

# Concurrence

# sections critiques

Rémi Forax

# Qu'affiche ce code ?

```
public class Counter {
    private int value;

    public void add10000() {
        for(int i = 0; i < 10_000; i++) {
            value++;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable runnable = counter::add10000;
        Thread t1 = new Thread(runnable);
        Thread t2 = new Thread(runnable);
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println(counter.value);
    }
}
```

# Qu'affiche ce code ? (2)

plusieurs exécutions affichent (suivant la météo?) :

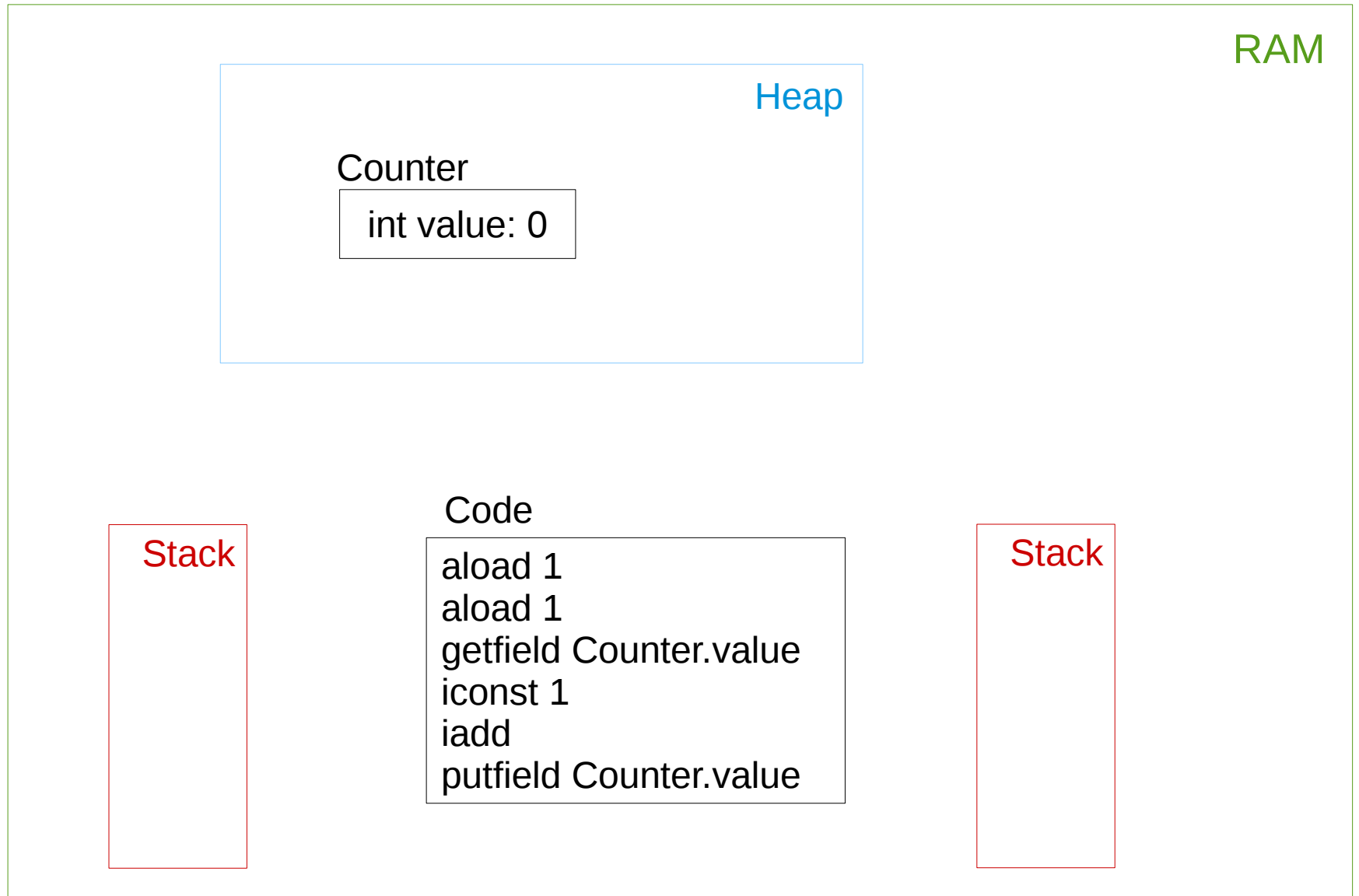
10 363 <-- euh ?

20 000 <-- ahh !

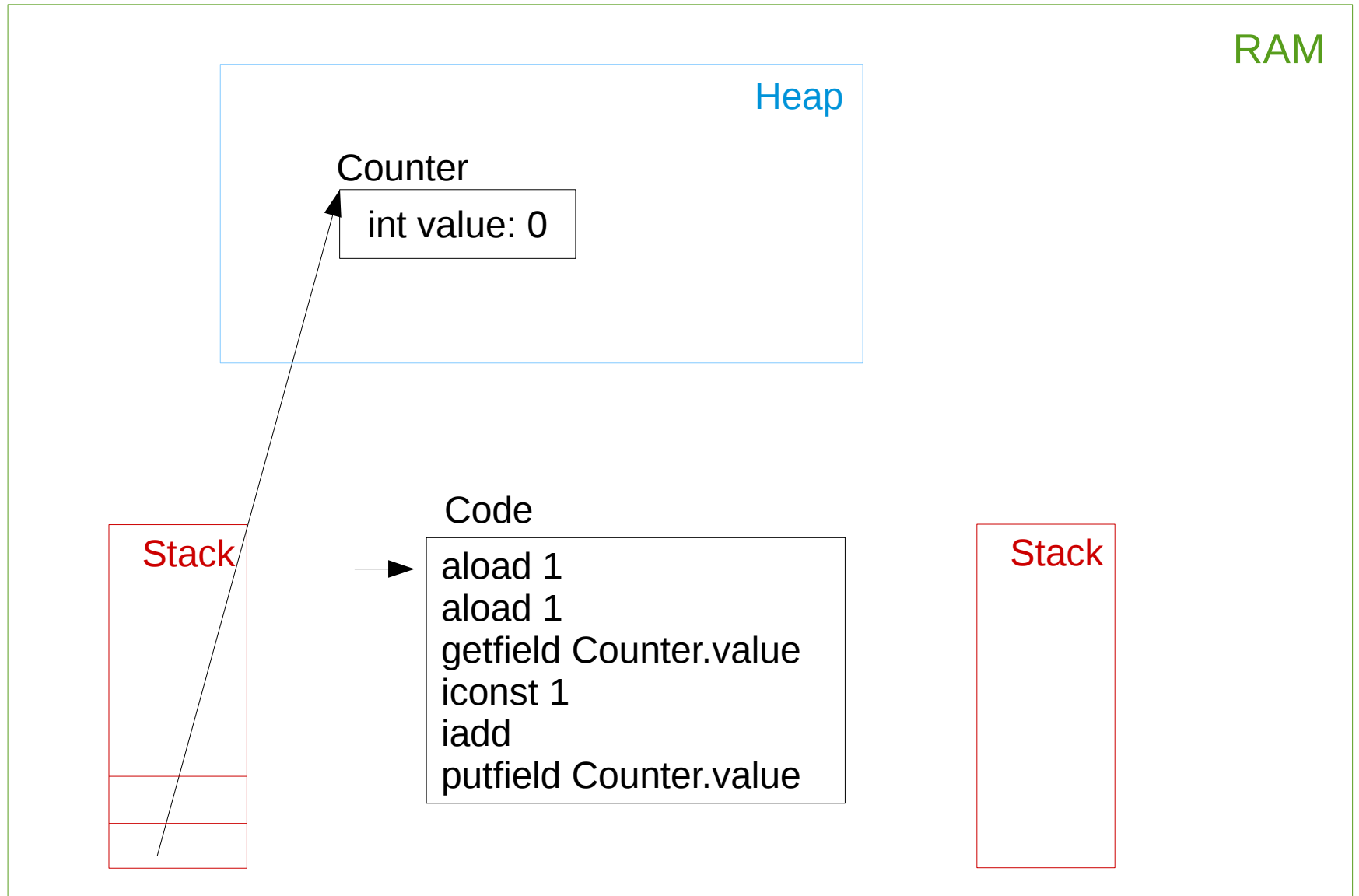
14 957 <-- re-euh ?

pourquoi ?

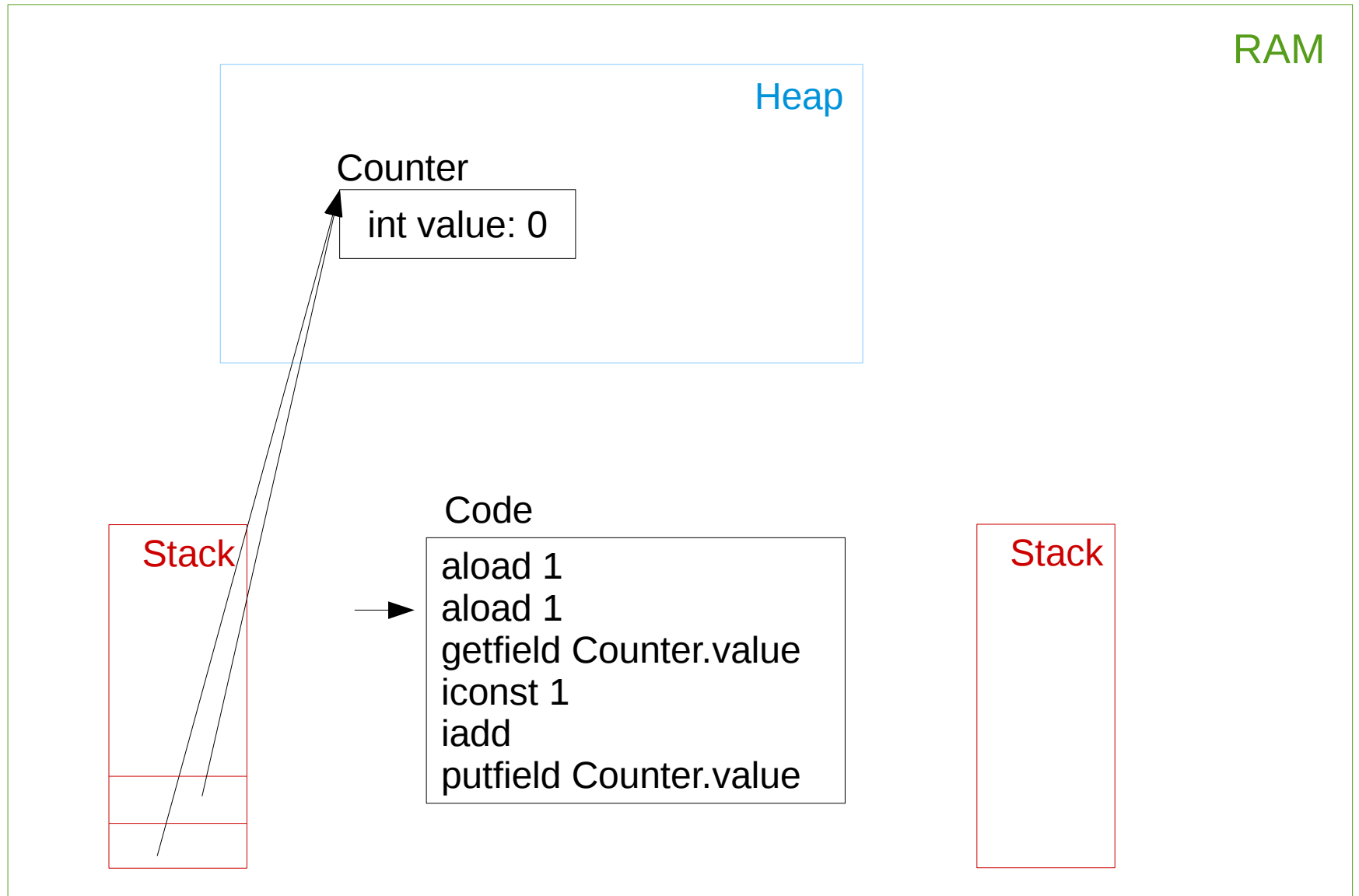
# En mémoire (1/7)



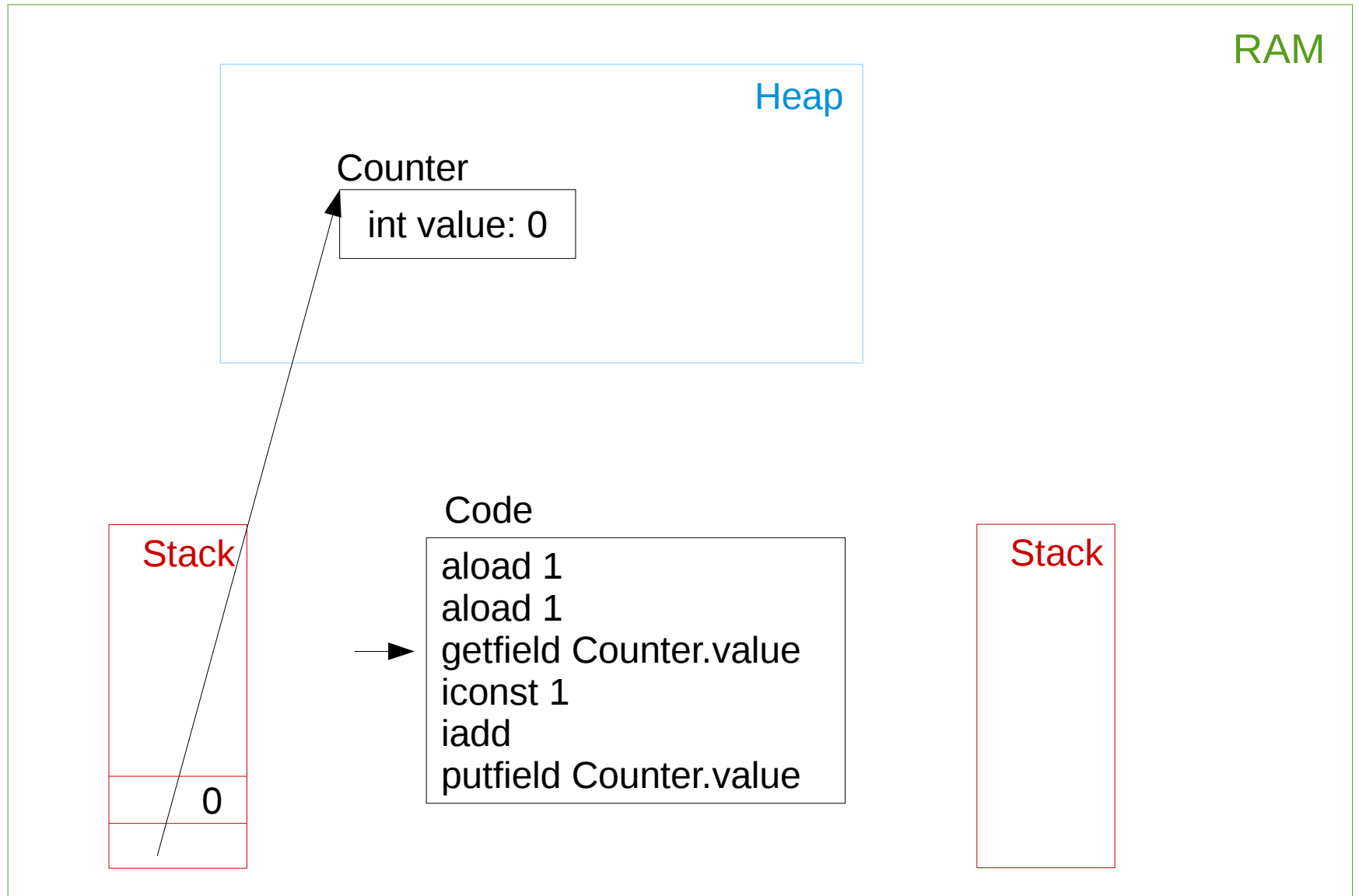
# En mémoire (2/7)



# En mémoire (3/7)

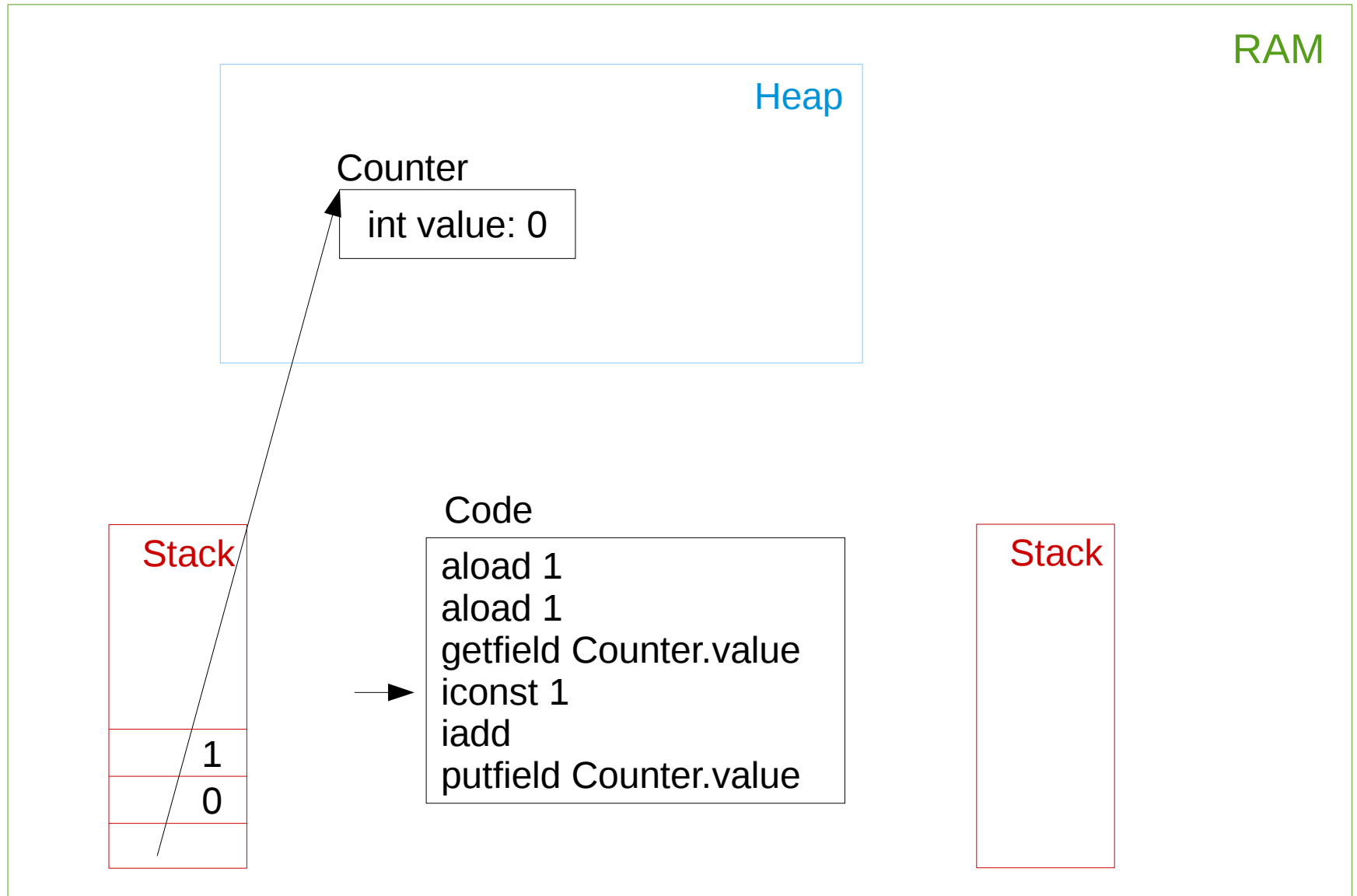


# En mémoire (4/7)



# En mémoire (5/7)

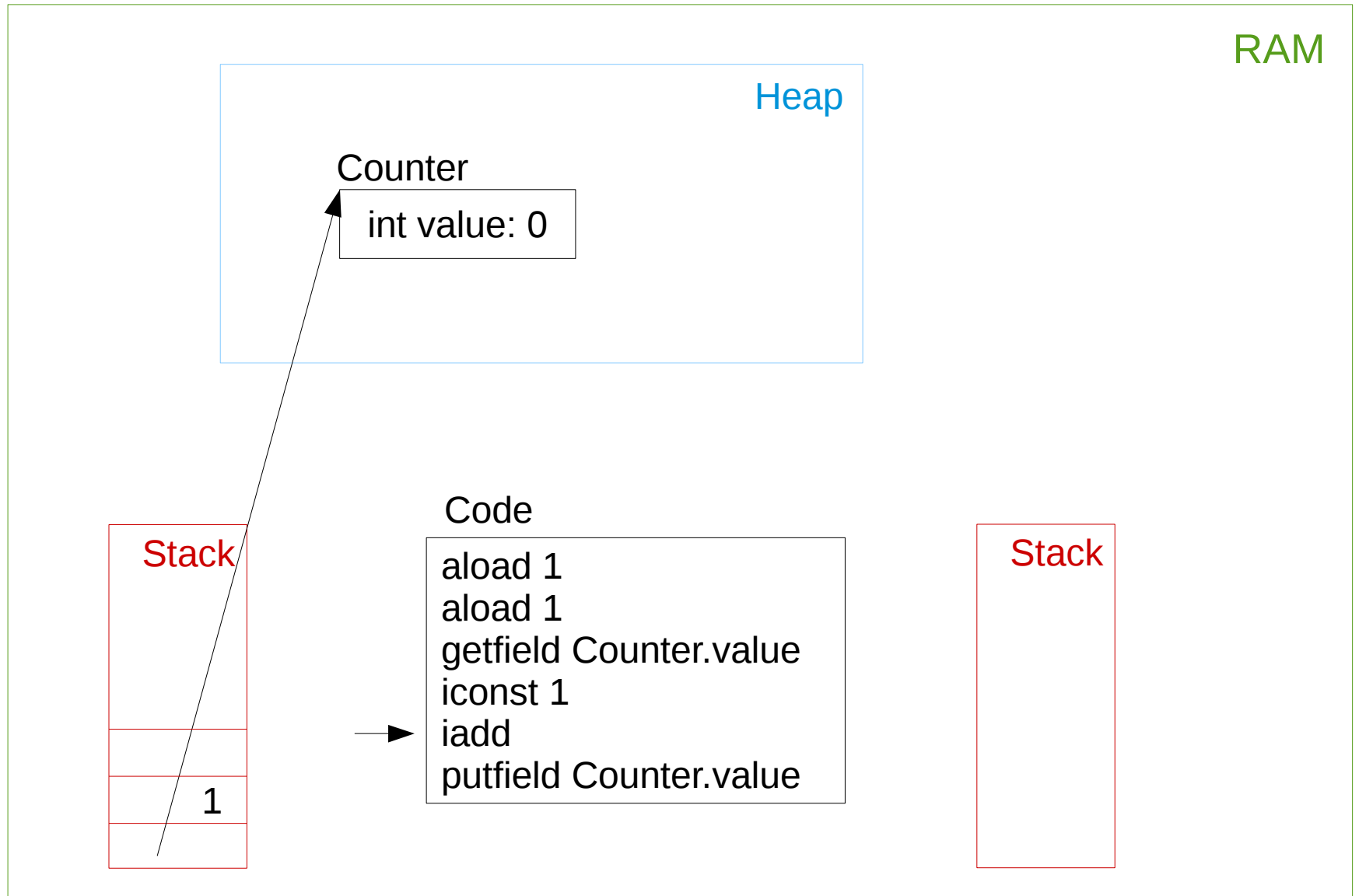
RAM



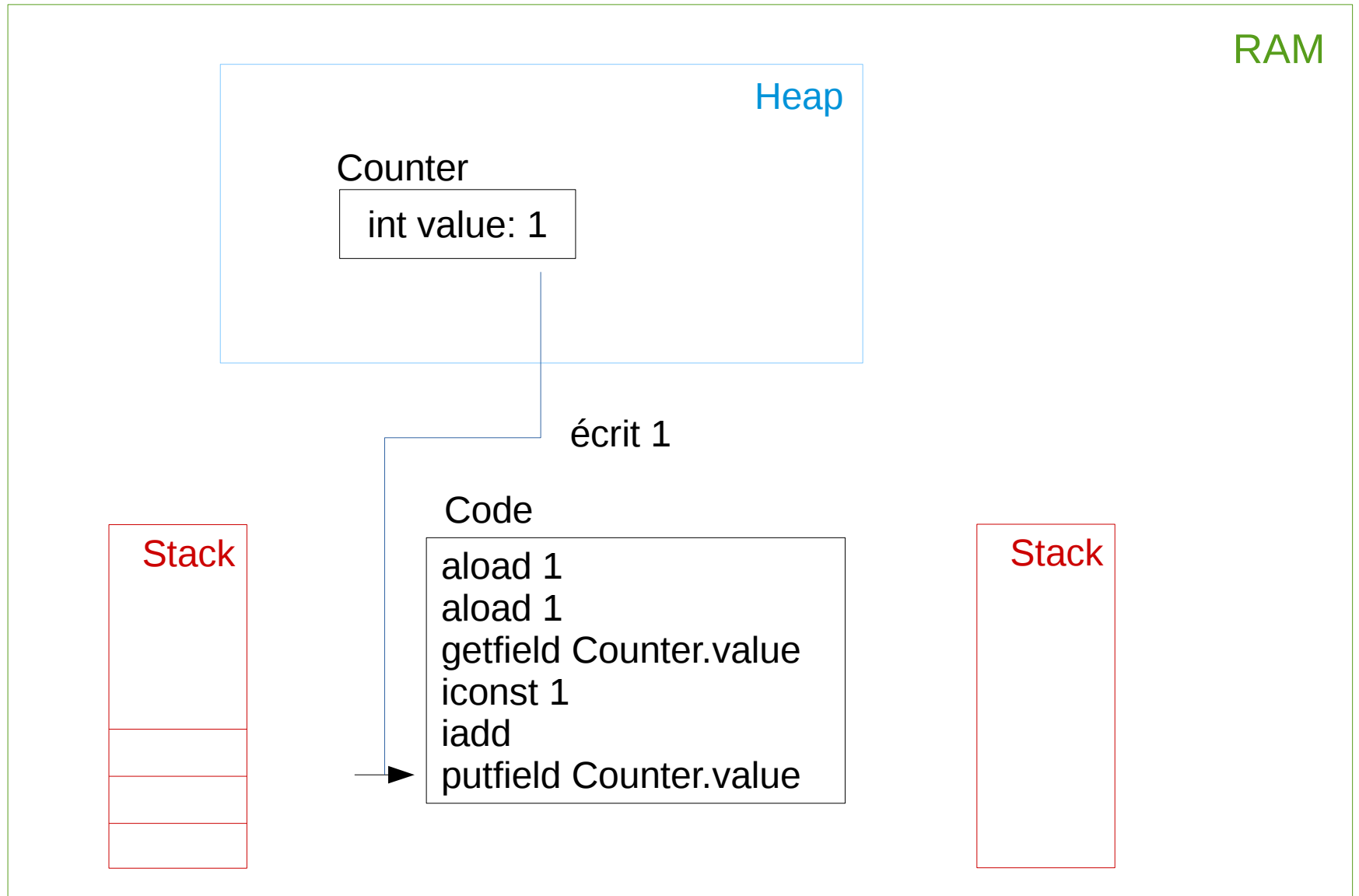


# En mémoire (6/7)

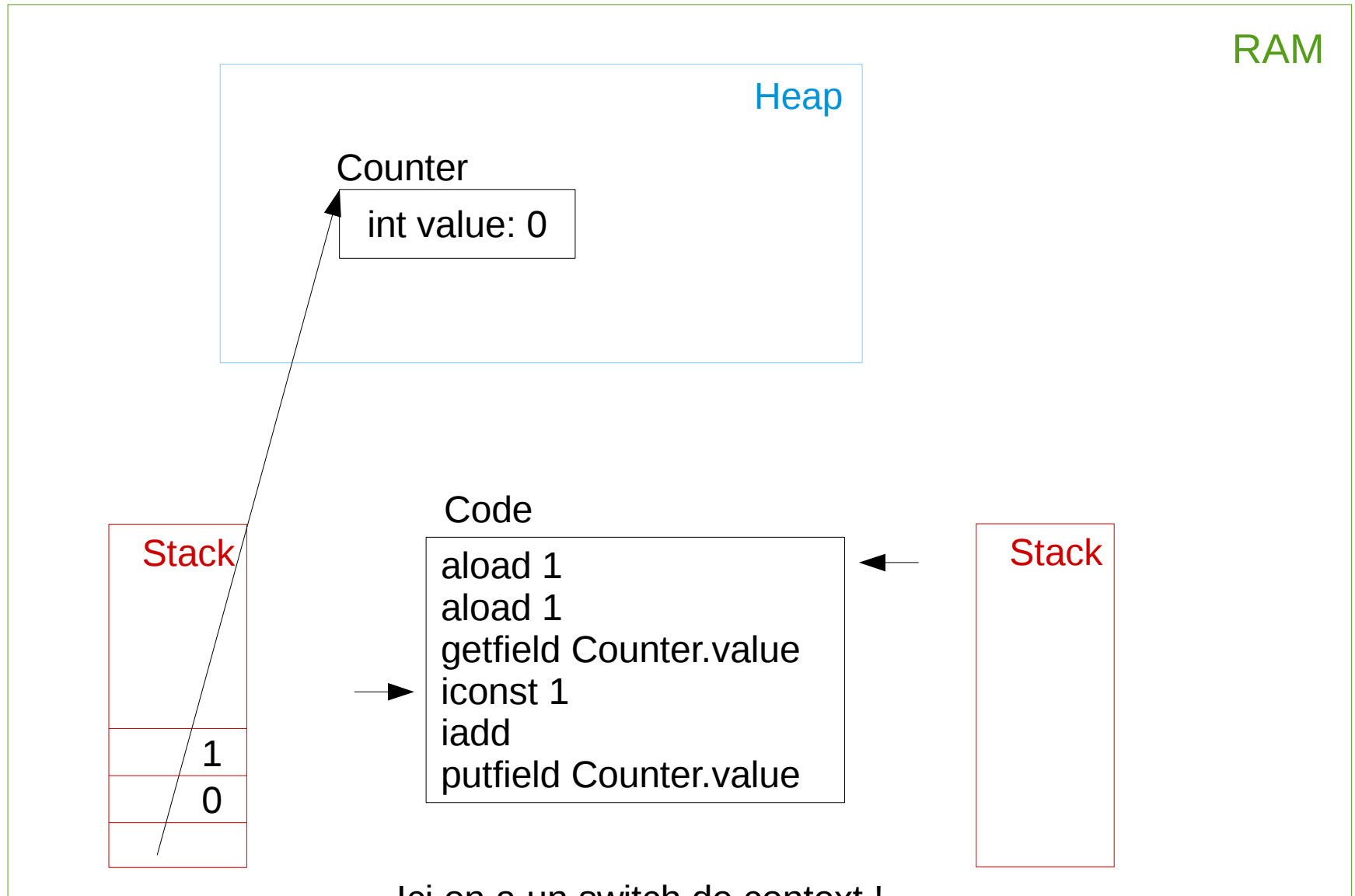
RAM



# En mémoire (7/7)

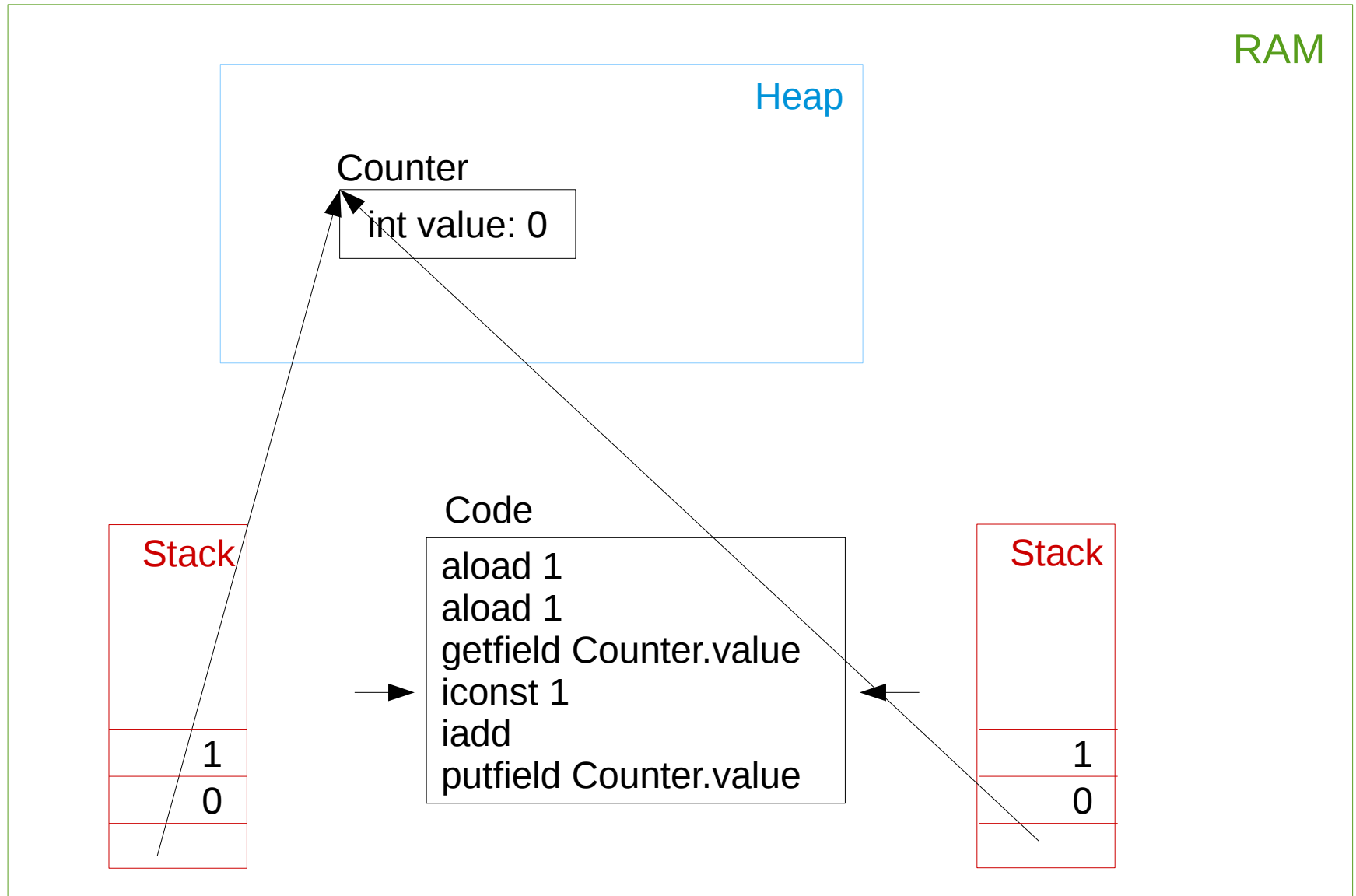


# En mémoire (5/7) avec 2 threads

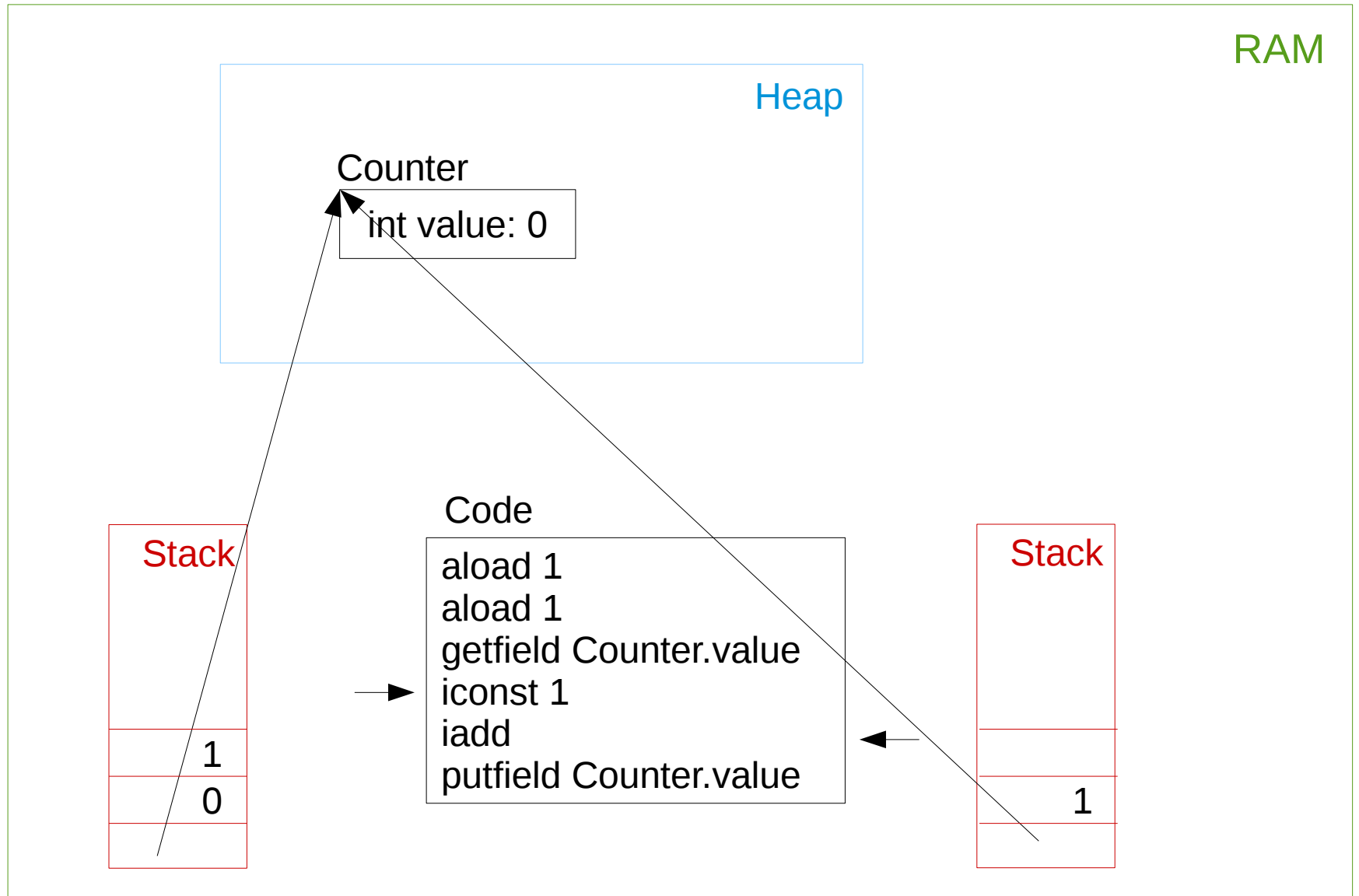


Ici on a un switch de context !

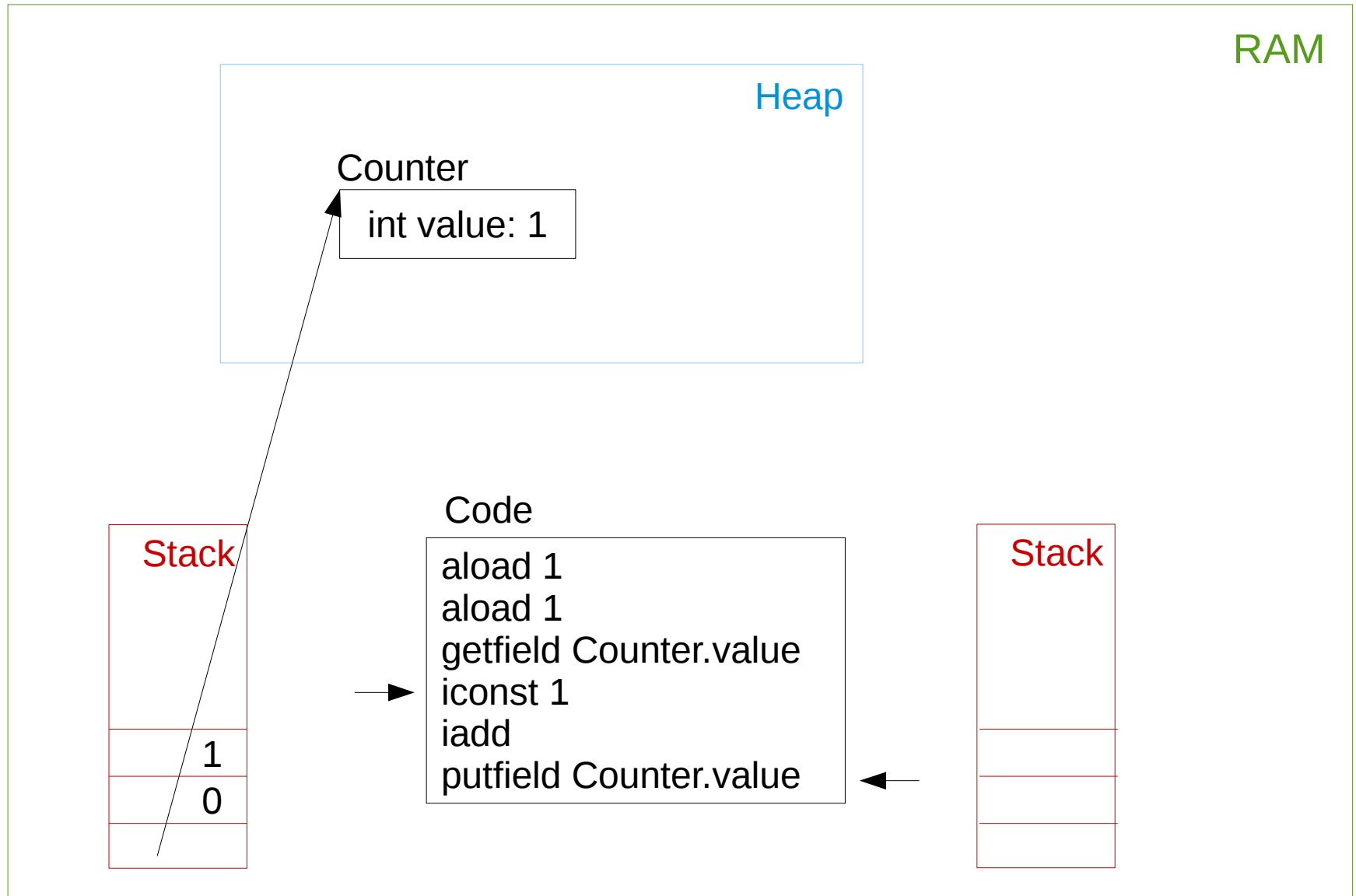
# En mémoire avec 2 threads (2/7)



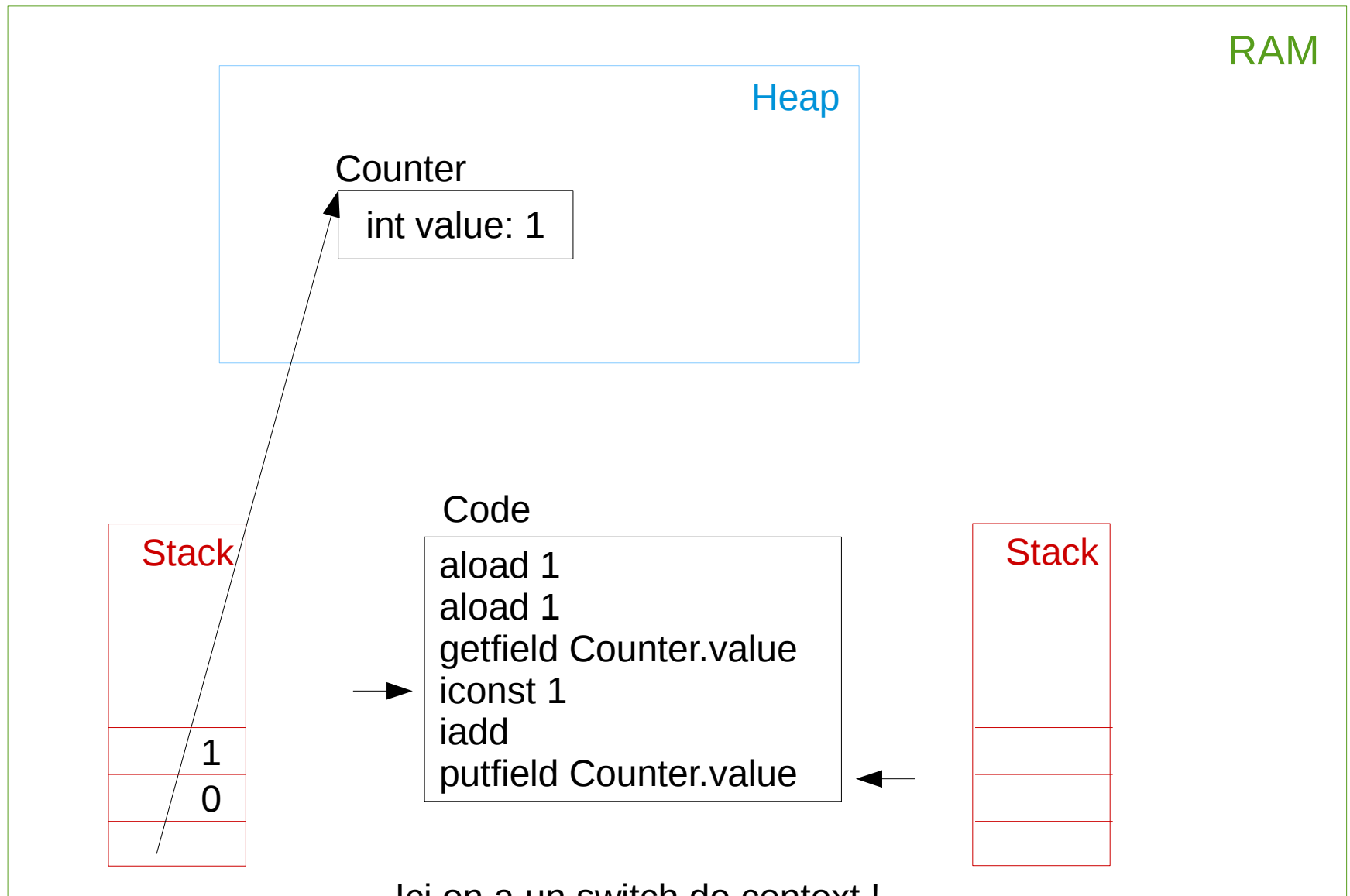
# En mémoire avec 2 threads (3/7)



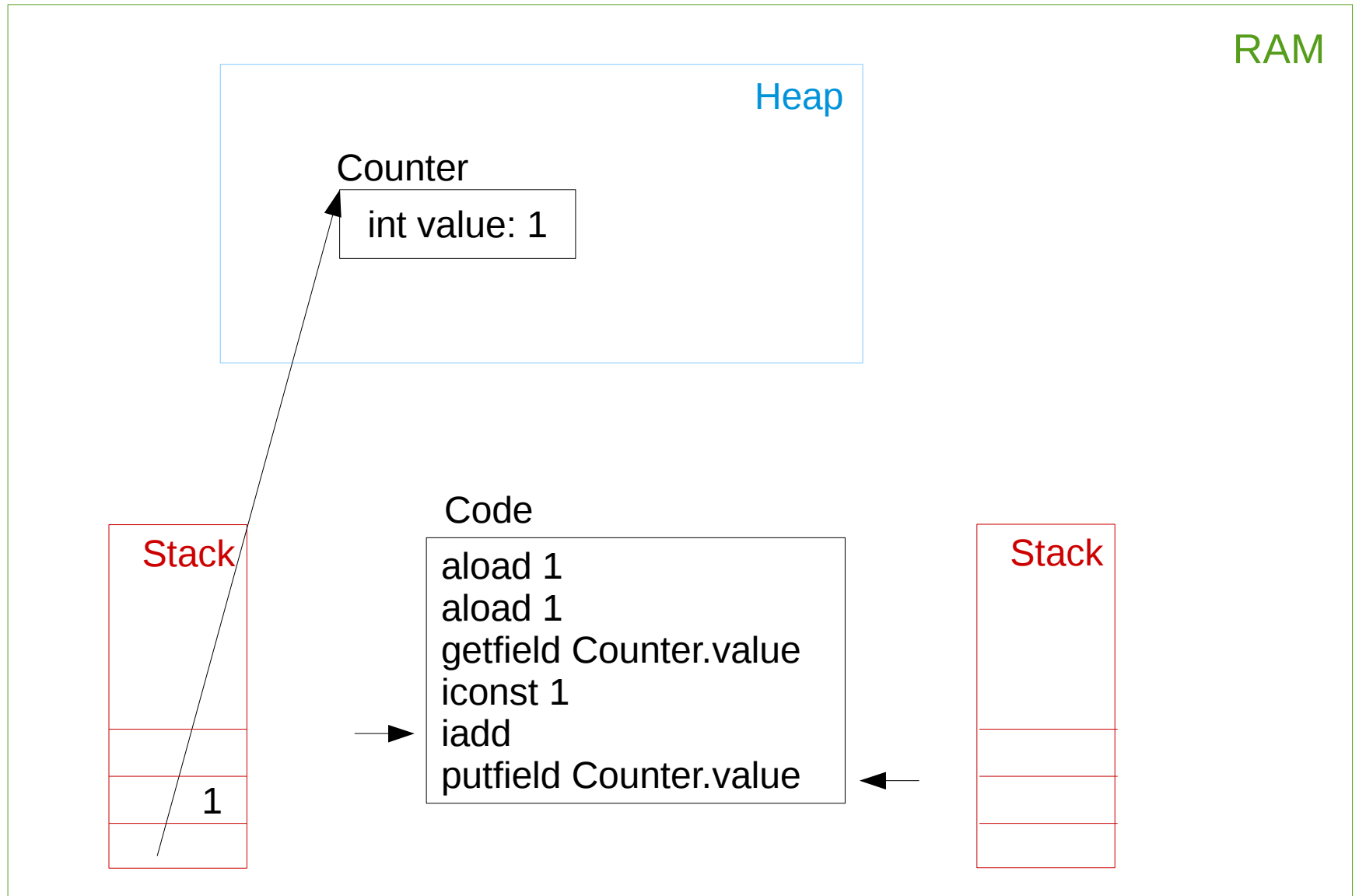
# En mémoire avec 2 threads (4/7)



# En mémoire avec 2 threads (5/7)

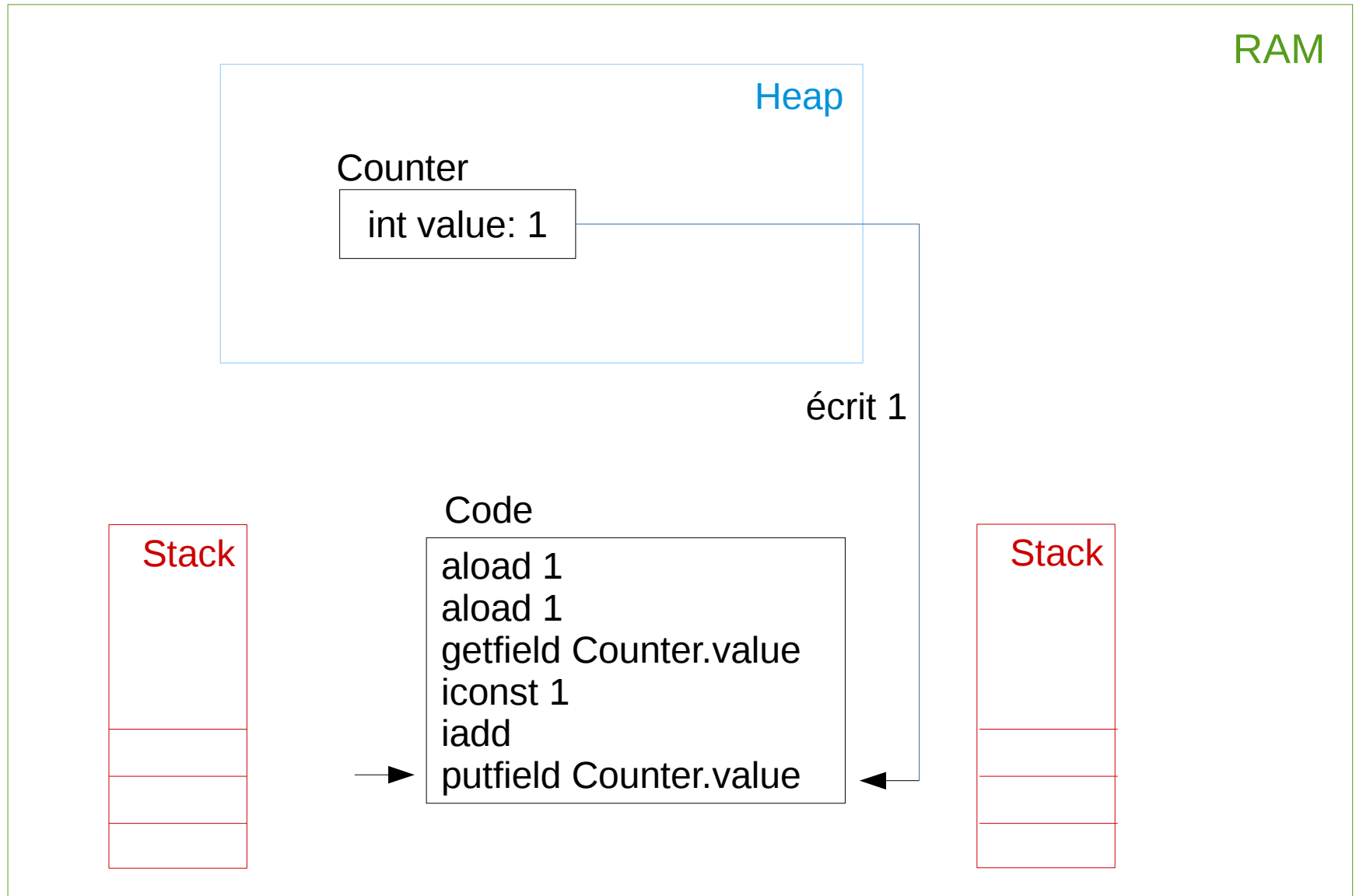


# En mémoire avec 2 threads (6/7)





# En mémoire avec 2 threads (7/7)



# Explication

Si une thread est dé-schedulé alors qu'elle a déjà lu la valeur alors on perd des incrémentation

Même si on écrit `i++` en Java (ou en C), l'opération n'est pas atomique pour le CPU, c'est une lecture puis une écriture

# Conspiration !

En fait, le scheduler n'est pas le seul facteur qui montre que l'opération n'est pas atomique

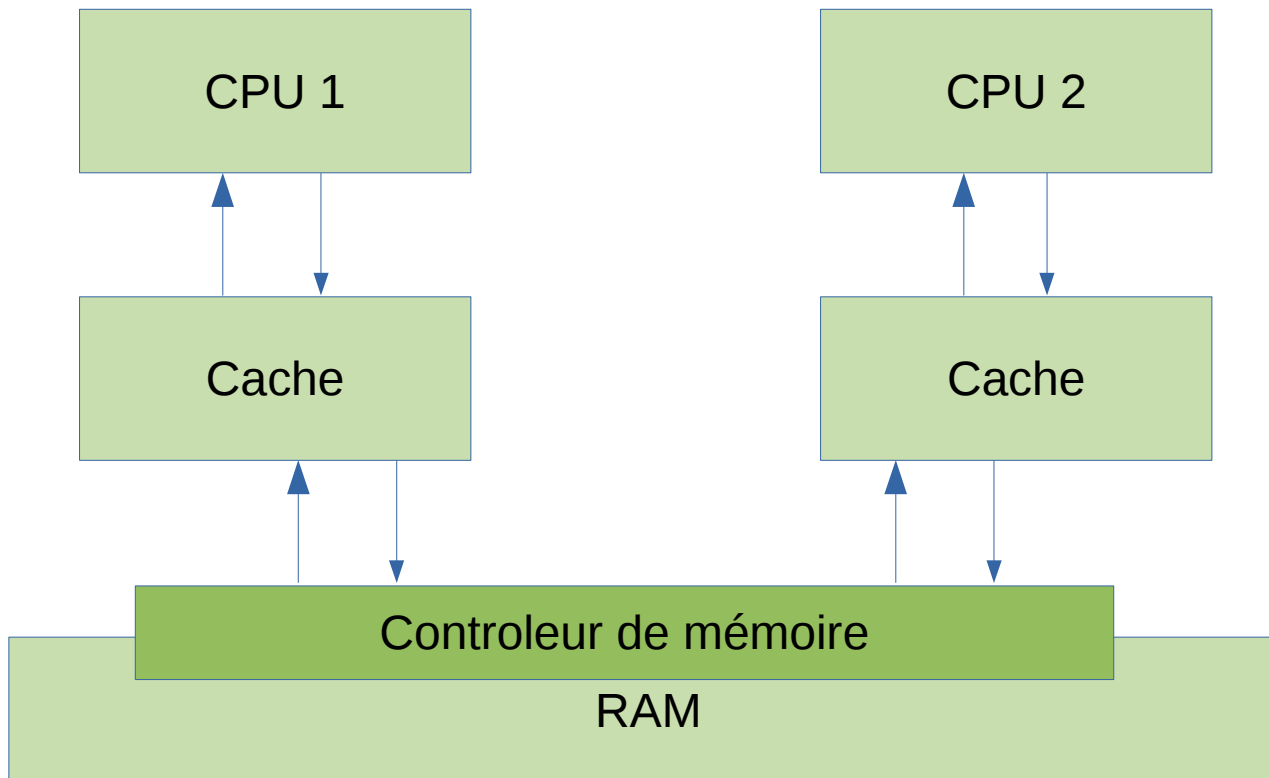
Si on regarde un peu plus,

- L'architecture CPU (caches !)
- Optimisation du compilateur (C) ou JIT (Java)

montre aussi le même problème

# Architecture CPU

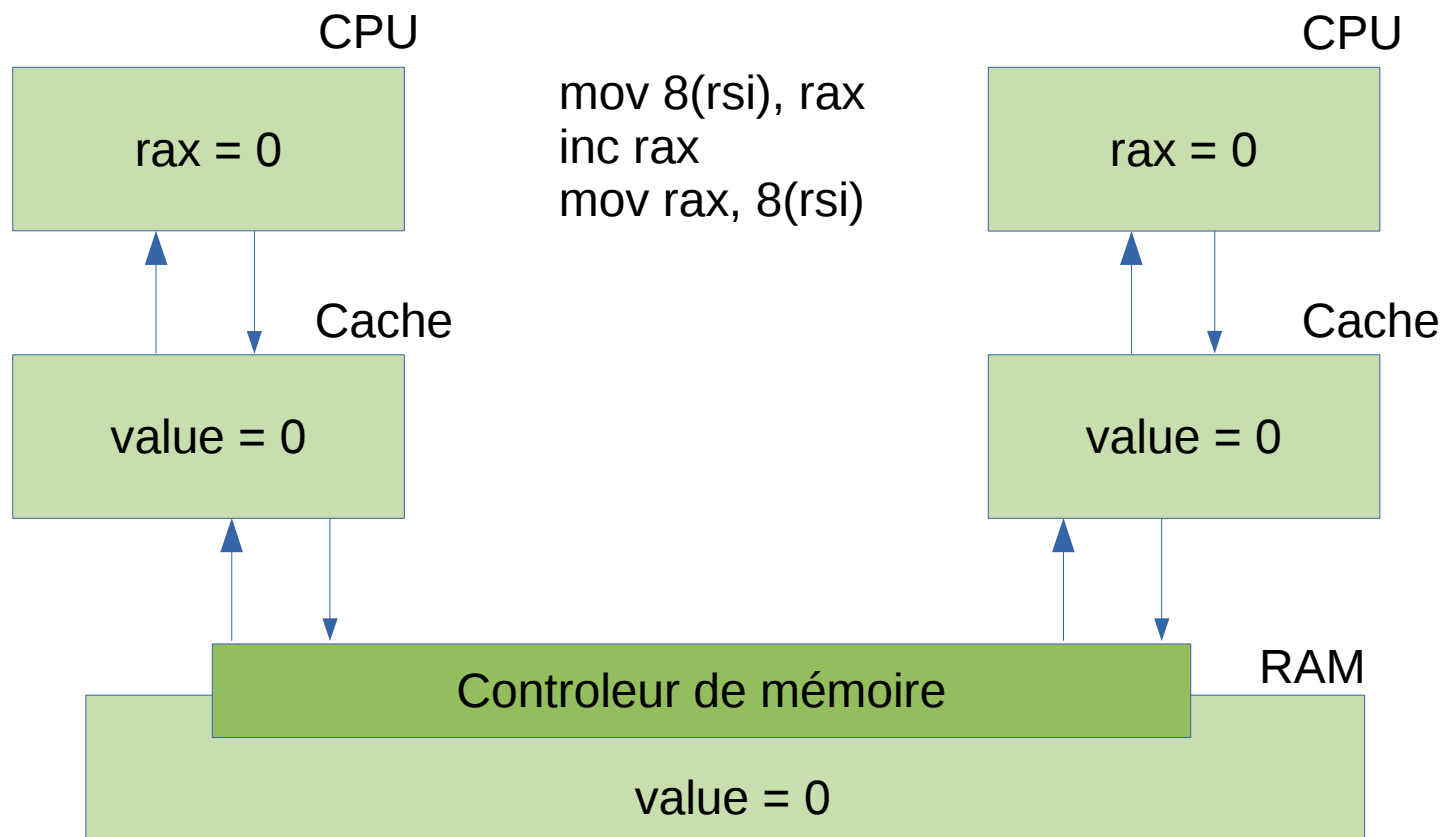
Les CPUs modernes ont un cache (en fait plusieurs en cascade) entre la RAM et les registres internes



# Architecture CPU (2)

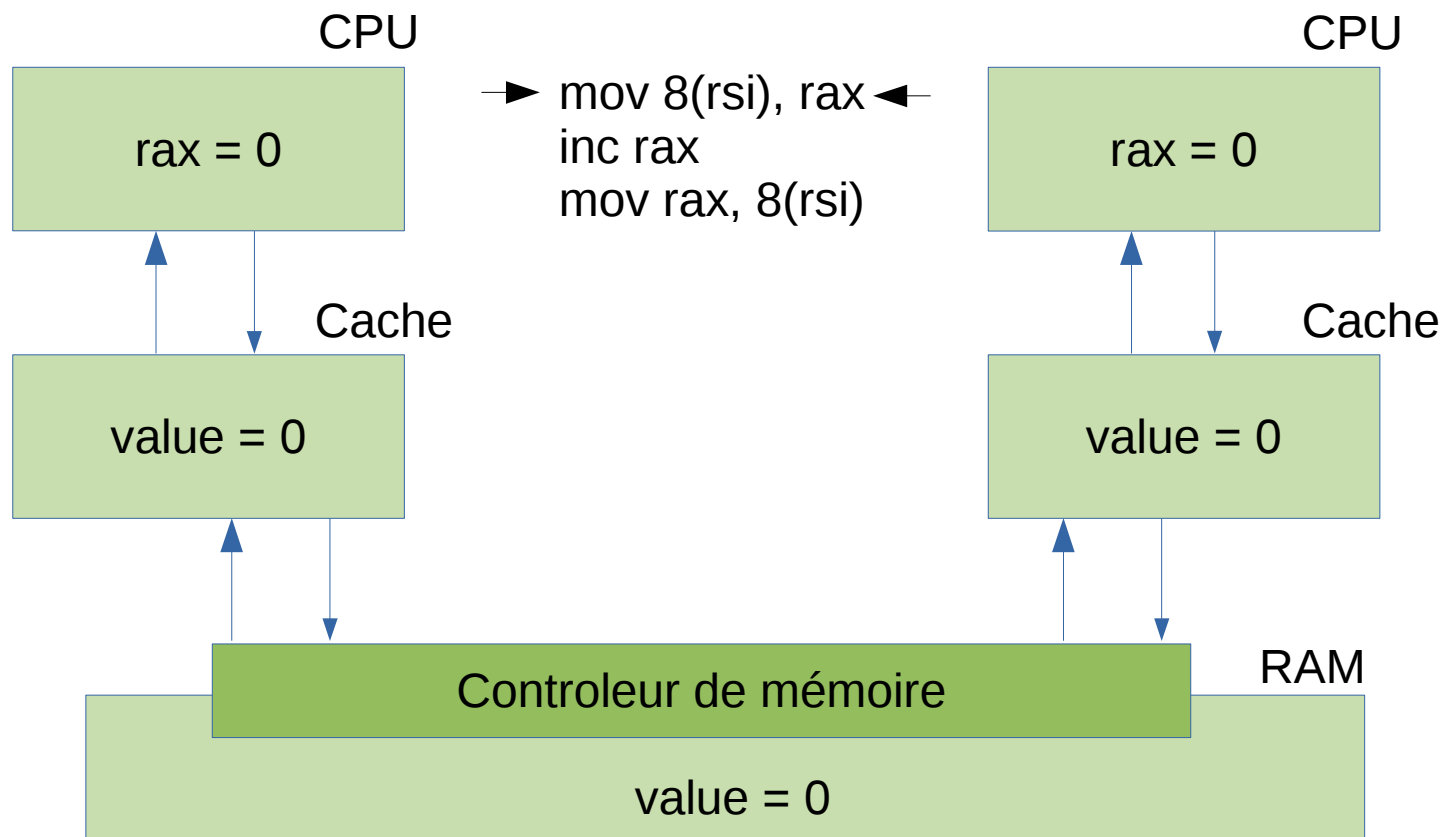
rsi contient l'adresse de 'counter'

8 est l'offset du champ 'value'



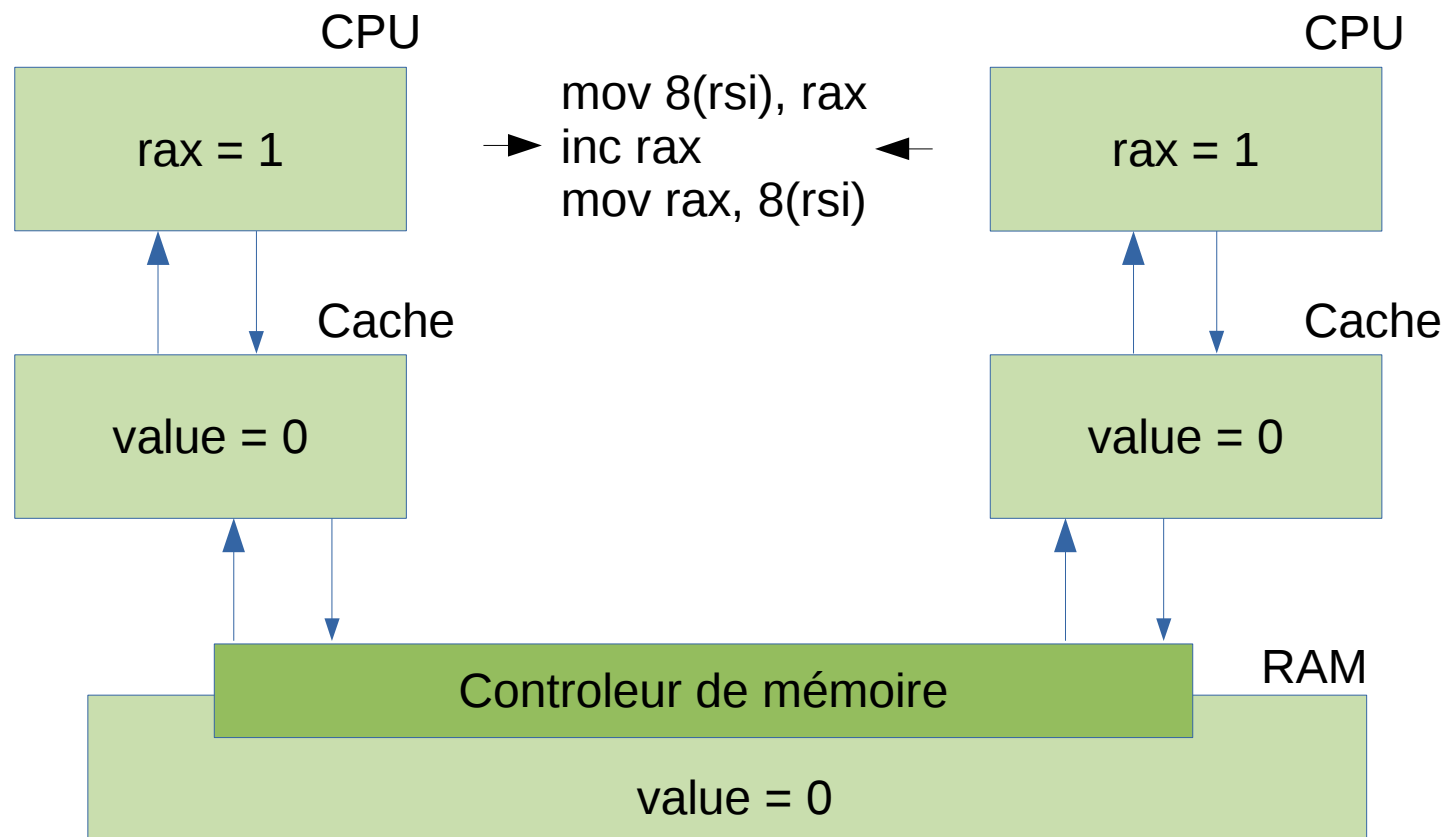
# Architecture CPU (2)

Supposons que les 2 CPUs exécutent les instructions en même temps



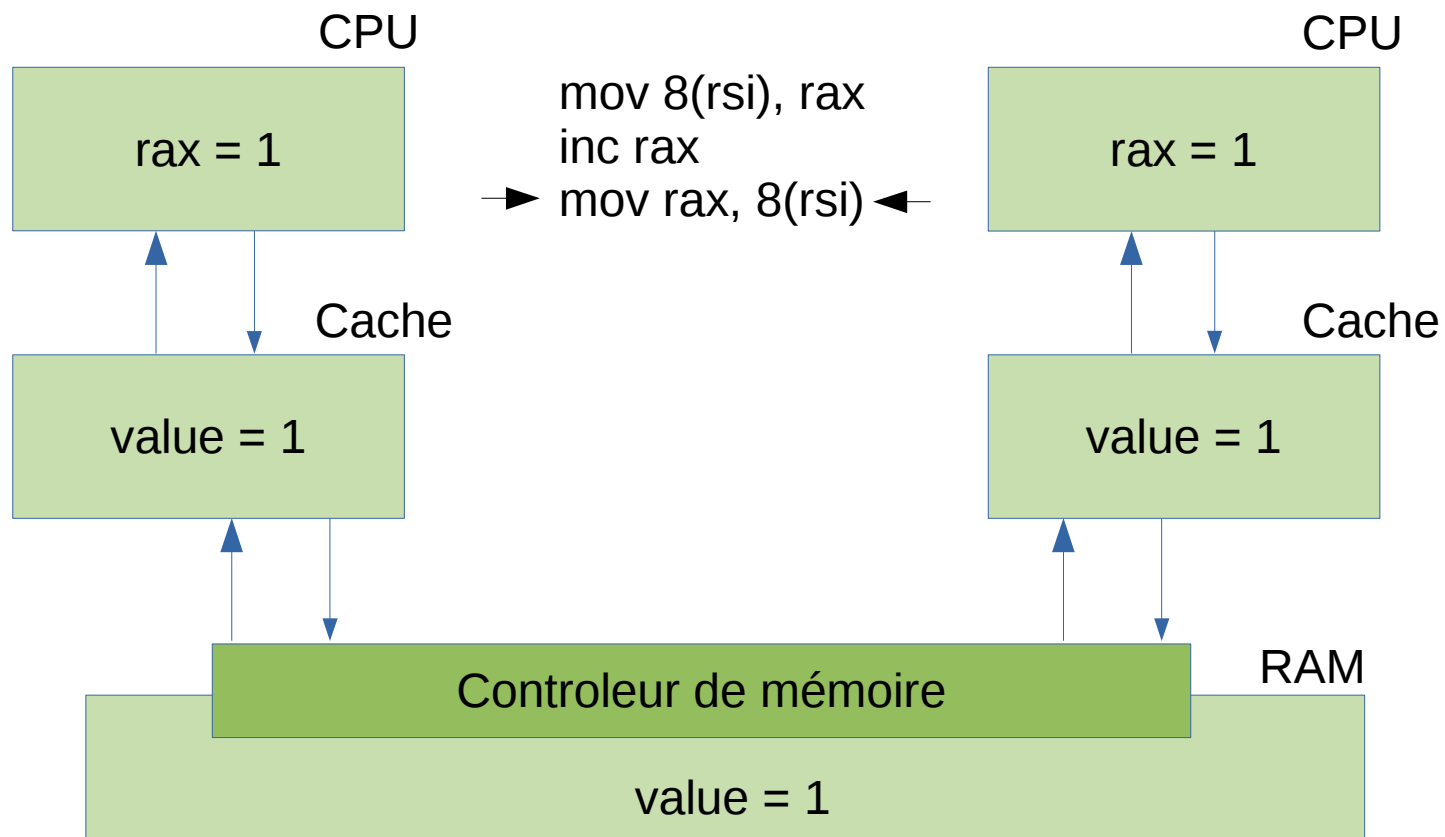
# Architecture CPU (2)

Supposons que les 2 CPUs exécutent les instructions en même temps



# Architecture CPU (2)

Les deux écrivent 1 dans la RAM à la même adresse !





# Architecture CPU

Le fait d'avoir 2 CPUs et des caches ou des registres spécifiques est suffisant pour “perdre” des incrémentations

Même problème d'avec le scheduler !

# Optimiseur de code ou JIT

Le code que l'on écrit dans un langage de haut niveau (C, Java) est rarement celui qui est exécuté

L'optimiseur de code transforme le code pour qu'il soit plus efficace

Par exemple :

```
public void add10000() {  
    for(int i = 0; i < 10_000; i++) {  
        this.value++;  
    }  
}
```

# Introduction d'une variable locale

devient :

```
public void add10000() {  
    int localValue = this.value;  
    for(int i = 0; i < 10_000; i++) {  
        localValue++;  
    }  
    this.value = localValue;  
}
```

l'aller retour vers la RAM est fait **1 seul fois**

# En résumé

Par défaut, une instruction ou un code ne s'exécute pas de façon atomique

Le code d'une thread s'exécute sans connaissance de l'état d'exécution des autres threads

Si on a plusieurs threads, avec au moins une lecture/écriture, on peut voir les effets de bords de cette non-atomicité

# Atomique -> section critique

Comment faire pour que `value++` soit atomique ?

- Impossible !  
`value++` n'est pas atomique pour le CPU

Tricher !

- Empêcher les lectures tant que l'on a pas fini l'écriture !

# Section critique

```
public class Counter {  
    private int value;  
  
    public void add10000() {  
        for(int i = 0; i < 10_000; i++) {  
            // début section critique  
            value = value + 1;  
            // fin section critique  
        }  
    }  
}
```

une seule thread peut rentrer dans la section critique à la fois !

# Section critique

Et si on a plusieurs code ?

```
public class Counter {  
    private int value;  
  
    public void add10000() {  
        for(int i = 0; i < 10_000; i++) {  
            // début section critique  
            value = value + 1;  
            // fin section critique  
        }  
    }  
  
    public void add1() {  
        // début section critique  
        value = value + 1;  
        // fin section critique  
    }  
}
```

Il faut un moyen d'indiquer que les sections critiques sont reliés ?

# Section critique et moniteur

On utilise un objet appelé moniteur qui sert de point de rendez-vous

```
public class Counter {  
    private int value;  
    private final Object unObjet = new Object();  
  
    public void add10000() {  
        for(int i = 0; i < 10_000; i++) {  
            // début section critique (unObjet)  
            value = value + 1;  
            // fin section critique (unObjet)  
        }  
    }  
  
    public void add1() {  
        // début section critique (unObjet)  
        value = value + 1;  
        // fin section critique (unObjet)  
    }  
}
```

Incantation magique,  
on verra après pourquoi ?



# Section critique

En Java, on utilise un bloc **synchronized**.

```
public class Counter {
    private int value;
    private final Object monitor = new Object();

    public void add10000() {
        for(int i = 0; i < 10_000; i++) {
            synchronized (monitor) {
                value = value + 1;
            }
        }
    }

    public void add1() {
        synchronized (monitor) {
            value = value + 1;
        }
    }
}
```

# Synchronized et jeton

Lorsqu'une thread entre dans un bloc synchronized, elle prend **un jeton associé à l'objet moniteur**, lorsqu'elle sort du bloque synchronized, elle rend le jeton.

**Une seule thread peut avoir le jeton à la fois !**

```
public class Counter {  
    private int value;  
    private final Object monitor = new Object();  
  
    public void add10000() {  
        for(int i = 0; i < 10_000; i++) {  
            synchronized (monitor) {  
                value = value + 1;  
            }  
        }  
    }  
}
```

# Synchronized et jeton

Le fait de prendre ou relacher un jeton est inerte pour l'objet en Java.

Cela ne modifie pas le comportement de objet

L'objet sert juste de référence commune, de point de rendez vous !

L'objet utilisé par synchronized

- est un Object donc pas un primitif
- pas null !

# Synchronized et interning

Interning est une technique utilisée par le JDK pour éviter d'allouer plusieurs fois un objet représentant la même chose

- Par exemple,
  - Les String literals: "foo"
  - Les `java.lang.Integer` avec `Integer.valueOf` ou auto-boxing
    - `Object o = 3; // appel implicitement Integer.valueOf(3)`
  - Les classes `java.lang.Class` avec `".class"` ou `Class.forName()`

On utilise pas ces objets en tant que moniteur !!

=> deadlock !

# Modèle de mémoire

L'entrée et la sortie d'un bloc synchronized, on un effet sur les variable en mémoire

- L'acquisition (**acquire**) d'un verrou associé à un moniteur force la relecture de tous les variables depuis la RAM (valeurs en cache invalidées)
- Le relâchement (**release**) d'un verrou associé à un moniteur force l'écriture de toutes les variables modifiés vers la RAM

# Modèle de mémoire vs JIT

L'entrée ou la sortie d'un bloc synchronized requiert que l'optimisateur/JIT

- ne sortent pas les instructions à l'intérieur d'un bloc synchronized
  - Le contraire est possible (lock coarsening)
- Invalide les registres à l'entrée, oblige la ré-écriture des registres vers la RAM en sortie
- émette des instructions CPU forçant
  - Une barrière en lecture à l'entrée (pas nécessaire si TSO)
  - Une barrière en écriture à la sortie

A l'intérieur d'un bloc synchronized les instructions peuvent être ré-organisés sans problème

# Modèle de mémoire

Un bloc synchronized n'empêche pas la mémoire (RAM) d'être dans un état incohérent au milieu d'un bloc

Mais empêche une autre thread de voir ces états incohérents

# Efficacité et bloc synchronized

Si un bloc synchronized englobe trop d'instructions, utiliser des threads n'a pas d'intérêt

// exemple idiot

```
public static void main(String[] args) {  
    Object lock = new Object();  
    for(int i = 0; i < 4; i++) {  
        synchronized(lock) {  
            new Thread() -> {  
                // ...  
            }.start();  
        }  
    }  
}
```

Implicite final  
car effectivement final

Le code d'un bloc synchronized doit être réduit au strict minimum



# Ré-Entrance

Une même thread peut prendre plusieurs fois le même moniteur

```
public class Foo {  
    private int value;  
    private final Object monitor = new Object();  
  
    public void setValue(int value) {  
        synchronized(monitor) {  
            this.value = value;  
        }  
    }  
  
    public void reset() {  
        synchronized(monitor) {  
            setValue(0);  
        }  
    }  
}
```

# Ré-Entrance

En Java, les moniteurs sont toujours ré-entrant

En terme d'implantation,

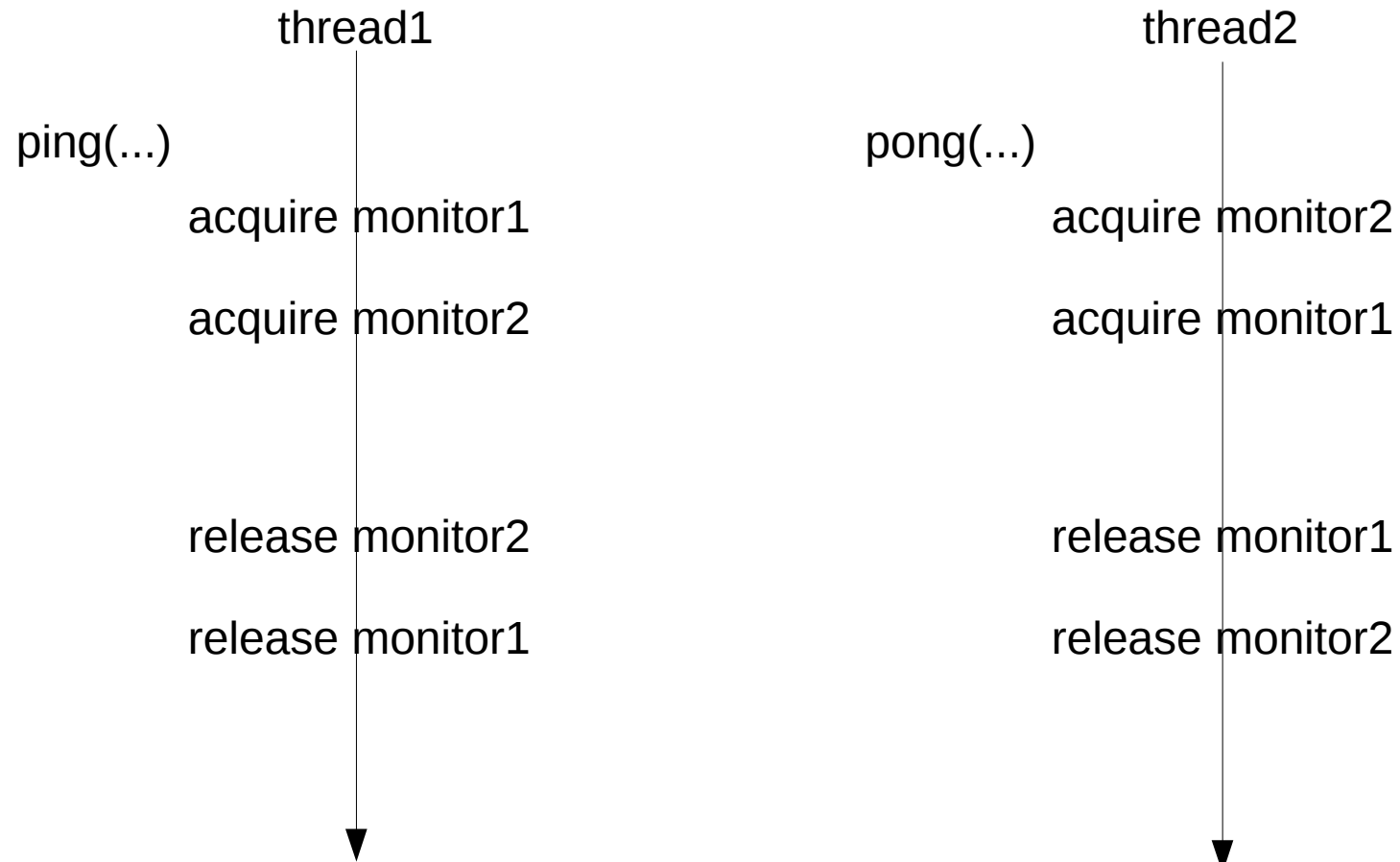
- La VM maintient un compteur associé à chaque verrou
- Lors de l'entrée dans un bloc synchronized, on incrémente le compteur
- Lors de la sortie dans un bloc synchronized, on décrémente le compteur

# Ping / Pong

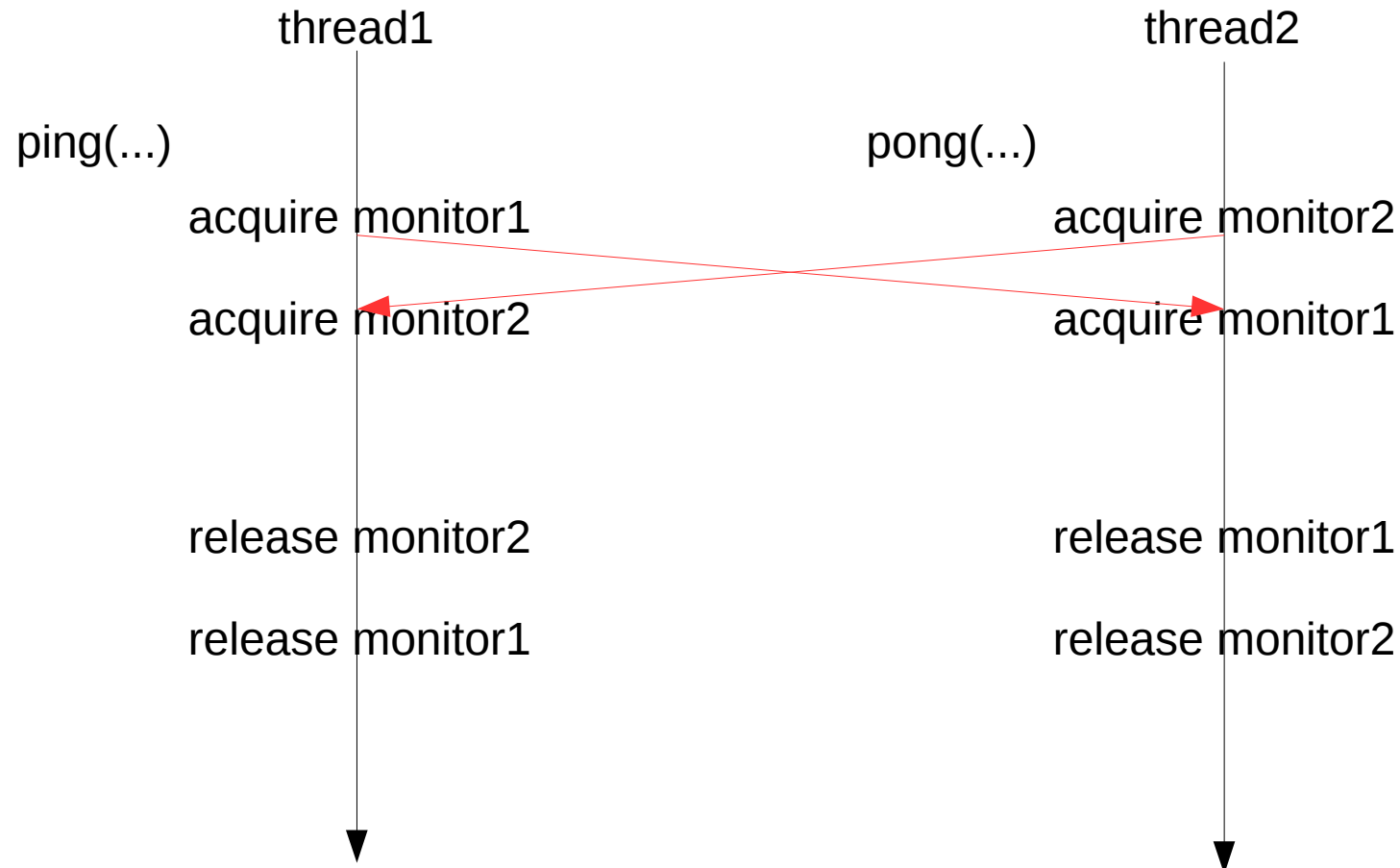
Que se passe t'il si une thread exécute ping et une autre pong ?

```
public class PingPong {  
    private final Object monitor1 = new Object();  
    private final Object monitor2 = new Object();  
  
    public void ping() {  
        synchronized (monitor1) {  
            synchronized (monitor2) {  
                // ...  
            }  
        }  
    }  
}  
  
    public void pong() {  
        synchronized (monitor2) {  
            synchronized (monitor1) {  
                // ...  
            }  
        }  
    }  
}
```

# Ping / Pong



# Ping / Pong



On appelle ce problème un deadlock !

# DeadLock

Si plusieurs threads (au moins 2), on prennent plusieurs moniteurs (au moins 2), dans un ordre différents, il risque d'y avoir un problème de deadlock

Deux solutions:

- N'utiliser qu'un seul moniteur
- Prendre les moniteurs toujours dans le même ordre

# Encapsulation POO / synchronized

Pour éviter les deadlocks, on évite de rendre les moniteurs visibles hors du code nécessaire

On utilise le principe d'encapsulation de la POO

- On déclare les champs stoquant un moniteur 'private'
- Pas de getter !

# Choix de l'objet moniteur

On utilise un object stocké dans un champ private final (ou une variable local effectivement final)

- si il y a déjà un objet de ce type dans la classe on l'utilise
- sinon, on se crée son propre Object

```
public class SynchronizedArrayList {  
    private final ArrayList<String> list = ...  
    public void add(String element) {  
        synchronized(list) {  
            list.add(element);  
        }  
    }  
}
```

Oh, un objet déclaré private final qui traîne





# POO / Classe ThreadSafe

On appelle threadsafe une classe qui peut être utilisée par plusieurs threads sans avoir d'états incohérents

- Cela dépend donc du contrat spécifié par la classe

Une classe non-mutable est par définition threadsafe

# ThreadSafe et JDK

## Conventions

- Par défaut, une classe est pas threadsafe sauf si c'est explicitement spécifié dans la javadoc
- Les classes de `java.util.concurrent` sont toutes threadsafe

## Exemple

- `java.lang.String`, threadsafe car immutable
- `java.util.HashMap`, pas threadsafe
- `java.util.Random`, threadsafe car marquée dans la doc

# Synchronized sur les méthodes

En Java, on peut déclarer une méthode 'synchronized'

Pour une méthode d'instance, c'est équivalent à un `synchronized(this) { ... }` sur le corps de la méthode

Pour une méthode statique, c'est équivalent à un `synchronized(TheCurrentClass.class) { ... }` sur le corps de la méthode

# Synchronized sur les méthodes

Dans les deux cas, cela pose problème, car c'est référence sont trop visible (attention au deadlocks) !

- “this” est trop visible, car connu de l'extérieur de la classe
- La référence à une classe (TheCurrentClass.class) est visible de n'importe quel endroit du code

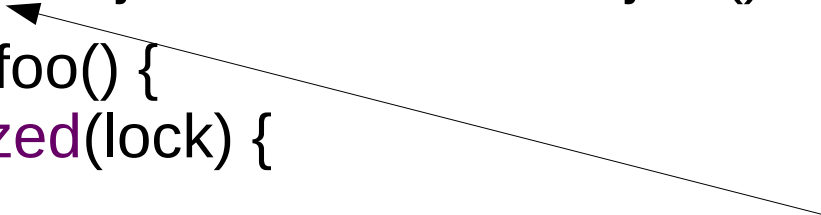
On ne déclare pas une méthode synchronized !

# Final ??

Pourquoi les champs donc la valeur est utiliser pour un bloc synchronized doivent être déclarés final ?

```
public class ThreadSafeHolder {  
    private final Object lock = new Object();  
    public void foo() {  
        synchronized(lock) {  
            // ...  
        }  
    }  
}
```

ahhhhhh



Il est dans ce cas possible pour une thread de voir le champ lock pas encore initialisé => NullPointerException

# java.util.concurrent.locks

Autre mécanisme permettant de définir des sections critiques introduit par Java 5,

Une interface Lock

Une implantation ReentrantLock

Utile si

- Forte contention (bcp de thread en attente)
- Operations autres que prendre/rendre un jeton

# ReentrantLock

- Mécanisme de verrou (appelé aussi mutex)
- Deux méthodes principales
  - lock() qui permet de prendre le verrou
  - unlock() qui permet de rendre le verrou
- Un appel à lock() doit toujours être suivi d'un appel à unlock()
  - Le code entre les deux est une section critique

# lock()/unlock() et exception

En Java, à peu près n'importe quel code peut lever une exception

donc le code ci-dessous ne marche pas

```
public class StupidLockDemo {  
    private final ReentrantLock lock = new ReentrantLock();  
  
    public void increment() {  
        lock.lock();  
        ... // do something here  
        lock.unlock();  
    }  
}
```

Si une exception est levée ici, unlock() n'est pas exécuté



# lock()/unlock() et finally

Il **faut** utiliser un block try/finally

```
public class LockDemo {  
    private final ReentrantLock lock = new ReentrantLock();  
  
    public void increment() {  
        lock.lock();  
        try {  
            ... // do something here  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

L'appel à lock() est hors du try

# ReentrantLock et fairness

Il existe deux implantations des ReentrantLock

- La version fair (équitable)
- La version non-fair qui est celle par défaut.

La version équitable garantie que si plusieurs threads sont en attente sur un verrou, après que la thread qui avait appelée lock() appel unlock(), c'est la première thread qui était en attente qui sera débloquée

- Sinon c'est une thread au hasard !

# Fairness et constructeur

Il existe deux constructeurs à ReentrantLock

- `ReentrantLock(boolean fairness)` qui permet de choisir si on veut un verrou équitable ou non
- `ReentrantLock()` sans paramètre qui est équivalent à `ReentrantLock(false)`

En terme d'efficacité, la version équitable (fair) est plus lente que la version non équitable

# Fairness et scheduler

La fairness d'un ReentrantLock n'a rien avoir avec la priorité que le scheduler associe au thread.

La thread qui sera débloquée n'est pas forcément celle qui est la plus prioritaire pour le scheduler mais celle qui a attendu le plus

# lock()/unlock() et modèle de mémoire

De la même façon que pour un bloc  
synchronized

- L'appel à la méthode lock() (acquire) est une barrière en lecture
- L'appel à la méthode unlock() (release) est une barrière en écriture

# Et si on mélange ?

Est ce que ce code est valide ?

```
public class StupidLockMix {  
    private final ReentrantLock lock = new ReentrantLock();  
  
    public void increment() {  
        synchronized(lock) {  
            // code ici  
        }  
    }  
}
```

Le code marche car on peut utiliser n'importe quel object comme moniteur

- Mais le code est stupide car on n'a pas besoin de créer un object ReentrantLock là où un simple Object suffit

# Plus de méthodes

Un ReentrantLock possède aussi les méthodes

- tryLock() qui permet d'acquies un verrou (comme lock()) mais qui renvoie false au lieu de bloquer si une autre thread possède déjà le verrou
  - L'opération est atomique
- lockInterruptibly() throws InterruptedException permet de prendre un verrou (comme lock()) mais qui peut être interrompu si la thread est bloquée en attendant qu'une autre thread qui possède déjà le verrou

# tryLock() et isLocked()

ReentrantLock possède une méthode de debug isLocked qui renvoie vrai si le verrou est déjà pris

– donc tryLock() est équivalent à

```
boolean tryLock() {  
    if (isLocked()) {  
        return false;  
    }  
    lock();  
    return true;  
}
```

en fait, **NON**, cette implémentation a un gros bug !



# isLocked() est une méthode de debug

Et si le scheduler arrête le code après le if ?

```
boolean tryLock() {  
    if (isLocked()) {  
        return false;  
    }  
    lock();  
    return true;  
}
```

← Le scheduler schedule une autre thread ici !

L'autre thread peut elle même prendre le verrou et dans ce cas, l'appel à lock() dans tryLock() devient un appel bloquant

# Lock vs synchronized

La structure de gestion des threads en attente est différente

- Pour un ReentrantLock, grosse structure de donnée, rapide si contention
- Pour un block synchronized, petite structure de donnée, lente si contention

ReentrantLock possède des méthodes supplémentaires

Dans le cas classique où il n'y a pas de contention, synchronized est plus efficace que ReentrantLock