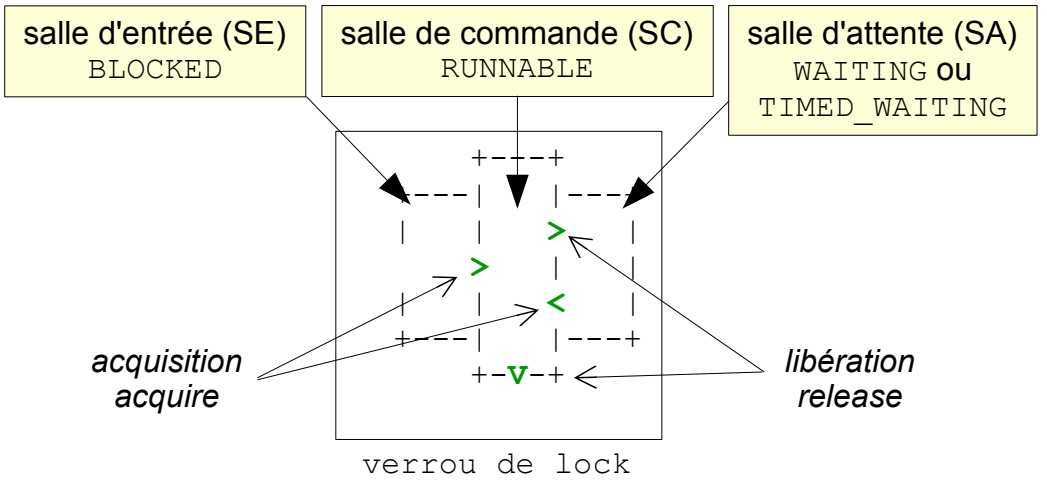
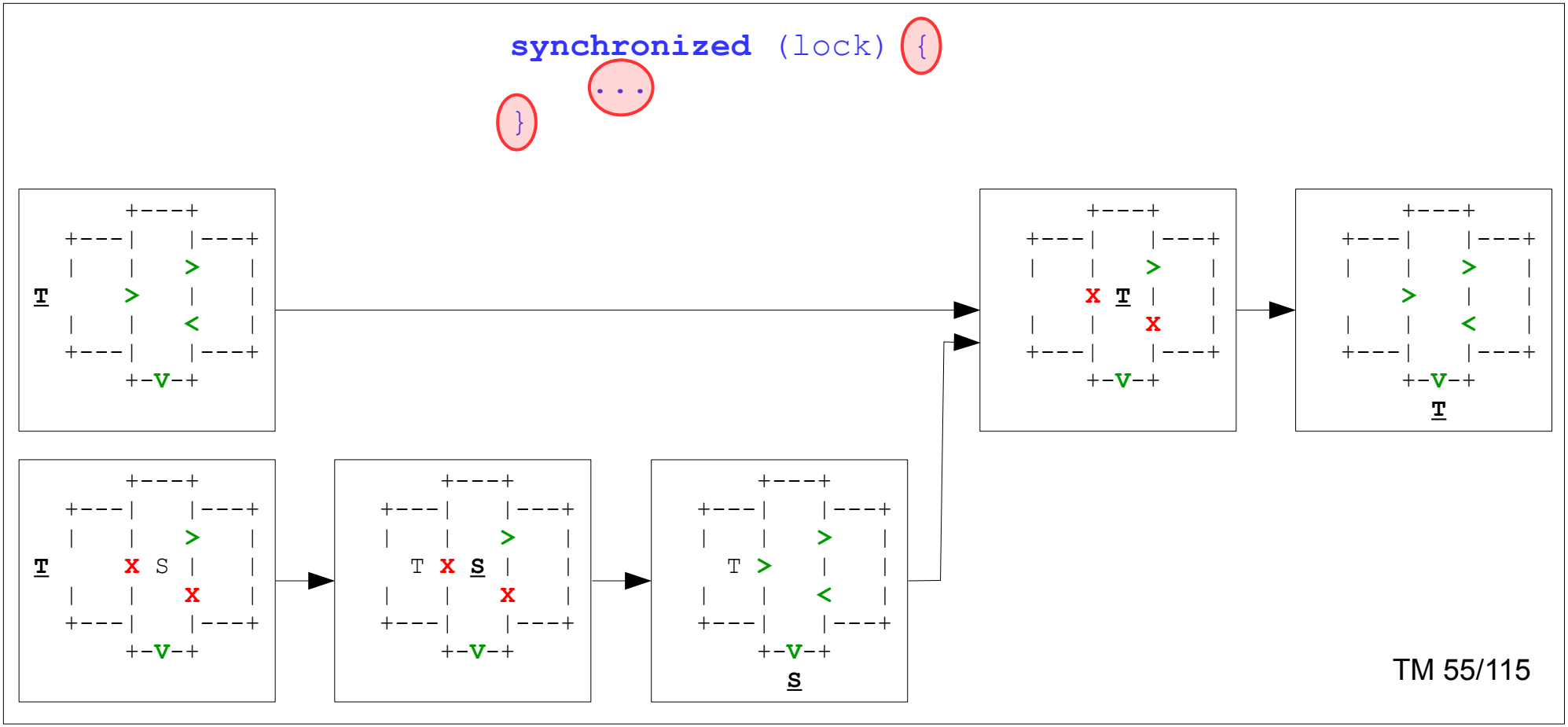
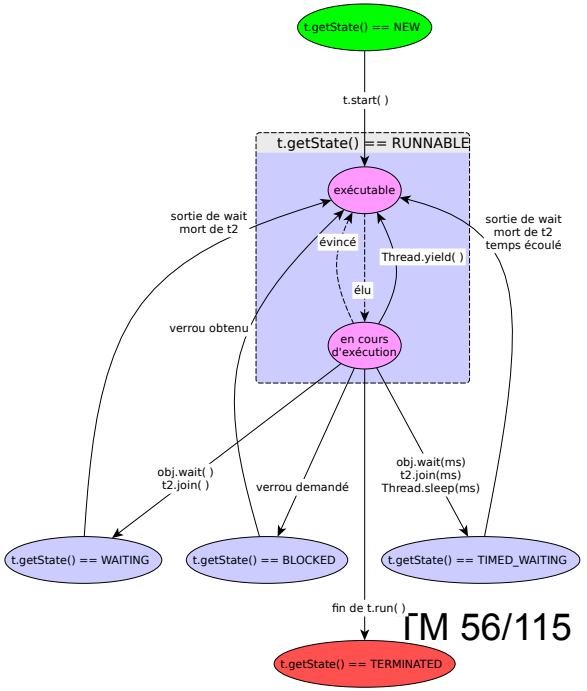

salle d'entrée (SE)
BLOCKED
salle de commande (SC)
RUNNABLE
salle d'attente (SA)
WAITING ou
TIMED_WAITING
acquisition
acquire
libération
release
verrou de lock


synchronized (lock) {
...
}
TM 55/115

Méthodes `wait` et `notify`

Synchronisation obligatoire !

- **`void`** `wait()`
  - thread courant en attente de notification
- **`void`** `wait(`**`long`** `x`**`[`**`, `**`int`** `y`**`]`**`)`
  - *idem* mais pour une durée limitée (`x` ms + `y` ns)
- **`void`** `notify()`
  - notifie un thread en attente
- **`void`** `notifyAll()`
  - notifie tous les threads en attente

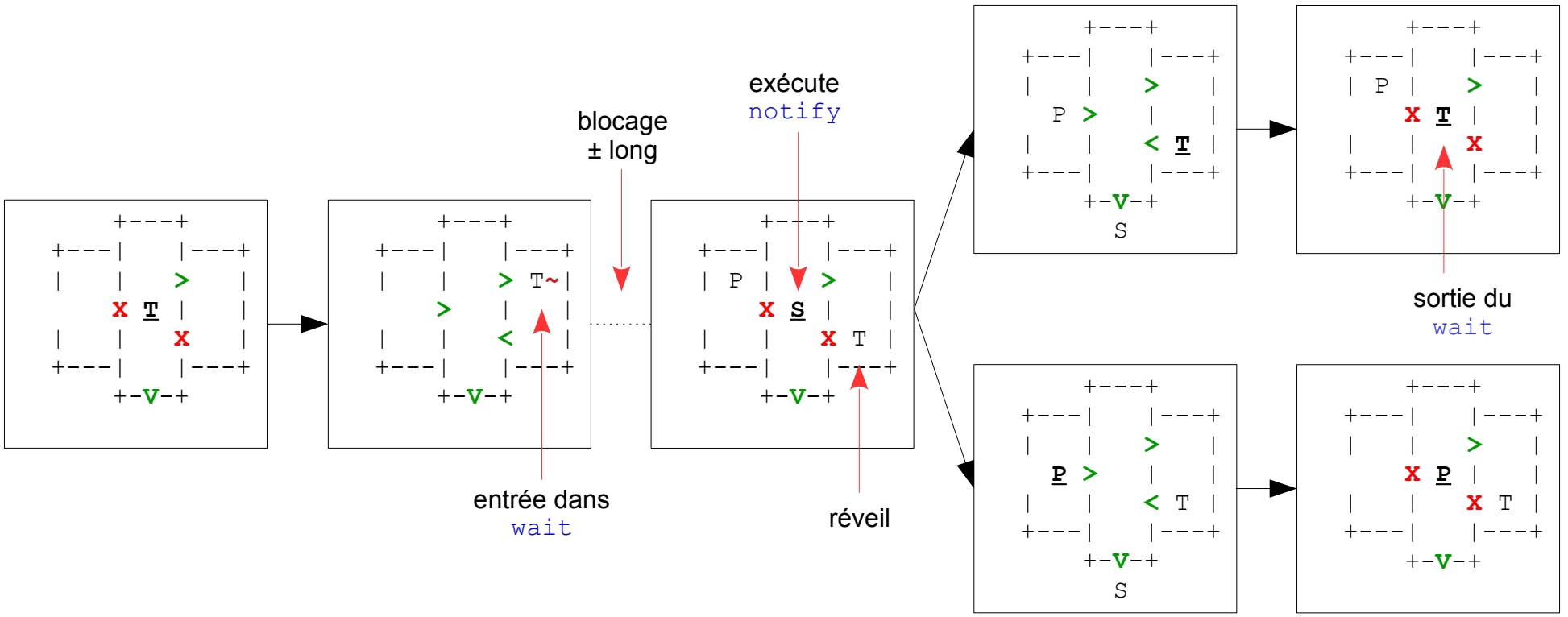


code cible de T

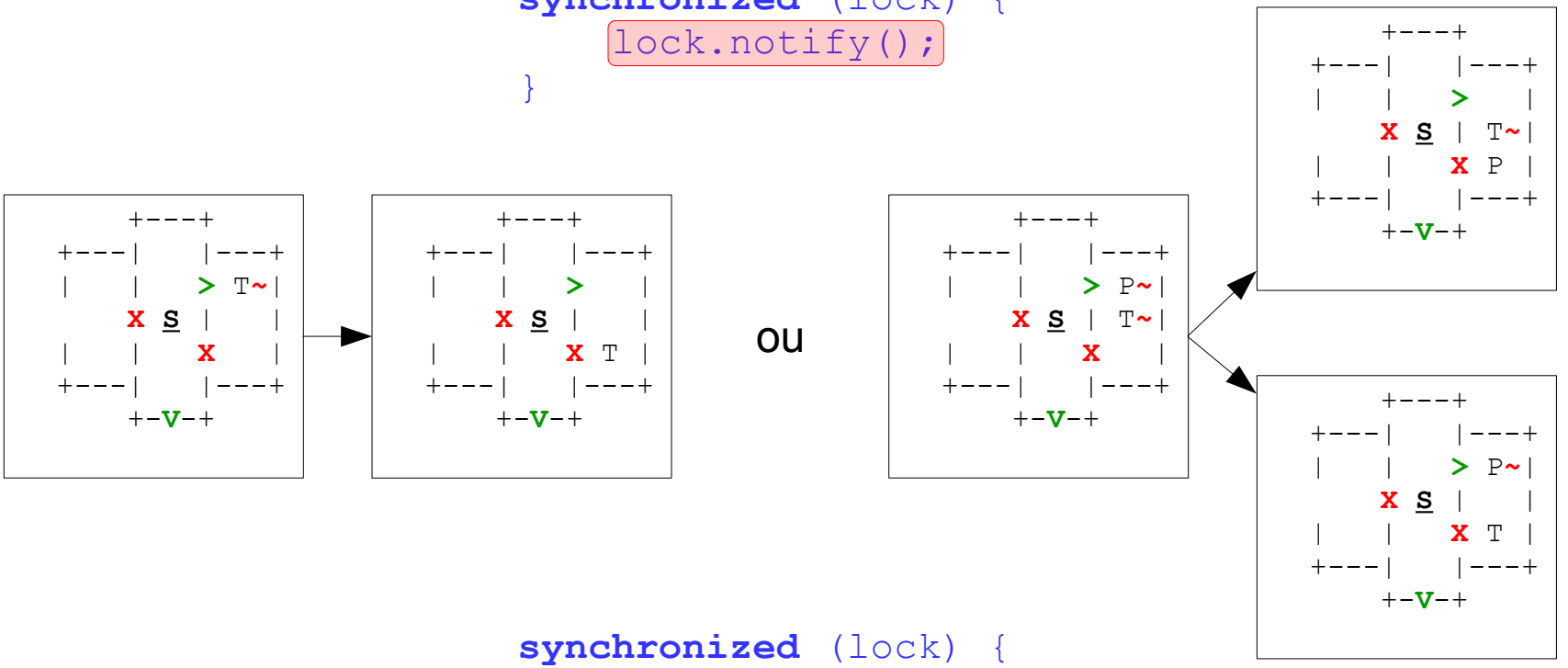
```
synchronized (lock) {
    lock.wait();
}
```

code cible de S

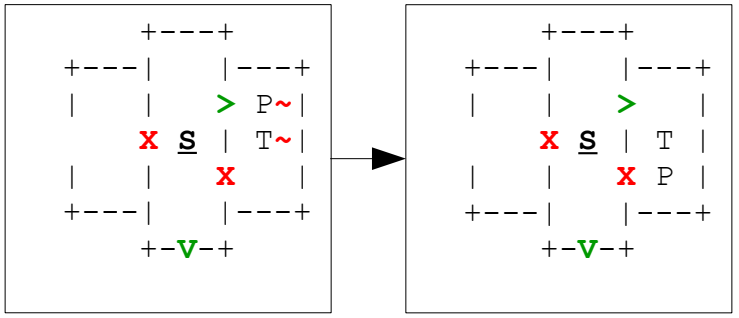
```
synchronized (lock) {
    lock.notify();
}
```



```
synchronized (lock) {
    lock.notify();
}
```




```
synchronized (lock) {
    lock.notifyAll();
}
```



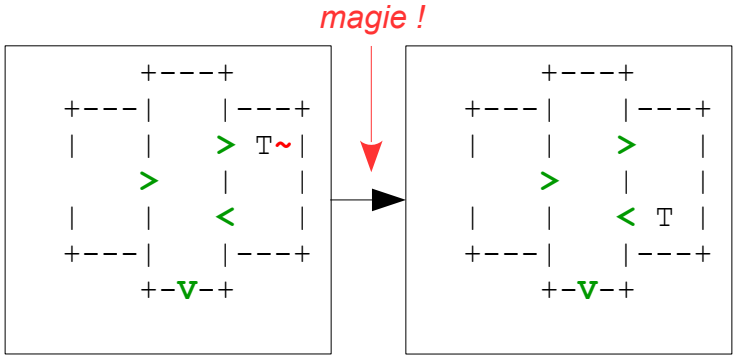
```
1 synchronized (lock) {
2     if (!dataAvailable) {
          try {
3,4,5,6,7     lock.wait();
          } catch (InterruptedException e) {
              // ...
          }
      }
8     consumeData();
:     dataAvailable = false;
  }
9
```

```
1 synchronized (lock) {
2     fillData();
:     dataAvailable = true;
3     lock.notify();
  }
4
```

A1 - t acquiert le verrou de lock (→ SC)
A2 - t teste la condition (données non disponibles)
A3 - t commence l'exécution de wait()
A4 - t libère le verrou de lock (SC → SA)
B1 - s acquiert le verrou de lock (→ SC)
B2 - s valide la condition
B3 - s exécute notify()
A5 - t est candidat au réveil (se présente à la sortie de SA)
B4 - s libère le verrou de lock (SC →)
A6 - t acquiert le verrou de lock (SA → SC)
A7 - t termine l'exécution de wait()
A8 - t invalide la condition
A9 - t libère le verrou de lock (SC →)

Rq. *Spurious wakeup*

- Dans la documentation de la classe `Thread` :
  - « *A thread can wake up without being notified, interrupted, or timing out, a so-called spurious wakeup. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied.* »



- Conclusion :
  - appel à `wait` → toujours inclus dans une boucle

```
1 synchronized (lock) {
2     if (!dataAvailable) {
          try {
3,4,5,6,7     lock.wait();
          } catch (InterruptedException e) {
              // ...
          }
      }
8     consumeData();
:     dataAvailable = false;
  }
9
```

```
1 synchronized (lock) {
2     fillData();
:     dataAvailable = true;
3     lock.notify();
  }
4
```

il faut coder
`while (!dataAvailable)...`
s'il y a plusieurs threads comme t

A1 - t acquiert le verrou de lock (→ SC)
A2 - t teste la condition (données non disponibles)
A'1 - t' est bloqué sur le verrou de lock (→ SE)
A3 - t commence l'exécution de wait()
A4 - t libère le verrou de lock (SC → SA)
B1 - s acquiert le verrou de lock (→ SC)
B2 - s valide la condition
B3 - s exécute notify()
A5 - t est candidat au réveil
B4 - s libère le verrou de lock (SC →)
A'1 - t' acquiert le verrou de lock (SE → SC)
A'2 - t' teste la condition (données disponibles)
A'8 - t' invalide la condition
A'9 - t' libère le verrou de lock (SC →)
A6 - t acquiert le verrou de lock (SA → SC)
A7 - t termine l'exécution de wait()
A8 - t tente d'invalider la condition... en supposant à tort qu'elle est valide !

```
1 synchronized (lock) {
2     while (!barrierOpened) {
          try {
3,4,5,6,7     lock.wait();
          } catch (InterruptedException e) {
              // ...
          }
      }
  }
8
```

```
1 synchronized (lock) {
2     barrierOpened = true;
3     lock.notify();
  }
4
```

A1 - t acquiert le verrou de lock (→ SC)
A2 - t teste la condition (données non disponibles)
A3 - t commence l'exécution de wait()
A4 - t libère le verrou de lock (SC → SA)

A'1 - t' acquiert le verrou de lock (→ SC)
A'2 - t' teste la condition (données non disponibles)
A'3 - t' commence l'exécution de wait()
A'4 - t' libère le verrou de lock (SC → SA)

B1 - s acquiert le verrou de lock (→ SC)
B2 - s valide la condition
B3 - s exécute notify()

A5 - t est candidat au réveil

B4 - s libère le verrou de lock (SC →)

A6 - t acquiert le verrou de lock (SA → SC)
A7 - t termine l'exécution de wait()
A8 - t libère le verrou de lock (SC →)

t' est bloqué à jamais

```
1 synchronized (lock) {
2     while (!barrierOpened) {
          try {
3,4,5,6,7     lock.wait();
          } catch (InterruptedException e) {
              // ...
          }
      }
  }
8
```

```
1 synchronized (lock) {
2     barrierOpened = true;
3     lock.notifyAll();
  }
4
```

A1 ... A2
A3 - t commence l'exécution de wait()
A4 - t libère le verrou de lock (SC → SA)

A'1 ... A'2
A'3 - t' commence l'exécution de wait()
A'4 - t' libère le verrou de lock (SC → SA)

B1 - s acquiert le verrou de lock (→ SC)
B2 - s valide la condition
B3 - s exécute notifyAll()

A5 - t est candidat au réveil

A'5 - t' est candidat au réveil

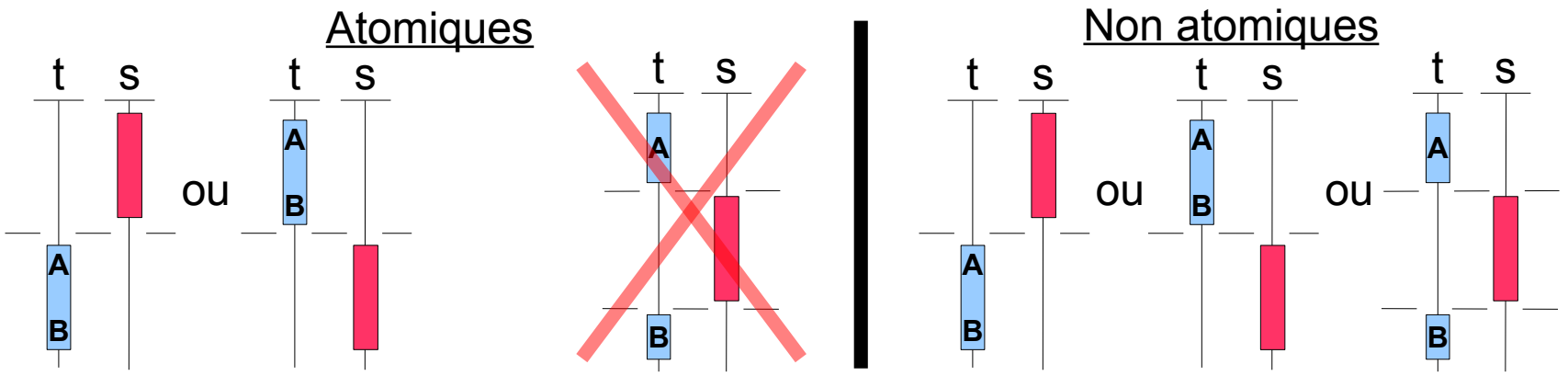
B4 - s libère le verrou de lock (SC →)

A6 - t acquiert le verrou de lock (SA → SC)
A7 - t termine l'exécution de wait()
A8 - t libère le verrou de lock (SC →)

A'6 - t' acquiert le verrou de lock (SA → SC)
A'7 - t' termine l'exécution de wait()
A'8 - t' libère le verrou de lock (SC →)

Opération atomique

- **Opération atomique relativement à une autre**

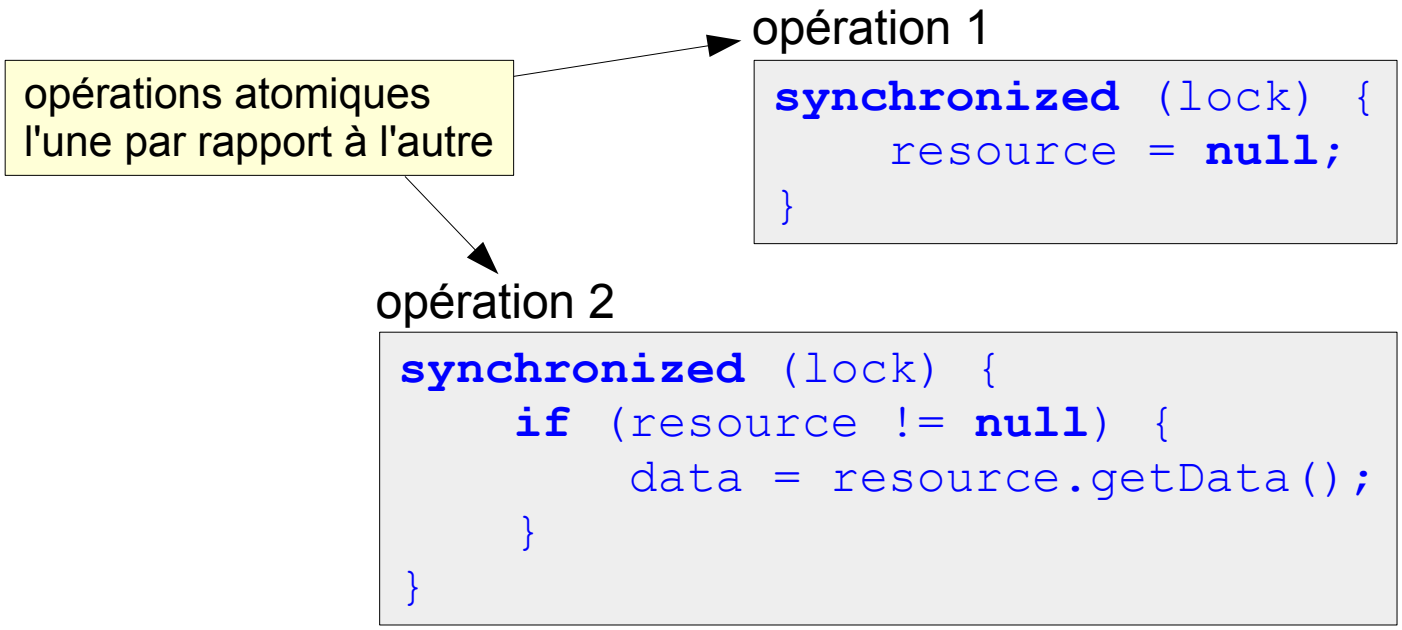


- **Opération atomique** = atomique relativement à toute opération* *qui partage des variables avec elle*

* y compris elle-même

Synchronisation et atomicité

- Exercice



  - La première opération est-elle atomique par rapport à la seconde ?
  - La seconde opération est-elle atomique ?
  - Sous quelle condition le sera-t-elle ?

Variable volatile

- Variable déclarée avec **`volatile`**
  - toujours mise à jour à partir de la mémoire partagée avant et après son utilisation

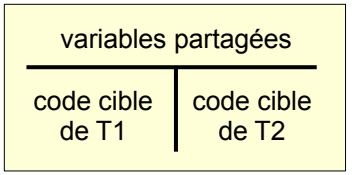
```
class Timer extends Thread {
    private volatile boolean stop = false;
    public void terminate() {
        stop = true;
    }
    public void run() {
        while (!stop) {
            doSomething();
            try { Thread.sleep(100); } ...
        }
    }
}
```

```
final Timer t = new Timer();
t.start();
Thread killer = new Thread(new Runnable() {
    public void run() {
        t.terminate();
    }
});
killer.start();
```

~~optimisation :~~
~~`stop` clonée dans un registre du thread~~
~~puis jamais mise à jour~~

La programmation multithread est contre intuitive

- Modification de l'ordre d'exécution




```
int a = 0; int b = 0;
```

`int x = a;`	`int y = b;`
`b = 2;`	`a = 1;`

(optimisation) transformation autorisée →

```
int a = 0; int b = 0;
```

`b = 2;`	`a = 1;`
`int x = a;`	`int y = b;`

intuition :

- termine par `b = 2;` → `(x,y) == (*,0)`
- termine par `a = 1;` → `(x,y) == (0,*)`
- donc : `x == 0` ou `y == 0`

exécutions possibles

```
int a = 0; int b = 0;
```

`b = 2;`	
`int x = a;`	
	`a = 1;`
	`int y = b;`

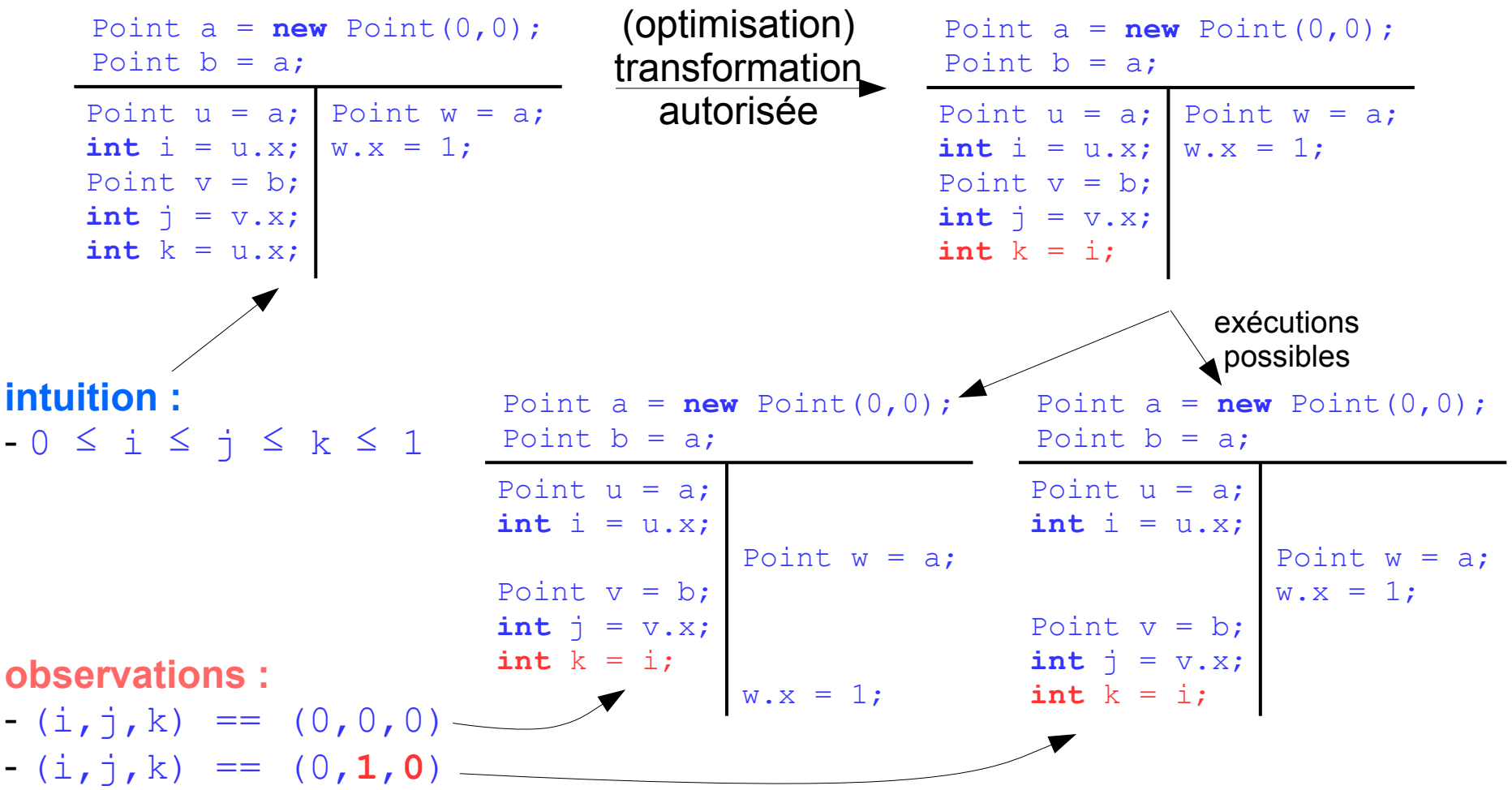
```
int a = 0; int b = 0;
```

`b = 2;`	
	`a = 1;`
`int x = a;`	
	`int y = b;`

observations :

- `(x,y) == (0,2)`
- `(x,y) == (1,2)`

- Élimination d'expressions



- Écritures non atomiques

```
boolean a = false;
int b = 0;
long c = 0;
```

`int x = 45;`
`b = x;`
`c = -x;`
`a = true;`
	`while (!a) {`
	`  Thread.yield();`
	`}`
	`print(b + " " + c);`

observations possibles :

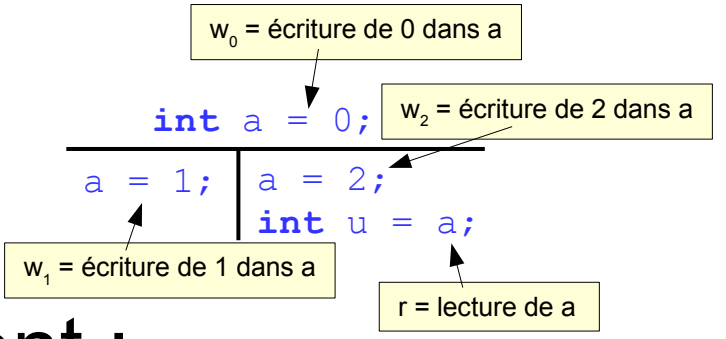
1 - affiche "45 -45"
2 - affiche "0 0"
3 - affiche "0 -45"
4 - affiche "45 0"
5 - boucle indéfiniment
6 - affiche "0 4294967251"
7 - affiche "0 -4294967296"
8 - affiche "45 4294967251"
9 - affiche "45 -4294967296"

```
  0 -> 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
-45 -> 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11010011

00000000 00000000 00000000 00000000 11111111 11111111 11111111 11010011 ->  4294967251
11111111 11111111 11111111 11111111 00000000 00000000 00000000 00000000 -> -4294967296
```

Modèle de mémoire Java

- **MMJ**
  - décrit les interactions entre threads et mémoire
  - définit un ordre partiel (R, AS, T) sur les actions
    - relation d'antériorité : si x ≺ y alors
      - x est atomique relativement à y
      - x est exécutée avant y
      - l'effet de x est observable depuis y

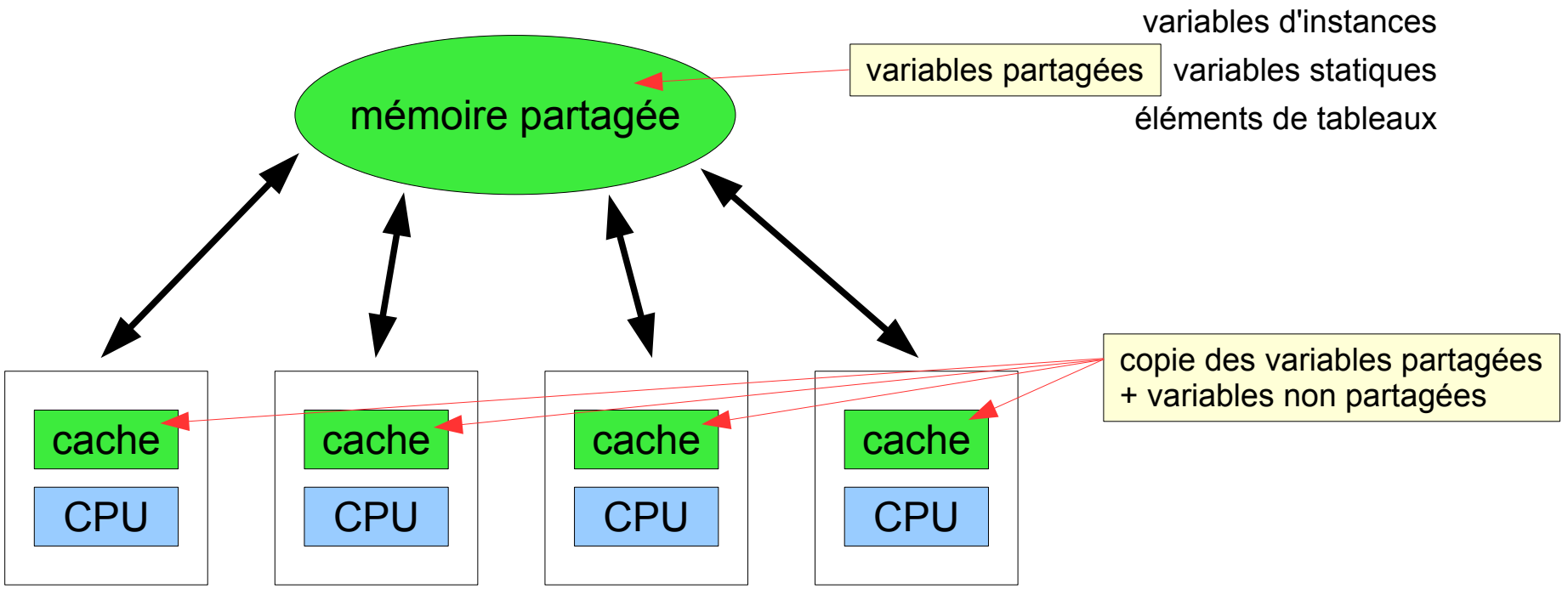


- Les **actions** sont essentiellement :
  - lecture / écriture de variables partagées
  - prise / libération de verrous
  - démarrage / terminaison de threads

Le MMJ garantit que :
$w_0 \prec w_1$
$w_0 \prec w_2 \prec r$
on en déduit que $u \neq 0$

Accès à la mémoire partagée

- Pour le MMJ les accès à la mémoire partagée sont correctement modélisés ainsi :



Garanties d'atomicité

Règle A1 (accès aux variables) :
l'accès (R/W) à toute variable d'un type de taille inférieure ou égale à 32 bits est atomique

```
int a = 0;
```

`a = 2;`	`int u = a;`

valeur de u :
0 ou 2

```
long a = 0;
```

`a = -45;`	`long u = a;`

valeur de u :
0 ou -45,
ou 4294967251
ou -4294967296

Règle A2 (accès aux variables volatiles) :
l'accès (R/W) à toute variable volatile est atomique

```
volatile long a = 0;
```

`a = -45;`	`long u = a;`

valeur de u :
0 ou -45

Garanties d'atomicité

Règle A3 (exécution des blocs synchronisés) :
deux blocs synchronisés sur un même verrou constituent des opérations atomiques entre elles

```
int a = 0;
int b = 0;
```

`a = 1;`	`a = 2;`
`b = 2;`	`b = 1;`
	`int u = a;`
	`int v = b;`

valeur de (u, v) :
(1, 1) ou (1, 2)
ou (2, 1) ou (2, 2)

```
int a = 0;
int b = 0;
```

`synchronize (M) {`	`synchronize (M) {`
`  a = 1;`	`  a = 2;`
`  b = 2;`	`  b = 1;`
`}`	`}`
	`synchronized (M) {`
	`  int u = a;`
	`  int v = b;`
	`}`

valeur de (u, v) :
(1, 2) ou (2, 1)

Définition de la relation d'antériorité

```
int a = 0;
```

```
a = 1;
int u = a;
```

valeur de u :
1

Règle H1 (ordonnancement dans un thread) :
si une action x précède une action y dans le code cible d'un thread alors $x \prec y$

Règle H2 (transitivité) :
si $x \prec y$ et $y \prec z$ alors $x \prec z$

Définition de la relation d'antériorité

Règle H3 (verrous) :
si x est une libération de verrou et y une acquisition ultérieure du même verrou
alors x ≺ y

release —*succession chronologique*→ acquire (*lié à ce release*)

Règle H4 (variables volatiles) :
si x est une écriture de variable volatile et y une lecture ultérieure de la même variable
alors x ≺ y

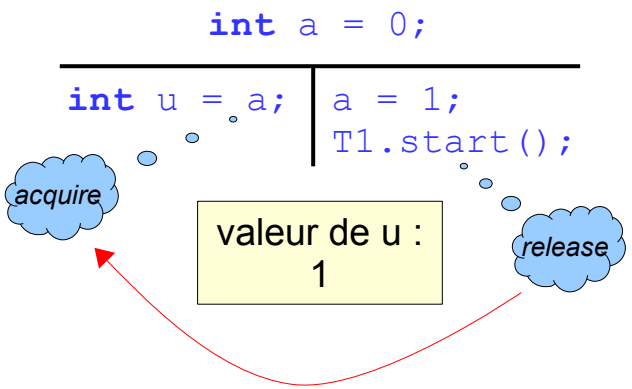
```
int a = 0;
int b = 0;
```

`b = 1;`	`int u = 0;`
`synchronize (M) {`	
`  a = 1;`	
`}`	`synchronize (M) {`
	`  u = a;`
	`}`
	`int v = b;`

release *acquire*

valeur de (u, v) sous l'hypothèse de succession chronologique indiquée : (1, 1)

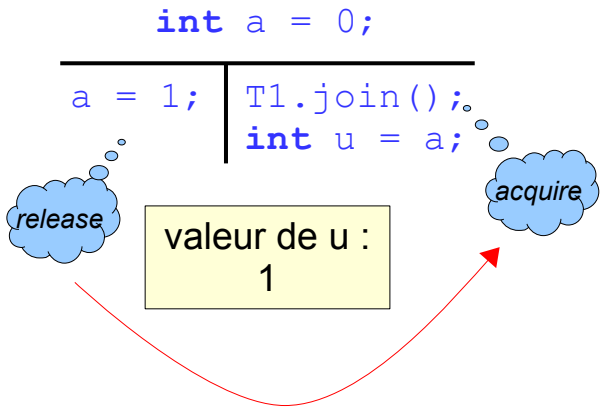
Définition de la relation d'antériorité

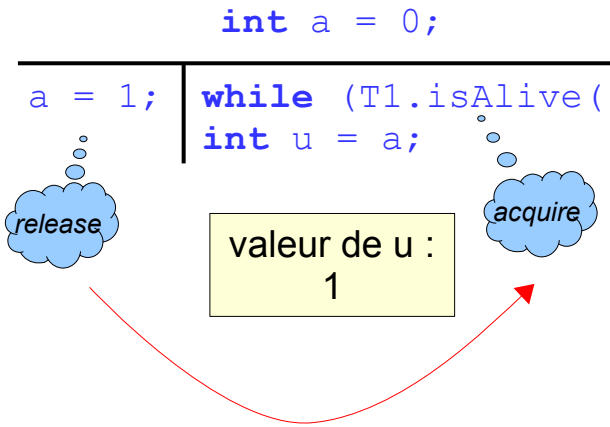


Règle H5 (démarrage d'un thread) :
si x est le démarrage d'un thread et y la première action du code cible de ce thread
alors x ≺ y

release — *succession chronologique* → acquire (*lié à ce release*)

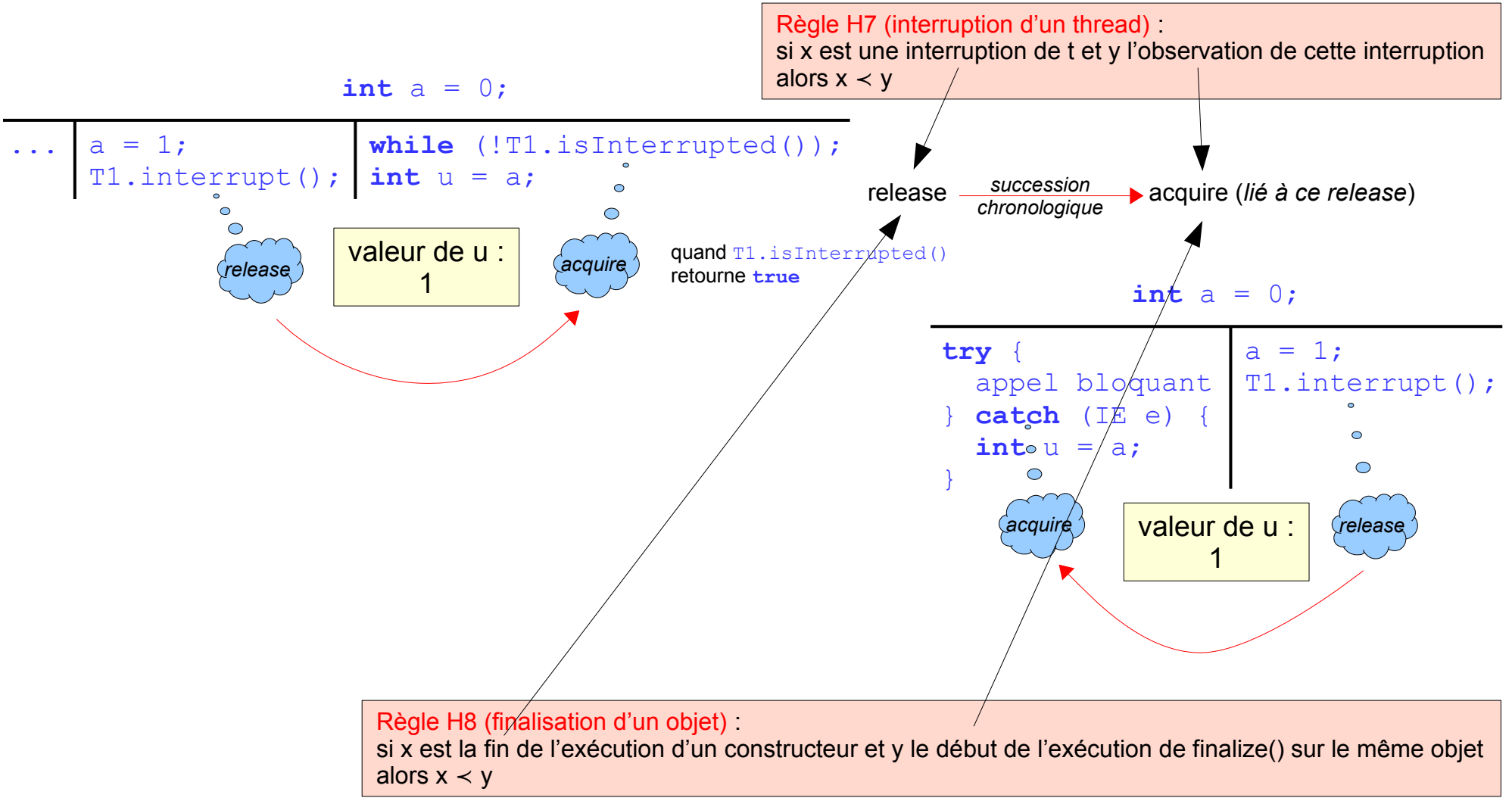
Règle H6 (terminaison d'un thread) :
si x est la dernière action du code cible d'un thread et y une détection de cette terminaison
alors x ≺ y





quand `T1.isAlive()` retourne **`false`**

Définition de la relation d'antériorité



Mise en garde : initialisation d'attributs non finaux

```
X a = null;
```

`a = new X();`	`int u = -1;`
	`if (a != null)`
	`  u = a.n;`

valeur de u :
-1, 0 ou 1

```
public class X {
    public int n;
    public X() { n = 1; }
}
```

Garanti par le MMJ :
- initialisation standard des attributs

```
X a = null;
```

`a = new X();`	`int u = -1;`
	`if (a != null)`
	`  u = a.n;`

valeur de u :
-1 ou 1

```
public class X {
    public final int n;
    public X() { n = 1; }
}
```

Garanti par le MMJ :
- initialisation complète des attributs **`final`**

Mise en garde : volatilité et atomicité

- Attention à ++
  - lecture + écriture → opération non atomique
  - même sur une variable volatile :

```
volatile int a = 0;
```

`int u = ++a;`	`int v = ++a;`

valeur de (u, v) :
(2, 1)
(1, 2)
(1, 1)

!