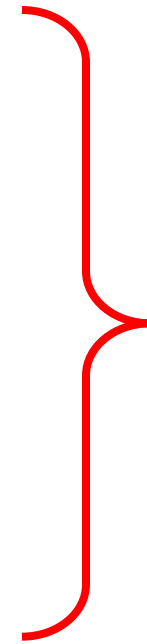
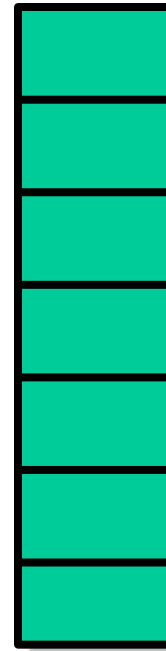


Atomic snapshot

Atomic Snapshot

» deux opérations

update



scan

Interface

```
public interface Snapshot {  
    public int update(int v);  
    public int[] scan();  
}
```


Thread **i** écrit **v** dans son registre

```
public interface Snapshot {  
    public int update(int v);  
    public int[] scan();  
}
```

Interface

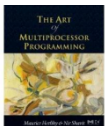
« snapshot » des registres des threads

```
public interface Snapshot {  
    public int update(int v);  
    public int[] scan();  
}
```



Specification séquentielle

- » Un scan retourne pour chacun des éléments la dernière valeur écrite (la valeur initiale si il n'y pas eu d'écriture),
- » un `update(v)` réalisé par le thread `i` écrit `v` dans l'entrée `i` du tableau



Atomique Snapshot

- Collect
 - lire toutes les valeurs des registres
- Problème
 - des collectes concurrents peuvent être incohérents
 - Résultat non linéarisable

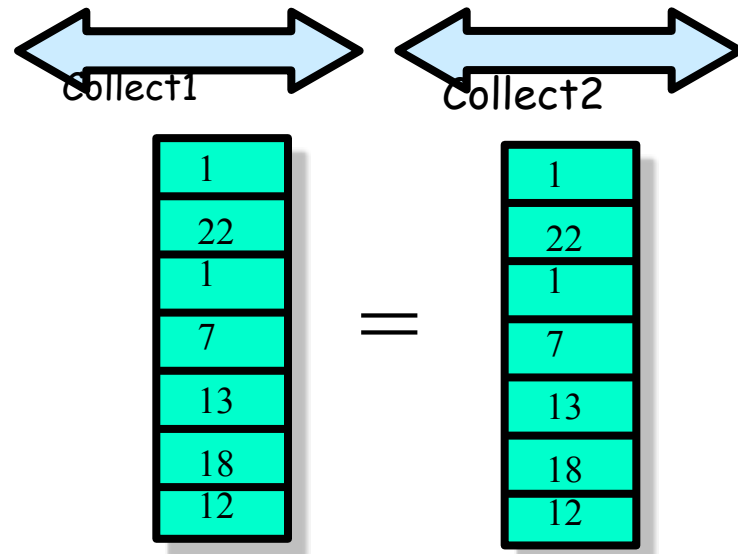
« Clean » Collects

- Clean Collects:
 - rien n'a été modifié pendant le collect
- dans ce cas le collect est un scan

comment assurer un clean collect?
comment le détecter?

Simple Snapshot

- chaque entrée à une étiquette croissante
- deux Collects
- S'ils donnent le même résultat c'est gagné
- Sinon
 - On recommence



Simple Snapshot: Update

```
public class SimpleSnapshot implements Snapshot {  
    private AtomicMRSWRegister[] register;  
  
    public void update(int value) {  
        int i = Thread.myIndex();  
        LabeledValue oldValue = register[i].read();  
        LabeledValue newValue =  
            new LabeledValue(oldValue.label+1, value);  
        register[i].write(newValue);  
    }  
}
```

Simple Snapshot: Update

```
public class SimpleSnapshot implements Snapshot {  
    private AtomicMRSWRegister[] register;
```

```
    public void update(int value) {  
        int i = Thread.myIndex();  
        LabeledValue oldValue = register[i].read();  
        LabeledValue newValue =  
            new LabeledValue(oldValue.label+1, value);  
        register[i].write(newValue);  
    }
```

Un seul SWMR par thread

Simple Snapshot: Update

```
public class SimpleSnapshot implements Snapshot {  
    private AtomicMRSWRegister[] register;  
  
    public void update(int value) {  
        int i = Thread.myIndex();  
        LabeledValue oldValue = register[i].read();  
        LabeledValue newValue =  
            new LabeledValue(oldValue.label+1, value);  
        register[i].write(newValue);  
    }  
}
```

à chaque fois une étiquette plus grande

Simple Snapshot: Collect

```
private LabeledValue[] collect() {  
    LabeledValue[] copy =  
        new LabeledValue[n];  
    for (int j = 0; j < n; j++)  
        copy[j] = this.register[j].read();  
    return copy;  
}
```

Simple Snapshot

```
private LabeledValue[] collect() {  
    LabeledValue[] copy =  
        new LabeledValue[n];  
    for (int j = 0; j < n; j++)  
        copy[j] = this.register[j].read();  
    return copy;  
}
```

Lecture simple des registres

```
public int[] scan() {  
    LabeledValue[] oldCopy, newCopy;  
    oldCopy = collect();  
    collect: while (true) {  
        newCopy = collect();  
        if (!equals(oldCopy, newCopy)) {  
            oldCopy = newCopy;  
            continue collect;  
        }  
    }  
    return getValues(newCopy);  
}}
```

Simple Snapshot: Scan

```
public int[] scan() {  
    LabeledValue[] oldCopy, newCopy;  
    oldCopy = collect();  
    collect: while (true) {  
        newCopy = collect();  
        if (!equals(oldCopy, newCopy)) {  
            oldCopy = newCopy;  
            continue collect;  
        }  
    }  
    return getValues(newCopy);  
}
```

Collect

Simple Snapshot: Scan

```
public int[] scan() {  
    LabeledValue[] oldCopy, newCopy;
```

```
    oldCopy = collect();
```

Premier Collect

```
    collect: while (true) {
```

```
        newCopy = collect();
```

Deuxième Collect

```
        if (!equals(oldCopy, newCopy)) {  
            oldCopy = newCopy;  
            continue collect;  
        }  
    }  
    return getValues(newCopy);  
}
```

Simple Snapshot: Scan

```
public int[] scan() {  
    LabeledValue[] oldCopy, newCopy;  
    oldCopy = collect();  
    collect: while (true) {  
        newCopy = collect();  
        if (!equals(oldCopy, newCopy)) {  
            oldCopy = newCopy;  
            continue collect;  
        }  
        return getValues(newCopy);  
    }  
}
```

si différent on
recommence

Simple Snapshot: Scan

```
public int[] scan() {  
    LabeledValue[] oldCopy, newCopy;  
    oldCopy = collect();  
    collect: while (true) {  
        newCopy = collect();  
        if (!equals(oldCopy, newCopy)) {  
            oldCopy = newCopy;  
            continue collect;  
        }  
        return getValues(newCopy);  
    }  
}
```

Si égal c'est bon



Simple Snapshot

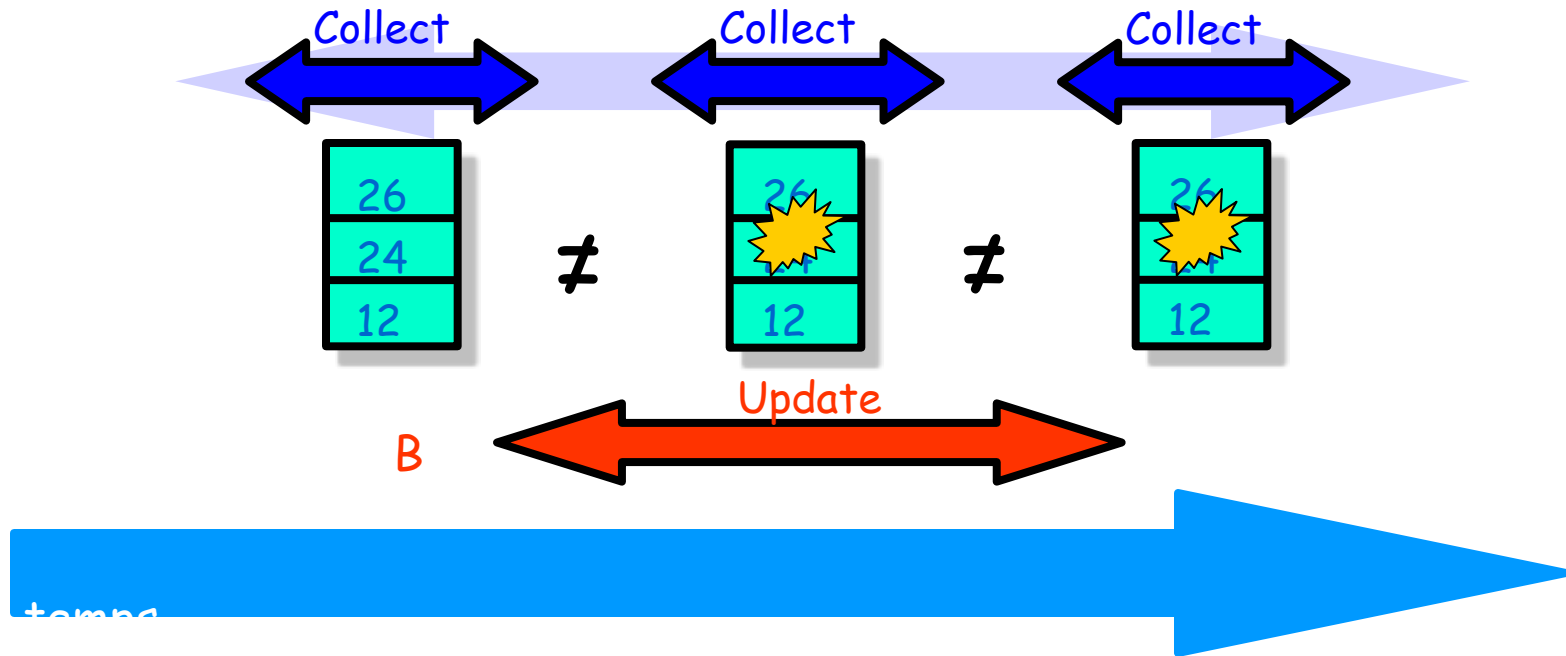
- Linéarisable
- Update wait-free
- Mais le Scan peut ne pas terminer
 - (toujours interrompu par des Update)

Wait-Free Snapshot

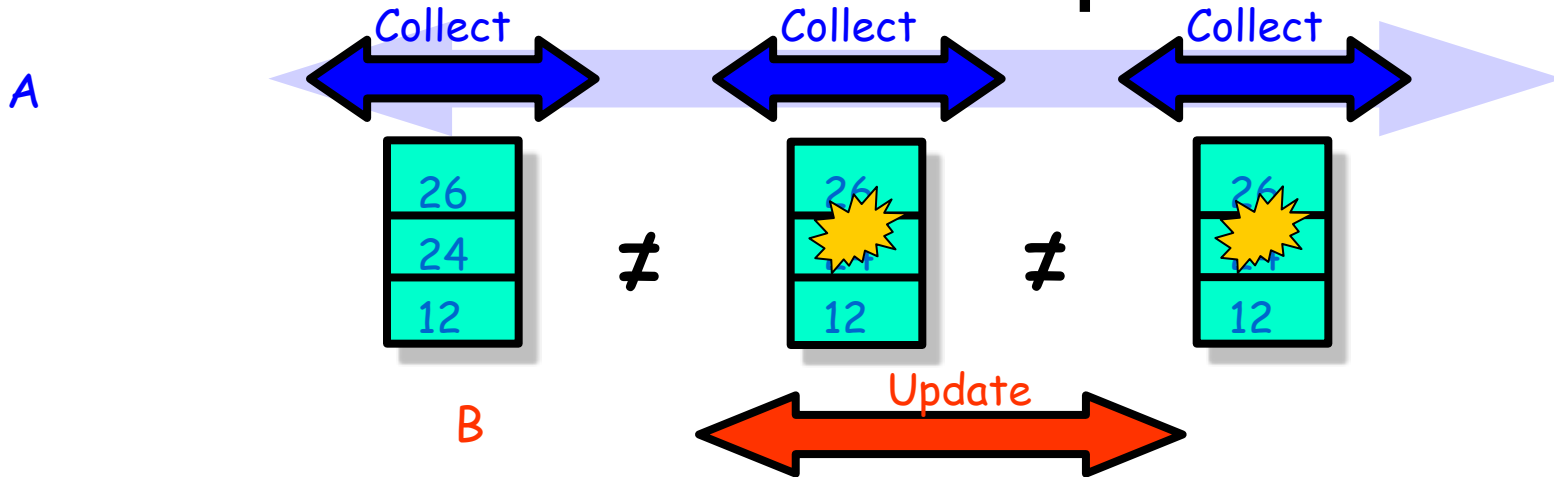
- faire un scan avant chaque update
- écrire le snapshot obtenu avec l'update
- Si le scan est interrompu par les updates, le scan prend le snapshot écrit dans l'update

Wait-free Snapshot

si le scan de A observe que B a bougé deux fois alors B a fait un update pendant le scan de A

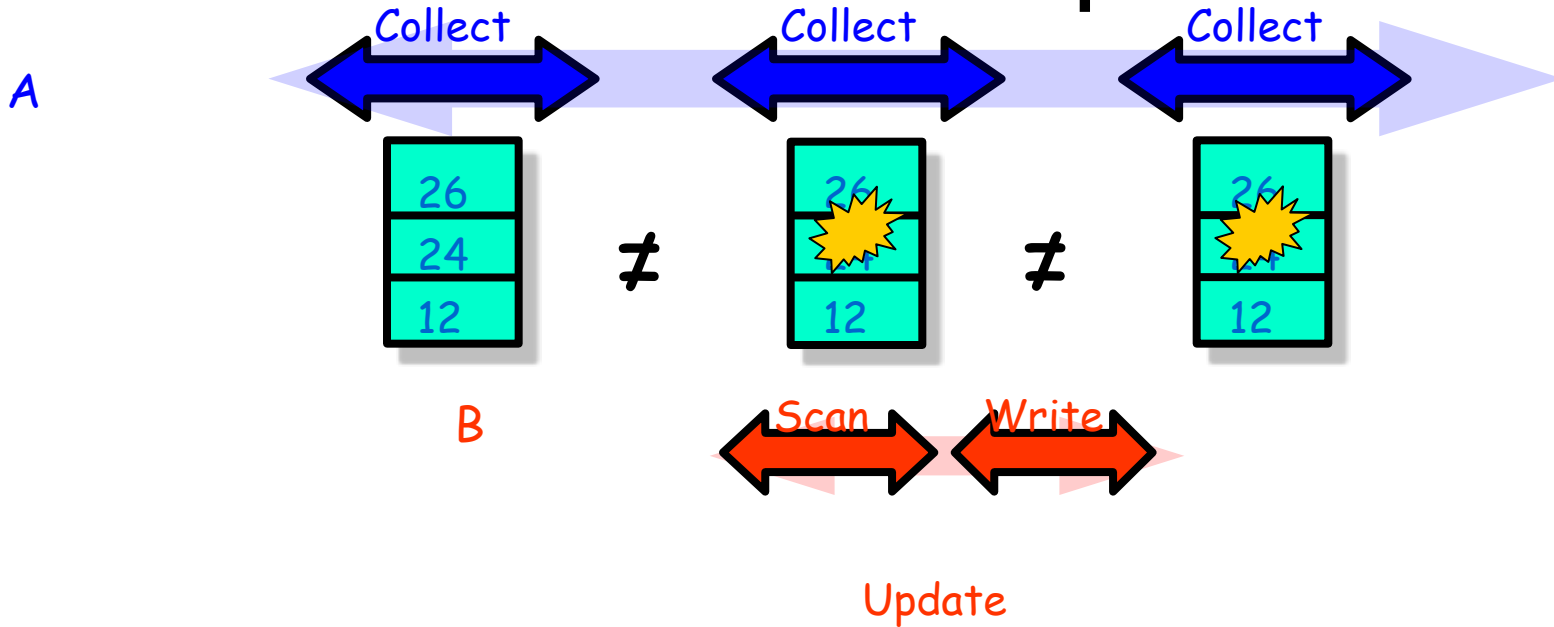


Wait-free Snapshot



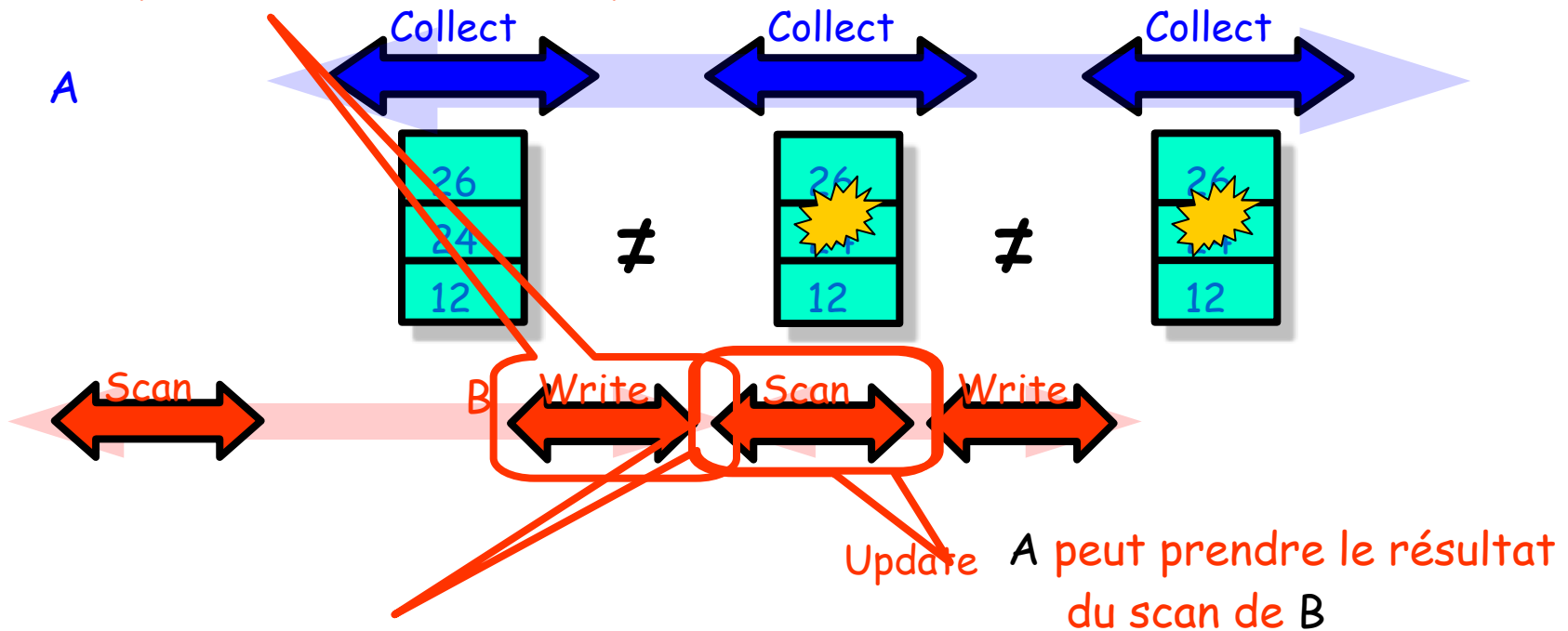
temp

Wait-free Snapshot



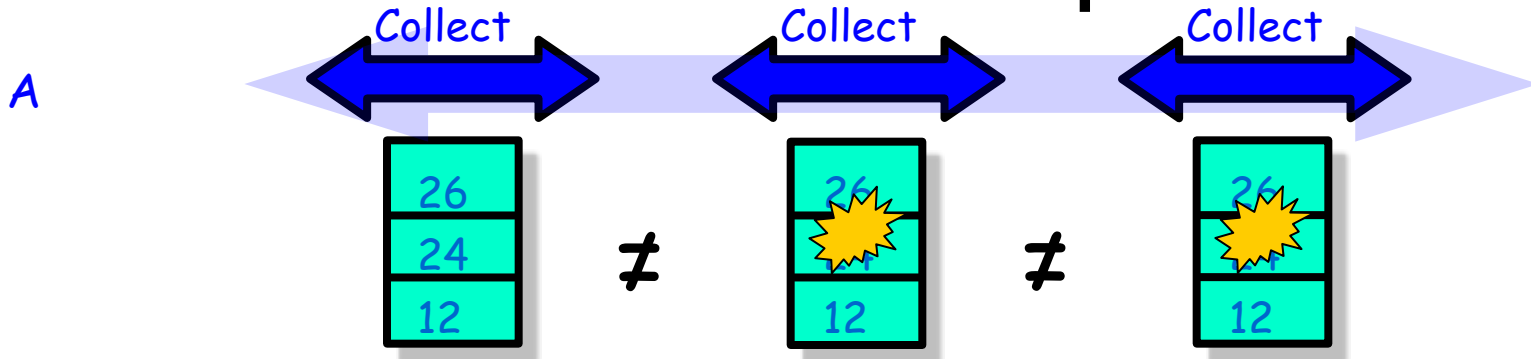
Wait-free Snapshot

1^{er} update de B a été écrit pendant le 1^{er} collect



le scan de B du deuxième update a eu lieu pendant l'intervalle du scan de A

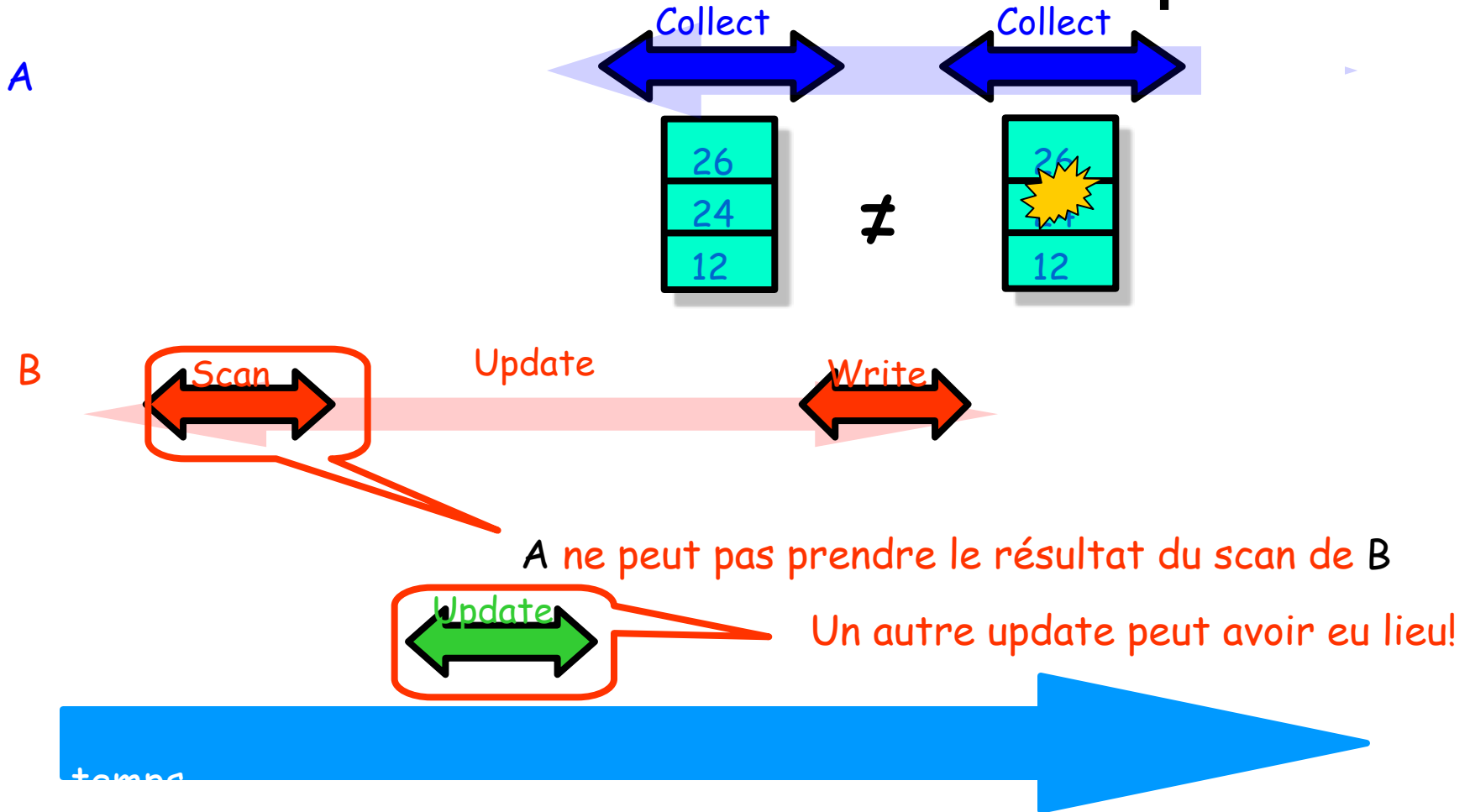
Wait-free Snapshot



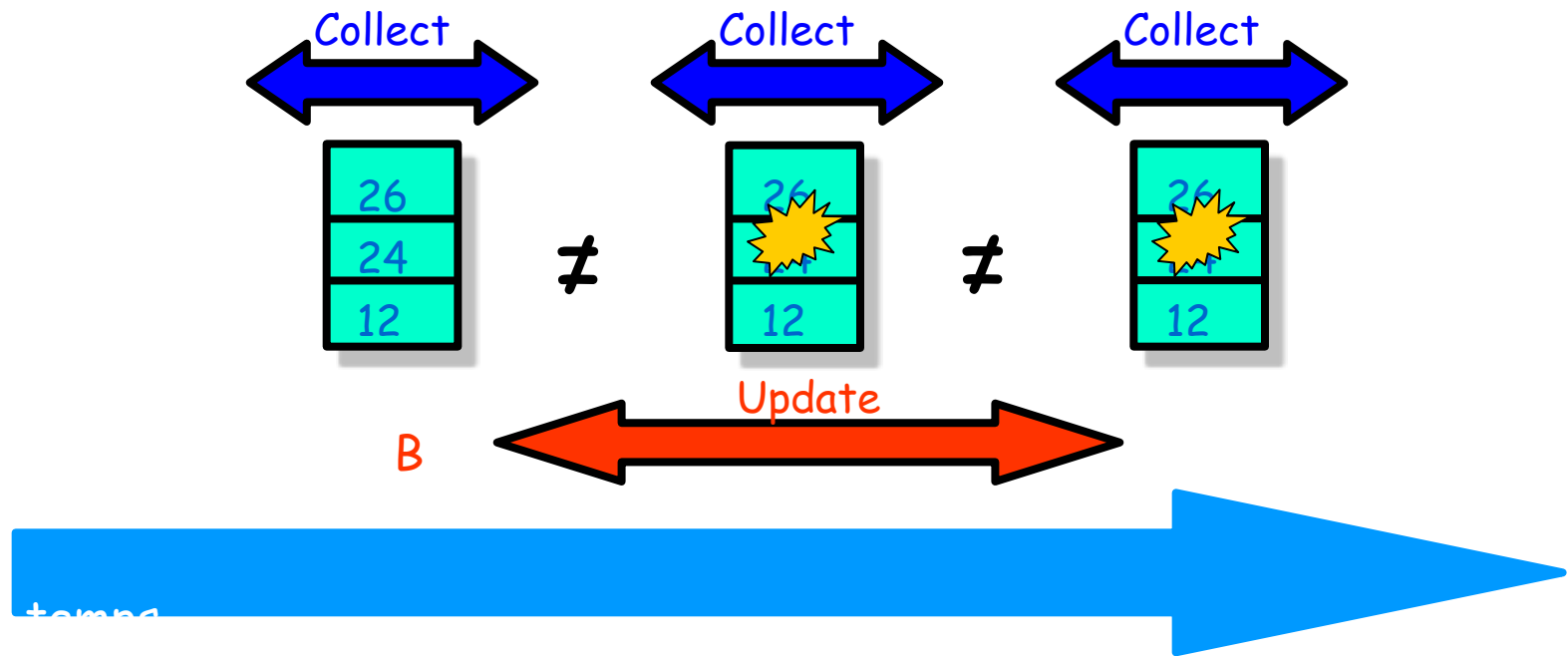
But no guarantee that scan
of B's 1st update can be used...
Why?



Une seule fois ne suffit pas

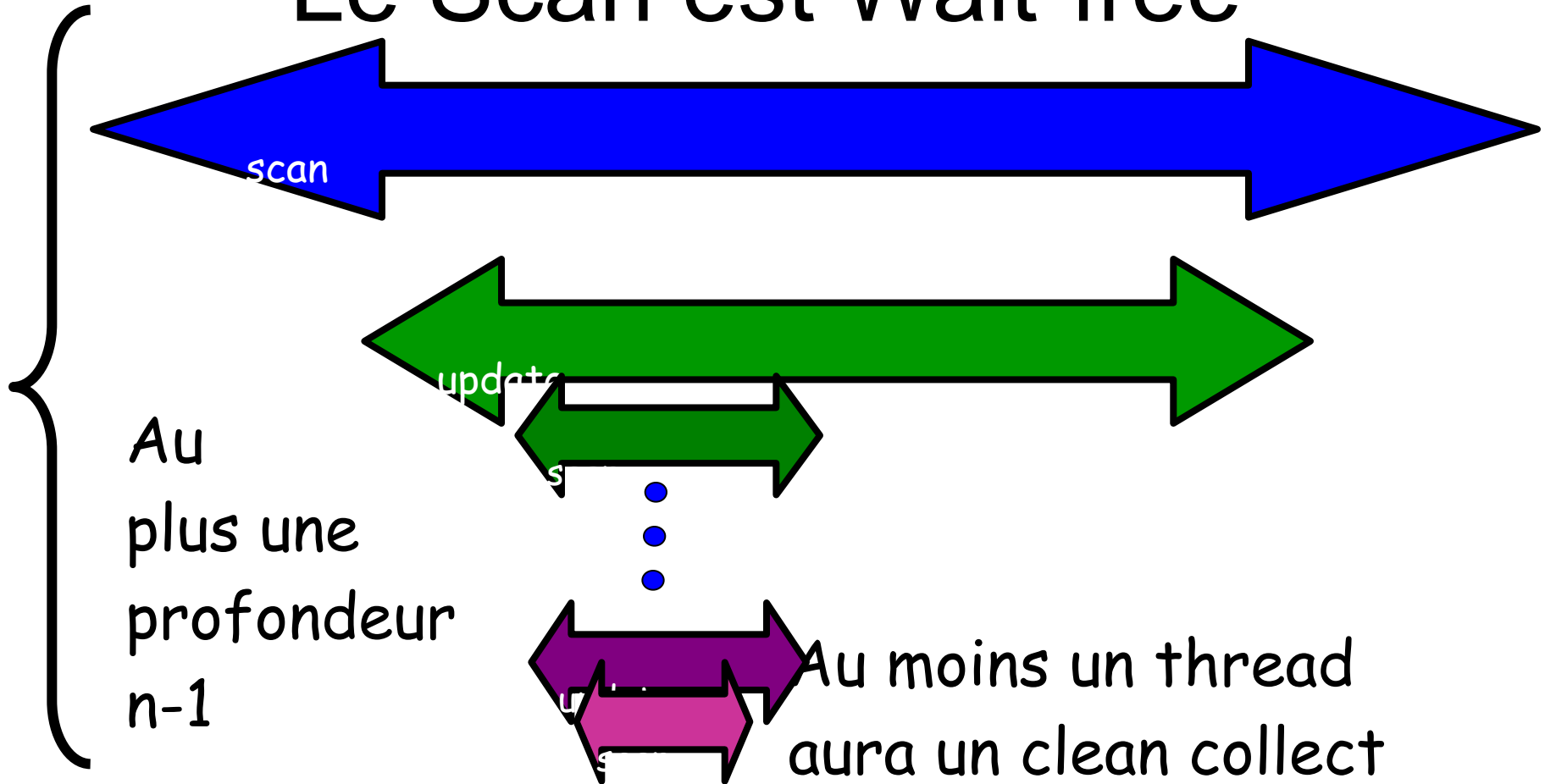


Quelqu'un bouge 2 fois!



Si on fait n collects différents...
au moins un thread a bougé 2 fois!

Le Scan est Wait-free

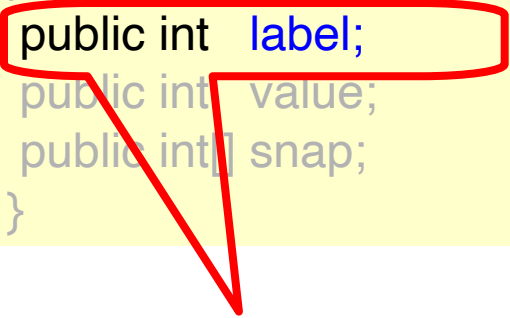


Wait-Free Snapshot étiquetage

```
public class SnapValue {  
    public int label;  
    public int value;  
    public int[] snap;  
}
```

Wait-Free Snapshot étiquetage

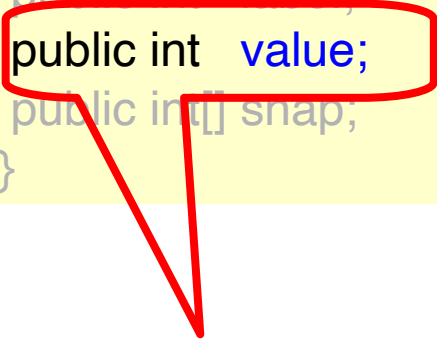
```
public class SnapValue {  
    public int label;  
    public int value;  
    public int[] snap;  
}
```



label incrémenté à
chaque snapshot

Wait-Free Snapshot étiquetage

```
public class SnapValue {  
    public int label;  
    public int value;  
    public int[] snap;  
}
```



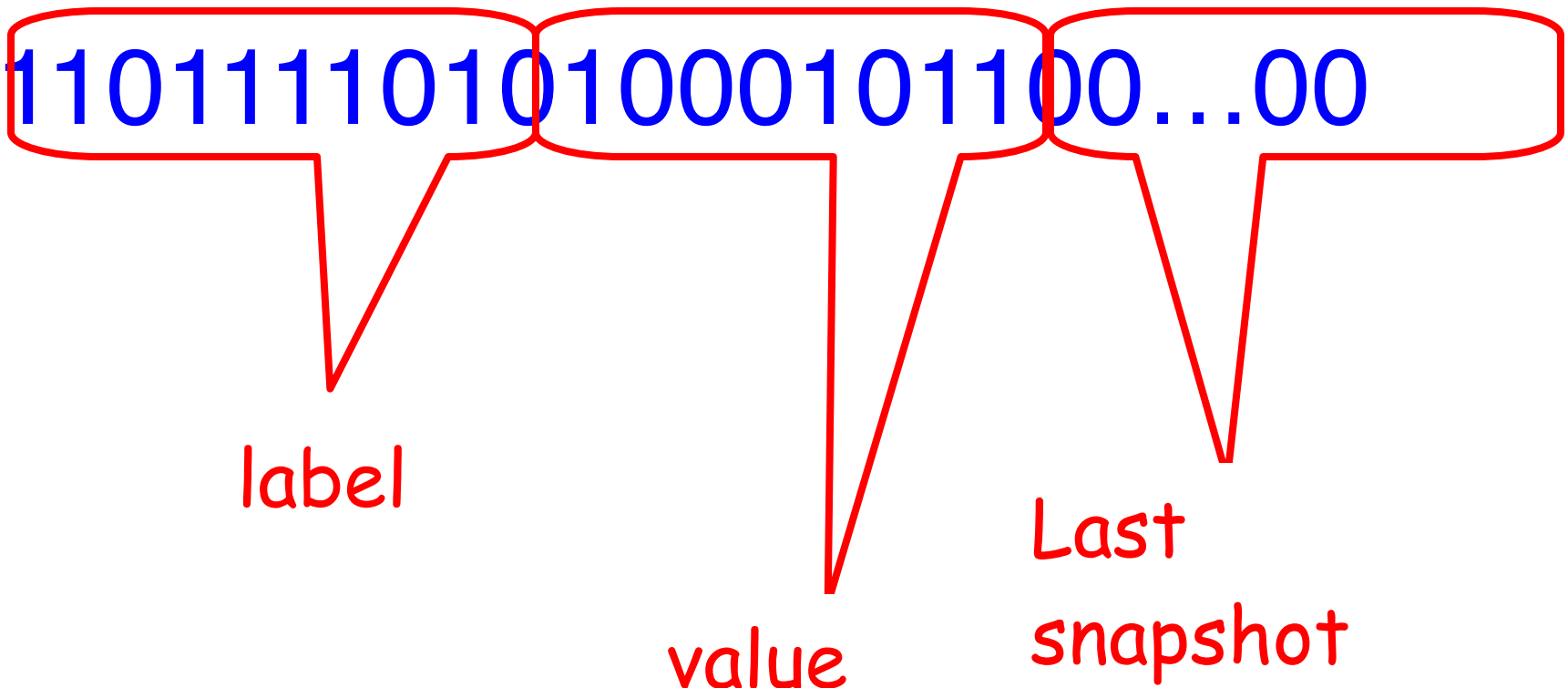
la valeur

Wait-Free Snapshot étiquetage

```
public class SnapValue {  
    public int  label;  
    public int  value;  
    public int[] snap;  
}
```

snapshot le plus récent

Wait-Free Snapshot Label



Wait-free Update

```
public void update(int value) {  
    int i = Thread.myIndex();  
    int[] snap = this.scan();  
    SnapValue oldValue = r[i].read();  
    SnapValue newValue =  
        new SnapValue(oldValue.label+1,  
                       value, snap);  
    r[i].write(newValue);  
}
```

Wait-free Scan

```
public void update(int value) {  
    int i = Thread.myIndex();  
    int[] snap = this.scan();  
    SnapValue oldValue = r[i].read();  
    SnapValue newValue =  
        new SnapValue(oldValue.label+1,  
            value, snap);  
    r[i].write(newValue);  
}
```

scan

Wait-free Scan

```
public void update(int value) {  
    int i = Thread.myIndex();  
    int snap = this.scan();  
    SnapValue oldValue = r[i].read();  
    SnapValue newValue =  
        new SnapValue(oldValue.label+1,  
                       value, snap);  
    r[i].write(newValue);  
}
```

scan

étiquetage du scan

Wait-free Scan

```
public int[] scan() {  
    SnapValue[] oldCopy, newCopy;  
    boolean[] moved = new boolean[n];  
    oldCopy = collect();  
    collect: while (true) {  
        newCopy = collect();  
        for (int j = 0; j < n; j++) {  
            if (oldCopy[j].label != newCopy[j].label) {  
                ...  
            }  
        }  
        return getValues(newCopy);  
    }  
}
```

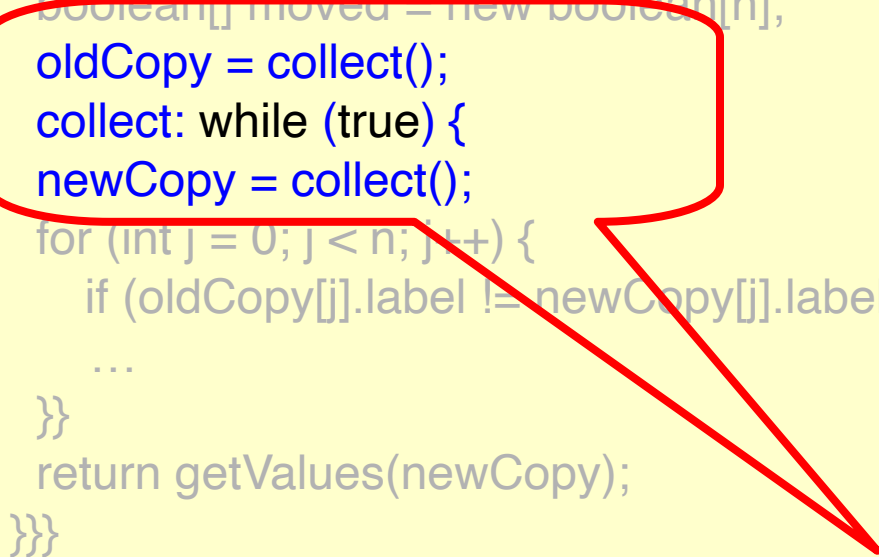
Wait-free Scan

```
public int[] scan() {  
    SnapValue[] oldCopy, newCopy;  
    boolean[] moved = new boolean[n];  
    oldCopy = collect();  
    collect: while (true) {  
        newCopy = collect();  
        for (int j = 0; j < n; j++) {  
            if (oldCopy[j].label != newCopy[j].label) {  
                ...  
            }  
        }  
        return getValues(newCopy);  
    }  
}
```

enregistrer ceux qui ont changé

Wait-free Scan

```
public int[] scan() {  
    SnapValue[] oldCopy, newCopy;  
    boolean[] moved = new boolean[n];  
    oldCopy = collect();  
    collect: while (true) {  
        newCopy = collect();  
        for (int j = 0; j < n; j++) {  
            if (oldCopy[j].label != newCopy[j].label) {  
                ...  
            }  
        }  
        return getValues(newCopy);  
    }  
}
```

A red speech bubble originates from the 'collect' loop in the code. The bubble's tail points to the 'collect' label and the 'while (true)' loop, and its main body encloses the lines 'oldCopy = collect();', 'collect: while (true) {', and 'newCopy = collect();'.

double collect répété

Wait-free Scan

```
public int[] scan() {  
    SnapValue[] oldCopy, newCopy;  
    boolean[] moved = new boolean[n];  
    oldCopy = collect();  
    collect: while (true) {  
        newCopy = collect();  
        for (int j = 0; j < n; j++) {  
            if (oldCopy[j].label != newCopy[j].label) {  
                ...  
            }  
        }  
        return getValues(newCopy);  
    }  
}
```

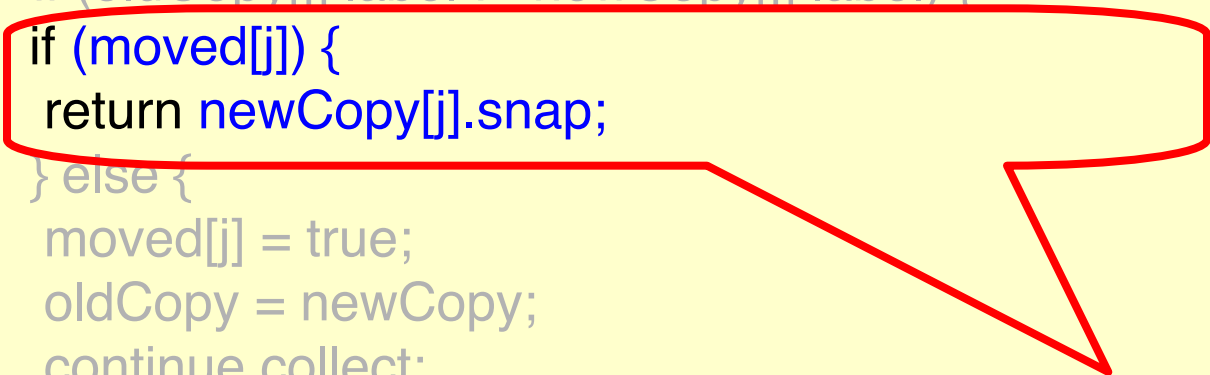
si différents...

Si différents

```
if (oldCopy[j].label != newCopy[j].label) {  
  if (moved[j]) { // second changement  
    return newCopy[j].snap;  
  } else {  
    moved[j] = true;  
    oldCopy = newCopy;  
    continue collect;  
  }  
}  
return getValues(newCopy);  
}
```

Si différents

```
if (oldCopy[i].label != newCopy[i].label) {  
  if (moved[j]) {  
    return newCopy[j].snap;  
  } else {  
    moved[j] = true;  
    oldCopy = newCopy;  
    continue collect;  
  }  
}  
return getValues(newCopy);  
}
```



si la thread a changé
deux fois, prendre son
second snapshot

Si différents

```
if (oldCopy[j].label != newCopy[j].label) {  
  if (moved[j]) { // second move  
    return newCopy[j].snap;  
  } else {  
    moved[j] = true;  
    oldCopy = newCopy;  
    continue collect;  
  }  
}  
return getValues(newCopy);  
}
```

Noter le changement

Observations

- Utilise des compteurs non bornés
 - Implementation avec compteurs bornes
- Chaque emplacement n'est écrit que par un seul thread (SWMR registres)
 - Extension : plusieurs threads peuvent mettre à jour un emplacement (MRMW registres)
 - Attention à l'estampille

Atomique snapshot

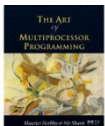
» Si la thread i exécute (x_i est strictement positif)

$update(x_i)$

$S_i = scan(); S_i = S_i - \{0\}$

—> on a $S_i \subseteq S_j$ ou $S_j \subseteq S_i$

» les snapshots sont inclus les uns dans les autres



immediate snapshot (IS)

Une operation:

WriteRead(v)

Un seul appel par
processus

Pour chaque processus

p_i :

$S_i := \text{WriteRead}_i(v_i)$

Un seul appel par
processus

Vecteurs S_1, \dots, S_N satisfont:

§ **auto-inclusion**: pour tout i : v_i
est dans S_i

§ **inclusion**: pour tout i et j : S_i
est un sous ensemble de S_j oo
 S_j est un sous-ensemble de S_i

§ **Immédiateté**: pour tout i et j :
si v_i est dans S_j , alors S_i est un
sous-ensemble de S_j

Spécification séquentielle

?

IS à partir d' AS

shared variables:

A_1, \dots, A_N – atomic snapshot objects, initially $[T, \dots, T]$

Upon WriteRead_i(v_i)

$r := N+1$

while true do

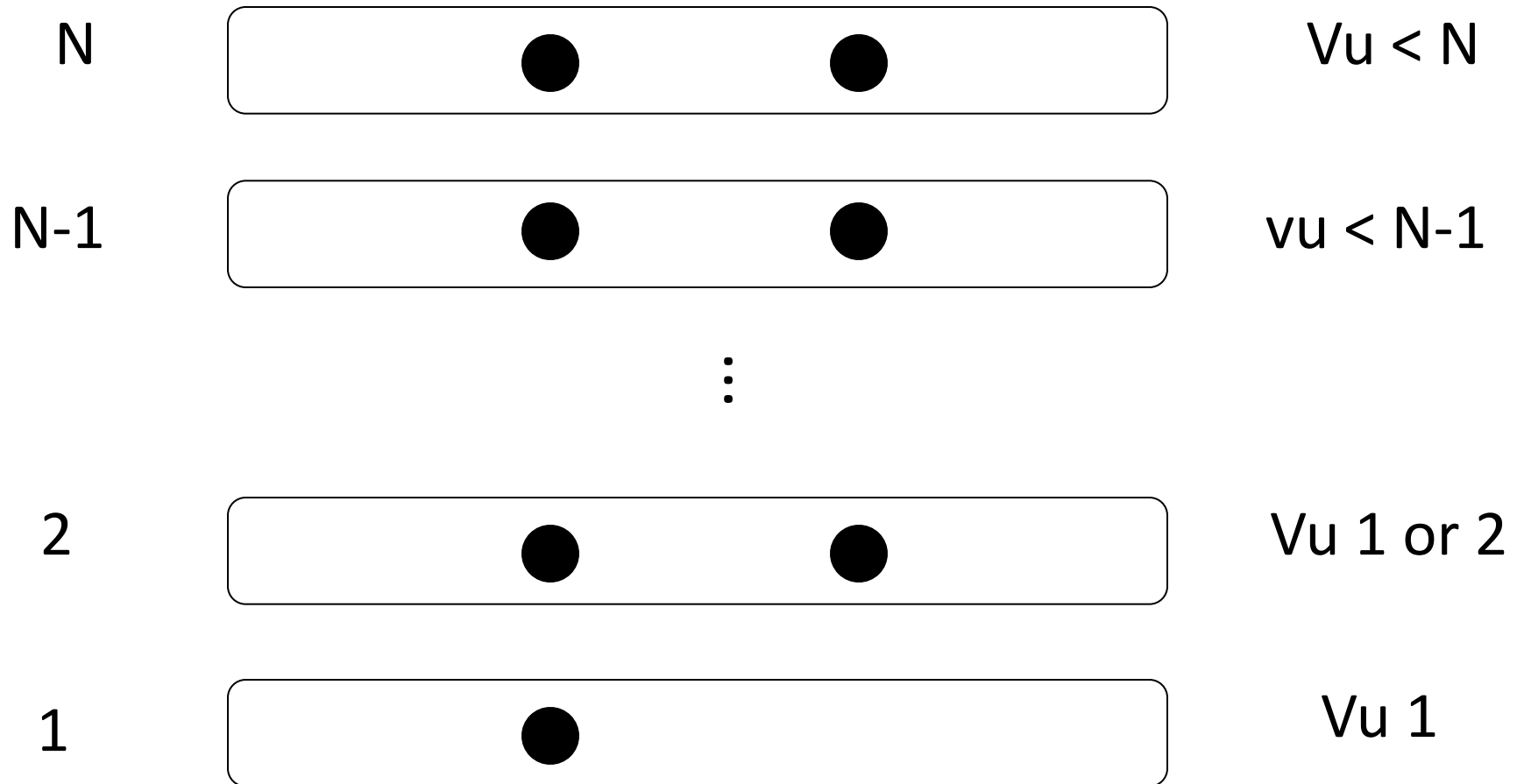
$r := r-1$

$A_r.\text{update}_i(v_i)$

$S := A_r.\text{snapshot}()$

 if $|S|=r$ then

 return S



Correction

La sortie de l'algorithme satisfait les 3 propriétés

- Par induction sur N :
 - Pour tout $N > 1$, l'algorithme est correct pour N
- Cas de base $N=1$: trivial

Correction

- On suppose l'algorithme correct pour $N-1$ processus
- N processus arrivent au niveau N
 - ✓ Au plus $N-1$ vont au niveau $N-1$ ou plus bas
 - ✓ (Au moins un processus reste au niveau N)
- Self-inclusion, Inclusion et Immediate sont vrai pour tous les proc qui vont au niveau $N-1$ ou plus bas par hyp induction
- Les processus sortant au niveau N retourne les N valeurs
 - ✓ les propriétés sont vrais pour les N processus

Résumé

- On vient de voir qu'on peut implementer (wait free) un snapshot MRMW multivalué
- A partir de registre sure SRSW binaire
- On peut aussi implementer un immediate snapshot
- Peut on faire mieux à partir de registres?

Suite

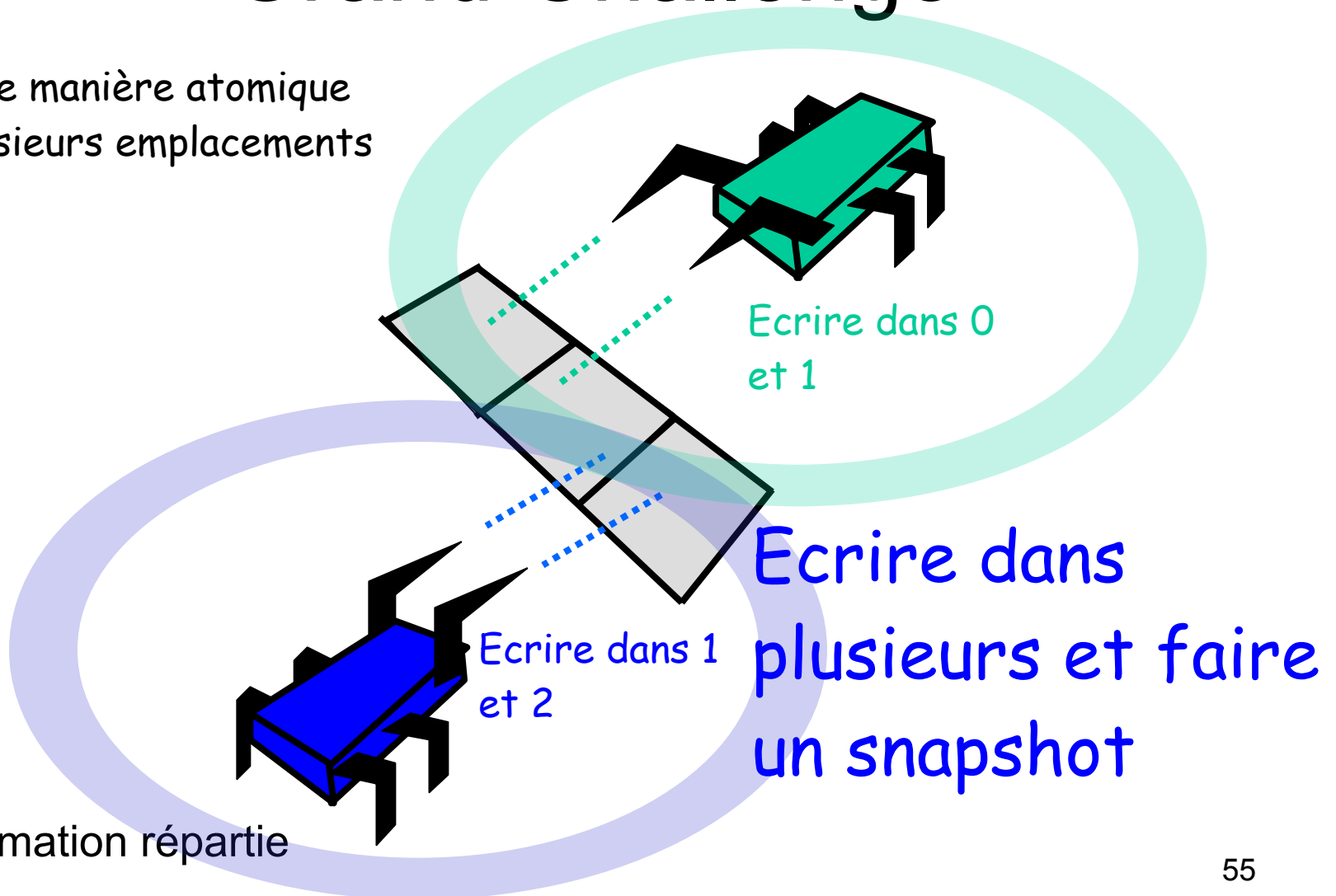
- Le Snapshot (linéarisable)
 - écrit un élément dans un tableau
 - lit tous les éléments du tableau

Ecrire (atomique) plusieurs éléments d'un tableau

Lire un ou plusieurs éléments du tableau?

Grand Challenge

Ecrire de manière atomique
dans plusieurs emplacements



Programmation répartie