

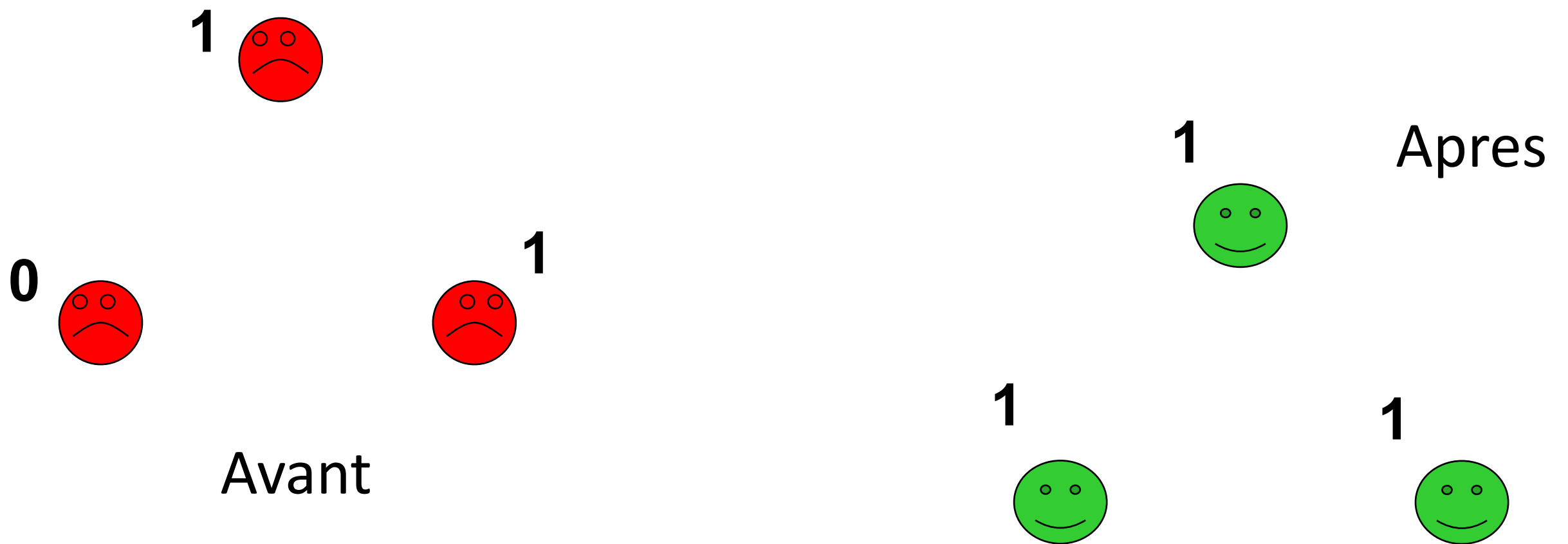
Consensus et objects

Modele

- N threads *asynchrones* (processus) p_0, \dots, p_{N-1} ($N \geq 2$)
- communication via des registres atomiques

Consensus

Processus *proposent* une valeur et doivent se mettre d'accord sur une valeur commune



Consensus comme Objects

- n -consensus objet : objet qui résoud le consensus entre n threads.
- spécification séquentielle
 - Méthode *propose*(v)
 - > propose la valeur v au consensus (v dans V un ensemble de valeurs initiales)
 - retourne la première valeur proposée
 - Le retour de *propose*(v) est la décision du processus
 - Un appel au plus par processus de *propose*()

Impossibilité du consensus wait-free [FLP85,LA87]

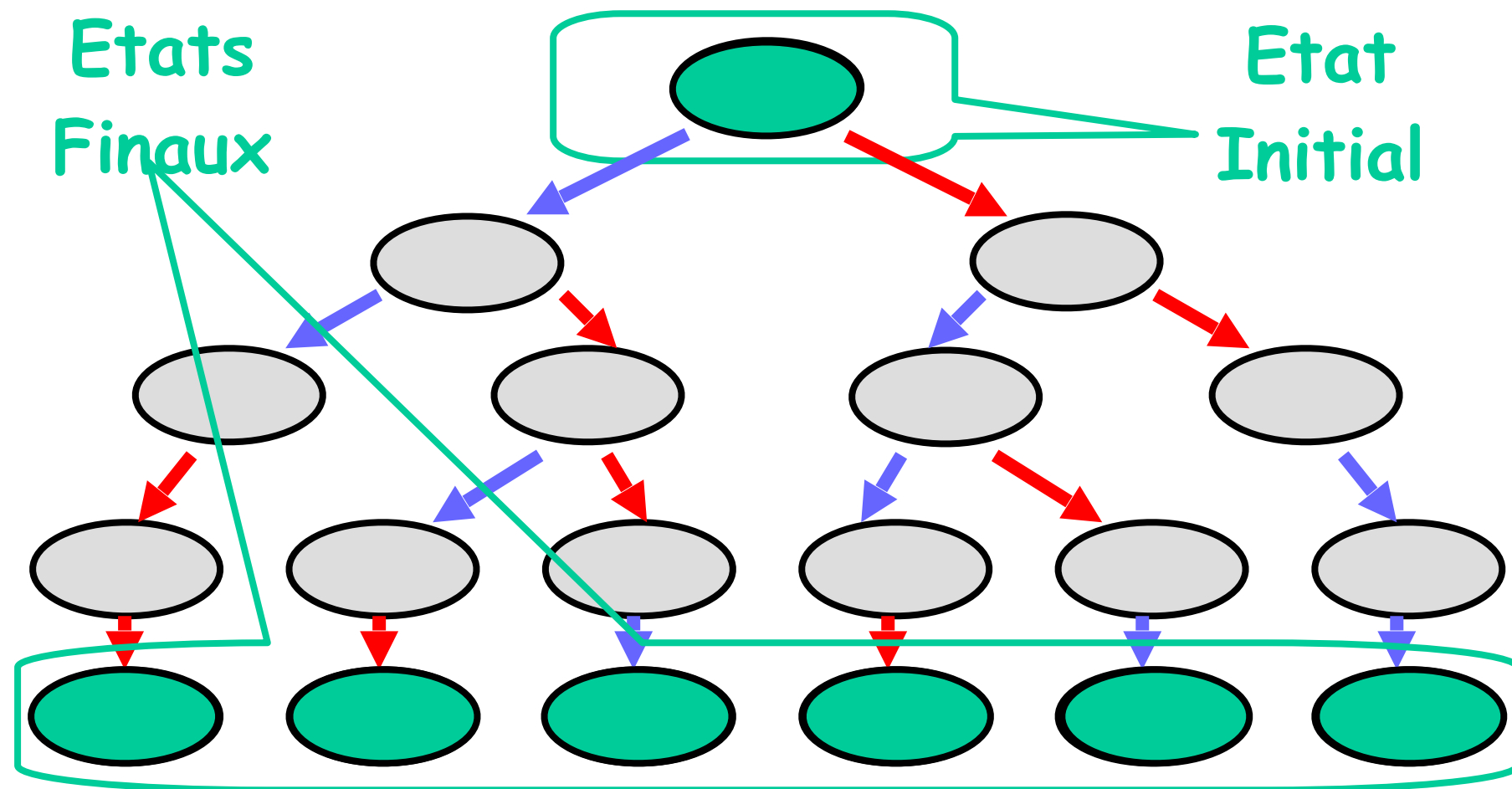
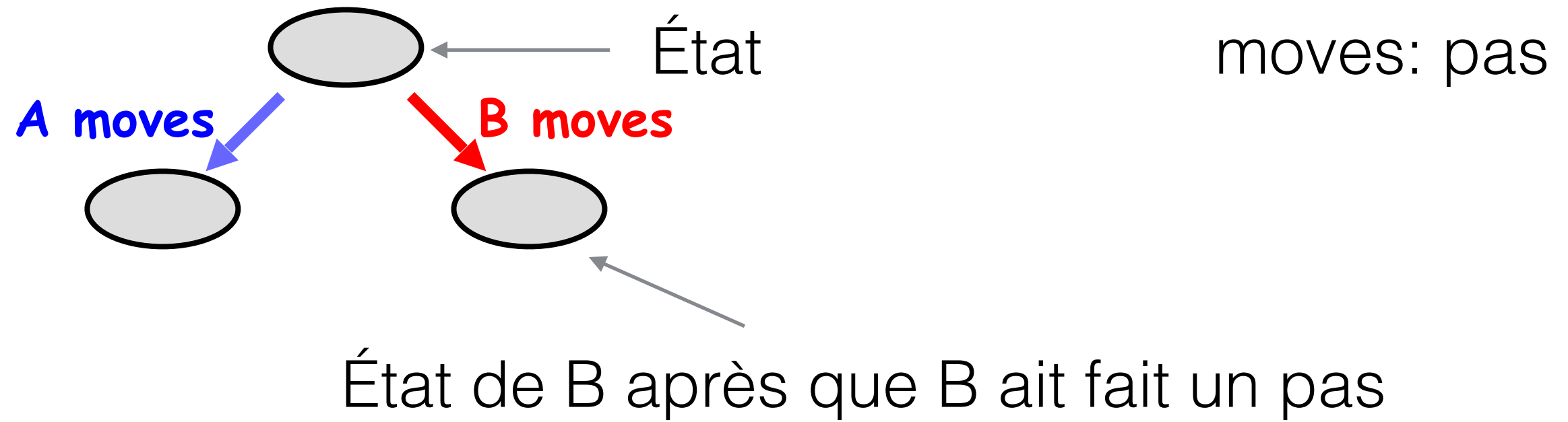
Theorem

Il n'y a pas d'implementation wait-free du n -consensus avec des registres pour $n \geq 2$

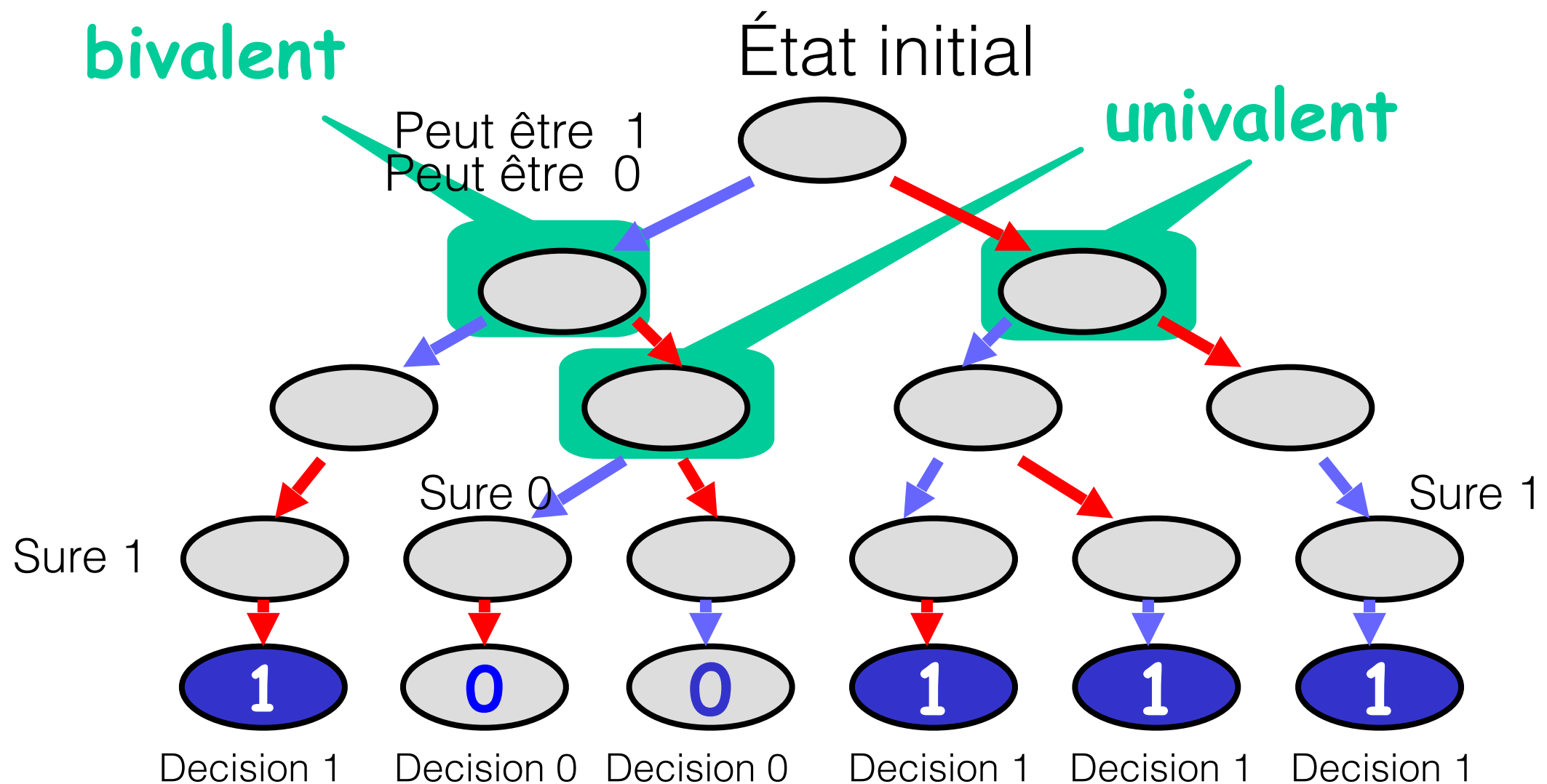
Preuve ensuite...

Modèle ...

- *Modélisation d'une exécution*
- 2 processus (A et B) déterministe
- consensus binaire (valeur initiale 0 ou 1)
 - «Pas» d'un processus (ici lecture ou écriture d'un registre)
 - «Configuration» (état de la mémoire, état du processus...)
 - « *Execution* » de l'algorithme: séquences de configurations + pas
 - *Ordonnancement* : séquence d'identifiant de processus (« wait-free »: à chaque fois n'importe quel processus peut faire un pas)
 - une configuration initiale et un ordonnancement définissent une exécution



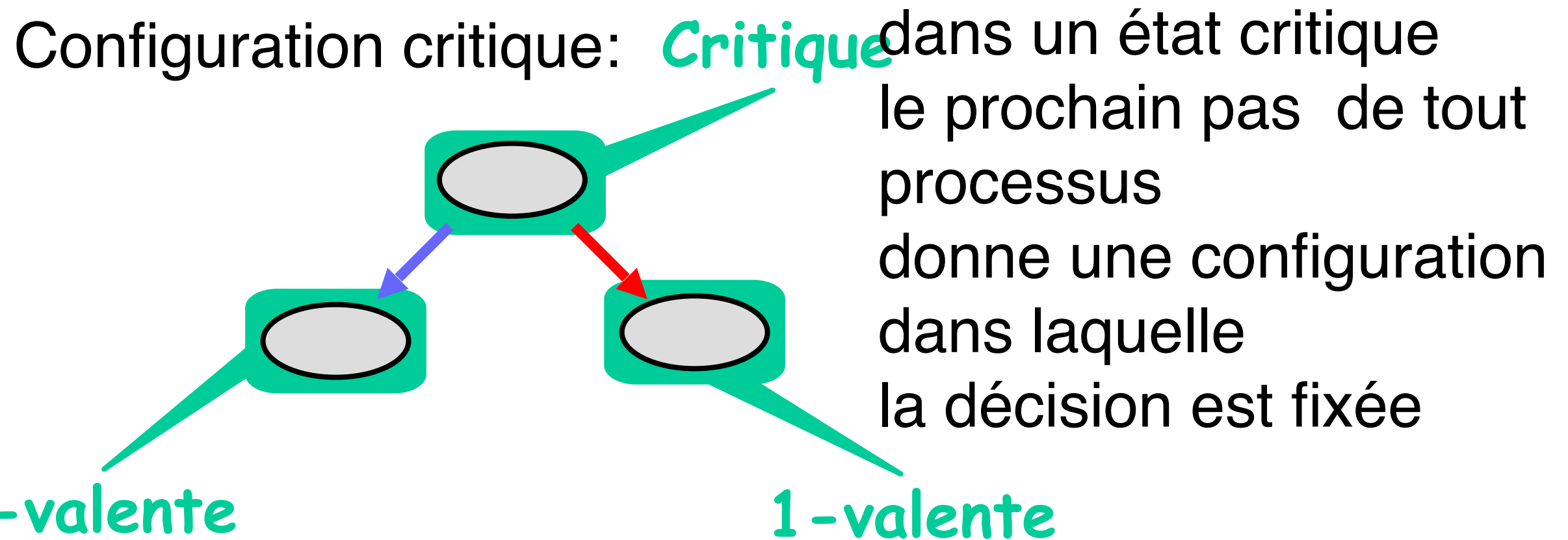
Consensus



bivalent: à partir de là 0 et 1 sont possibles
0-valent: à partir de là seulement 0
univalent: à partir de là seulement une valeur
1-valent: à partir de là seulement 1

Modèle

- Les exécutions wait-free forment un arbre (une forêt) (nœud: configurations; arête: pas) représentant toutes les exécutions possibles à partir d'une configuration initiale (à partir de toutes les configurations initiales)
- Valences:
 - Configurations *Bivalentes*:
 - Sortie non fixée
 - Configurations *Univalente*
 - Sortie fixée
 - (Mais peut être pas encore « connu »)
 - Configurations *1-Valente* ou *0-Valente* = univalente pour 1 ou 0



Propriétés:

Il existe une configuration initiale bivalente.

S'il y a un protocole de consensus wait-free, alors il y a un état critique

Il existe une configuration initiale bivalente

Une configuration initiale est bivalente:

considérons A avec l'état initial 0 et B avec l'état initial 1

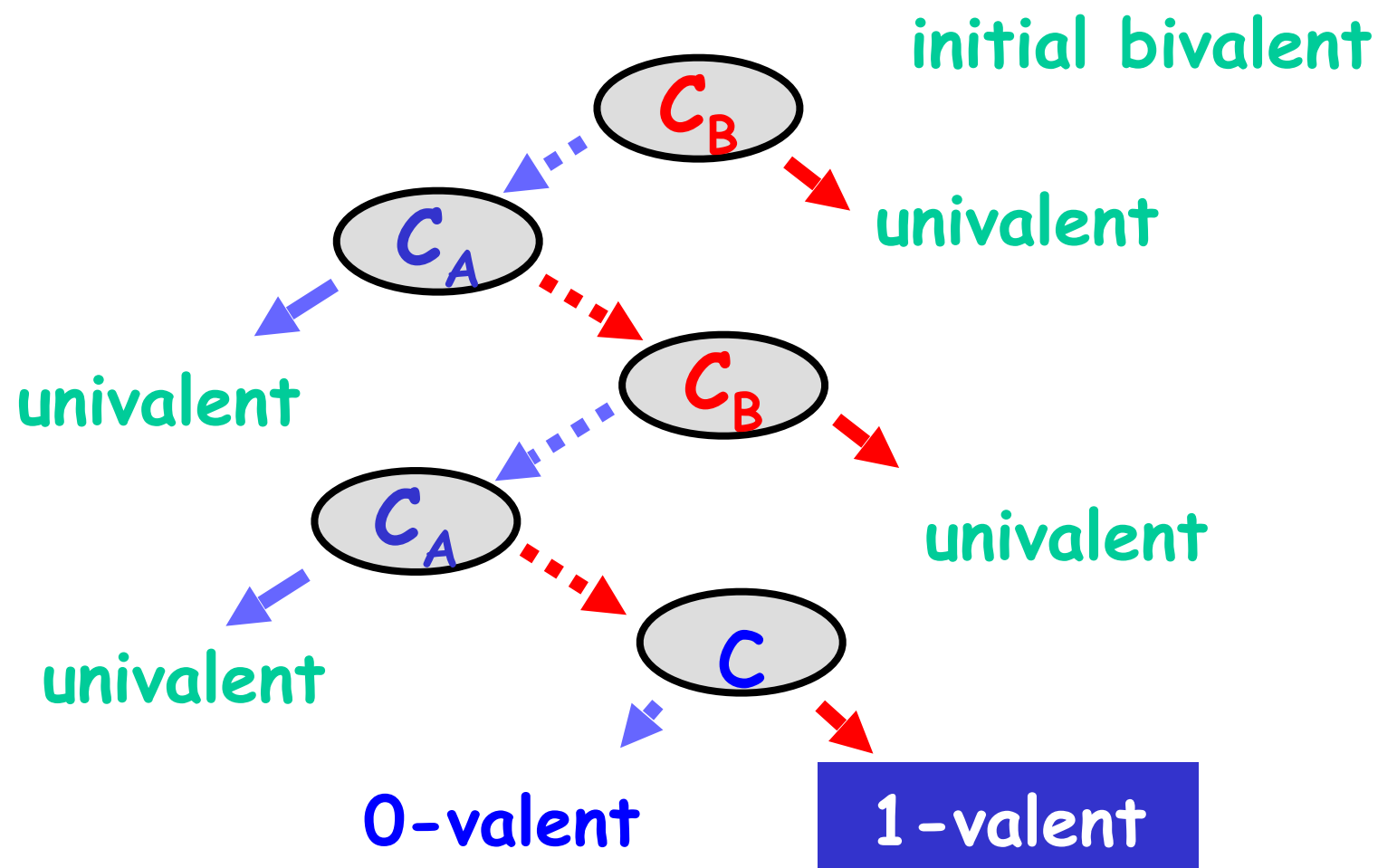
alors lorsque A s'exécute seul (wait-free) , il décide 0 car pour A il est impossible de distinguer la configuration où B a la valeur initiale 0, et la décision 0 est la seule autorisée.

alors lorsque B s'exécute seul, il décide 1 car pour B il est indiscernable que A ait la valeur initiale 1

=> la configuration est bivalente.

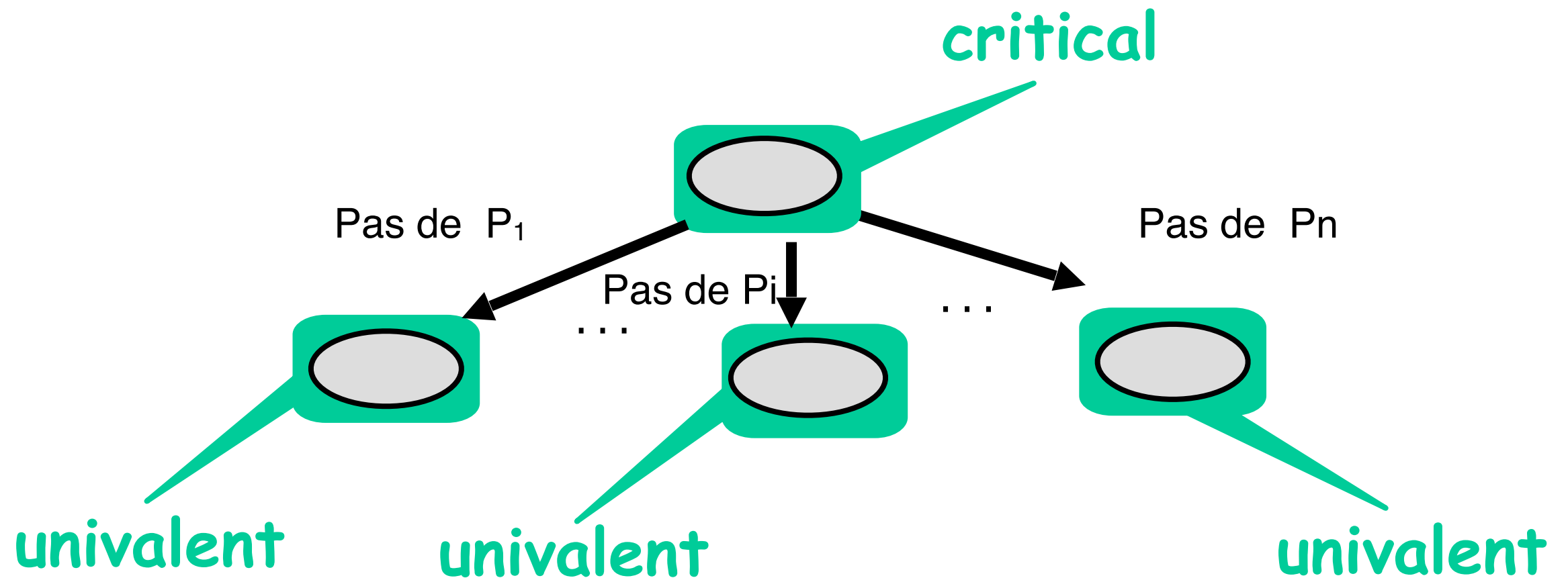
Configuration critique

Il y a un état critique



- Partir d'une configuration initiale bivalente
- Le protocole atteint une configuration critique
 - Sinon on resterait dans une configuration bivalente pour toujours
 - et le protocole ne serait pas wait-free

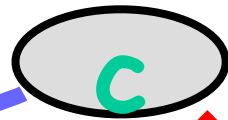
Donc: pour chaque protocole de consensus, il existe une étape critique



Impossibilité...

- en supposant qu'il existe un algorithme de consensus wait free:
 - il y a une configuration critique
 - considérer cette configuration et les pas des processus à partir de cette configuration

A s'exécute seul,
A finira par
decider 0 (wait
free)



B reads x

read

A s'exécute seul, A
finira par decider 1
(wait free)

Les configurations
sont les mêmes du
point de vue de A

contradiction



B writes x

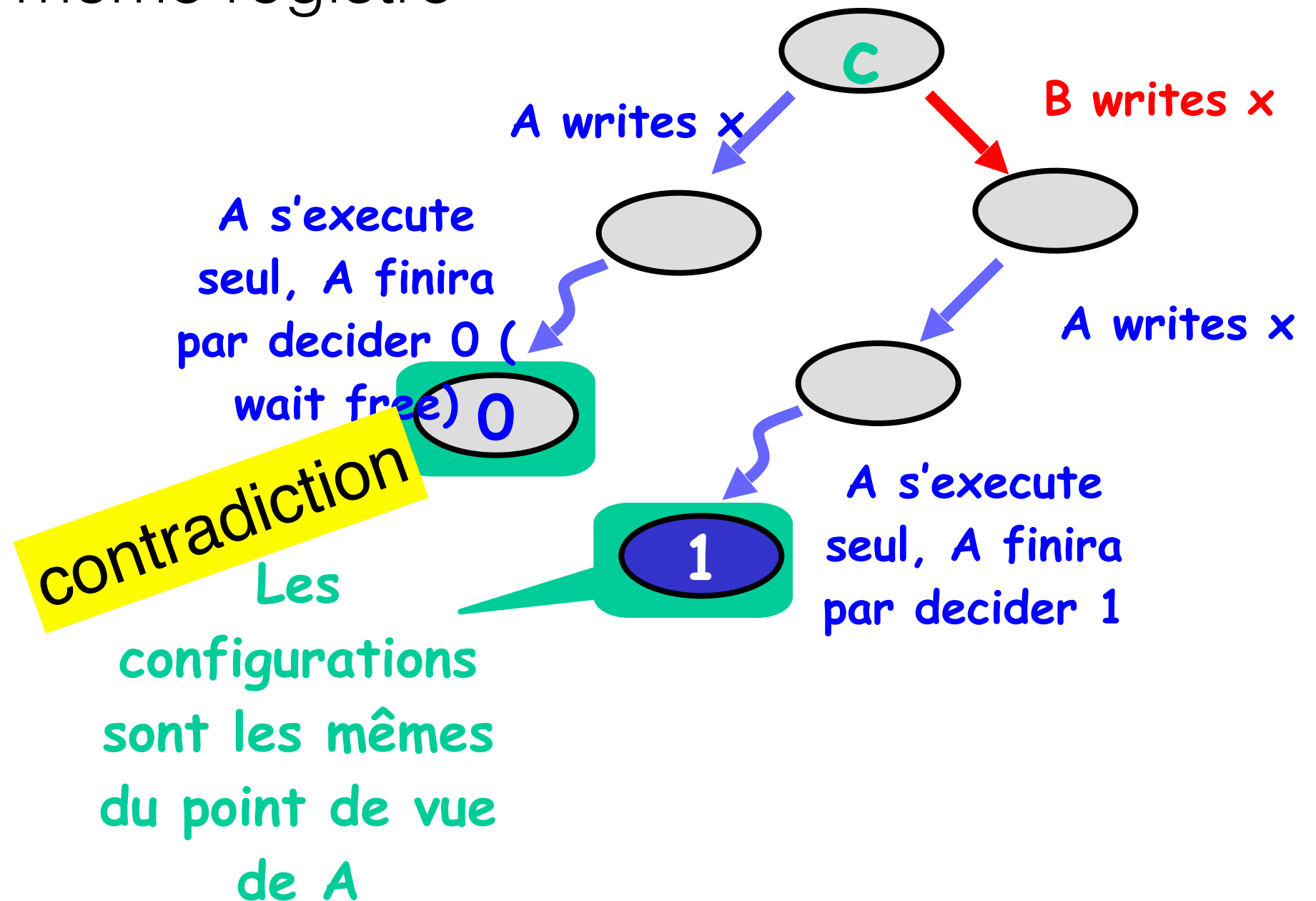
write sur différents
registres

A writes y

contradiction

Les configurations
sont les mêmes

Ecriture sur le même registre



Impossible d'obtenir du consensus avec des registres

Comparer les objets atomiques

- Objets atomiques: structures de données concurrentes (ou primitives matérielles)
- Quelle est la puissance des objets comme Test & Set, Compare & Swap, Pile, File...?
- «Comparer» les objets avec l'objet consensus

n-consensus

- n-consensus : consensus entre n processus
- n-consensus mesure la puissance des objets: numéro de consensus
- THEOREME: Il est impossible d'obtenir un n -consensus avec un nombre quelconque de $(n-1)$ -consensus et de registres
- Hiérarchie des objets: un objet peut être assez fort pour implémenter le y -consensus (avec des registres) et trop faible pour implémenter le $(y+1)$ -consensus

- THEOREME: Il est impossible d'obtenir un n -consensus avec un nombre quelconque de $(n-1)$ -consensus et de registres
- Preuve: par contradiction. Supposons qu'il existe un protocole.
 - il y a une configuration initiale bivalente
 - considérer cette configuration et les pas des processus dans cette configuration
 - Etudiez les cas....

- cas à considérer:
 - lecture / écriture sur les registres: ok
 - si les opérations commutent: (c'est le cas pour accès a consensus et accès à registre—accès à 2 consensus différents)
ok
 - alors les seules opérations à considérer sont des appels au même objet consensus

Numéro de Consensus

- Une classe C d'objets résout **n -consensus** s'il existe un protocole de consensus pour n processus utilisant un nombre quelconque d'objets de classe C et des registres atomiques
- Le **numéro de consensus** d'une classe C : $(h(C))$ est le plus grand n pour lequel cette classe résout le n -consensus.
- Définir une hiérarchie d'objets:
 - Si l'on peut implémenter un objet de classe C à partir d'objets de classe D (et de registres) alors $h(C) \leq h(D)$
 - si $h(C) > h(D)$ il n'y a pas d'implémentation d'objets de C avec des objets de D .

- Implication
 - Si X a pour numéro de consensus c
 - Et Y a pour numéro de consensus $d < c$
 - Alors il n'y a aucun moyen de construire une implémentation wait-free de X par Y
- $h(\text{registre}) = 1$

Fifo \rightarrow 2 consensus

- Fifo: enq(v) et deq()
- Algo pour 2 processus:

Fifo q: initialisé à Red, Green

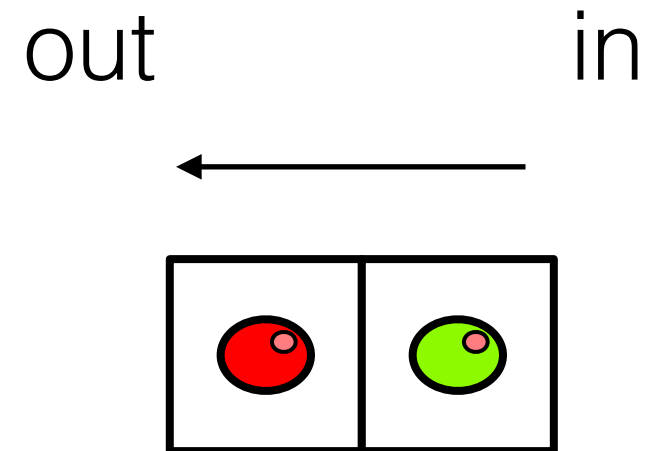
process i : code propose(v_i)

 proposed[i] := v_i

 V := q.deq()

 if V=Red then return v_i

 else return proposed[1-i]



In Java...

```
public interface Consensus {  
    Object proposeC(object value);  
}  
  
abstract class ConsensusProtocol<T>  
    implements Consensus {  
    protected T[] proposed = new T[N]  
    protected void propose(T value) {  
        proposed[ThreadID.get()] = value;  
    }  
    abstract public T decide(T value);  
}  
  
public class QueueConsensus  
    extends ConsensusProtocol {  
    private Queue queue;  
    public QueueConsensus() {  
        queue = new Queue();  
        queue.enq(Ball.RED);  
        queue.enq(Ball.GREEN);  
    }  
    public Object proposeC(object value) {  
        propose(value);  
        Ball ball = this.queue.deq();  
        if (ball == Ball.RED)  
            return proposed[i];  
        else  
            return proposed[1-i];  
    }  
}
```

- ★ un seul processus obtient le **RED** c'est le gagnant
- ★ le gagnant décide de sa valeur
- ★ le perdant peut trouver la valeur du gagnant dans le tableau

Théorème: $h(\text{Queue}) \geq 2$

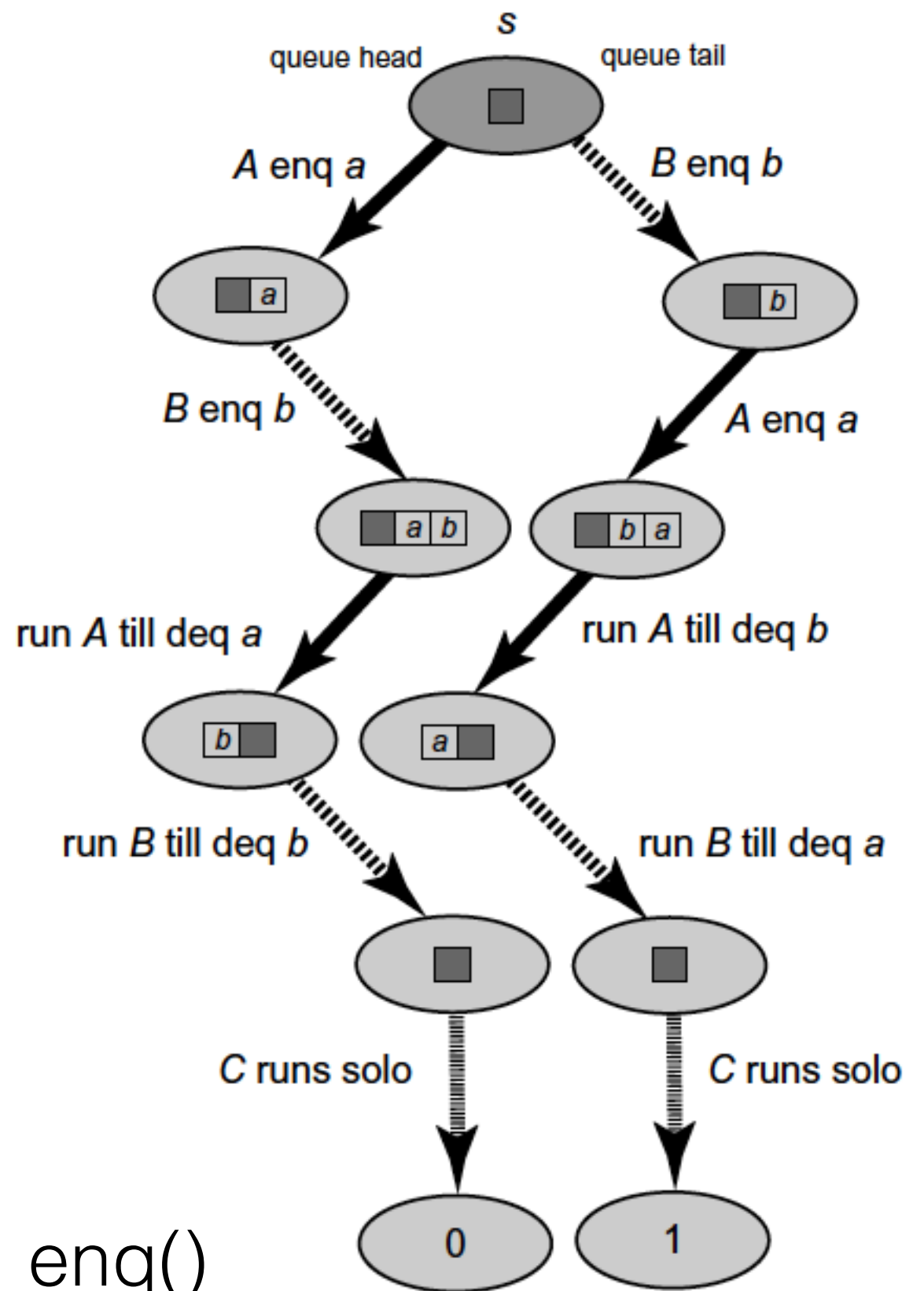
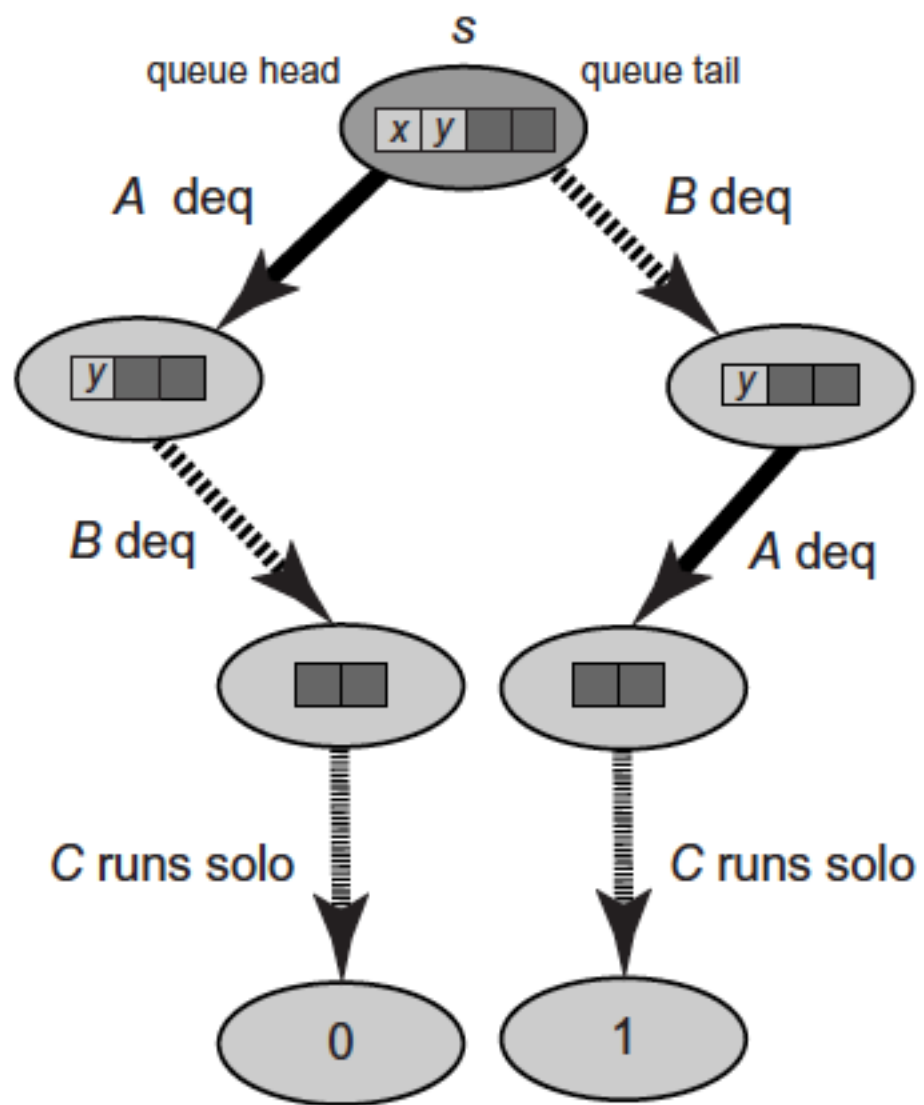
Il n'y a pas d'implementation wait-free d'une file par des registres

Réciproquement

Preuve $h(\text{Queue}) < 3$

Preuve

- supposons le contraire: des processus A, B, C et un protocole de consensus
- il y a une configuration critique:
- cas à considérer:
 - lecture / écriture sur les registres: ok
 - si les opérations commutent: ok
 - alors les seules les opérations à considérer sont la même opération (deq ou enq ()) sur la même file d'attente



deq()

Théorème: $h(\text{queue})=2$

enq()

Grand Challenge précédent

- Snapshot signifie
 - Écrire chaque élément du tableau
 - Lire plusieurs éléments de manière atomiqueEt que se passe-t-il si
 - Ecrire plusieurs éléments de manière atomique
→ m
 - Lire plusieurs éléments de manière atomique → n
- → affectation multiple (m, n)

Théorème de l'affectation multiple

- On ne peut pas réaliser l'affectation multiple avec des registres atomiques
 - Single location write/multi location read OK
 - Multi location write/multi location read impossible

Preuve

- On considère une affectation $(2,3)$
 - On montre qu'avec cet objet et des registres on peut faire du consensus pour 2 processus
 - Le consensus pour 2 processus est impossible
- Donc:
 - On ne peut pas réaliser une $(2,3)$ affectation wait free

Preuve

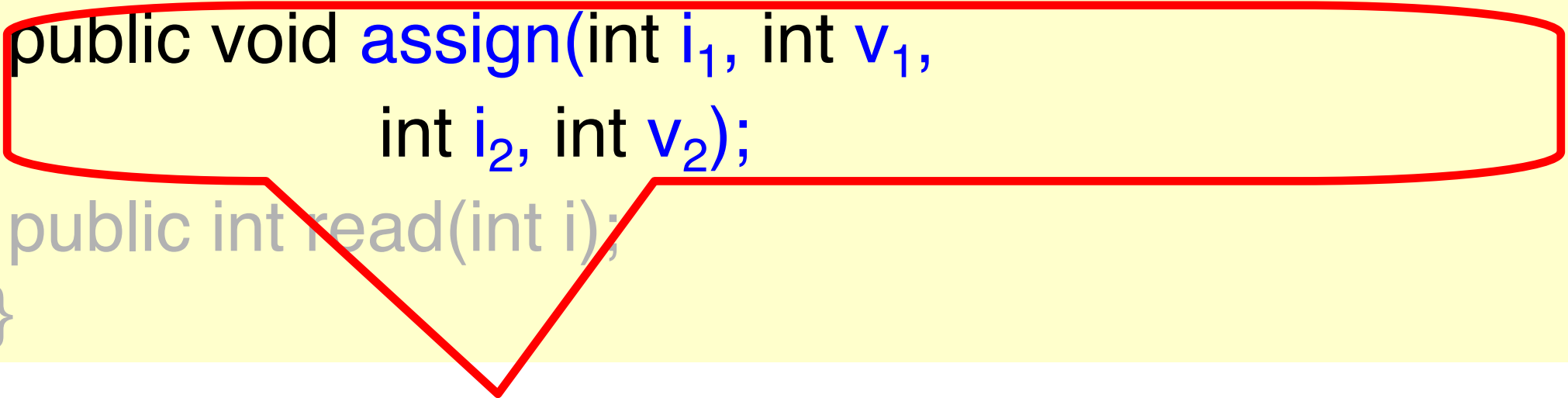
- Un tableau de 3 éléments
- A écrit de manière atomique dans les cases 0 et 1
- B écrit de manière atomique dans les cases 1 et 2
- Les threads peuvent faire un scan de n'importe quel ensemble de location

Affectation double Interface

```
interface Assign2 {  
    public void assign(int  $i_1$ , int  $v_1$ ,  
                      int  $i_2$ , int  $v_2$ );  
    public int read(int i);  
}
```

Double Assignment Interface

```
interface Assign2 {  
    public void assign(int i1, int v1,  
                      int i2, int v2);  
    public int read(int i);  
}
```



Atomically assign
value[i₁] = v₁
value[i₂] = v₂

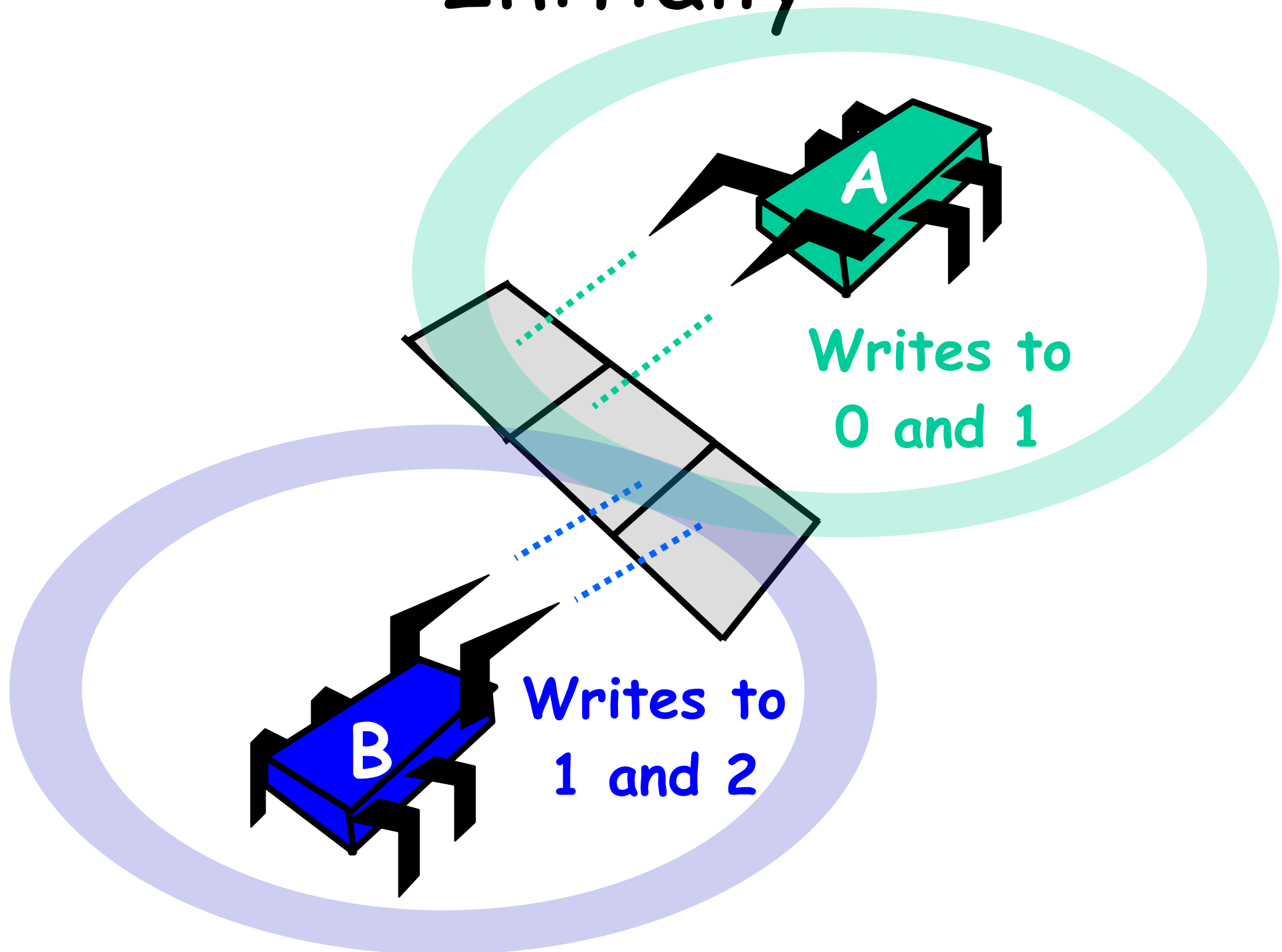
Double Assignment Interface

```
interface Assign2 {  
    public void assign(int i1, int v1,  
                      int i2, int v2);
```

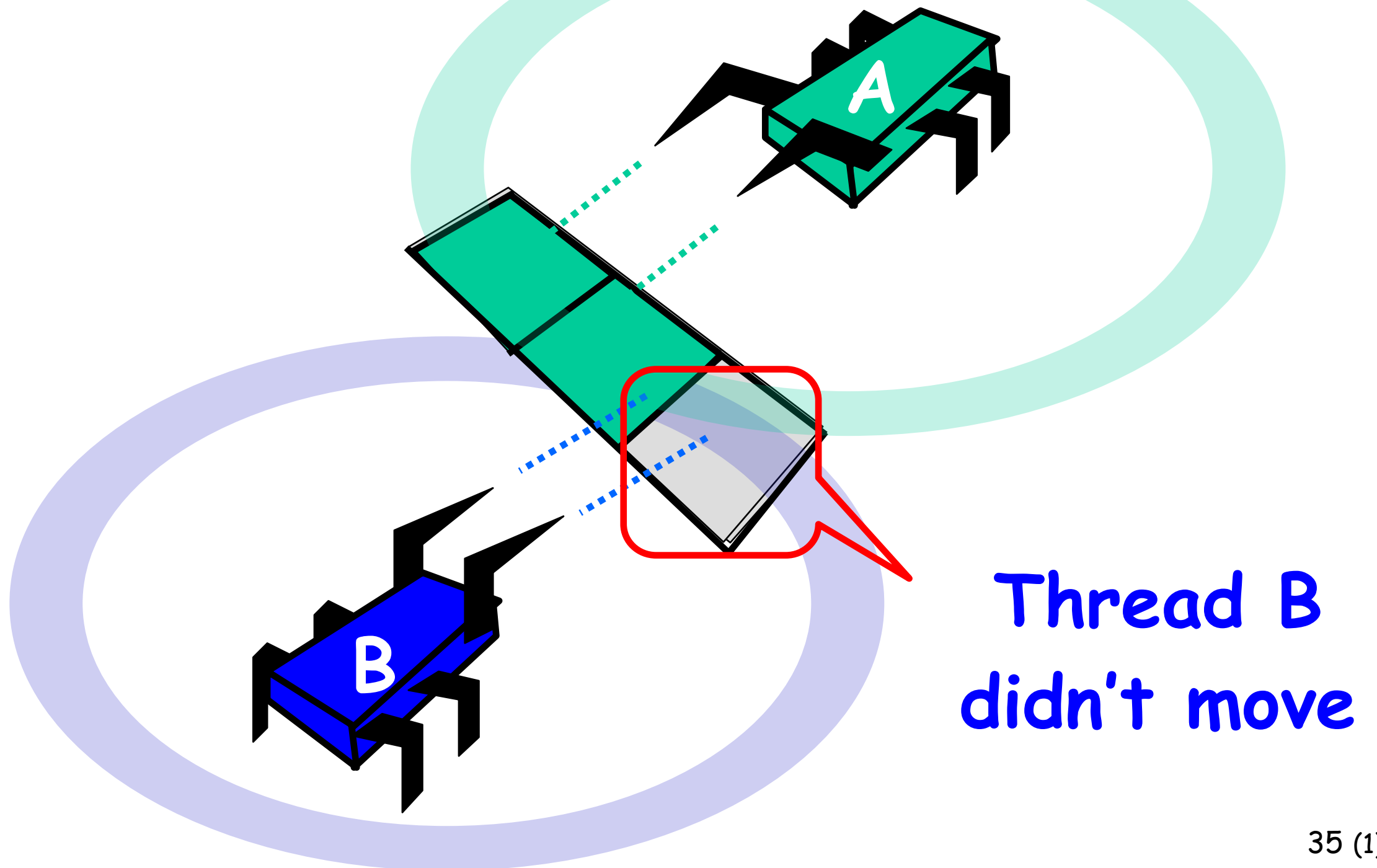
```
    public int read(int i);  
}
```

Return i-th value

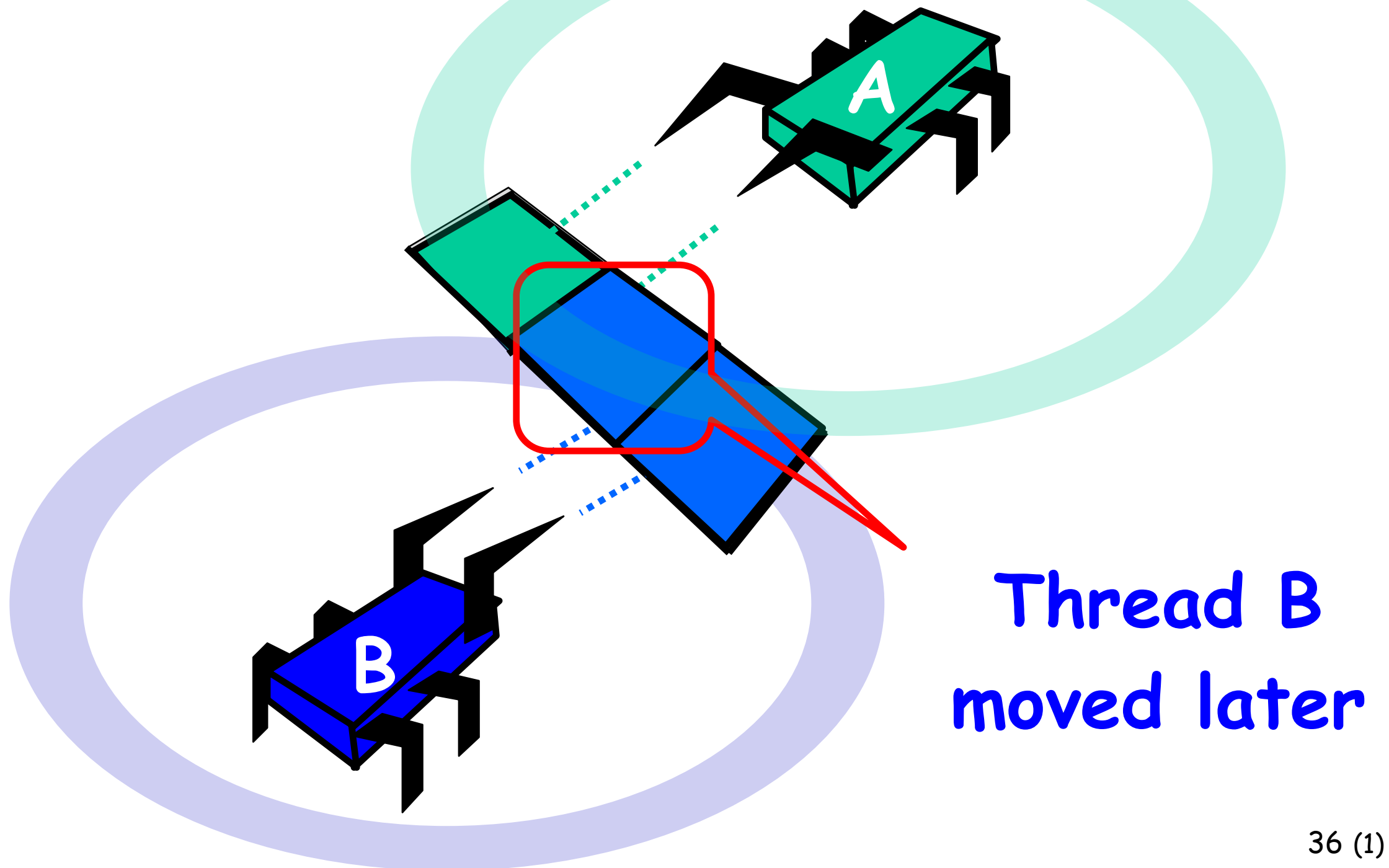
Initially



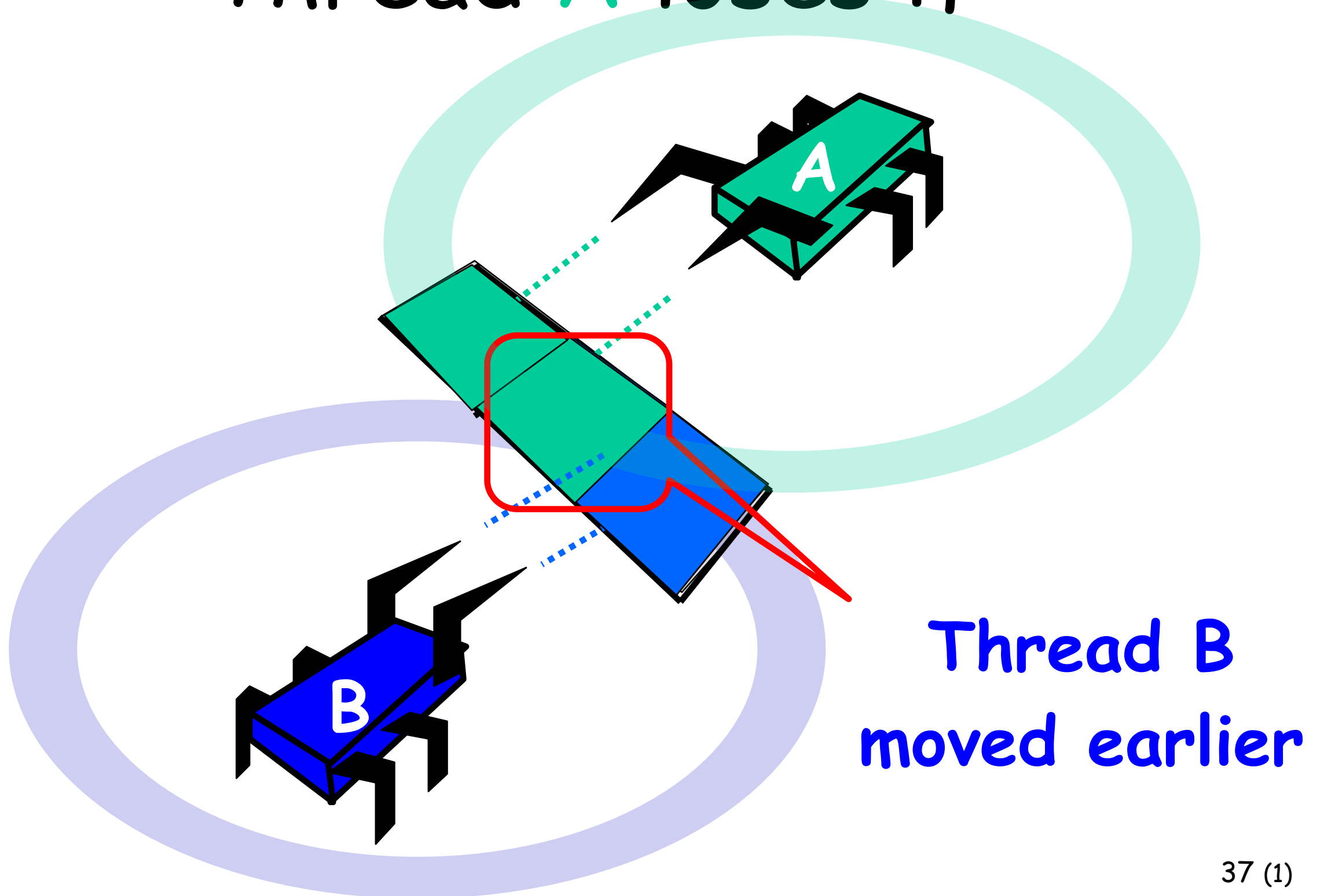
Thread **A** wins if



Thread *A* wins if



Thread *A* loses if



Multi-Consensus Code

```
class MultiConsensus extends ...{
    Assign2 a = new Assign2(3, EMPTY);
    public Object decide(object value) {
        proposed[i]=value;
        a.assign(i, i, i+1, i);
        int other = a.read((i+2) % 3);
        if (other==EMPTY||other==a.read(1))
            return proposed[i];
        else
            return proposed[1-i];
    }
}
```

Multi-Consensus Code

```
class MultiConsensus extends ...{  
    Assign2 a = new Assign2(3, EMPTY);  
    public Object decide(object value) {  
        proposed[i]=value;  
        a.assign(i, i, i+1, i);  
        int other = a.read((i+2) % 3);  
        if (other==EMPTY||other==a.read(1))  
            return proposed[i];  
        else  
            return proposed[1-i];  
    }  
}
```

Multi-Consensus Code

```
class MultiConsensus extends ... {  
    Assign2 a = new Assign2(3, EMPTY);  
    public Object decide(object value) {  
        proposed[i]=value;  
        a.assign(i, i, i+1, i);  
        int other = a.read((i+2) % 3);  
        if (other==EMPTY||other==a.read(1))  
            return proposed[i];  
        else  
            return proposed[1-i];  
    }  
}
```

**3 cases init à
EMPTY**

Multi-Consensus Code

```
class MultiConsensus extends ...{  
    Assign2 a = new Assign2(3, EMPTY);  
    public Object decide(object value) {  
        proposed[i]=value;  
a.assign(i, i, i+1, i);  
        int other = a.read((i+2) % 3);  
        if (other==EMPTY||other==a.read(1))  
            return proposed[i];  
        else  
            return proposed[1-i];  
    }  
}
```

**Affecte id 0 aux
cases 0,1 (ou id 1 to
cases 1,2)**

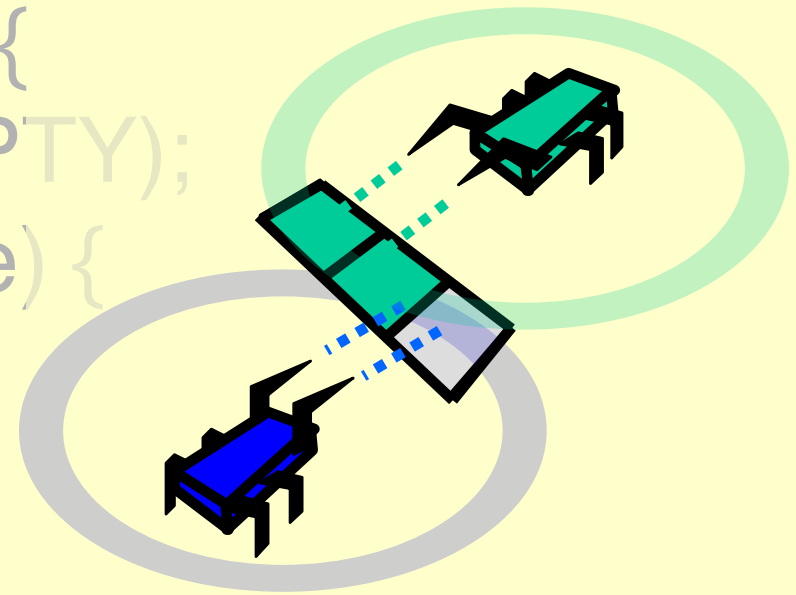
Multi-Consensus Code

```
class MultiConsensus extends ...{  
    Assign2 a = new Assign2(3, EMPTY);  
    public Object decide(object value) {  
        proposed[i]=value;  
        a.assign(i, i, i+1, i);  
        int other = a.read((i+2) % 3);  
        if (other==EMPTY||other==a.read(1))  
            return proposed[i];  
        else  
            return proposed[1-i];  
    }  
}
```

**Lit le registre non
affecté à la thread**

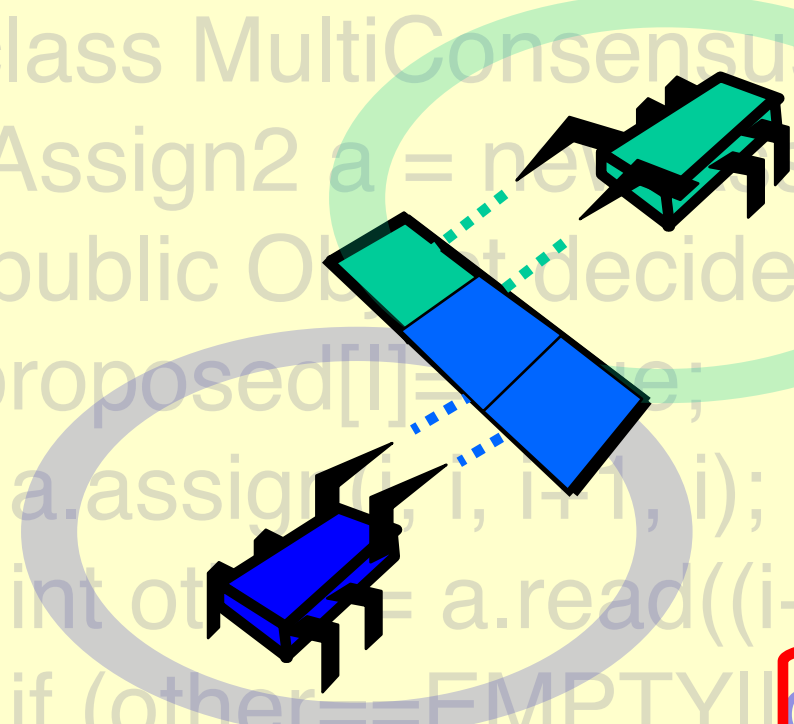
Multi-Consensus Code

```
class MultiConsensus extends ...{  
    Assign2 a = new Assign2(3, EMPTY);  
    public Object decide(object value) {  
        proposed[i]=value;  
        a.assign(i, i, i+1, i);  
        int other = a.read((i+2) % 3);  
        if (other==EMPTY || other==a.read(1))  
            return proposed[i];  
        else  
            return proposed[1-i];  
    }  
}
```



**L'autre thread n'a
pas bougé, je gagne**

Multi-Consensus Code



```
class MultiConsensus extends ...{
  Assign2 a = new Assign2(3, EMPTY);
  public Object decide(object value) {
    proposed[i] = value;
    a.assign(i, i+1, i);
    int other = a.read((i+2) % 3);
    if (other == EMPTY || other == a.read(1))
      return proposed[i];
    else
      return proposed[1-i];
  }
}
```

**L'autre thread a
bougé en second. Je
gagne**

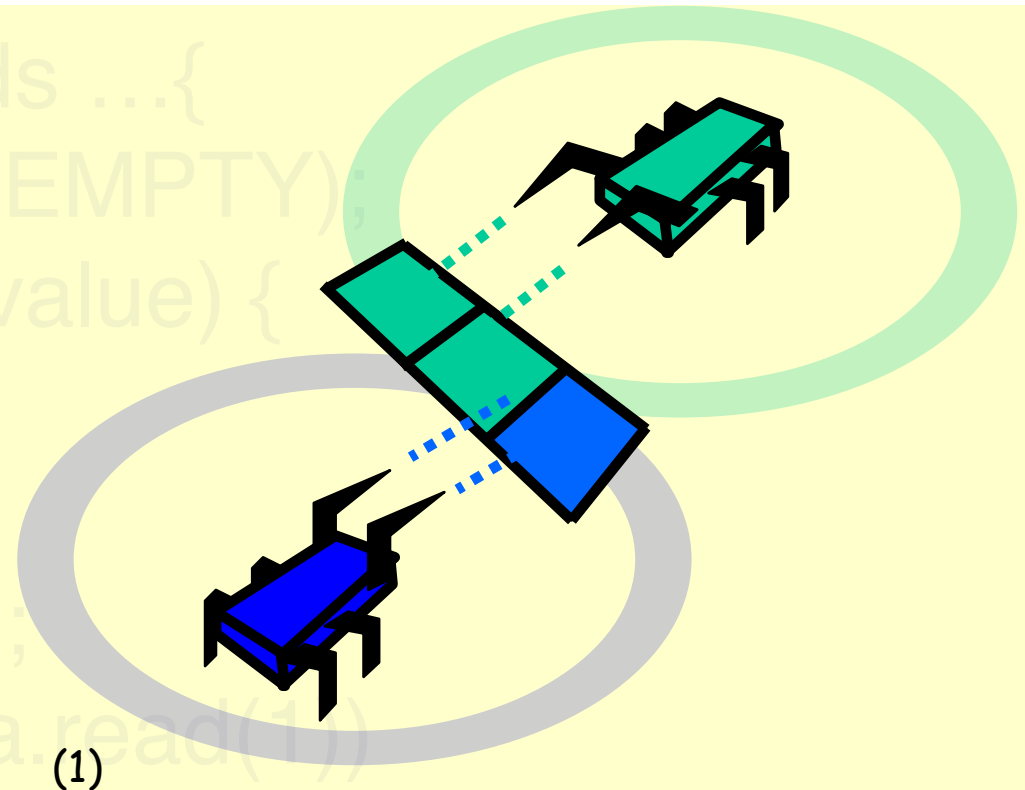
Multi-Consensus Code

```
class MultiConsensus extends ...{  
    Assign2 a = new Assign2(3, EMPTY);  
    public Object decide(object value) {  
        proposed[l]=value;  
        a.assign(i, i, i+1, i);  
        int other = a.read((i+2) % 3);  
        if (other==EMPTY||other==a.read(1))  
            return proposed[i];  
        else  
            return proposed[1-i];  
    }  
}
```

OK, je gagne

Multi-Consensus Code

```
class MultiConsensus extends ...{
  Assign2 a = new Assign2(3, EMPTY);
  public Object decide(object value) {
    proposed[i]=value;
    a.assign(i, i, i+1, i);
    int other = a.read((i+2) % 3);
    if (other==EMPTY||other==a.read(1))
      return proposed[i];
    else
      return proposed[1-i];
  }
}
```



**L'autre thread a
bouge en premier je
perds**

En résumé

- Si on a du (2,3) affectations on peut faire du 2 consensus
- Donc
 - Il n'y a pas de (2,3) affectations wait free a partir de registres atomiques