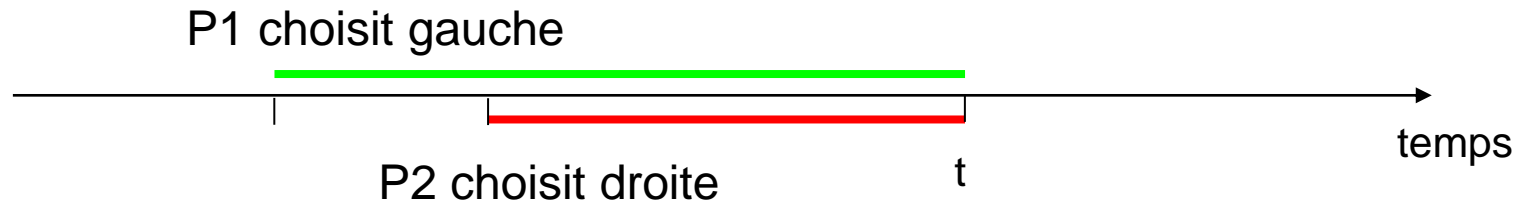
**Cas 3:** P1 ou P2 (exclusif) essaye de saisir la fourchette commune



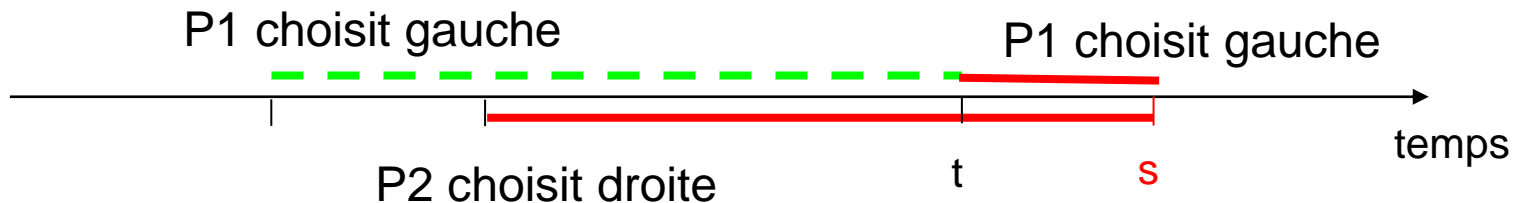
P2 trouve la fourchette commune libre et mange

# Lemme 2

## Cas 4:

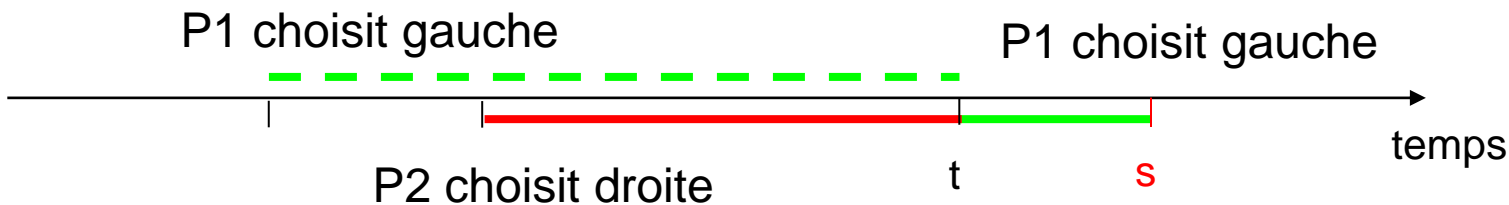


P1 ne trouve pas la fourchette commune sur la table et son prochain tirage aléatoire est à un temps  $s > t$  et *avant que P2 essaye de saisir la fourchette commune*. Avec probabilité  $\frac{1}{2}$  P1 va à nouveau sélectionner la fourchette gauche. On est dans la situation qui correspond au **cas 1**.



# Lemme 2

**Cas 5:** P1 ne trouve pas la fourchette commune sur la table et son prochain tirage aléatoire est à un temps  $s > t$  et *après que P2 essaye de saisir la fourchette commune*.



Dans cette situation, c'est P2 qui commence à manger.

D'après les différents cas considérés, soit P1 soit P2 commence à manger avec un probabilité positive.

# Configuration

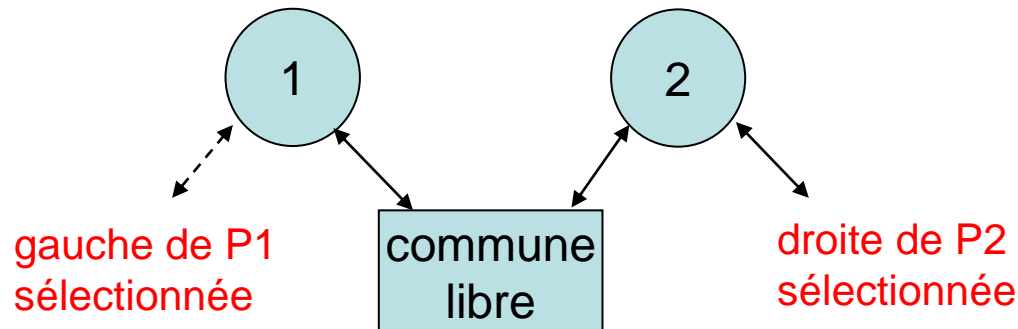
A chaque instant correspond une **configuration** qui consiste en les valeurs des derniers choix effectués par les philosophes.

Des configurations, à des temps différents A et B sont **disjointes** si chaque philosophes à effectué un tirage aléatoire entre A et B.

# Lemme 3

## Lemme 3:

Si au temps  $t$  la configuration est  $A$  alors il arrive avec probabilité 1 une configuration  $B$ , disjointes de  $A$ , et telle qu'il P1 ait choisit sa fourchette gauche (dans  $B$ ) et P2 sa fourchette droite.



# Lemme 3

Depuis la configuration A, chaque philosophe va effectuer un nouveau tirage aléatoire. En effet, il se saisit une infinité de fois d'une fourchette et donc, il effectue une infinité de tirage aléatoire. Il existe donc une configuration G disjointe de A.

Si G satisfait l'énoncé du lemme on a fini.

Sinon, dans G tous les philosophes ont choisis gauche ou tous les philosophes ont choisis droite. Tous les philosophes vont refaire des tirages aléatoires par hypothèse et la probabilité qu'ils sélectionnent toujours soit gauche soit droite est nulle. Avec probabilité 1, on a donc existence d'une configuration disjointes de A qui satisfait l'énoncé du lemme.



# Le théorème

## **Théorème:**

Si l'ordonnanceur assure que toutes les configurations se produisent avec égale probabilités alors les philosophes finissent toujours par manger.

# Insuffisance de ressources

L'algorithme n'assure pas que tous les philosophes mangent à un moment donné (starvation).

On considère deux philosophes P1, P2 adjacents et l'exécution suivante:

1. P1 saisit deux fourchettes et mange
2. P2 test la fourchette commune qui n'est pas disponible
3. P1 restitue les deux fourchettes
4. P1 saisit les deux fourchettes et mange
5. P2 test la fourchette commune qui n'est pas disponible
6. etc.

Dans le cas général on montre qu'il existe un ordonnancement des processus tel que  $n-1$  philosophes ne mangent jamais.

Dans ce cas l'ordonnanceur est supposé un adversaire qui ne laisse pas les configurations se produire au hasard.

# Java en pratique

# Possibilité d'interblocage

On considère l'implémentation suivante d'un objet qui implémente la méthode *swap*

```
class BCell {  
    int value;  
    public synchronized int getValue();  
        return value;  
}  
    public synchronized void setValue(int i) {  
        value = i;  
    }  
    public synchronized void swap(BCell x) {  
        int temp = getValue();  
        setValue(x.getValue());  
        x.setValue(temp);  
    }
```

# Possibilité d'interblocage

BCell p,q;

Processus 1

p.swap(q);

Processus 2

q.swap(p);

et l'exécution:

P1.lock(p) // processus 1 obtient le verrou sur p

P2.lock(q) // processus 2 obtient le verrou sur q

q.getValue // processus 1 doit obtenir le verrou, il attend


p.getValue // processus 2 doit obtenir le verrou, il attend

La solution consiste à demander les verrous dans le même ordre

# Swap sans interblocage

```
class Cell {  
    int value;  
    public synchronized int getValue() {  
        return value;  
    }  
    protected synchronized void doSwap(Cell x) {  
        int temp = getValue();  
        setValue(x.getValue());  
        x.setValue(temp);  
    }  
    public void swap(Cell x) {  
        if (this == x) return;  
        else if (system.identityHashCode(this) < System.identityHashCode(x))  
            doSwap(x);  
        else  
            x.doSwap(this);  
    }  
}
```

accessible seulement depuis une méthode de la classe



# Data race

Dans un programme concurrent le principal problème est dû aux problèmes de course sur les données qui peut avoir lieu si

*Une variable est accédée par plusieurs threads et au moins l'un deux écrit (modifie le contenu de la variable).*

Pour résoudre ces problèmes, on peut

1. Ne pas partager des variables entre plusieurs threads
2. Ne pas permettre de les modifier
3. Synchroniser les accès à la variable

# Data race

Les problèmes de course surviennent lorsque l'exactitude du résultat de l'exécution des threads dépend de l'ordonnancement (timing) des instructions atomiques.

Exemple: On veut créer un objet uniquement au moment où on doit l'utiliser, l'objet doit être créé une seule fois.

```
Public class LazyInitRace{  
    private ExpensiveObject instance = null;  
    public ExpensiveObject getInstance() {  
        if (instance==null)  
            instance = new ExpensiveObject();  
        return instance;  
    }  
}
```



# Data Race

Dans cet exemple, l'objet peut-être créer plusieurs fois selon l'ordonnancement des instructions. Une solution consiste a synchroniser la méthode getInstance()

```
public synchronized ExpensiveObject getInstance() { ...}
```

# Data Race

C'est le même problème qui survient lorsqu'on exécute *compteur++*;

Pour les manipulations atomiques des données, Java propose des classes adhoc.

AtomicLong

AtomicInteger

AtomicBoolean

....

(voir `java.util.concurrent.atomic`)

# AtomicLong

Par exemple pour implémenter un compteur partagés par plusieurs threads.

```
Public class CountingFactorizer implements Servlet {  
    private final AtomicLong count = new AtomicLong(0);  
  
    public long getCount() {return count.get();}  
  
    public void service(ServletRequest req, ServletResponse resp) {  
        ...  
        count.incrementAndGet(); // ici count++; ne marche pas, service est appelée  
        ...                       // par plusieurs threads  
        ...  
    }  
}
```

# AtomicLong

Une autre solution consiste à synchroniser la méthode service

```
Public synchronized void service(ServletRequest req, ServletResponse resp) {...}
```

D'un point de vue des performances cette solution n'est pas acceptable.

D'une manière générale, on utilise un verrou (lock) chaque fois qu'une opération composée (count++) doit être atomique. Le verrou doit

1. être le même partout où la variable est accédée
2. chaque accès à la variable doit être protégé avec le verrou

# Consistance

Si un ensemble de variables prend des valeurs qui doivent être consistantes alors tous les accès aux variables doivent être protégés par le même verrou.

Les lectures et écritures de variables de type standard sont exécutés atomiquement **sauf** pour les variables *double* ou *long* qui ne sont pas *volatile*. Ces variables sur 64 bits peuvent être accédées comme deux variables de 32 bits. S'il y a plusieurs threads, une lecture peut retourner une valeur qui n'a pas été écrite par un autre thread (*out-of-thin-air*).

Les variables long et double partagées doivent être volatile ou protégées par un verrou.

# Visibilité

Les mécanismes de synchronisation permettent d'exécuter plusieurs instructions de manière atomique. Ce n'est pas la seule fonction de ces mécanismes. La synchronisation permet d'assurer que des modifications (écritures) réalisées sur des variables sont visibles par d'autres threads.

Sans synchronisation, il n'y a pas de garantie qu'une écriture dans une variable soit visible pour un autre processus (écriture en mémoire cache ou dans un registre).

En plus, la synchronisation limite le ré-ordonnancement des instructions.

# Visibilité

```
Public class NoVisibility {  
    private static boolean ready;  
    public static int number;  
    public static class ReaderThread extends Thread {  
        public void run() {  
            while(!ready)  
                Thread.yield();  
            system.out.println(number);  
        }  
    }  
  
    public static void main(String[] args) {  
        new ReaderThread().start();  
        number = 42;  
        ready = true;  
    }  
}
```

# Visibilité

Avec ce programme on peut observer que

1. Le thread `ReaderThread` ne progresse pas, la condition d'attente dans la boucle `while` est toujours valide. Cela peut se produire parce que rien n'oblige l'écriture de la variable `ready` par le thread `main` à être visible.
2. Le thread `ReaderThread` affiche 0 à l'écran. Cela peut se produire si le compilateur ré-ordonne les instructions du thread `main` et l'écriture de `ready` est visible.

Ces comportements sont possibles car le compilateur peut ré-ordonner les instructions pour autant que la sémantique du code pour une exécution séquentielle ne soit pas modifiée et aussi utiliser les registres du processeur comme mémoire cache.



# Visibilité

```
Public class MutableInteger {  
    private int value;  
  
    public int get() {return value;}  
    public void set(int value) {this.value = value; }  
}
```

Un thread qui appelle get() peut ne pas voir la valeur mise à jour par un précédent appel à set(...). Par contre, c'est garanti si on synchronise les routines

```
public synchronized int get() {return value;}  
public synchronized void set(int value) {this.value = value; }
```

# Visibilité - Synchronisation

Synchroniser les routines assure qu'il y a une relation d'ordre entre les appels aux routines, c'est-à-dire l'un apparait après/avant l'autre.

C'est vrai pour les données accédées dans le bloc synchronisé, mais aussi pour toutes les autres variables accédées précédemment dans le code du programme.

# Visibilité - synchronisation

Thread A

y = 1



lock M



x = 1



unlock M



Thread B



lock M



i = x



unlock M



j = y



Toutes les actions avant  
le unlock M sont visibles

# Visibilité - Synchronisation

Les verrous ne permettent pas seulement d'assurer l'exclusion mutuelle lors de l'accès aux variables mais aussi que les modifications des variables soient visibles.

ATTENTION: pour que ca soit vrai, il faut toujours utiliser le même verrou pour protéger l'accès aux variables.

# Visibilité - Volatile

Les variables de type **volatile** assure aussi que les mises-à-jours sont visibles par les autres threads.

Les compilateurs ne sont pas autorisés à ré-ordonner des opérations sur des variables volatiles avec d'autres opérations.

Une lecture d'une variable volatile retourne toujours la valeur correspondant à la dernière mise-à-jour.

Lire une variable volatile assure aussi que toutes les mises-à-jours effectuées par le thread qui a écrit la variable volatile sont visibles. Une écriture à le même effet que libérer un verrou (unlock M) et une lecture le même effet que d'acquérir un verrou (lock M), sans bloquer les processus.

# Visibilité - Volatile

Thread A

$y = 1$



$x = 1$



$M = \dots$



Toutes les actions avant  
l'assignation à  $M$   
sont visibles



lecture de  $M$



$i = x$



$j = y$



# Volatile - exemple

Un exemple typique d'utilisation d'une variable volatile est quand un thread attend qu'un autre thread ait terminé une action en testant une variable de condition.

```
volatile boolean asleep;
```

```
....
```

```
while (!asleep)  
    do something
```

ATTENTION: une variable volatile n'assure pas l'atomicité des accès. Dans l'exemple où on utilise `count++` même si `count` est volatile, ça ne marche pas.

# Volatile

Les situations où une variable volatile est utile sont

1. L'écriture de la variable ne dépend pas de sa valeur actuelle
2. La variable ne doit pas satisfaire des conditions de cohérence avec d'autres variables
3. On n'a pas besoin d'utiliser un verrou pour accéder à la variable.



# Publication

Publier un objet consiste à le rendre accessible depuis des parties du programme qui ne font pas partie de son domaine de visibilité. On peut le faire

1. En utilisant une référence sur l'objet, référence qui possède un plus grand domaine de visibilité que l'objet
2. En utilisant une méthode non privée qui retourne une référence sur l'objet
3. En le passant en argument à une méthode d'une autre classe

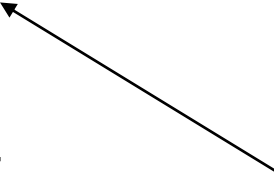
# Publication

Par exemple,

```
public static Set<Secret> knownSecrets;
```

```
Public void initialize() {  
    knownSecrets = new HashSet<Secret>();  
}
```

Peut-être accédé par  
un autre thread alors que  
le constructeur n'a pas terminé  
son exécution



Ou encore

```
class UnsafeStates {  
    private String[] = new String[] { 'A', 'B', ...};  
  
    public String[] getStates() {return states;}  
}
```

# Publication

Un objet se trouve dans un état prévisible et cohérent seulement après que son constructeur ait fini complètement de s'exécuter.

Publier un objet depuis son constructeur est dangereux. Par exemple, en démarrant (`start()`) un thread dans un constructeur et en publiant le pointeur `this` sur l'objet qui se construit.

La publication peut-être explicite, en passant le pointeur `this` en argument ou implicite si l'objet `Runnable` ou le thread exécuté est une sous-classe de l'objet.

# Spécifications

Depuis la version 1.5 de java les comportements des programmes concurrents sont formellement spécifiés.

Les actions inter-threads non synchronisées reconnues par le modèle sont:

- lecture d'une variable partagée (non volatile)
- écriture d'une variable partagée (non volatile)

Les actions inter-threads synchronisées sont:

- lecture d'une variable volatile
- écriture d'une variable volatile
- acquisition d'un verrou (lock)
- libération d'un verrou (unlock)
- la première et la dernière actions d'un thread
- une action qui démarre un thread ou qui détecte la fin d'un thread (`t.isAlive()` ou `t.join()`)

# Synch-order

Toutes les actions de synchronisation sont ordonnées selon un ordre total, c'est-à-dire que les machines virtuelles Java doivent assurer qu'une action est toujours exécutée avant ou après une autre. L'ordre résultant s'appelle synch-order.

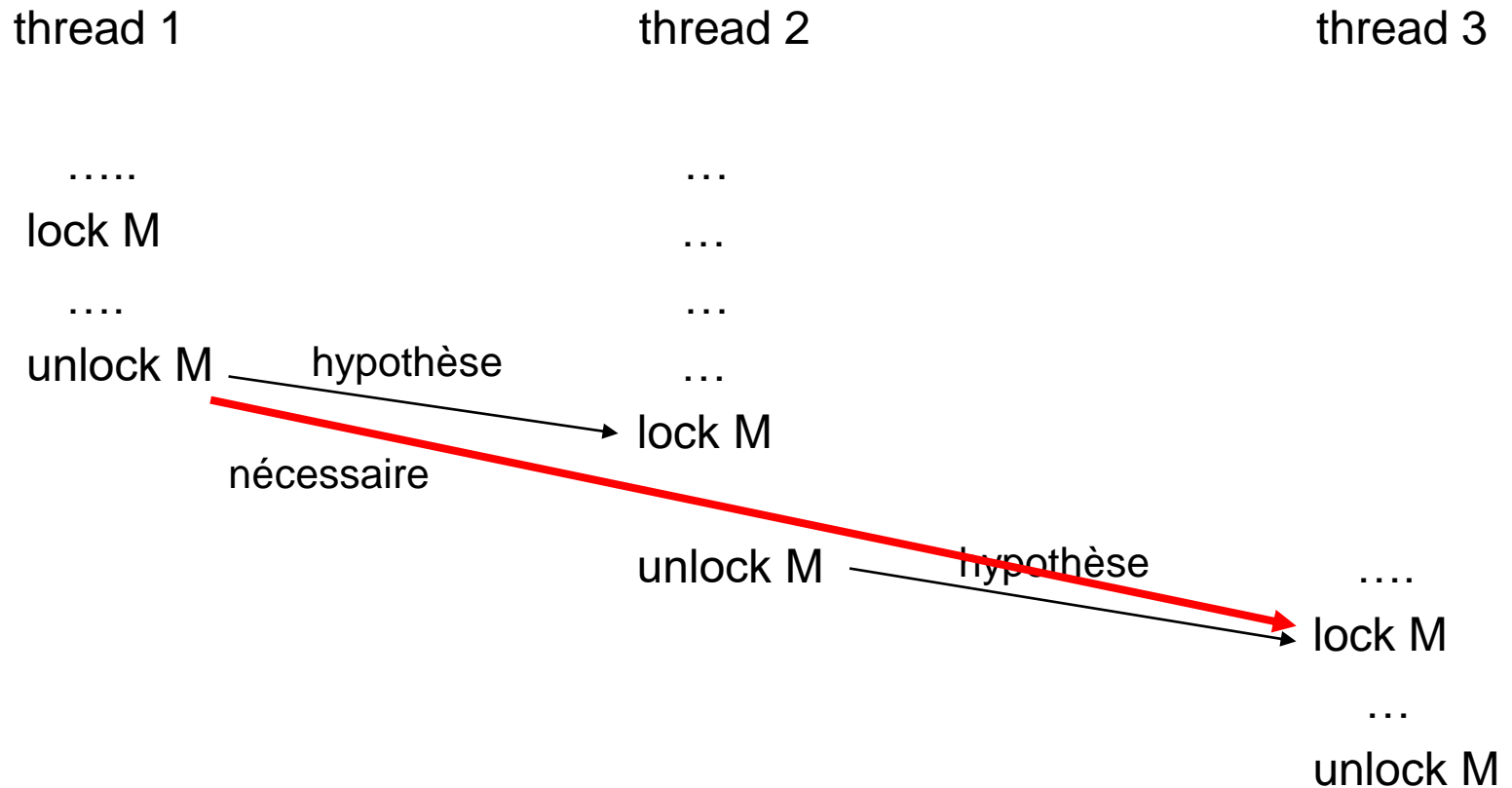
- un `unlock()` sur un verrou est ordonné par rapport aux `lock()` sur le même verrou.
- une écriture d'une variable volatile est ordonnée par rapport à toutes les lectures de la même variable par d'autres threads.
- une action qui démarre un thread apparaît avant la première instruction exécutée par le thread.
- l'écriture des valeurs par défaut (0, false ou null) pour chaque variable apparaît avant la première action de tous les threads

# Synch-order

- la dernière action d'un thread est ordonnée par rapport à toutes les actions d'une autre thread qui détecte que l'exécution du thread est terminée (en utilisant `isAlive` ou `join`)
- si un thread `t1` exécute `t2.interrupt()`, l'interruption est ordonnée par rapport à toutes les actions ou un thread (`t2` ou un autre) détecte l'interruption (en utilisant `isInterrupted`, `interrupted` ou parce qu'une exception `InterruptedException` est levée).

Lorsqu'on raisonne sur les entrelacements possibles, les seuls entrelacements valides sont ceux pour lesquels le synch-order est **total** (ou la relation synch-order peut-être étendue à une relation d'ordre totale)

# Synch-order



# Happen-before

On décrit une relation d'ordre qui s'appelle happen-before. Si une action **a** apparaît avant une action **b** par rapport à cette relation d'ordre, alors **a** est visible pour **b**. On note  $hb(a,b)$  pour indiquer que **a** se produit avant **b**.

- si **a** et **b** sont des actions d'un même thread et que **a** apparaît avant **b** dans l'ordre du programme alors  $hb(a,b)$ .
- si **a** apparaît avant **b** par rapport à l'ordre synchron-order alors  $hb(a,b)$ .
- si  $hb(a,b)$  et  $hb(b,c)$  alors  $hb(a,c)$ .

ATTENTION: la relation  $hb$  sur les actions d'un même thread n'entraîne pas que les actions sont réellement exécutées dans cet ordre, si la sémantique intra-thread n'est pas modifiée le compilateur peut les ordonner différemment. Par contre, les actions sont visibles par les autres threads dans l'ordre.



# Programmes correctement synchronisés

Deux accès à une variable partagée sont conflictuels si au moins un des accès est une écriture .

On a un problème de course sur les données (data-race) si des accès conflictuels ne sont pas ordonnés par la relation happen-before.

Un programme est bien synchronisé si pour tous les entrelacements valides il n'y a pas de problème de course sur les données

Synchronisation  
non  
bloquante  
  
(wait-free)

# Généralités

**But:** développer des mécanismes de synchronisation non bloquants.  
On espère

- Améliorer les performances
- Etre plus tolérant aux pannes en évitant la situation où un processus cesse de s'exécuter tout en possédant un ou plusieurs verrous.
- Eviter les problèmes d'interblocage

# Types de registres

On définit trois types de registres

- sûr (safe)
- régulier (regular)
- atomique (atomic)

Les différents types de registres sont caractérisés par leur comportement lorsque plusieurs processus les écrivent/lisent simultanément.

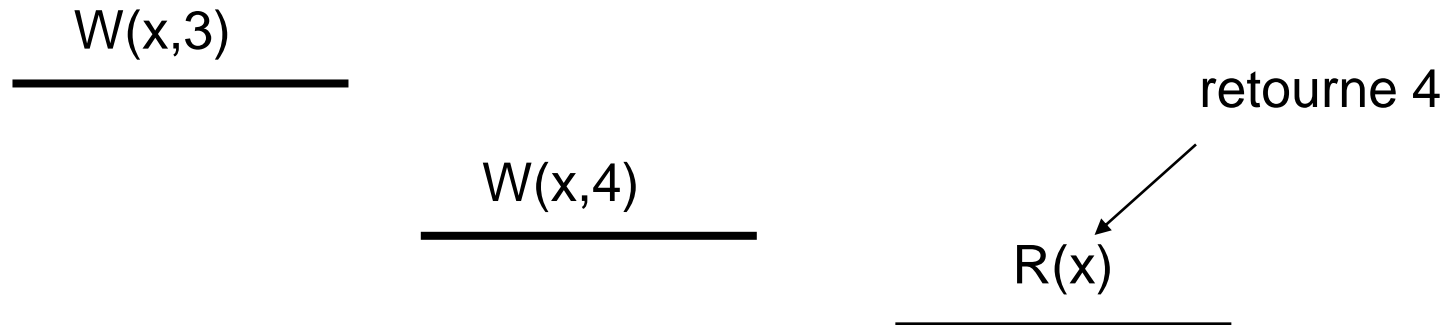
Chacun de ces registres peut être accédés par un seul rédacteur, par plusieurs rédacteurs, par un seul lecteur, par plusieurs lecteurs. Les différents types d'accès permis par les registres sont notés par SW, MW, SR, MR, SWMR,...

# Registres sûrs

Un registre est sûr (safe) si une lecture qui n'est pas simultanée à une écriture retourne la dernière valeur écrite dans le registre.



Dans cette situation le registre est sûr si la lecture  $R(x)$  retourne la valeur 3.



# Registres sûrs

W(x,3)

W(x,4)

R(x)

R(x) peut retourner 3 ou 4, les écritures étant simultanée on ne sait pas laquelle est effective avant l'autre.

# Registres sûrs

W(x,3)

R(x)

R(x) peut retourner n'importe quelle valeur, le comportement du registre n'est pas défini si une écriture et une lecture sont simultanées. De même dans la situation suivante.

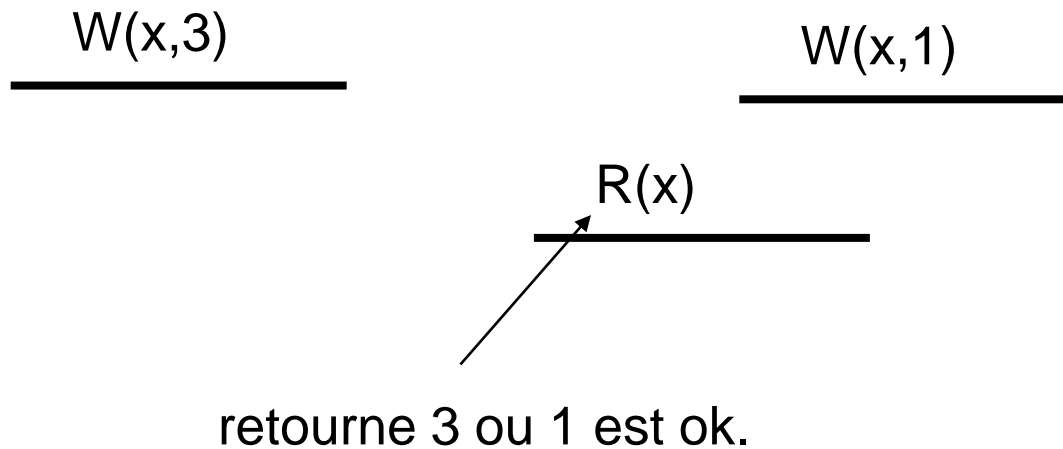
W(x,3)

W(x,1)

R(x)

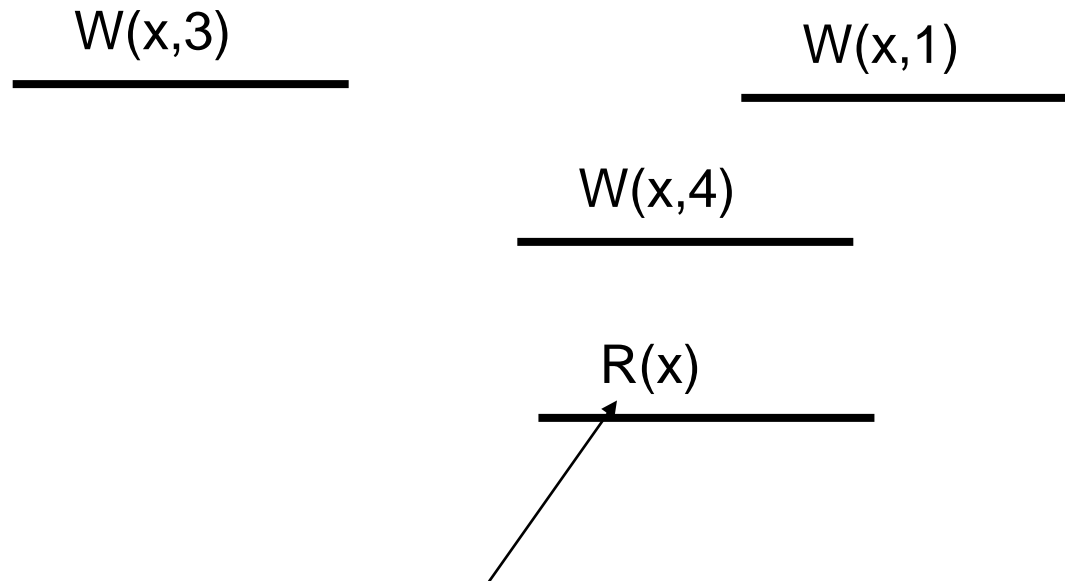
# Registre réguliers

Un registre régulier est sûr et lorsqu'une lecture et plusieurs écritures sont simultanées, il retourne (lecture) soit la dernière valeur écrite, soit une valeur écrite concurremment





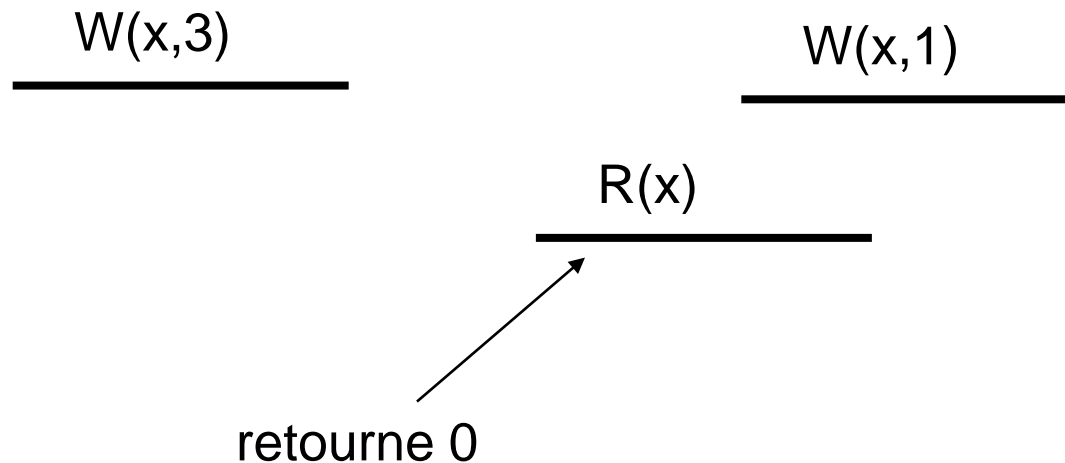
# Registres réguliers



retourne 4, 3 ou 1

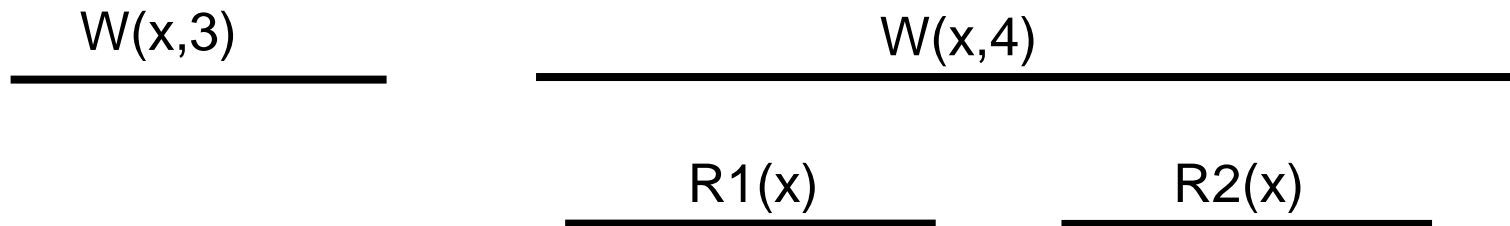
# Registres réguliers

L'exemple ci-dessous montre une exécution où le registre n'est pas régulier, la valeur retournée ne correspond pas ni à la dernière écriture ni à l'écriture concurrente.



# Registres atomiques

Un registre est atomique s'il est régulier et linéarisable. La condition de linéarisabilité détermine le comportement du registre lorsque plusieurs lectures sont séquentiellement effectuées. Le registre est linéarisable si des lectures successives sont bien vues successives par le registre.



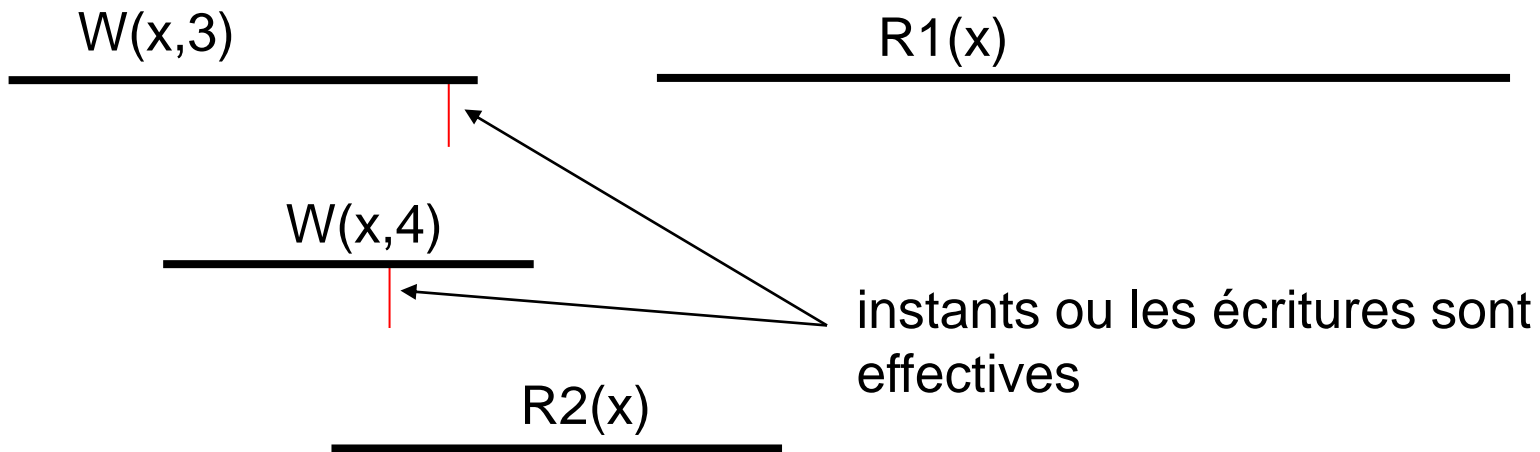
R1 retourne 3 et R2 retourne 3, ok

R1 retourne 3 et R2 retourne 4, ok

R1 retourne 4 et R2 retourne 3, pas atomique, mais régulier

R1 retourne 4 et R2 retourne 4, ok

# Registres atomiques



$R2$  retourne 4 et  $R1$  retourne 3 est atomique. Correspond à la situation où l'écriture de 3 est effective après l'écriture de la valeur 4.

# Registres atomiques

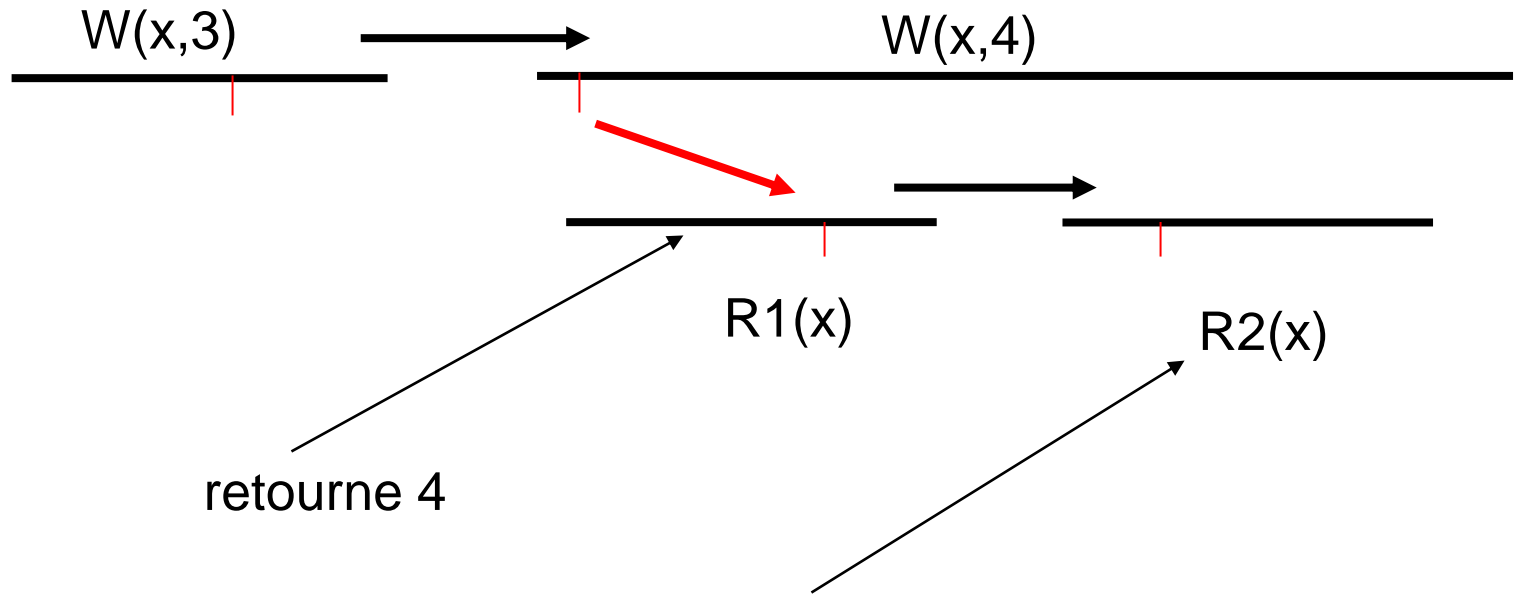
Pour définir la notion de registre atomique plus formellement, on introduit une notion d'ordre de précedence.

Les lectures/écritures d'un même processus sont toutes ordonnées par une relation de précedence décrite par l'ordre séquentiel du programme.

On imagine que les accès aux registres sont effectifs à un instant donné instantanément.

Une exécution particulière détermine un ordre inter-processus qui doit être **total**.

# Exemple



R2 ne peut retourner que 4 pour assurer que l'ordre soit total

# Définitions

On note  $v^i, i = 1, 2, \dots$  les différentes valeurs prises par un registre pendant une exécution donnée.

A chaque valeur correspond *une unique opération d'écriture*, on note ces opérations  $W^i, i=1, 2, \dots$  Tapez une équation ici.

On note  $R^i, i = 1, 2, \dots$  les lectures du registre qui retournent les valeurs  $v^i, i = 1, 2, \dots$

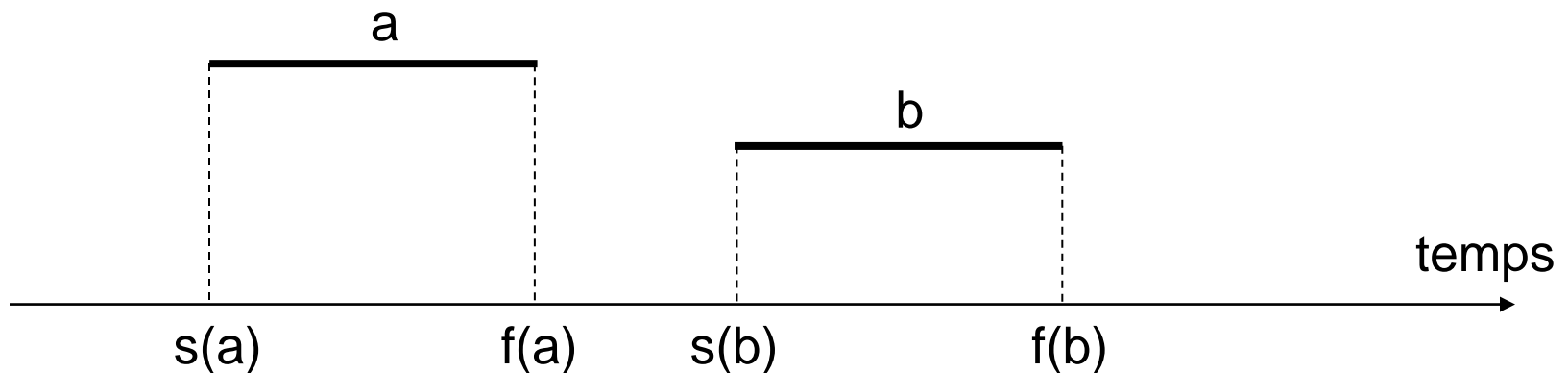
Une exécution particulière contient un unique  $W^i$  mais peut contenir plusieurs  $R^i$

# Définitions

A chaque opération  $a$  de lecture ou écriture on associe un temps de début  $s(a)$  et de fin  $f(a)$ . Pour deux opérations  $a$  et  $b$  on a

$$a \rightarrow b$$

si  $f(a) < s(b)$





# Définitions

Un registre est régulier si:

- Une lecture ne précède jamais une écriture, c'est-à-dire on a jamais  $R^i \rightarrow W^i$
- Une lecture ne retourne jamais une valeur passée qui à été modifiée, on a jamais  $W^i \rightarrow W^j \rightarrow R^i$

Un registre atomique satisfait en plus

$$si R^i \rightarrow R^j \text{ alors } i \leq j$$

Une registre est sûr s'il est atomique lorsqu'on se restreint aux exécutions sans chevauchement (overlap)

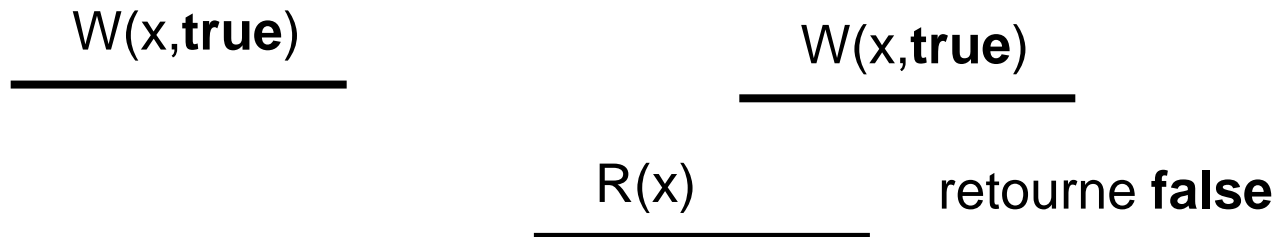
# Registre Java sûr

Le prototype du registre SRSW booléen sûr en Java est donné par le code suivant

```
class SafeBoolean {  
    boolean value;  
    public boolean getValue() {  
        return value;  
    }  
    public void setValue(boolean b) {  
        value = b;  
    }  
}
```

# Registre Java régulier

La seule situation où le registre précédent n'est pas régulier est la situation où deux écritures consécutives sont identiques (soit **false** soit **true**) et qu'une lecture concurrente retourne la valeur complémentaire.



Pour éviter cette situation, on ne réécrit pas la valeur du registre.

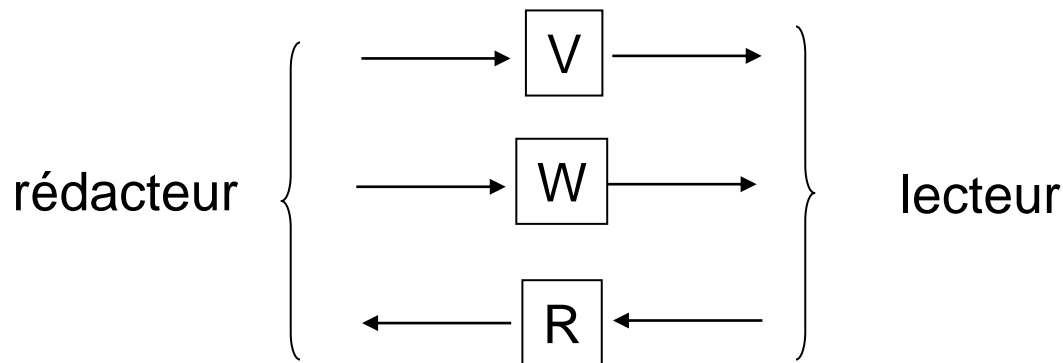
# Registre Java régulier

```
class RegularBoolean {  
    boolean prev; // on mémorise la valeur précédente, n'est pas partagée  
    SafeBoolean value; // un registre safe  
    public boolean getValue() {  
        return value.getValue();  
    }  
    public void setValue(boolean b) {  
        if (prev != b) {  
            value.setValue(b);  
            prev = b;  
        }  
    }  
}
```

# Registre atomique

Pour implémenter un registre atomique, on note la méthode **setValue()** par **change()** pour indiquer que l'écriture est effective seulement si la valeur est différente

Pour construire le registre V atomique on utilise deux registres W et R modifiés par le rédacteur et le lecteur respectivement (**registres réguliers**)



# registre atomique

Protocole rédacteur

change V

**if** W == R **then** change W

Protocole lecteur

1. **if** W == R **then return** v

2. read x := V

3. **if** W <> R **then** change R

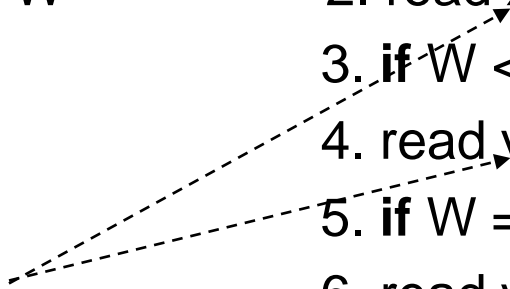
4. read y := V

5. **if** W == R **then return** v

6. read v := V

7. **return** x

variable locales



# Preuve

On montre pour commencer que l'on a jamais  $R^i \rightarrow W^i$

On procède par l'absurde, soit  $r$  la (sous-)lecture qui retourne la valeur erronée. Par définition, on a

$$f(r) < f(R) < s(W)$$

Le registre  $V$  est régulier, alors aucune lecture  $r$  ne peut satisfaire  $f(r) < s(W)$ .

# Preuve

On montre que le registre est régulier, en conjonction avec ce qu'on a déjà prouvé, il reste à voir qu'on n'a jamais  $W^i \rightarrow W^j(w) \rightarrow R^i(r)$

On procède par l'absurde. On note par  $r$  la (sous-)lecture du registre  $V$  qui retourne la valeur dans  $R^i$ , et  $w$  l'écriture dans  $W^j$ .

Le registre  $V$  est régulier, alors on doit avoir  $s(r) < f(w)$ . C'est possible si la lecture se termine à la ligne 1. du protocole du lecteur et retourne la valeur de la variable locale  $v$ .

Mais cela ne peut pas se produire sous les hypothèses puisque le protocole d'écriture modifie la valeur de  $W$  avant de terminer.

En fait, on a démontré que

$$W \rightarrow R \Rightarrow w \rightarrow r$$



# Preuve

Il reste à montrer que le registre est atomique, c'est-à-dire que  
*si  $R^i \rightarrow R^j$  alors  $i \leq j$ , c'est-à-dire  $\neg (W^j \rightarrow W^i)$*

On procède par l'absurde

soient  $r_i, r_j$  les sous-lectures du registre  $V$ . On note  $r_i \in R^i$   
pour indiquer que la sous-lecture a bien été effectuée pendant la  
lecture  $R^i$

ce qui est le cas si le lecteur quitte le protocole aux lignes 5 ou 7. S'il  
quitte le protocole à la ligne 1, alors la sous-lecture a été effectuée  
avant.

Pour les écritures on a pas d'ambiguïté et on a bien  $w_j \rightarrow w_j$

On suppose  $i \neq j$  et on a alors  $r_i \rightarrow r_j$  (on a bien deux lectures  
qui retournent des valeurs différentes)

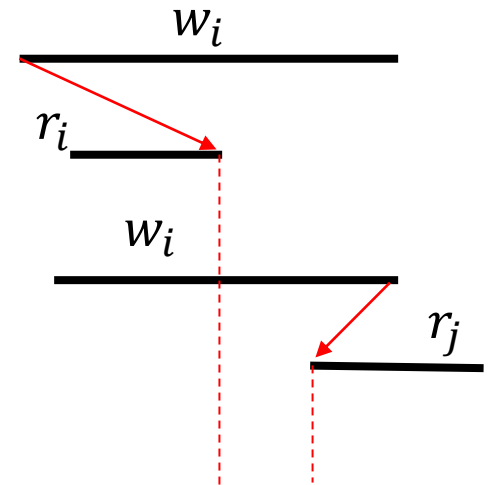
# Preuve

En tenant compte du fait que  $V$  est régulier, on doit nécessairement avoir

$$s(w_i) < f(r_i) \text{ car } \neg(r_i \rightarrow w_i)$$

et aussi

$$s(r_j) < f(w_i) \text{ car } \neg(w_j \rightarrow w_i \rightarrow r_j)$$



comme  $r_i \rightarrow r_j$ , on a  $s(w_i) < f(r_i) < s(r_j) < f(w_i)$

et la valeur de  $W$  ne change pas pendant l'intervalle  $[f(r_i), s(r_j)]$  (\*)

# Preuve

La lecture  $r_i$  peut s'exécuter aux lignes 2, 4 ou 6 du protocole.

**cas 1:** read  $x := V$  ligne 2. Alors  $R_i$  retourne la valeur de  $x$  à la ligne 7. de son protocole. On voit dans l'exécution du protocole qu'entre la ligne 2. et la ligne 7. la valeur de  $W$  doit nécessairement changer, ce qui contredit la remarque (\*) ( on a  $R^i \rightarrow R^j$  ).

**cas 2:** read  $v := V$  ligne 4. Alors  $R_i$  retourne à la ligne 5 de son protocole après avoir testé  $W == R$ . Comme la valeur de  $W$  ne change pas dans l'intervalle  $[f(r_i), s(r_j)]$  l'exécution du protocole pour  $R^j$  doit se terminer à la ligne 1. et les valeurs retournées sont les mêmes, une contradiction.

**cas 3:** read  $v := V$  ligne 6.  $R^i$  retourne à la ligne 1. de son protocole après avoir testé  $W == R$  et on obtient la même contradiction qu'as cas 2.

# Registres multivalués SRSW

Pour implémenter un registre multivalué SRSW atomique on utilise un tableau de registres **booléens atomique SRSW**.

L'idée est de représenter un nombre compris dans l'intervalle **0 ... maxVal-1** par un bit positionné à **true** à la position correspondante dans le tableau.

Par exemple, on code la valeur 3 par le tableau:

0	1	2	3	4
false	false	false	true	false

# Registres multivalués SRSW

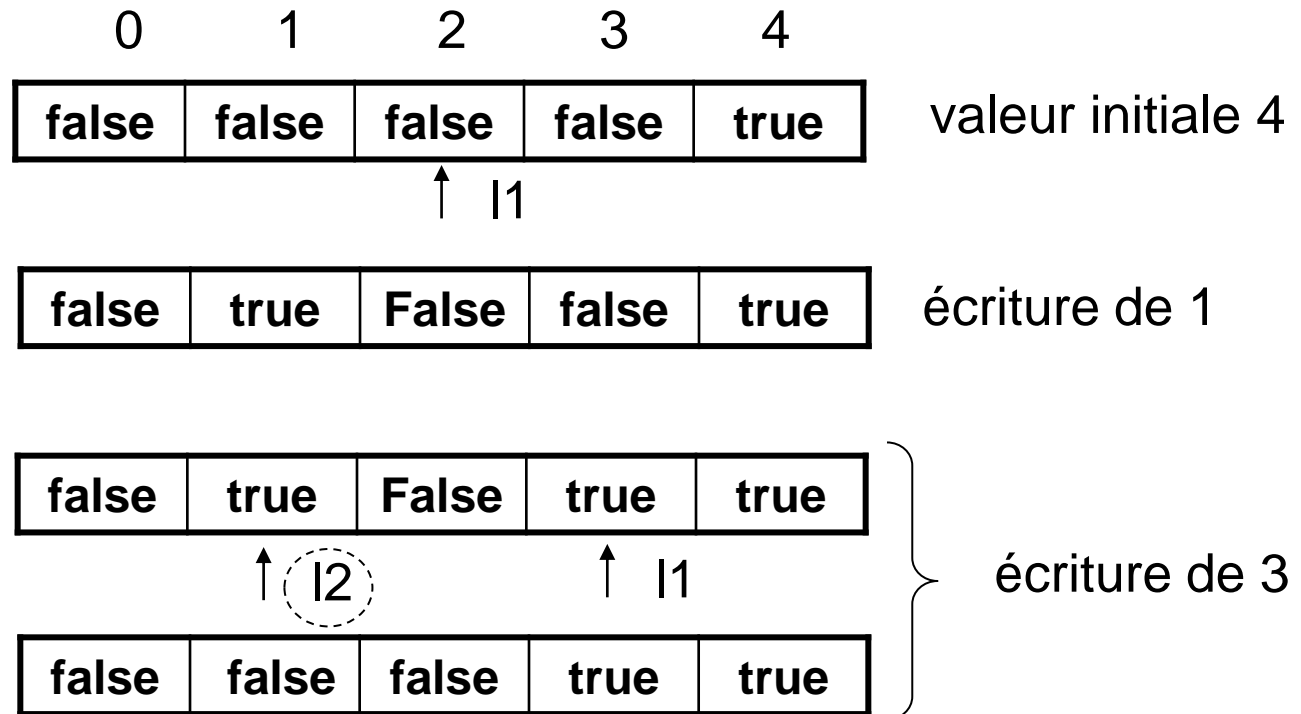
Pour l'écriture la procédure est de la forme:

```
public void setValue(int x) {  
    A[x] = true;                // on dépose la valeur  
    for( int i = x - 1; i >= 0; i--)  
        A[i] = false;          // on supprime la valeur précédente si <  
}
```

Il est nécessaire de positionner l'entrée correspondante dans un premier temps et de supprimer les autres entrées. Sinon, un lecteur pourrait trouver toutes les entrées à **false**.

# Registres multivalués MRSW

Pour le lecteur, on ne peut pas se contenter de parcourir le tableau dans l'ordre croissant des indices pour que le registre soit atomique. En effet, on considère l'exécution où on écrit 4 puis 1 puis 3 dans le registre

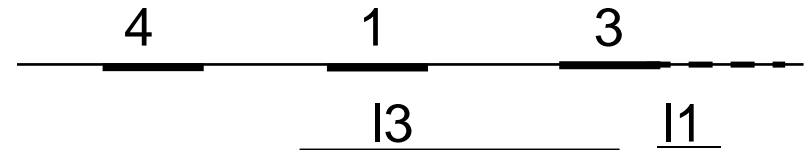


I2, lecture 2 après la première lecture I1

# Registres multivalués SRSW

On a les écritures suivantes  $4 \rightarrow 1 \rightarrow 3$

La première lecture retourne 3



La deuxième lecture retourne 1

Le registre n'est pas atomique puisque la seconde lecture retourne la valeur précédant l'écriture concurrente et la première lecture la valeur écrite concurremment.

Le registre est régulier. En fait c'est un registre MRSW régulier (même si les registres SRSW sont atomiques).

# Registres multivalués atomique SRSW

```
class MultiValued {  
    int n = 0;  
    boolean A[] = null;  
    public MultiValued(int maxVal, int initVal) {  
        n = maxVal; A = new boolean [n];  
        for(int i = 0; i < n; i++) A[i] = false;  
        A[initVal] = true;  
    }  
    public void setValue(int x) {  
        A[x] = true;                // on dépose la valeur  
        for( int i = x - 1; i >= 0; i--)  
            A[i] = false;          // on supprime la valeur précédente si <  
    }
```



# Registres multivalués atomique SRSW

```
public int getValue() {  
    int j = 0;  
    while(!A[j]) j++;           // forward scan  
    int v = j;  
    for(int i = j-1; i >= 0; i--) } backward scan  
        if (A[i]) v = i;  
    return v;  
}  
}
```

Cette construction est aussi valable pour plusieurs lecteurs, on a un registre MRSW atomique.

# registre MRSW atomique

Pour permettre plusieurs lecteur d'accéder simultanément le registre, l'idée est de maintenir un sous-registre par lecteur, c'est-à-dire utiliser un tableau de SRSW registres  $V[n]$  (des registres **réguliers**)

Le rédacteur écrit les nouvelles valeurs dans tous les sous-registres.

Le registre ainsi obtenu n'est pas atomique.

Il faut qu'un lecteur s'assure après la lecture de son registre qu'il a obtenu la dernière valeur écrite.

Pour cela, les registres SRSW de base sont constitués de deux champs, **value** et **ts** pour la valeur et un **timestamp**.

Un tableau **Comm[i][j]** indique la valeur lue par le lecteur i au lecteur j.

# registre MRSW atomique

```
class MRSW {  
    int n = 0;  
    SRSW V[] = null; // tableau de registres  
    SRSW Comm[][] = null; // communication inter lecteur  
    int seqNo = 0;  
    public MRSW(int readers, int initVal) {  
        n = readers;  
        V = new SRSW[n];  
        for(int i = 0; i < n; i++)  
            V[i].setValue(initVal,0); // initialisation de la valeur et timestamp = 0  
        Comm = new SRSW[n][n];  
        for(int i = 0; i < n; i++) for(int j = 0, j < n; j++)  
            Comm[i][j] .setValue(initVal,0);  
    }  
}
```

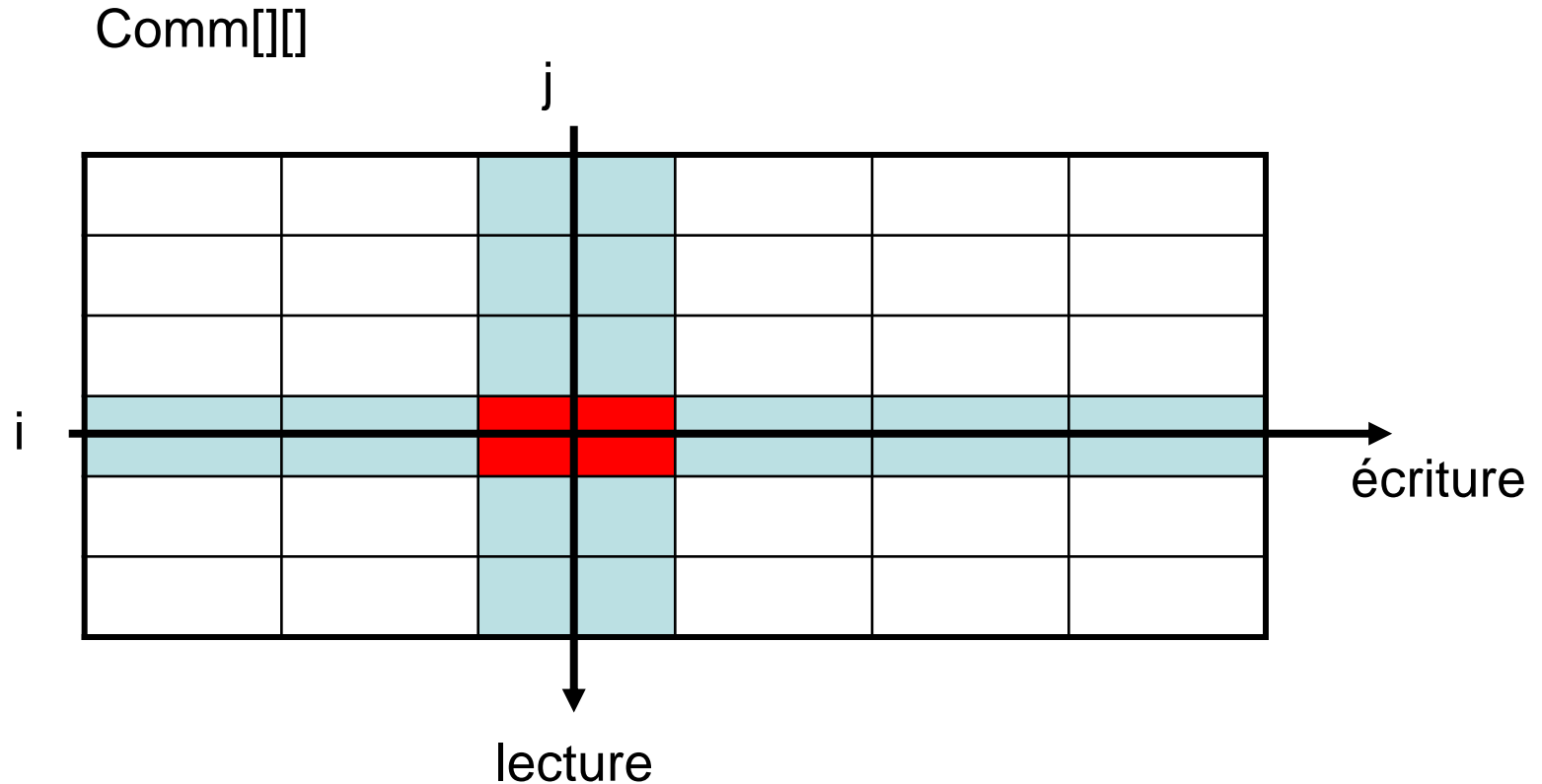
# Registre MRSW atomique

```
public int getValue(int r) { // lecteur r lit son propre registre
    SRSW tsv = V[r]; // variable locale
    for(int i = 0; i < n; i++)
        if (Comm[i][r].getTS() > tsv.getTS())
            tsv = Comm[i][r];
    for(int i = 0; i < n; i++) {
        Comm[r][i].setValue(tsv);
    }
    return tsv.getValue();
}
```

# Registre MRSW atomique

```
public void setValue(int x){  
    seqNo++; // numéro de séquence suivant  
    for(int i = 0; i < n; i++)  
        V[i].setValue(x,seqNo);  
    }  
}
```

# Registre MRSW atomique



*si  $R^i \rightarrow R^j$  alors  $\neg (W^i \rightarrow W^j)$*

# Remarque

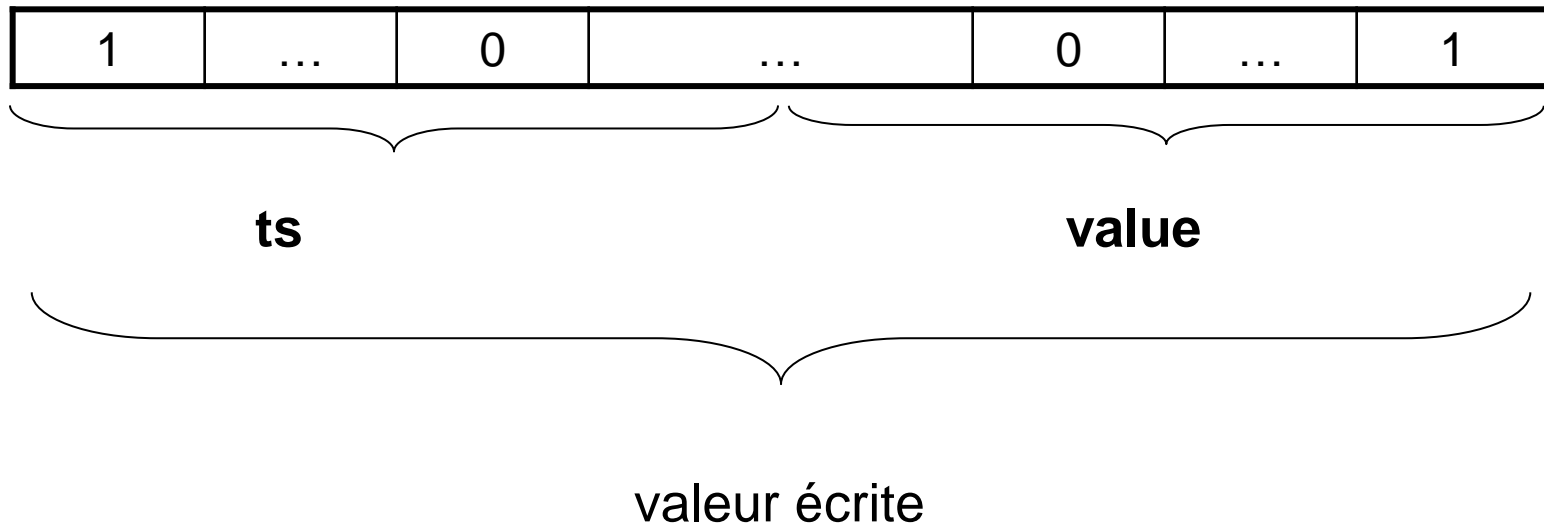
L'intérêt de travailler avec des objets atomiques (linéarisables) est que n'importe quelle exécution peut s'interpréter comme un entrelacement d'exécutions atomiques.

Pour spécifier la sémantique d'un objet on doit spécifier le préconditions admissibles et les postconditions correspondantes. (La documentation augmente linéairement avec le nombre d'objets)

La propriété de linéarisabilité est locale, c'est-à-dire que si tous les objets d'un programme le sont, le programme est linéarisable aussi.

# Remarque

Pour maintenir un timestamp **ts** ainsi qu'une valeur **value** on peut décomposer la valeur écrite dans le registre multivalué atomique comme:





# Autres solutions

La version 1.5 propose le paquetage `java.util.concurrent.atomic`, qui contient, entres autres, la classe

**`AtomicStampedReference<V>`**

Une autre solution utilise un tableau de registres multivalués atomiques.

Cette solution a d'autres applications et est purement algorithmique.

# ThreadLocal

Pour la programmation séquentielle, Java propose des objets de types statiques et non statiques. Les objets non statiques sont répliqués dans chaque instances de la classe, les objets statiques sont communs à toutes les instances de la classe.

Un objet de type **ThreadLocal** est partagé par toutes les instances d'une classe **d'un même thread** et sont différents pour des threads différents.

# Consensus

Le problème du consensus est le suivant:

- i) A chaque processus est assigné une valeur initiale,
- ii) Les processus doivent se mettre d'accord sur une valeur commune

Cette valeur commune doit appartenir à l'ensemble des valeurs initiales pour éviter une solution triviale qui consiste à toujours choisir une même valeur. L'algorithme doit être *wait-free*.

Un objet consensus doit implémenter l'interface suivante

```
public interface consensus {  
    public void propose(int pid, int value);  
    public int decide(int pid);  
}
```

# Consensus – registres atomiques

Le problème du consensus peut-être résolu pour un seul processus avec des registres atomiques.

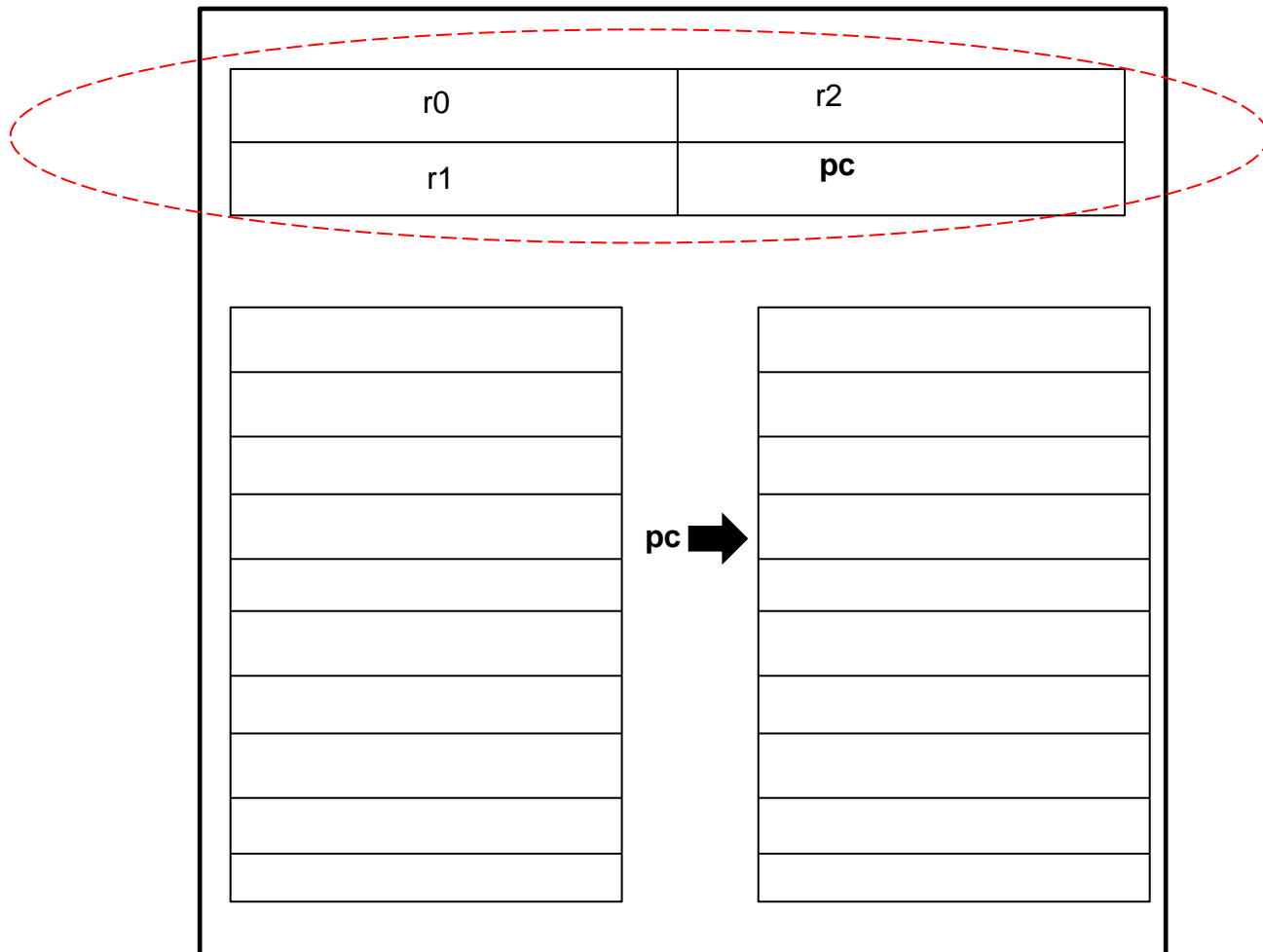
**Pour deux processus, il n'y a pas de solution wait-free au problème du consensus.**

Pour deux processus, le protocole doit donc permettre aux processus de se mettre d'accord sur une des deux valeurs initiales.

Le protocole est dans un *état bivalent* à un instant donné si les deux valeurs initiales peuvent être choisies selon l'exécution à venir.

Un état bivalent est *critique* si quelque soit la prochaine action exécutée l'état du protocole est non bivalent.

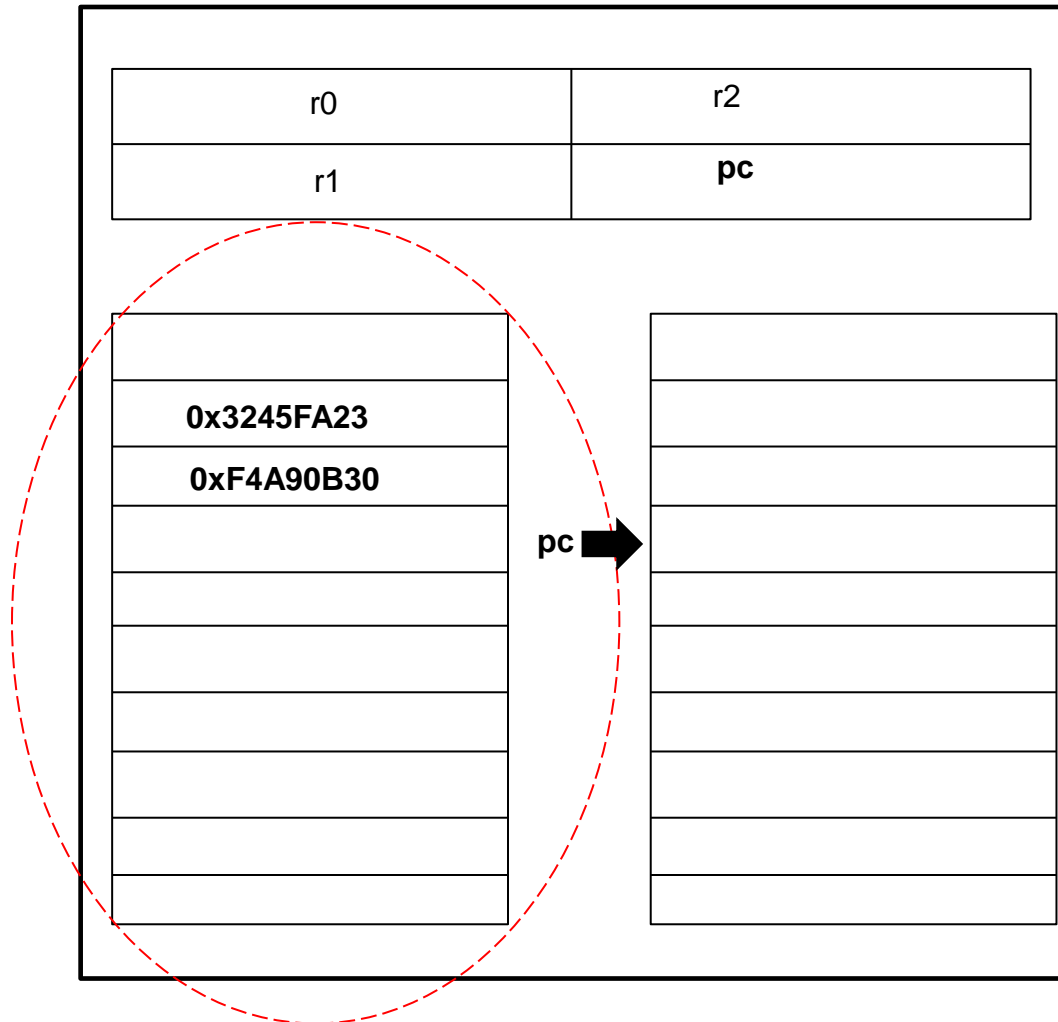
# Rappel – état d'un thread



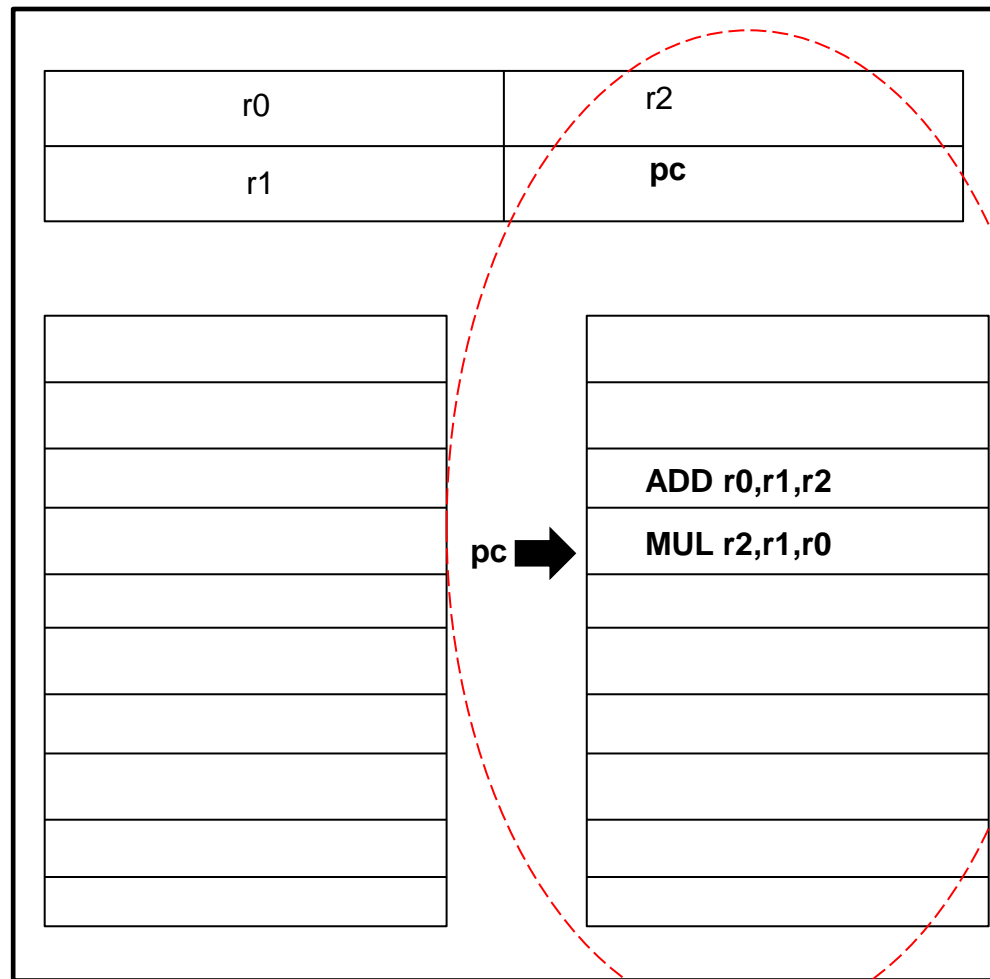
Les registres  
(ADD r0,r1,r2)

# Rappel – état d'un thread

La mémoire  
pour les variables



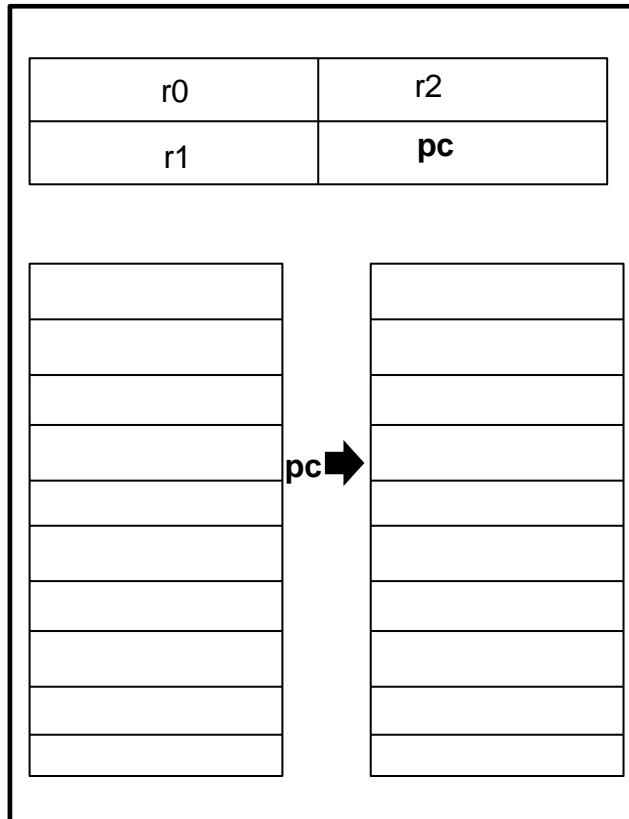
# Rappel – état d'un thread



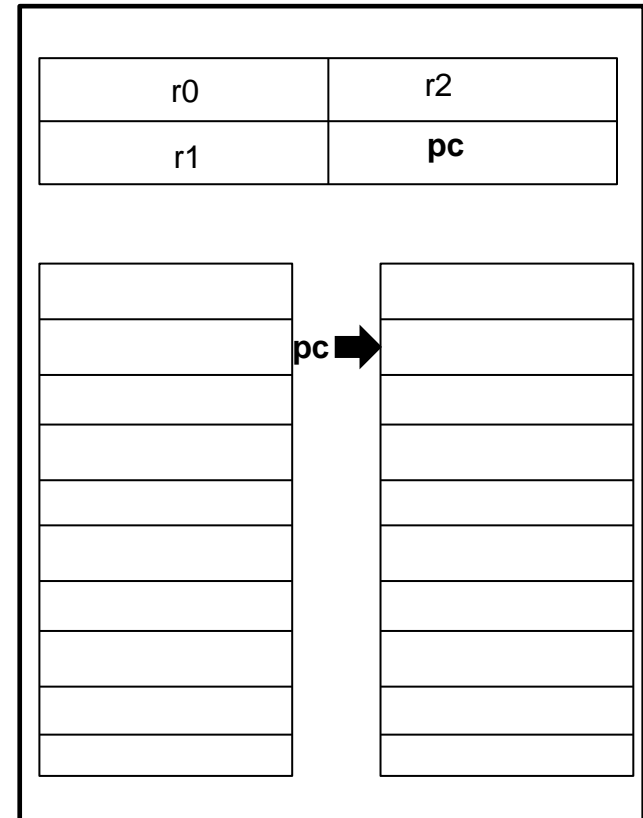
La mémoire  
pour le programme

# Rappel – état d'un système

**P0**



**P1**

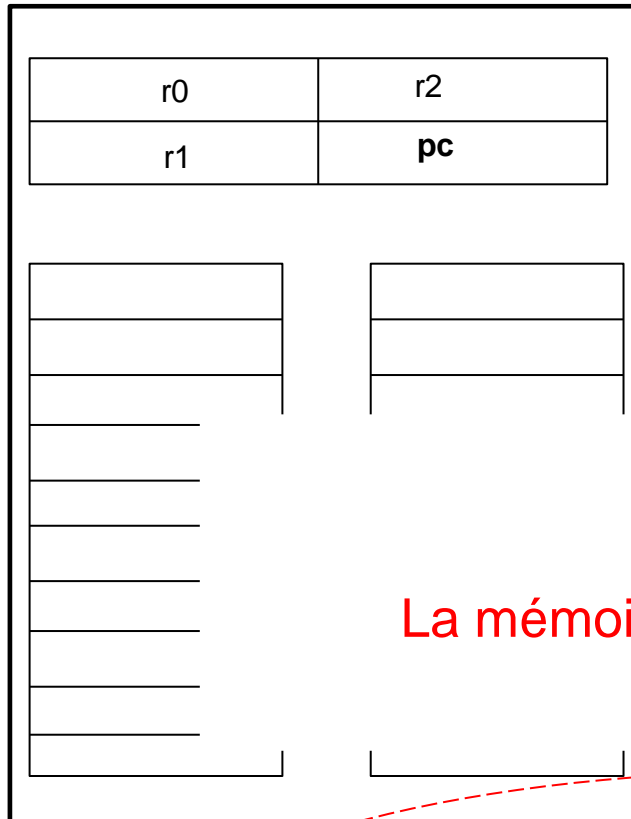


0x0AA4120C			
0x123ACB45			

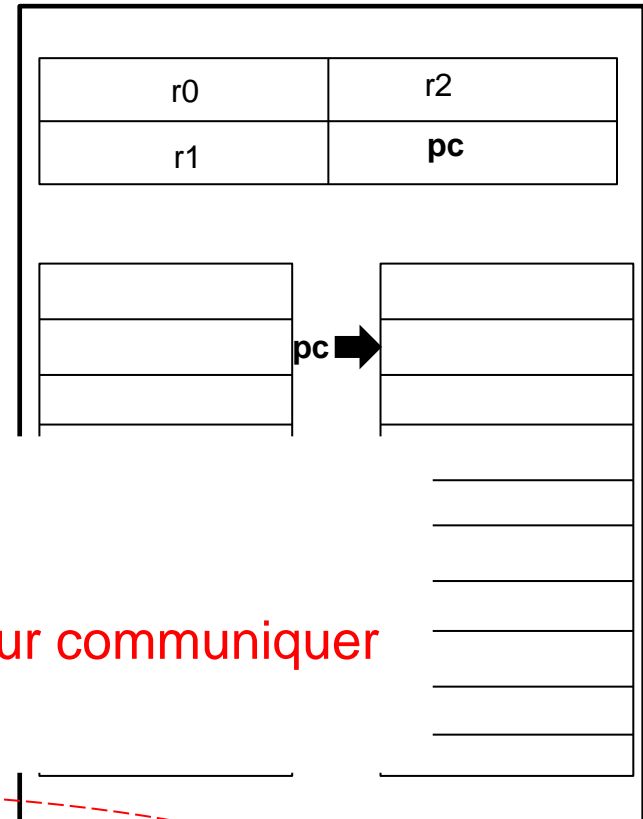


# Rappel – état d'un système

**P0**



**P1**

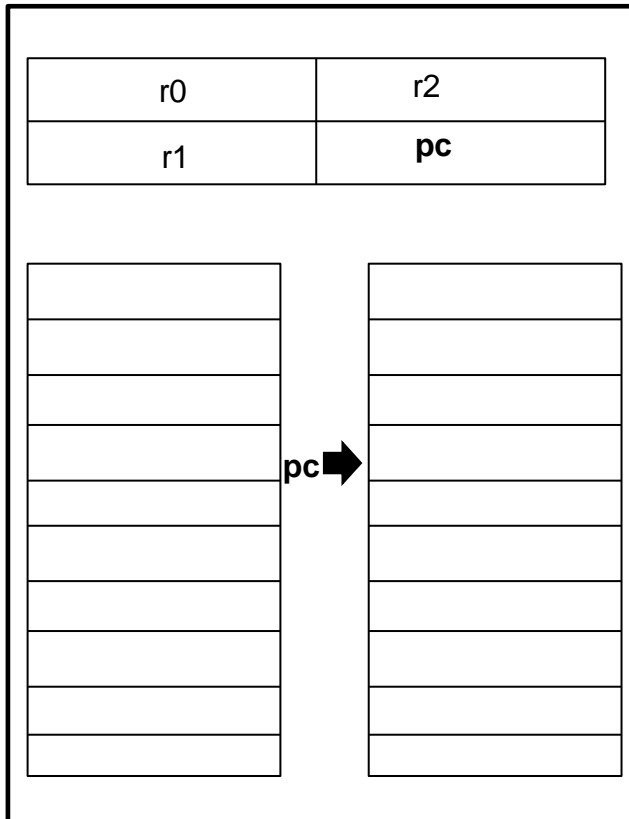


La mémoire partagée pour communiquer

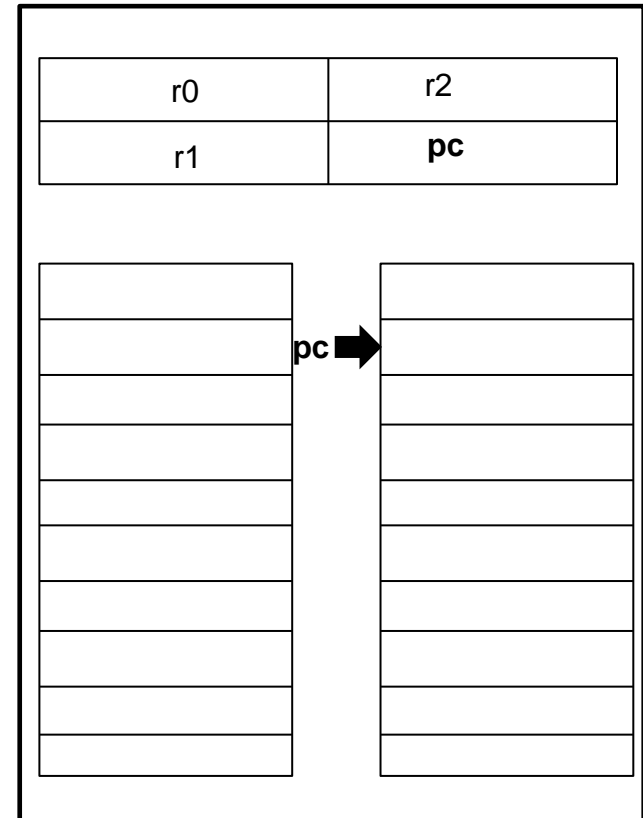
0x0AA4120C			
0x123ACB45			

# état d'un système P0

**P0**



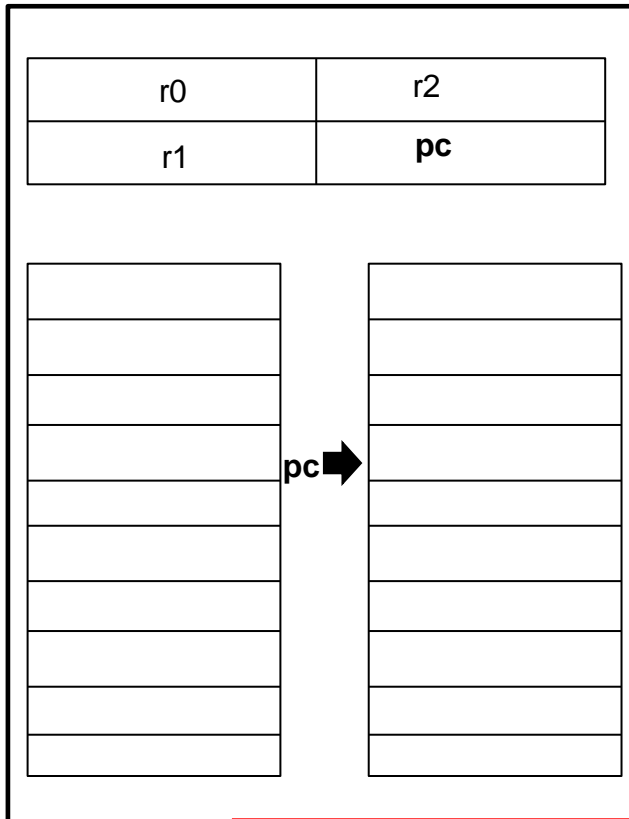
**P1**



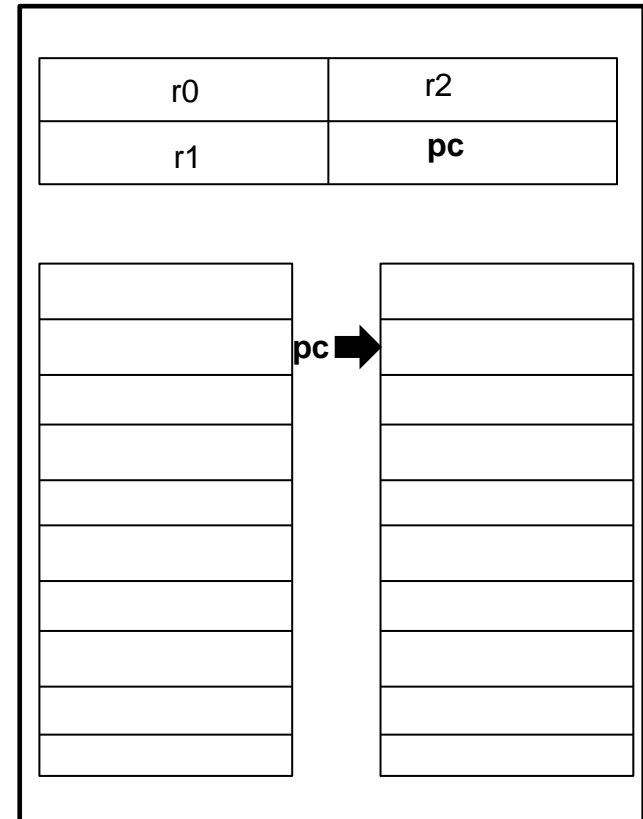
0x0AA4120C			
0x123ACB45			

# état d'un système P1

**P0**



**P1**



0x0AA4120C			
0x123ACB45			

# Etat initial

## **L'état initial du protocole est bivalent:**

On suppose que les deux valeurs initiales sont différentes, P0 obtient 0 et P1 1.

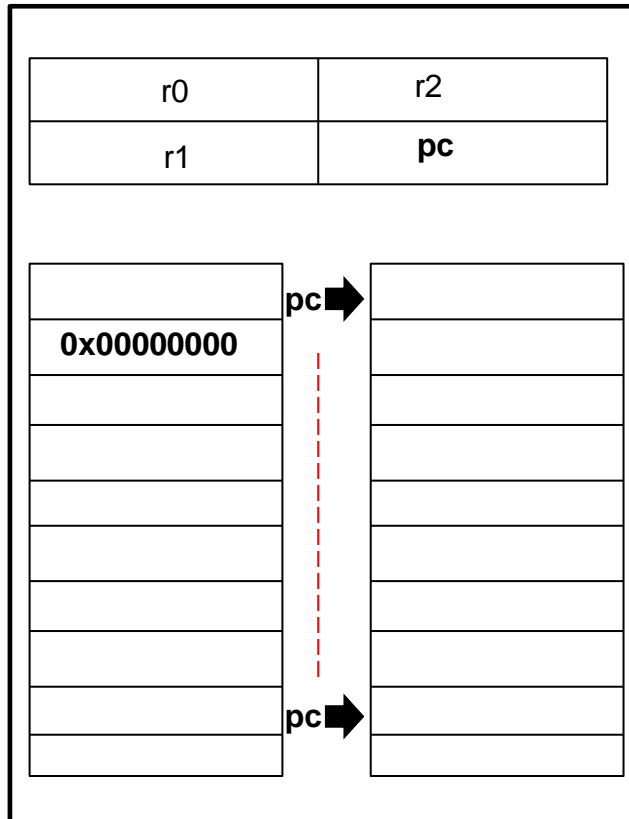
Une première exécution consiste en l'exécution complète du protocole par P0. Comme le protocole est wait-free, le protocole doit choisir la valeur 0 après un nombre fini d'étapes. Ensuite, le protocole est exécuté par P1 qui doit, lui aussi, choisir la valeur 0.

La deuxième exécution consiste en l'exécution du protocole par P1. Cette exécution doit se terminer par le choix de la valeur 1.

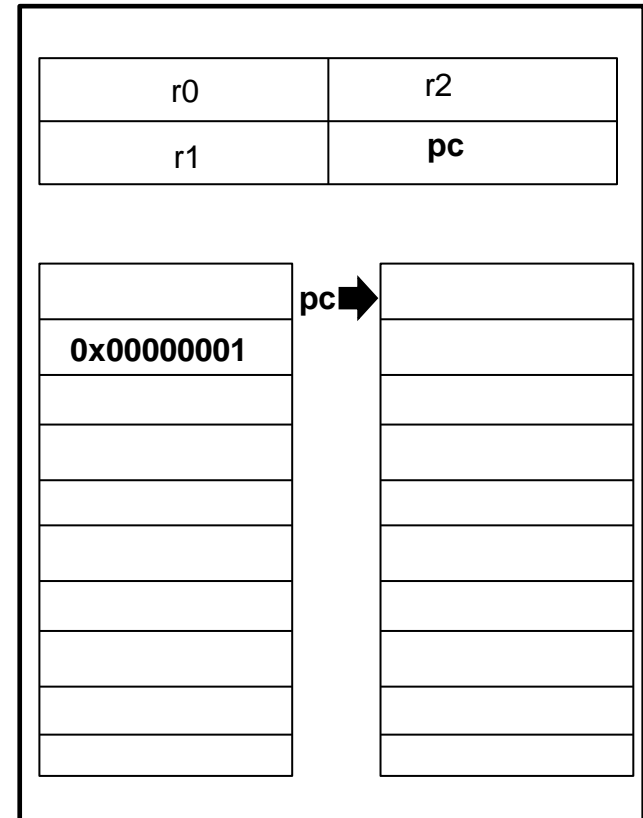
Ces deux exécutions montrent que l'état initial est bivalent.

# Illustration P0 s'exécute seul

**P0**



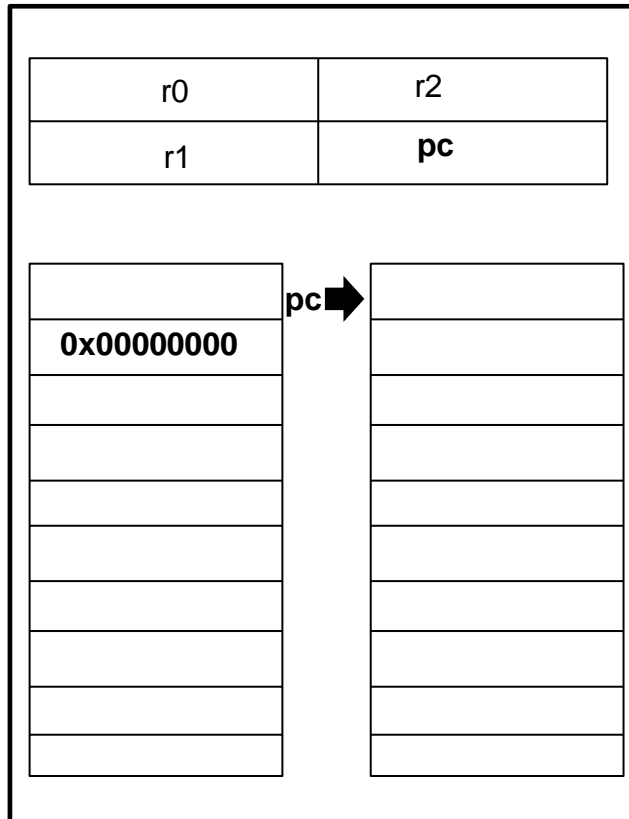
**P1**



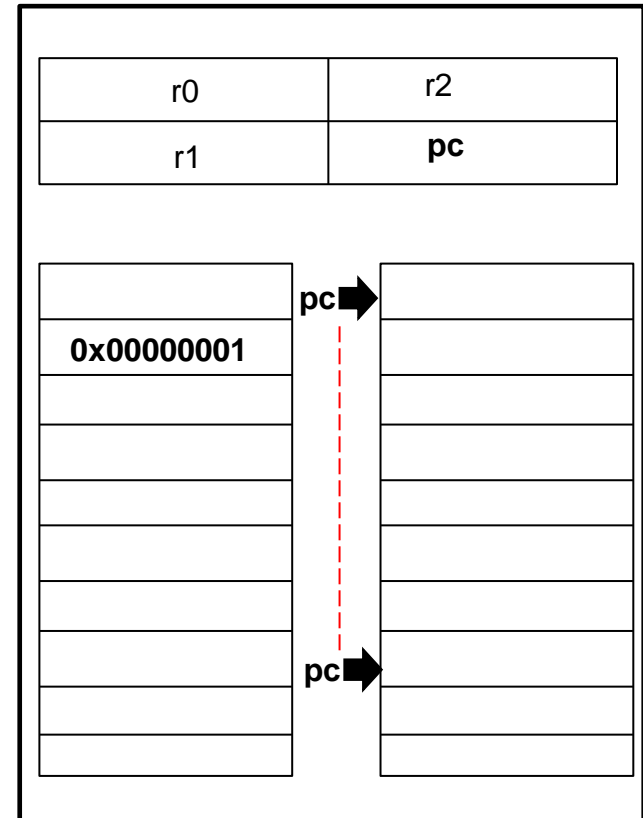
0x0AA4120C			
0x123ACB45			

# Illustration P1 s'exécute seul

**P0**



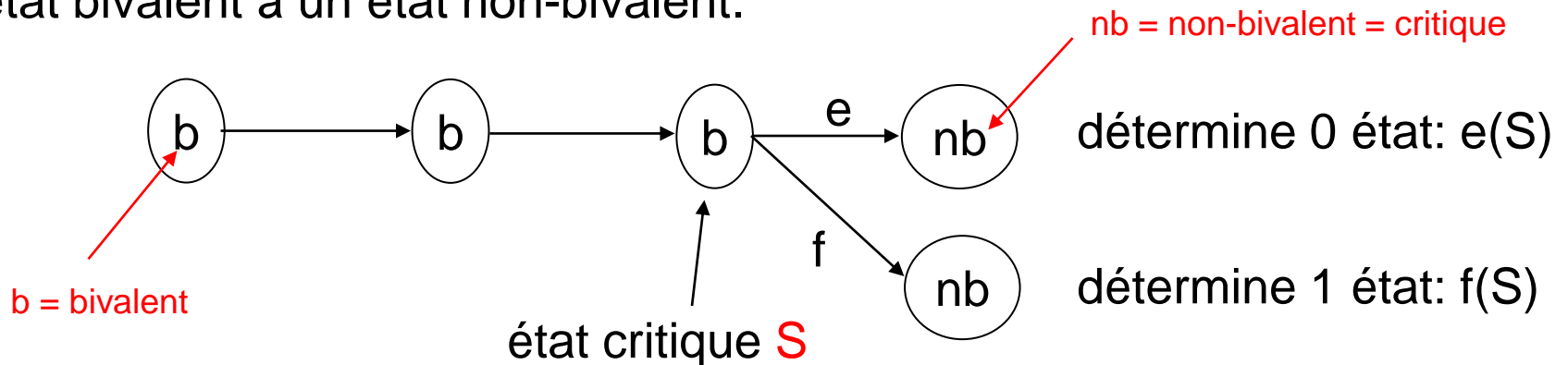
**P1**



0x0AA4120C			
0x123ACB45			

# Etats bivalents

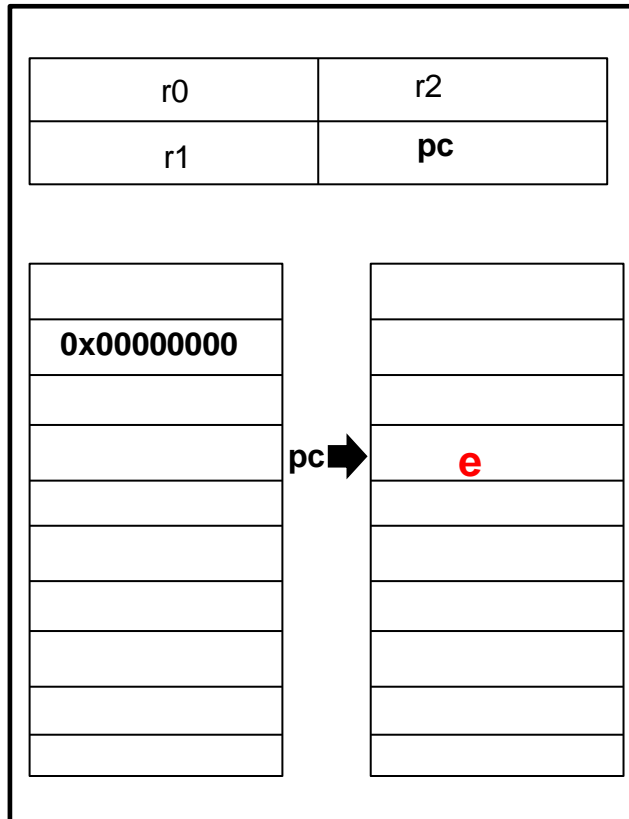
On considère l'exécution du protocole par les deux processus. Comme le protocole est wait-free, il arrive un instant où le protocole passe d'un état bivalent à un état non-bivalent.



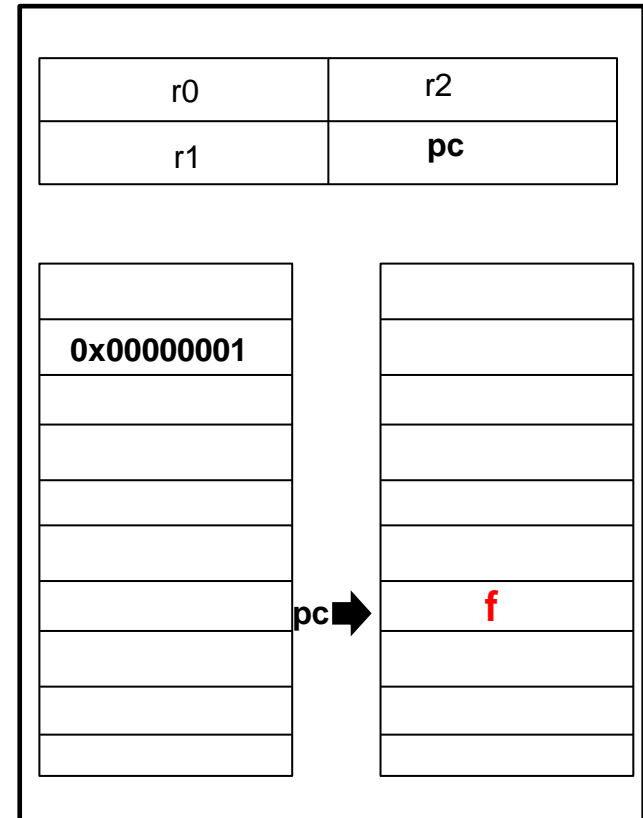
On appelle  $S$  l'état critique. On note  $e$  l'instruction de  $P_0$  et  $e(S)$  l'état du protocole après l'exécution de  $e$ . De même, on note  $f$  l'instruction de  $P_1$  et  $f(S)$  l'état du protocole après l'exécution de  $f$ . La valeur décidée en  $e(S)$  est différente de celle en  $f(S)$ .

# Illustration

**P0**



**P1**



0x0A4120C0			
0x05634F56			



# Actions e et f

Les instruction e et f peuvent être:

- Lecture atomique mémoire privée
- Lecture atomique mémoire partagée
- Ecriture atomique mémoire privée
- Ecriture atomique mémoire partagée

Pour chacune de ces instructions

- e et f accèdent la même adresse mémoire (partagée)
- e et f accèdent des adresses mémoires différentes (partagée ou locale)

Finalement e et f peuvent être des instructions 'internes', ADD, SUB,...

# Actions e et f

Il y a beaucoup de combinaisons mais l'analyse est un peu plus rapide.

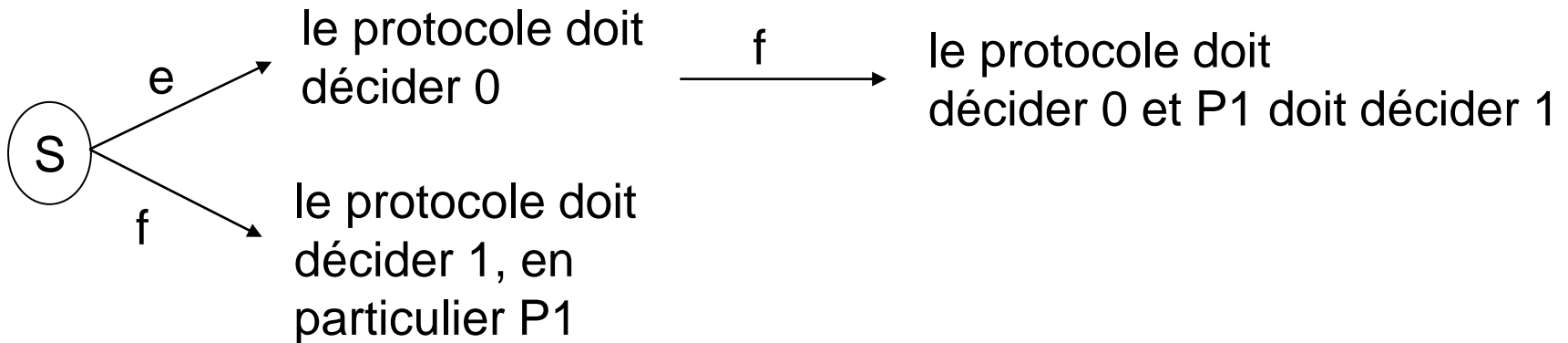
On va distinguer

- **Cas 1** soit e soit f est une lecture ou une instruction interne
- **Cas 2** e et f sont à des accès à des adresses mémoire différents
- **Cas 3** e et f sont à des accès à une même adresse mémoire

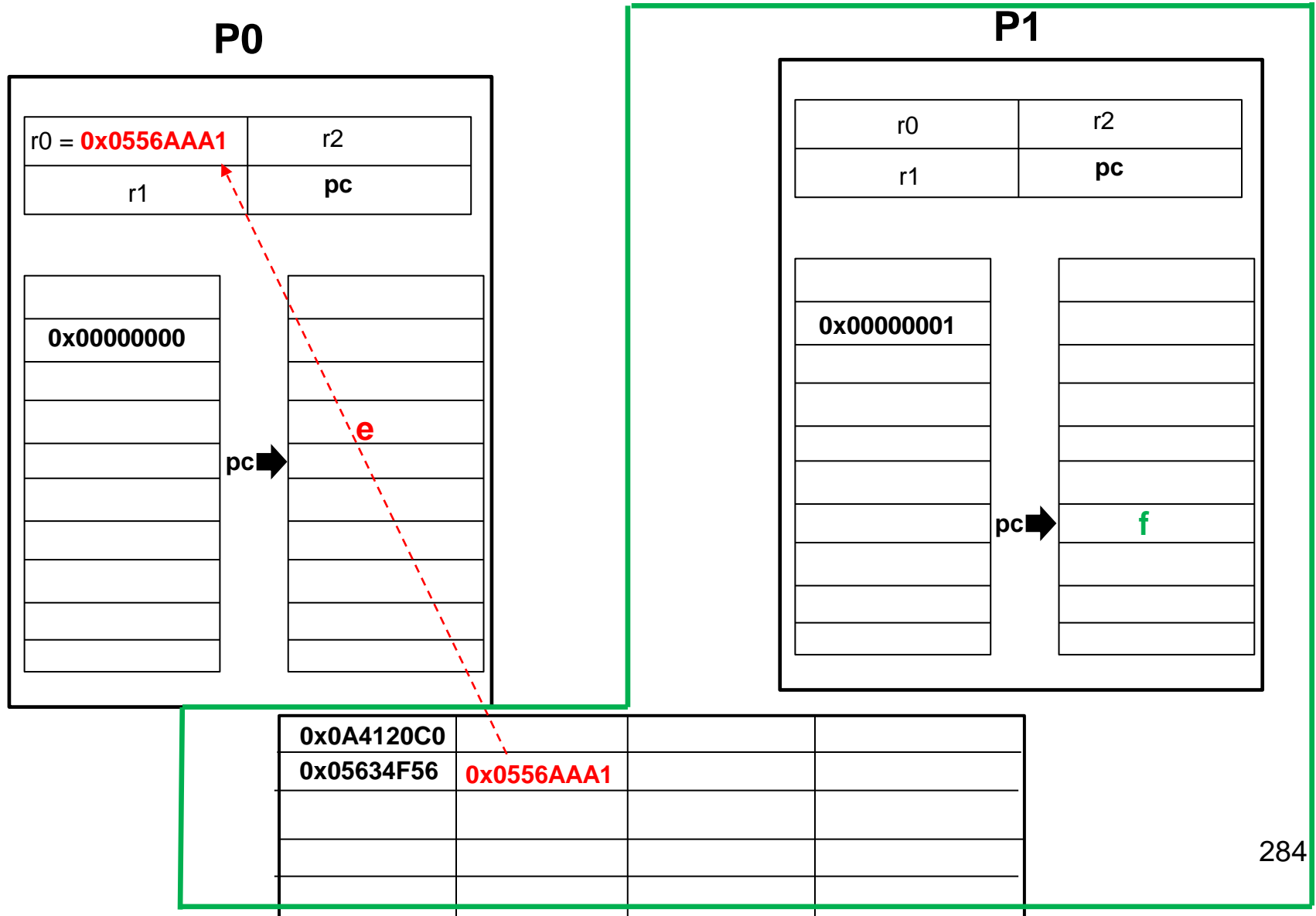
# Analyse

On discute les différentes formes de e et f.

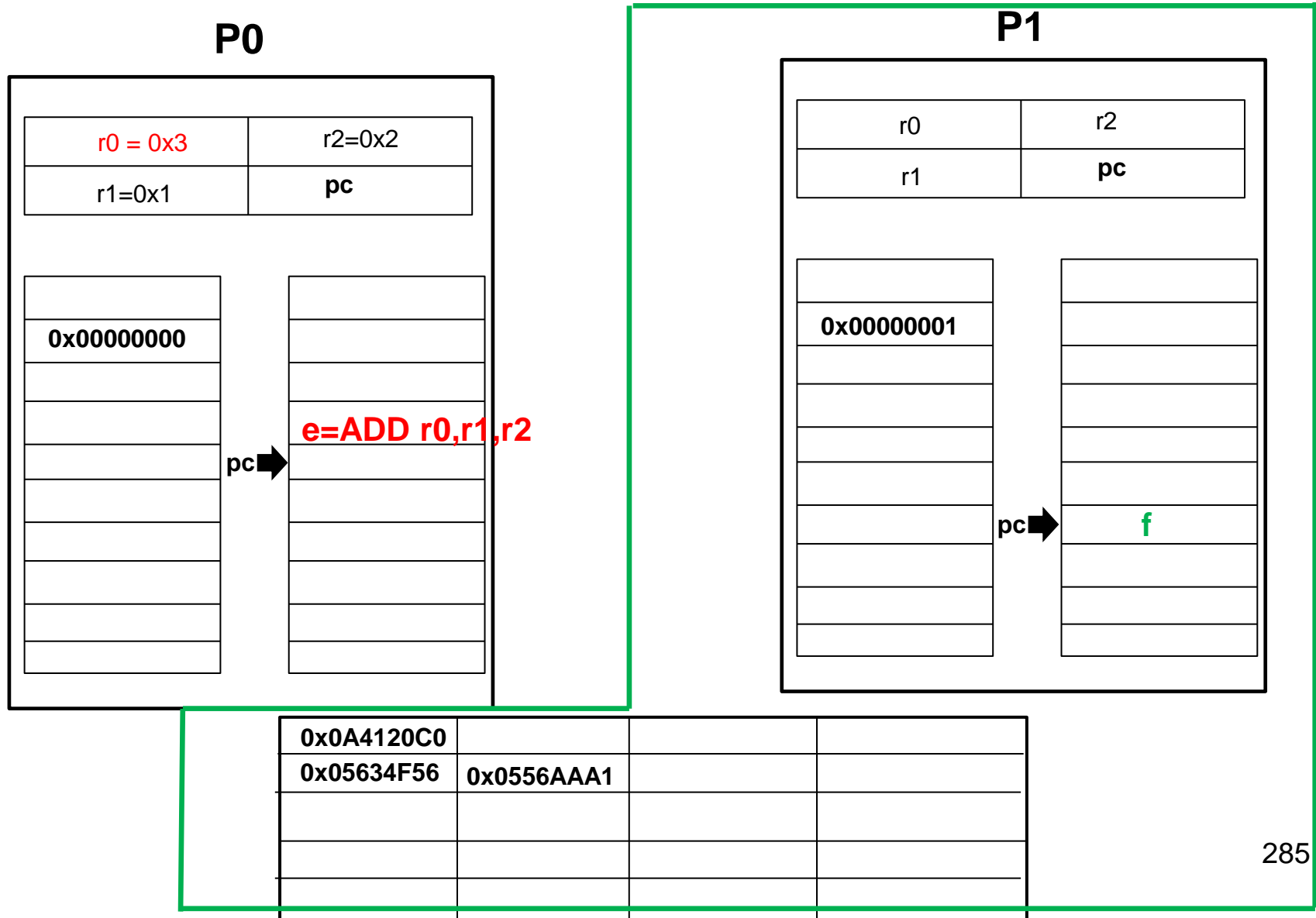
**Cas 1:** Soit e soit f est une lecture ou une instruction interne , disons e. Lorsque P0 exécute e, l'état de P1 ne change pas. Alors l'exécution de P1 depuis S ou e(S) est la même et cela contredit le fait que les valeurs choisies sont différentes



# Illustration e est une lecture

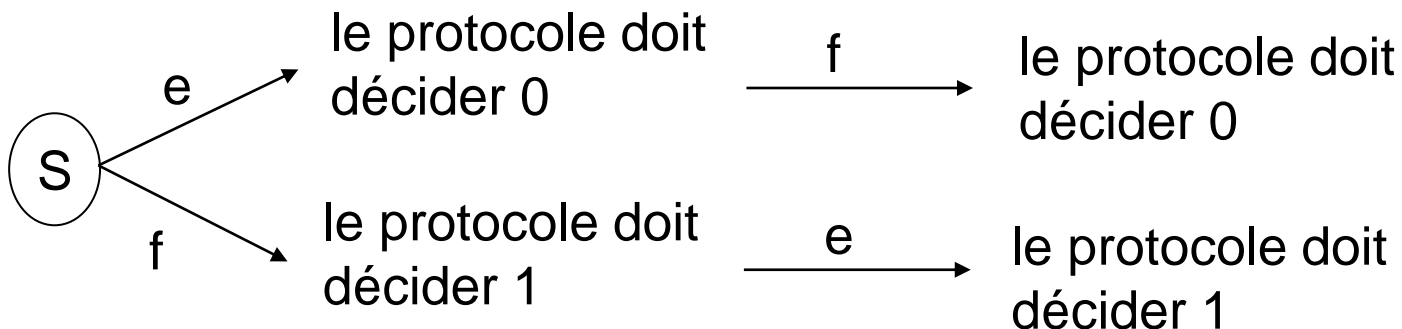


# Illustration e est une instruction interne



# Analyse

**Cas 2:** les instructions e et f correspondent à des écritures à des adresses mémoires différentes. Dans cette situation le résultat de e puis de f est le même que celui de f puis de e.



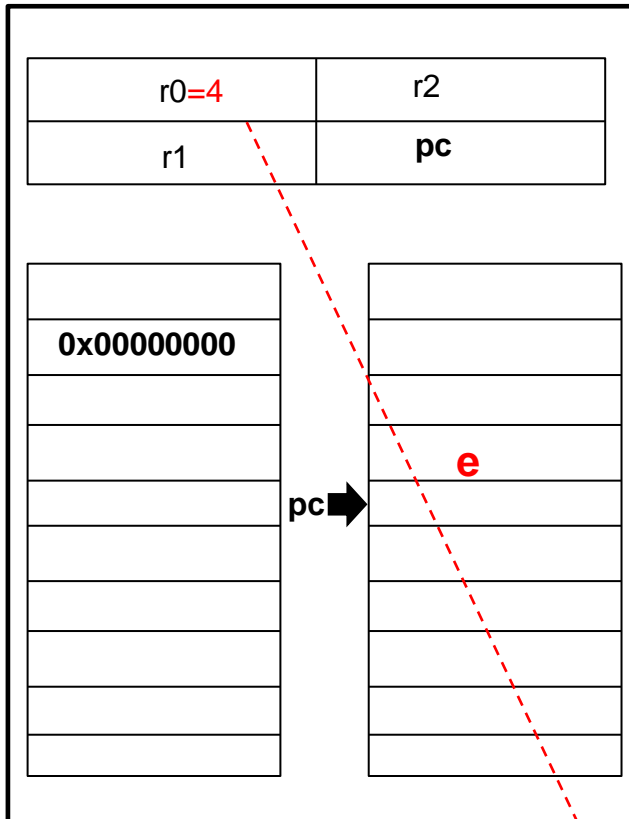
# Analyse

On a une contradiction car le protocole doit se trouver dans le même état après exécution de  $e$  puis  $f$  ou de  $f$  puis  $e$ , c'est-à-dire l'état du système  $e(f(S))$  est le même que  $f(e(S))$ .

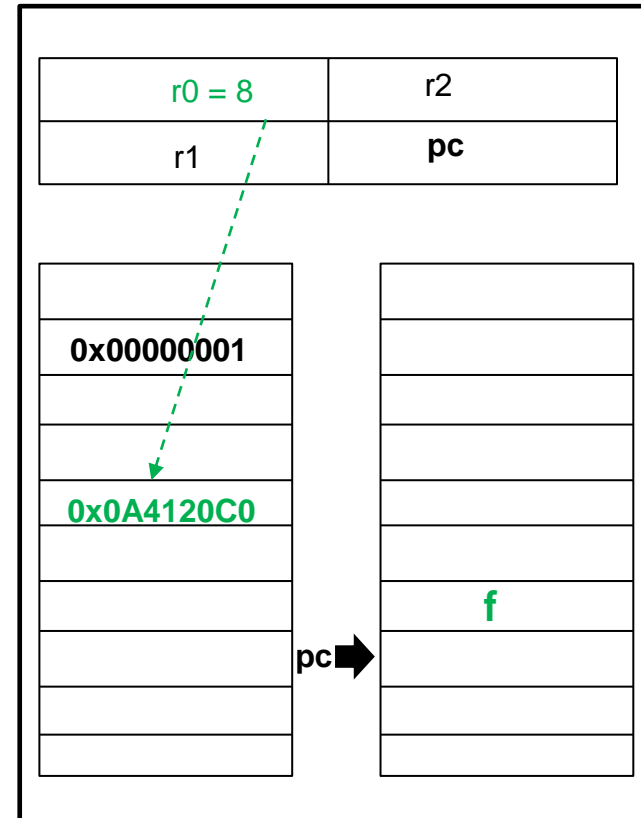
Puisque c'est le même état le système (déterministe) ne peut pas choisir deux valeurs distinctes.

# Illustration accès registres différents (écritures)

P0



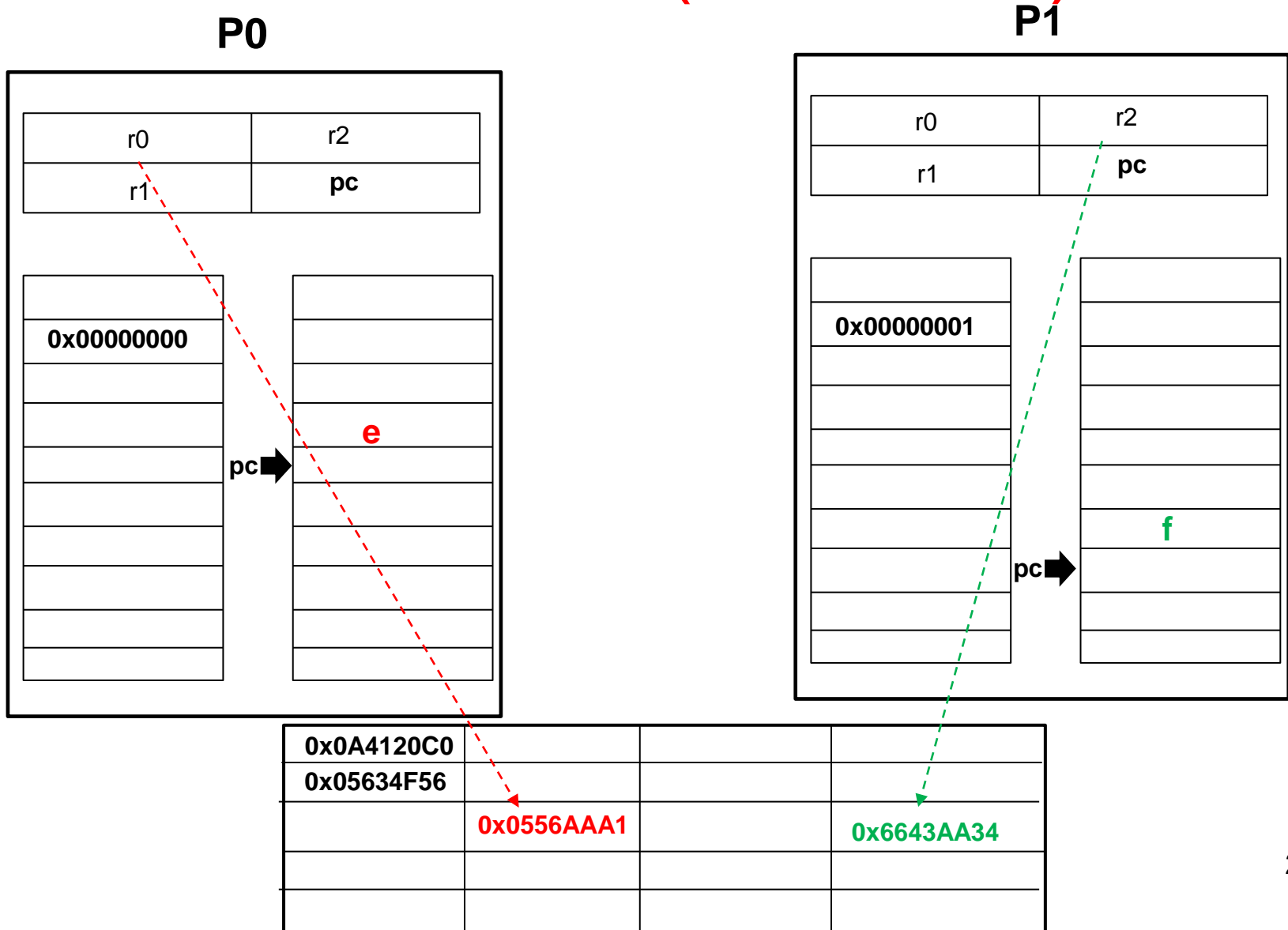
P1



0x05634F56	0x05634F56		

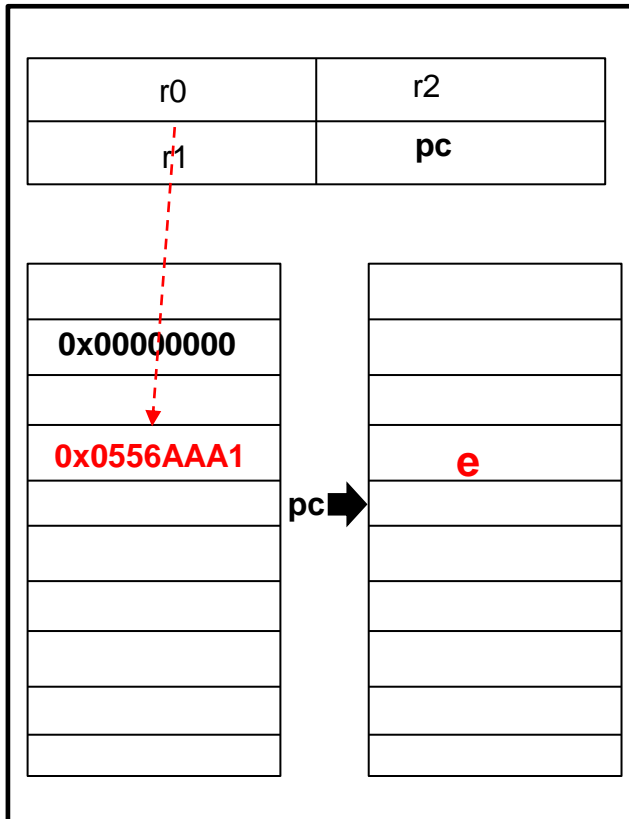


# Illustration accès registres différents (écritures)

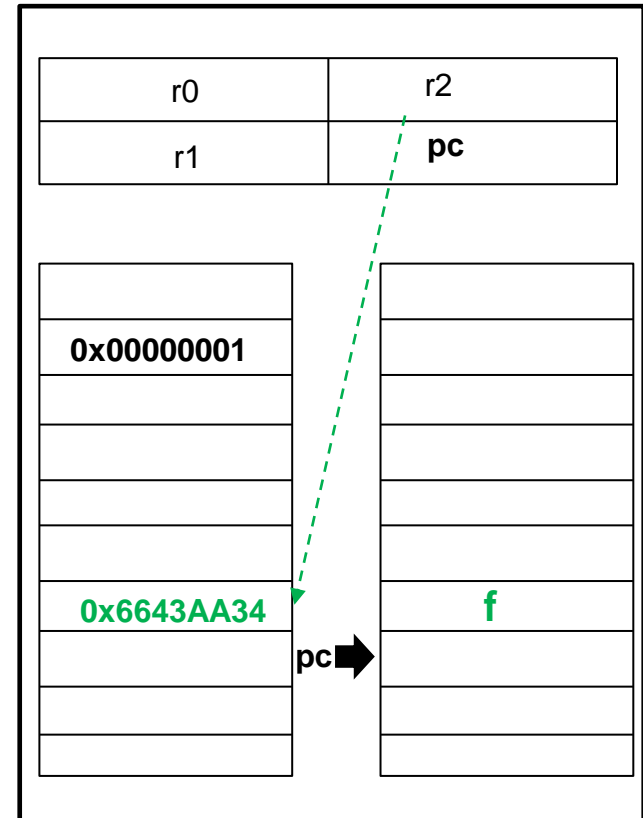


# Illustration accès registres différents (écritures)

**P0**



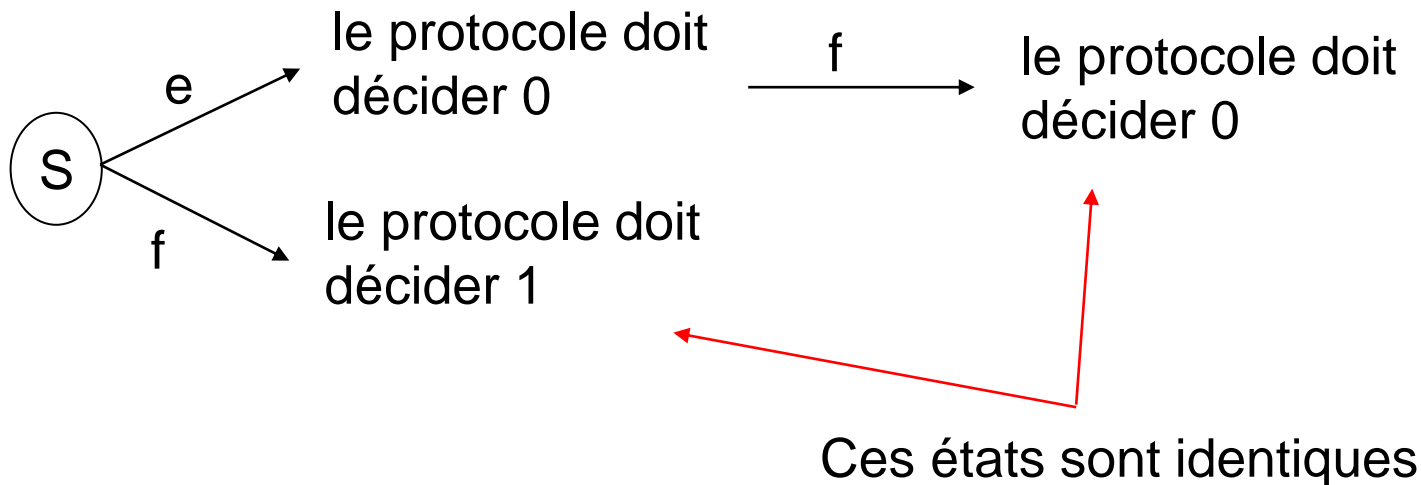
**P1**



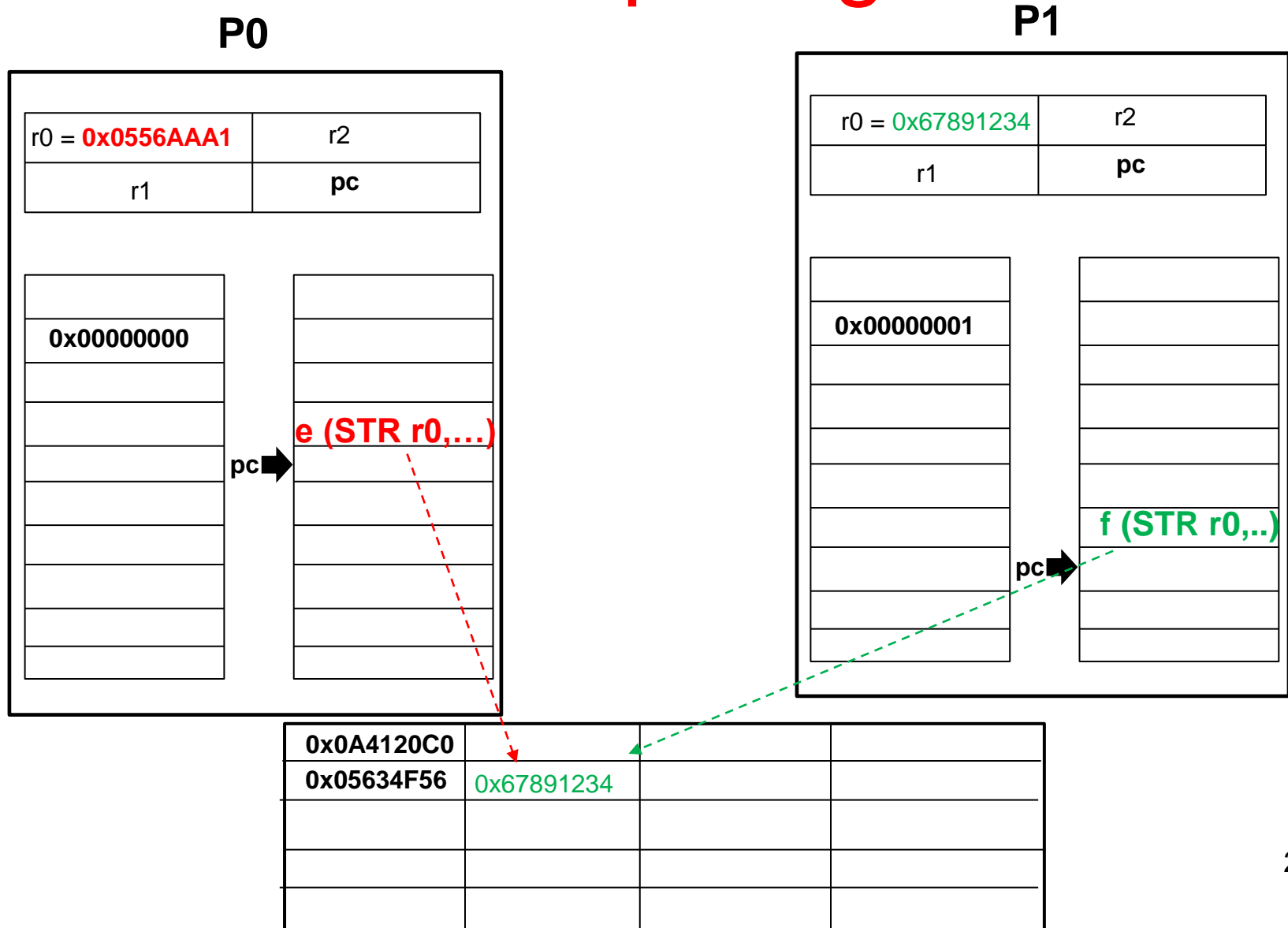
0x0A4120C0			
0x05634F56			

# Analyse

**Cas 3:**  $e$  et  $f$  sont des écritures à une même adresse mémoire. A nouveau les états du protocole en  $f(S)$  et  $f(e(S))$  sont identiques **pour P1** puisque l'écriture par P1 annule l'écriture par P0 dans  $f(e(S))$ . Donc si à partir de  $f(S)$  et de  $f(e(S))$  on exécute **seulement P1** alors P1 doit choisir 0 et 1 et c'est une contradiction puisqu'il ne distingue pas les états.

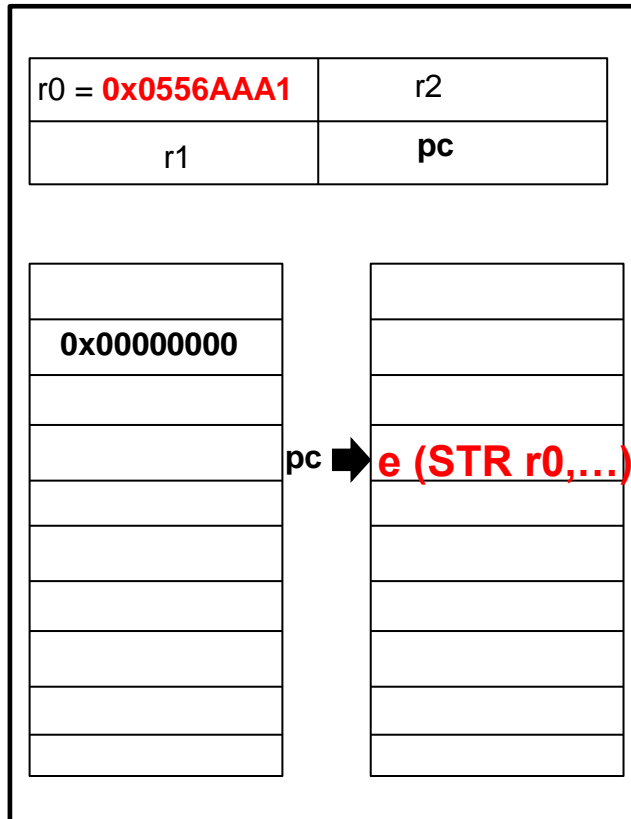


# Illustration accès même registre = mémoire partagée e->f

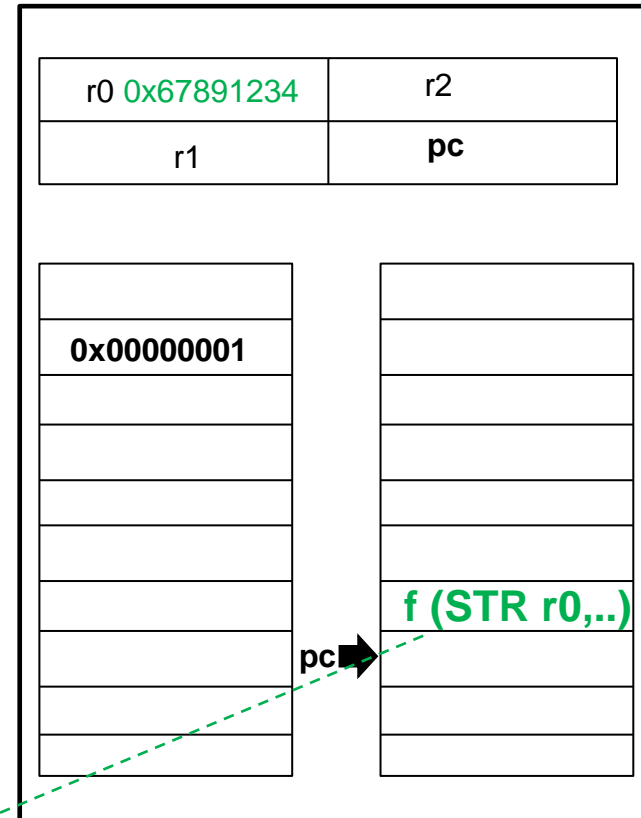


# Illustration accès même registre = mémoire partagée f seul

P0



P1



0x0A4120C0			
0x05634F56	0x67891234		

# Consensus – test-and-set

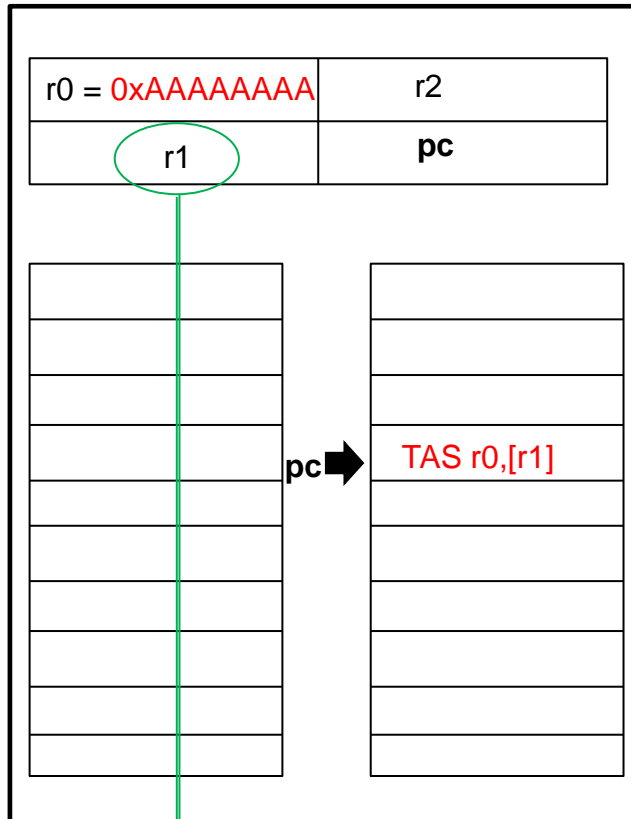
On rappelle l'implémentation de la fonction atomique test-and-set

```
public class TestAndSet {  
    int myValue = -1;  
  
    public synchronized int testAndSet(int newValue) {  
        int oldValue = myValue;  
        myValue = newValue;  
        return oldValue;  
    }  
}
```

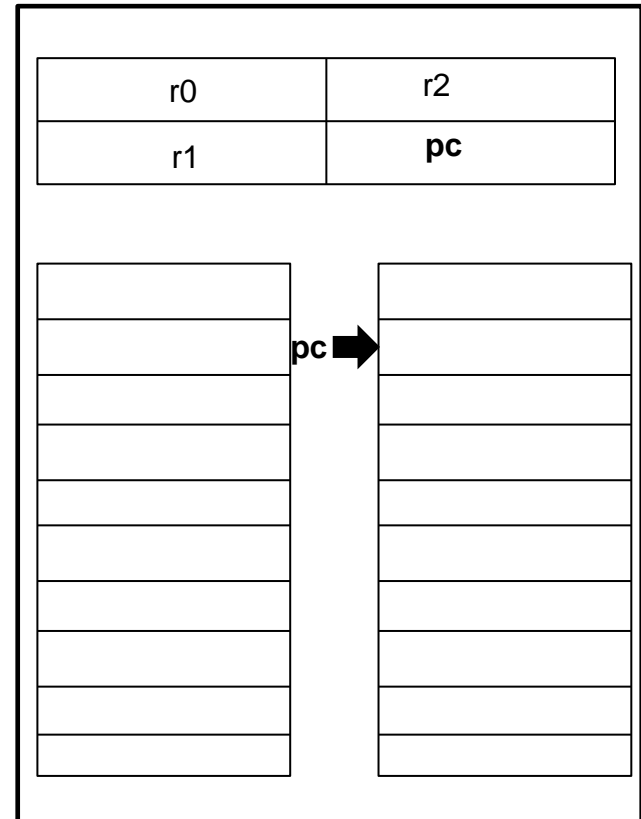
Cette primitive permet de résoudre le problème du consensus pour 2 processus

# TAS - avant

P0



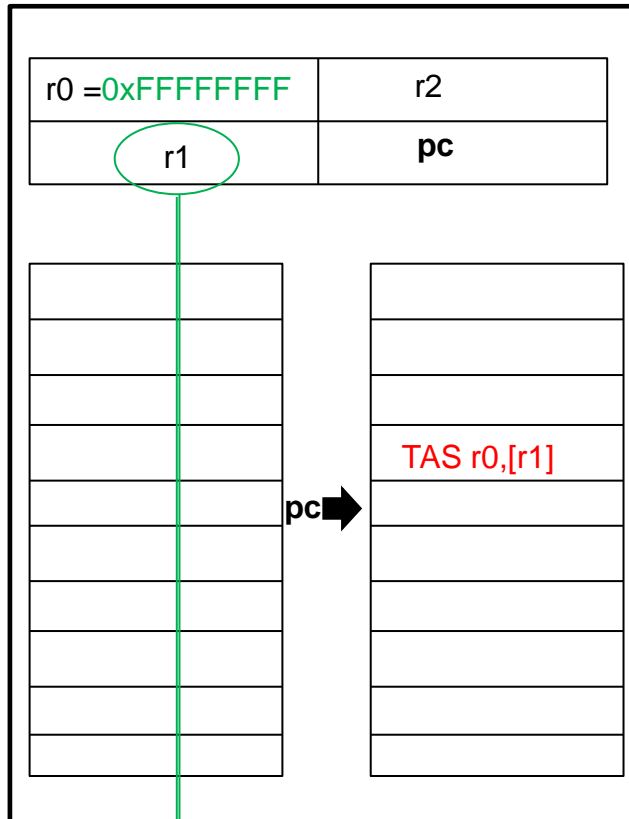
P1



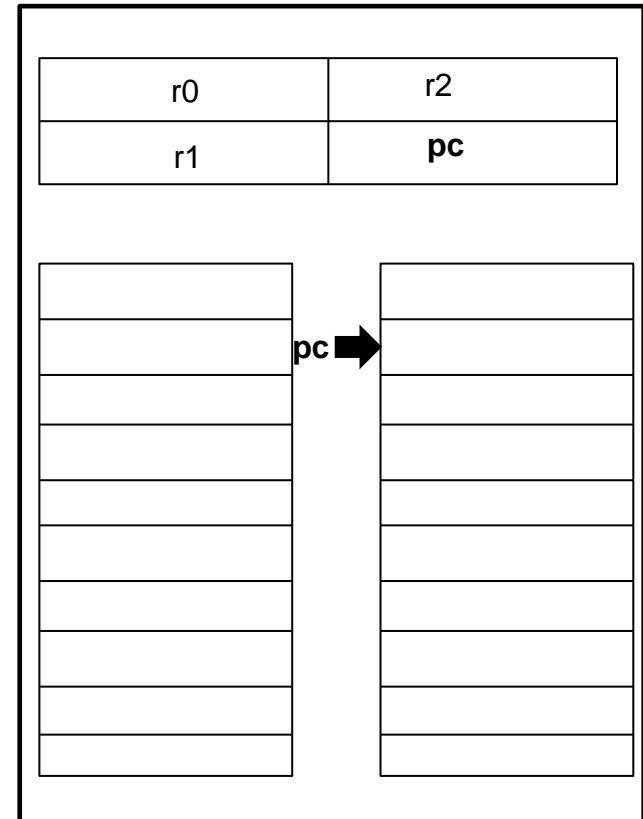
0x0AA4120C			
0x123ACB45			
0xFFFFFFFF			

# TAS - après

**P0**



**P1**



0x0AA4120C			
0x123ACB45			
0xAAAAAAAA			



# Consensus – test-and-set

```
class TestAndSetConsensus implements Consensus {  
    TestAndSet x;  
    int proposed[] = {0, 0};  
    public void propose(int pid, int v) {  
        proposed[pid] = v;  
    }  
    public int decide(int pid) {  
        if (x.testAndSet(pid) == -1) return proposed[pid];  
        else return proposed[1-pid];  
    }  
}
```

# Consensus TAS impossibilité

On montre qu'on ne peut pas résoudre le problème du consensus pour trois processus avec la fonction testAndSet.

On a trois processus P0, P1, P2.

L'ordonnanceur exécute P0 et P1, alternativement une instruction atomique pour chaque thread. P2 n'est pas exécuté pendant cette première phase qui dure jusqu'à atteindre un état critique.

Comme le protocole est wait-free le système atteint un état critique S.

Si P0 s'exécute (instruction e) alors on décide 0

Si P1 s'exécute (instruction f) alors on décide 1

# Consensus TAS impossibilité

Les instructions e et f peuvent être:

- Lecture atomique mémoire privée ou une instruction interne
- Lecture atomique mémoire partagée
- Ecriture atomique mémoire privée
- Ecriture atomique mémoire partagée

Pour chacun des instructions e et f on peut distinguer

- E et f accèdent le même registre/adresse mémoire
- E et f accèdent des registres /adresses mémoires différents

On ajoute l'exécution d'une instruction TAS

- Instruction TAS

Si e et f sont des instructions TAS alors on distingue si ce sont

- TAS sur le même objet
- Tas sur deux objets distincts

# Conensus TAS impossibilité

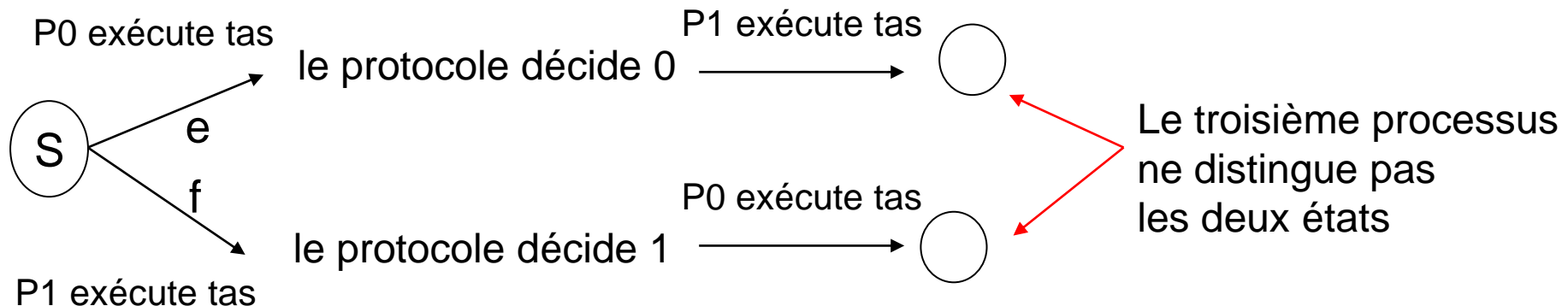
Pour les différents cas, si e et f ne sont pas des instructions TAS alors on procède comme pour la démonstration avec des registres atomiques avec deux processus P0 et P1 actifs pour montrer la contradiction.

On considère le cas où e et f sont des instructions TAS.

# Consensus – test-and-set

**Cas 1:** e et f sont des TAS sur des objets différents.

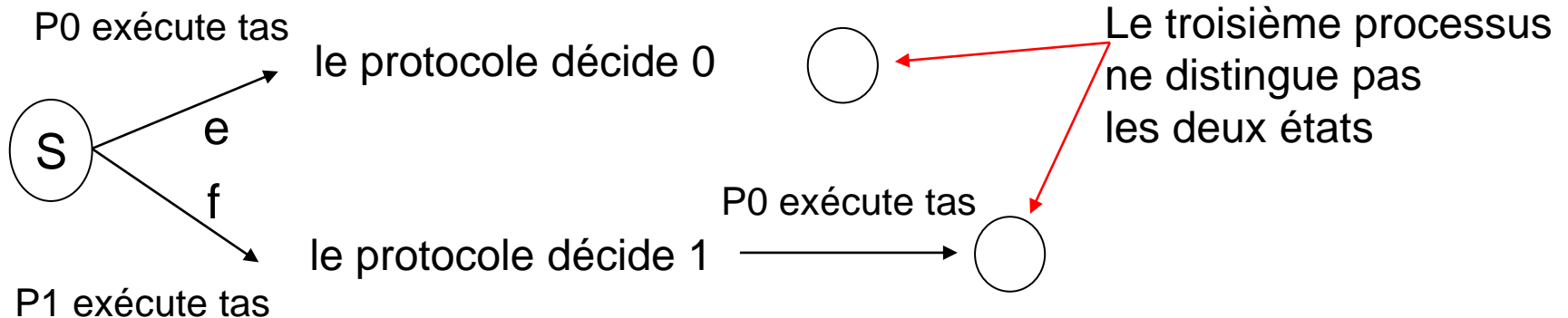
Dans ce cas les états  $e(f(S))$  et  $f(e(S))$  sont identiques mais le protocole doit choisir deux valeurs distinctes, c'est une contradiction.



# Consensus – test-and-set

**Cas 2:** e et f sont des TAS sur un même objet.

Dans ce cas les états  $e(f(S))$  et  $e(S)$  sont identiques **pour P2**. Donc depuis ces états si on exécute P2 sans arrêt il doit choisir la même valeur et c'est une contradiction.



# Consensus – test-and-set

**Corollaire:** on ne peut pas implémenter la fonction testAndSet avec des registres atomiques.

En effet, si c'était possible on pourrait résoudre le problème du consensus pour deux processus avec des registres atomiques.

# Consensus - Move

On considère l'opération atomique *move* qui copie la valeur d'un registre atomique dans un autre registre atomique

Cette primitive permet de résoudre le problème du consensus pour  $n$  processus avec les registres atomiques



# Consensus - Move

```
class MoveConsensus implements Consensus {  
    int proposed[] = {0, 0};  
    public void propose(int pid, int v) {  
        proposed[pid] = v; // atomique  
    }  
    public int decide(int pid) {  
        if (pid == 0) {  
            proposed[1]=proposed[0]; // atomique  
            return proposed[0];  
        }  
        else  
            move(proposed[1],proposed[0]);  
        return proposed[0];  
    }  
}
```

# Consensus - Move

Les registres `proposed[]` et l'opération `move` sont atomiques, on peut donc distinguer les situations où l'écriture dans les registres se produit avant ou après l'opération `move`.

Si `proposed[1]=proposed[0]`  $\longrightarrow$  `move(proposed[1],proposed[0])`  
alors la valeur initiale du processus 0 est choisie.

Sinon, c'est la valeur initiale du processus 1 qui est choisie.

Pour généraliser le protocole à  $n$  processus on utilise un tableau `r[0..n-1, 0..1]`. `r[i,0]=i` et `r[i,1]=i-1` initialement.

# Consensus - Move

Le processus  $i$  exécute le protocole suivant:

```
move(r[i,0],r[i,1])  
for(j=i+1; j< n; j++)  
    r[j,0] := j-1; // écriture atomique  
for(j=n-1; j>=0; j--)  
    if r[j,1]=j;  
    return j;
```

# Consensus - Move

Tous les processus décident la même valeur (agreement)

Supposons qu'un processus  $P_0$  décide la valeur  $j_0$  et un autre  $P_1$  la valeur  $j_1$  avec  $j_0 < j_1$ .

Pour décider  $j_0$ ,  $P_0$  doit lire  $r[j_1, 1] < j_1$ , c'est-à-dire que la lecture doit précéder le  $\text{move}(r[j_1, 0], r[j_1, 1])$ , de plus cette opération doit s'exécuter plus tard.

# Consensus - Move

Le numéro de  $P_0$  doit être supérieur à  $j_1$ , sinon il exécute  $r[j_1, 0] = j_1 - 1$ .

Comme il retourne  $j_0 < j_1 \leq \#P_0$ ,  $P_0$  doit nécessairement avoir  $r[\#P_0, 1] \neq \#P_0$ .

Donc un processus  $P_2$ ,  $\#P_2 < \#P_0$  a exécuté  $r[\#P_0, 0] = \#P_0 - 1$ . Avant d'exécuter cette opération, il a exécuté  $\text{move}(r[\#P_2, 0], [\#P_2, 1])$ .

Si  $\#P_2 > j_0$ , il existe un troisième processus  $P_3$ ,  $\#P_3 < \#P_2$  qui a exécuté  $r[\#P_2, 0] = \#P_2 - 1$  et ainsi de suite jusqu'à montrer qu'il existe un processus qui a exécuté  $r[j_1, 0] = j_1 - 1$ , une contradiction.

# Remarque

Le problème du consensus est utilisé pour comparer l'utilité des fonctions de synchronisation. Par exemple, on a vu qu'une fonction `testAndSet` ne peut pas être implémentée avec des registres atomiques.

Comme on peut résoudre le problème du consensus pour deux processus avec des registres atomiques, on dit qu'ils ont un numéro de consensus de 1.

La fonction `testAndSet` a un numéro de consensus de 2.

On sait qu'on ne peut pas implémenter une primitive avec un numéro de consensus  $n$  avec une primitive de numéro de consensus inférieur.

**On peut montrer l'inverse en toute généralité.** Par exemple, avec la fonction `testAndSet` on peut implémenter des registres atomiques.