

La lecture optimiste en Java

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

La programmation optimiste

La programmation à l'aide de verrous repose sur une perspective pessimiste de l'exécution des tâches concurrentes du système.

Il faut se méfier des variables ou des objets partagés !

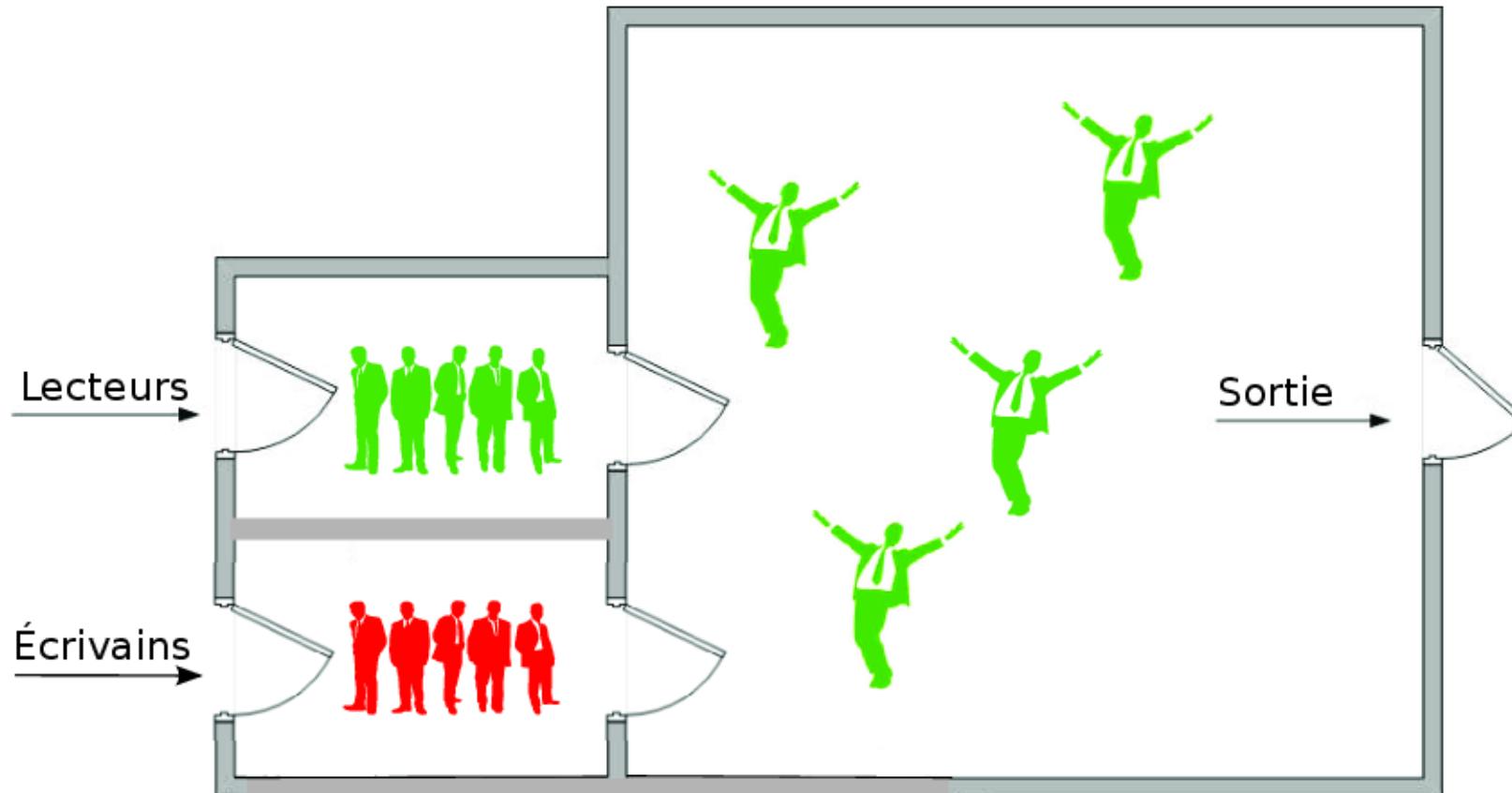
À l'inverse de l'approche classique, la **programmation optimiste** consiste à ne pas prendre de trop de précautions *a priori* mais à vérifier *a posteriori* qu'aucune interférence préjudiciable n'a eu lieu au cours de l'exécution de la << section critique. >>

- Pour les **opérations de lecture**, il suffit de recommencer la procédure si elle a échoué (quitte à prendre alors d'avantage de précautions), *car les données obtenues sont potentiellement corrompues* ;
- Pour les **opérations d'écriture**, en revanche, il faut parvenir à effectuer une modification complète et correcte des données ou bien aucune modification du tout, quitte à devoir retenter l'opération ultérieurement : on parle alors d'*écriture atomique conditionnelle*.



Sur les performances du verrou de lecture-écriture

Rappel : l'intérêt d'un verrou de Lecture/Ecriture



Plusieurs threads peuvent posséder le verrou de lecture simultanément ! Ça peut permettre d'améliorer les performances.

Quand peut-on espérer un gain de performance ?

Les verrous de lecture-écriture s'appuient sur une implémentation plus coûteuse en temps que les verrous simples. Leur emploi ne sera plus performant que si le gain obtenu par les lectures en parallèle est supérieur au coût de l'utilisation de ces verrous.

La Javadoc dit : « ReentrantReadWriteLocks can be used to improve concurrency in some uses of some kinds of Collections. This is typically worthwhile only when *the collections are expected to be large*, accessed by *more reader threads than writer threads*, and *entail operations with overhead that outweighs synchronization overhead*. »

Dans le cas contraire, utiliser ces verrous de lecture-écriture peut engendrer un **surcoût**.

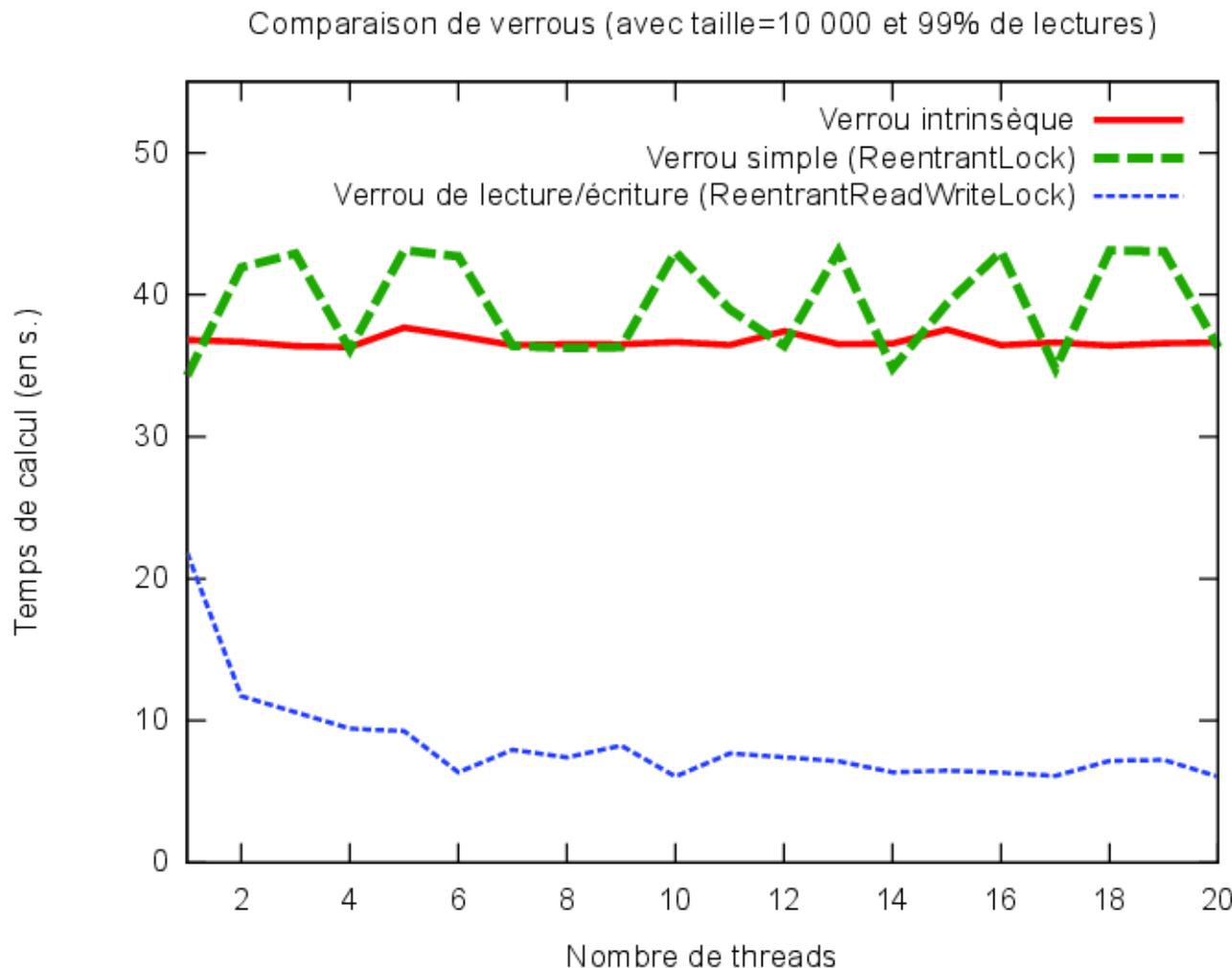
Benchmark adopté : un tableau de booléens

```
static final int taille = 10_000 ;
static final boolean[] drapeaux = new boolean[taille];
public void run() {
    for (int i=1 ; i <= part ; i++) {
        if (aléa.nextInt(100)<1) { // 1% d'écritures
            synchronized(verrou) { // en exclusion mutuelle
                drapeaux[aléa.nextInt(taille)] = aléa.nextBoolean() ;
            } // Écriture d'un booléen aléatoire dans une case aléatoire
        } else { // 99% de lectures
            synchronized(verrou) { // en exclusion mutuelle
                int somme = 0 ;
                for (int j=0 ; j<taille ; j++) if (drapeaux[j]) somme++ ;
            } // Calcule le nombre de booléens égaux à true
        }
    }
}
```

Le même chose avec un verrou de Lecture/Ecriture

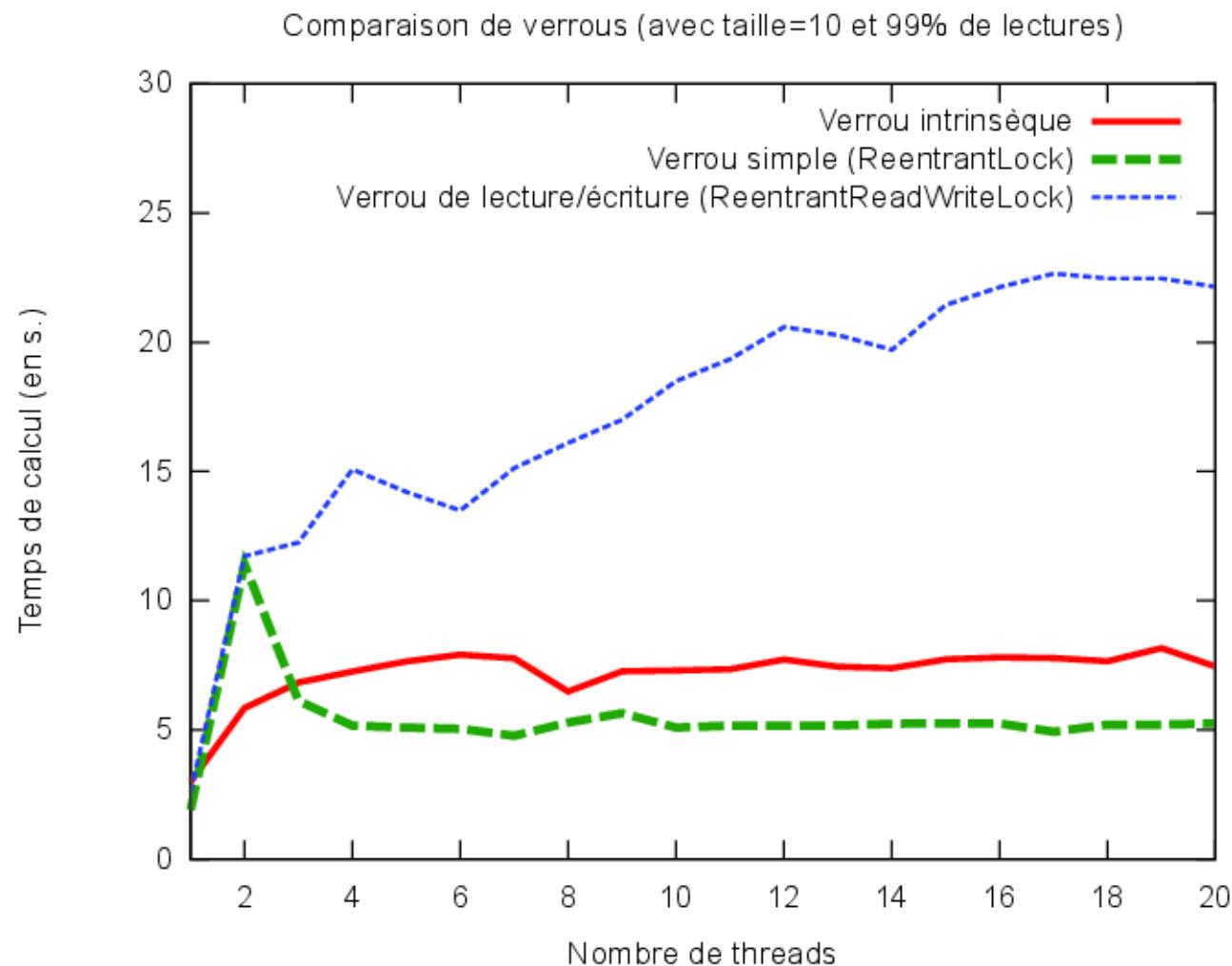
```
static ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();  
...  
for (int i=1 ; i <= part ; i++){  
    if (aléa.nextInt(100)<1) { // 1% d'écritures  
        rwl.writeLock().lock() ; // Je prends le verrou d'écriture  
        try {  
            drapeaux[aléa.nextInt(taille)] = aléa.nextBoolean() ;  
        }finally{ rwl.writeLock().unlock(); } // Je relâche le verrou  
    } else { // 99% de lectures  
        rwl.readLock().lock() ; // Je prends le verrou de lecture  
        try {  
            int somme = 0 ;  
            for (int j=0 ; j < taille ; j++) if (drapeaux[j]) somme++ ;  
        }finally{ rwl.readLock().unlock(); } // Je relâche le verrou  
    }  
}
```

Illustration du gain sur un tableau à 10 000 éléments



Le verrou de lecture-écriture permet des lectures en parallèle : un gain est observé lorsque le nombre de threads croît (jusqu'à ce qu'il atteigne le nombre de coeurs disponibles).

Le défaut des verrous RWL : cas d'un tableau à 10 éléments



Le gain produit par un verrou de lecture-écriture n'est pas systématique ! En particulier, l'opération de lecture doit être suffisamment longue. Ici le tableau est trop petit.

✓ *Sur les performances du verrou de lecture-écriture*

(→) *Le verrou timbré : « stamped lock »*

Introduction du verrou << timbré >>

Introduit dans Java 8, *le verrou timbré (StampedLock)* n'est pas véritablement un verrou ; en tout cas, il n'implémente pas l'interface **Lock**.

- Comme les verrous de Lecture/Écriture de **ReentrantReadWriteLock**, il permet de protéger les opérations de lecture, autorisées en parallèle, contre les opérations d'écriture et d'assurer l'exclusion mutuelle des opérations d'écriture ;
- Plus << léger >> que le verrou de Lecture/Écriture, il sera souvent **plus performant** que le verrou de Lecture/Écriture.
- Ce verrou **n'est pas réentrant !** Le risque de blocage est accru.
- Il permet en outre d'essayer une *lecture optimiste* durant laquelle les écritures ne seront pas bloquées : **les données lues peuvent alors être corrompues !**
~~> il faudra vérifier à la fin de la lecture qu'aucune écriture n'a eu lieu simultanément.
- Il est possible d'essayer de **transformer un timbre de lecture en timbre d'écriture**.

Utilisation du verrou timbré comme un verrou de lecture/écriture

Comme les verrous de Lecture/Écriture de **ReentrantReadWriteLock**, le verrou timbré permet de protéger différemment les opérations de lecture et d'écriture.

Acquisition en mode écriture

La méthode **writeLock()** permet d'acquérir le verrou *en mode d'écriture*; elle bloque si le verrou est déjà en mode écriture ou en mode lecture.

Elle retournera *un entier long appelé « timbre »* qui doit être utilisé lors du relâchement du verrou par un appel du type **unlockWrite(timbre)**.

Acquisition en mode écriture

La méthode **readLock()** permet d'acquérir le verrou *en mode lecture*; elle bloque si le verrou est en mode écriture.

Elle retournera également un timbre utilisé pour le relâchement du verrou par un appel du type **unlockRead(timbre)**.

Rappel : Compte bancaire avec un ReentrantReadWriteLock (1/2)

```
public class CompteBancaire {  
    private volatile long épargne;  
    private final ReadWriteLock verrou=new ReentrantReadWriteLock();  
  
    public CompteBancaire(long épargne) {  
        this.épargne = épargne;  
    }  
  
    public void déposer(long montant) {  
        verrou.writeLock().lock();  
        try {  
            épargne += montant;  
        } finally {  
            verrou.writeLock().unlock();  
        }  
    }  
}
```

Rappel : Compte bancaire avec un ReentrantReadWriteLock (2/2)

```
public void retirer(long montant) {  
    verrou.writeLock().lock();  
    try {  
        épargne -= montant;  
    } finally {  
        verrou.writeLock().unlock();  
    }  
}  
  
public long solde() {  
    verrou.readLock().lock();  
    try {  
        return épargne;  
    } finally {  
        verrou.readLock().unlock();  
    }  
}
```

Compte bancaire avec un verrou timbré (1/2)

```
public class CompteBancaire {  
    private volatile long épargne;  
    private final StampedLock verrou = new StampedLock();  
  
    public CompteBancaire(long épargne) {  
        this.épargne = épargne;  
    }  
  
    public void déposer(long montant) {  
        long timbre = verrou.writeLock();  
        try {  
            épargne += montant;  
        } finally {  
            verrou.unlockWrite(timbre);  
        }  
    }  
}
```

Compte bancaire avec un verrou timbré (2/2)

```
public void retirer(long montant) {
```

```
    long timbre = verrou.writeLock();
```

```
    try {
```

```
        épargne -= montant;
```

```
    } finally {
```

```
        verrou.unlockWrite(timbre);
```

```
    }
```

```
}
```

```
public long solde() {
```

```
    long timbre = verrou.readLock();
```

```
    try {
```

```
        return épargne;
```

```
    } finally {
```

```
        verrou.unlockRead(timbre);
```

```
    }
```

```
}
```

```
}
```

Recette applicable pour une lecture optimiste

La lecture optimiste consiste à ne pas prendre le verrou de lecture et à **espérer** qu'aucune écriture n'aura lieu pendant la lecture des données.

Au cours d'une lecture optimiste, le verrou pourra être acquis en mode écriture : les données pourront donc être modifiées. La lecture effectuée n'est alors pas atomique et les données lues peuvent par conséquent être inconsistentes.

Une lecture optimiste consiste

- ① à récupérer un **timbre** marquant le début de la lecture ;
- ② à lire les données en les recopiant localement ;
- ③ à valider la lecture réalisée, à l'aide du timbre, et recommencer sinon !
- ④ et enfin, à exploiter les copies locales des données lues.

Les deux méthodes à utiliser pour une lecture optimiste

La méthode **tryOptimisticRead()** renvoie un *timbre de lecture optimiste*.

La méthode **validate(timbre)** renvoie un booléen qui indique que le verrou n'est pas passé en mode écriture depuis la délivrance du timbre par la méthode **tryOptimisticRead()** : il indique donc si la lecture réalisée est *valide*.

Code typique pour une lecture optimiste

```
public long solde() {  
    long timbre = verrou.tryOptimisticRead();  
    long copie = épargne;  
    // Il serait hasardeux d'exploiter la copie dès à présent!  
    if (!verrou.validate(timbre)) {  
        // La copie obtenue est potentiellement corrompue !  
        timbre = verrou.readLock();  
        try {  
            // Le verrou est à présent en mode lecture  
            copie = épargne;  
        } finally {  
            verrou.unlockRead(timbre);  
        }  
    }  
    return copie; // Cette copie est fiable!  
}
```

- ✓ *Sur les performances du verrou de lecture-écriture*
- ✓ *Le verrou timbré : « stamped lock »*
- 👉 *Performances de la lecture optimiste*

Retour au benchmark (1/2)

```
import java.util.concurrent.locks.StampedLock;  
...  
static private StampedLock verrou = new StampedLock();  
...  
public void run(){  
    long timbre ;  
    for (int i = 0 ; i < part; i++) {  
        if (aléa.nextInt(100) < 1) {          // 1% d'écritures  
            timbre = verrou.writeLock(); // en exclusion mutuelle  
            try {  
                drapeaux[aléa.nextInt(taille)] = aléa.nextBoolean() ;  
            } finally { verrou.unlockWrite(timbre); }  
        }  
        else {                                // 99% de lectures
```

Retour au benchmark (2/2)

```
else {                                     // 99% de lectures

    long timbre = verrou.tryOptimisticRead();

    int somme = 0;

    for (int j=0 ; j < taille ; j++) {
        if (drapeaux[j]) somme++ ;

    }

    if (! verrou.validate(timbre)){ // Il y a eu corruption!
        timbre = verrou.readLock(); // Plus de risque...
        try {

            somme = 0;

            for (int j=0; j < taille; j++) {
                if (drapeaux[j]) somme++ ;

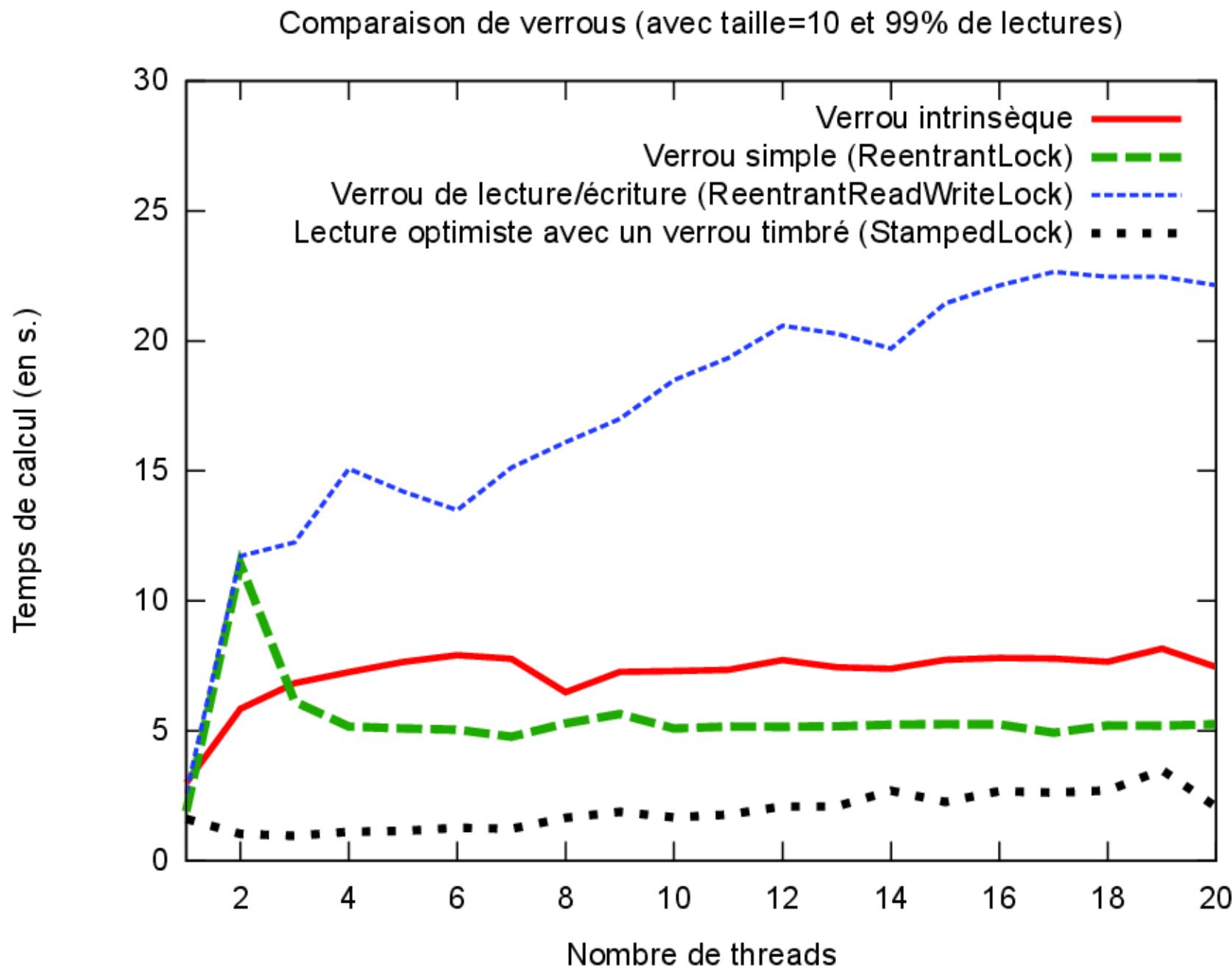
            }

        } finally { verrou.unlockRead(timbre); }

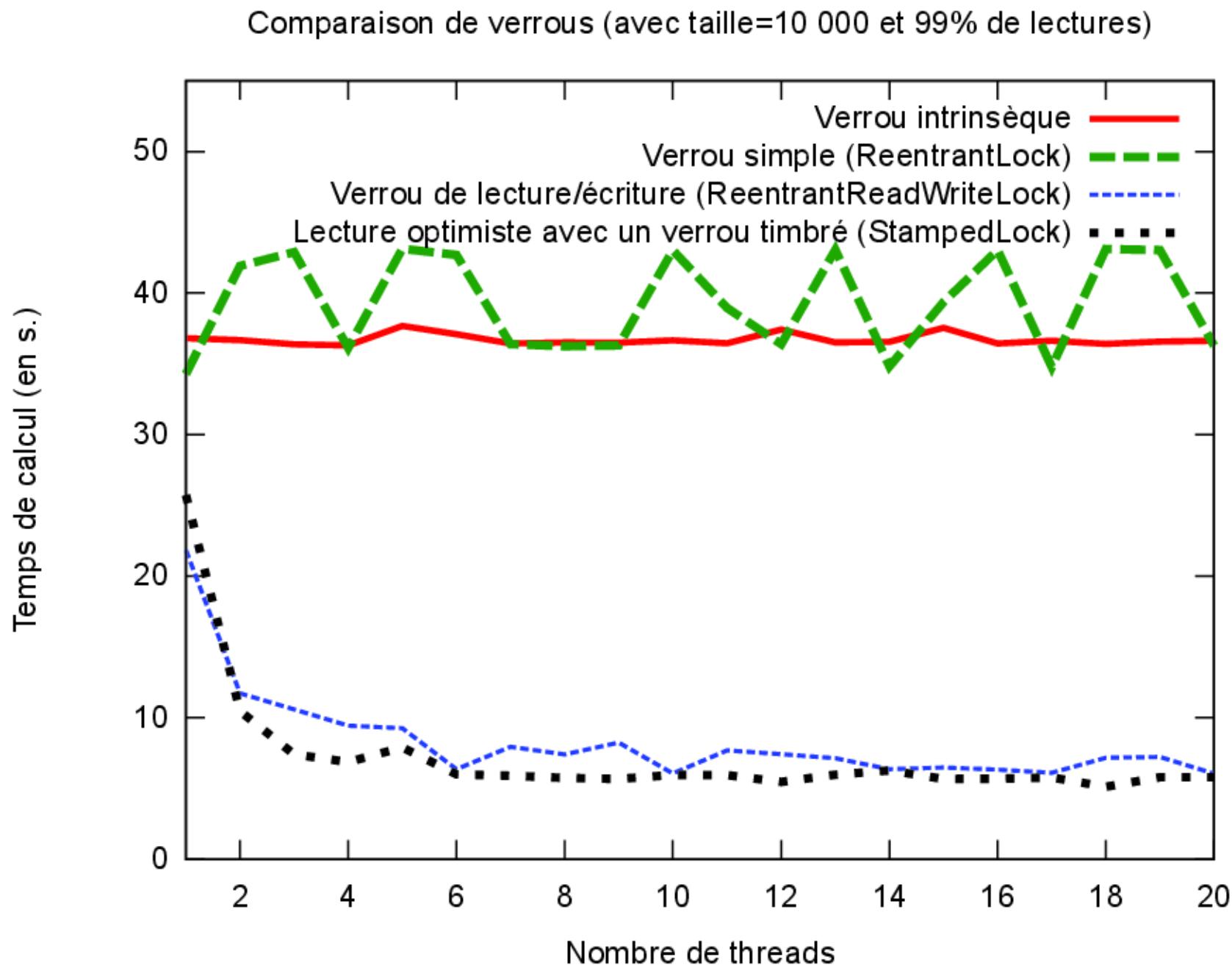
    }

}
```

Performances sur un tableau à 10 éléments



Performances sur un tableau à 10 000 éléments



- ✓ *Sur les performances du verrou de lecture-écriture*
 - ✓ *Le verrou timbré : « stamped lock »*
 - ✓ *Performances de la lecture optimiste*
-  *La maison ne fait pas crédit !*

Le compte bancaire doit rester positif

```
public class CompteBancaire {  
    private volatile long épargne;  
    public CompteBancaire(long épargne) {  
        this.épargne = épargne;  
    }  
    public synchronized void déposer(long montant) {...}  
  
    public synchronized boolean retirer(long montant) {  
        if (montant > épargne) return false;  
        épargne -= montant;  
        return true;  
    }  
  
    public synchronized long solde() {...}  
}
```

Avec un verrou ReentrantReadWriteLock

```
public boolean retirer(long montant) {  
    verrou.readLock().lock();  
    try {  
        if (montant > épargne) return false;  
    } finally {  
        verrou.readLock().unlock();  
    }  
    verrou.writeLock().lock();  
    try {  
        if (montant > épargne) return false; // Pourquoi?  
        épargne -= montant;  
    } finally {  
        verrou.writeLock().unlock();  
    }  
    return true;  
}
```

Avec un verrou timbré

```
public boolean retirer(long montant) {  
    long timbre = verrou.readLock() ; // C'est un timbre de lecture  
    try {  
        if (montant > épargne) return false ;  
        long nouveauTimbre = verrou.tryConvertToWriteLock(timbre) ;  
        if (nouveauTimbre==0) { // La tentative de conversion a échoué  
            verrou.unlockRead(timbre) ;  
            timbre = verrou.writeLock() ;  
            if (montant > épargne) return false ; // Intérêt ?  
        } else { timbre = nouveauTimbre ; }  
                // timbre est désormais un timbre d'écriture  
        épargne -= montant ;  
        return true ;  
    } finally {  
        verrou.unlock(timbre) ; // Timbre d'écriture ou de lecture!  
    }  
}
```

Conversion de timbre

La méthode `tryConvertToWriteLock(timbre)` essaie de transformer le mode du verrou vers le mode d'écriture, et renvoie un nouveau timbre en cas de succès.

Cette conversion est possible et renvoie un timbre *non nul* :

- si le timbre en paramètre est déjà un timbre d'écriture : la méthode renvoie alors le timbre initial ;
- si le timbre en paramètre est un timbre de lecture et aucun autre thread ne dispose d'un timbre de lecture ;
- si le timbre en paramètre est un timbre de lecture optimiste et le verrou n'est ni en mode lecture, ni en mode écriture.

Programmation sans verrou

Master Informatique — Semestre 1 — UE obligatoire de 3 crédits

À propos des verrous

Les verrous sont un moyen simple de se prémunir contre toute interférence préjudiciable entre les parties du programme qui peuvent s'exécuter en parallèle, en garantissant l'**atomicité** de parties de code « critiques. »

Néanmoins les verrous pâtissent d'un certain nombre d'inconvénients bien connus :

- Les **interblocages** sont un risque principal, qui demande de la rigueur ;
- Les **performances sont réduites** s'il y a inutilement trop de verrous ;
- Les **performances sont réduites** si un thread *lent* saisit une série de verrous ;
- Les **inversions de priorité**, lorsqu'un thread *non-prioritaire* possède un verrou ;
- **L'intolérance aux fautes**, si un thread s'arrête en possédant un verrou ;
- Les **structures de données** proscrivent les mises-à-jour en parallèle.

Ce cours vise à présenter quelques techniques usuelles de programmation sans verrou : celles-ci s'appuient toutes sur les *objets atomiques de Java*.

Compte bancaire sans verrou

```
public class CompteBancaire {  
    private final AtomicLong épargne;  
    public CompteBancaire(long épargne) {  
        this épargne = new AtomicLong(epargne);  
    }  
    public void déposer(long montant) {  
        épargne.addAndGet(montant);  
    }  
    public void retirer(long montant) {  
        épargne.addAndGet(-montant);  
    }  
    public long solde() {  
        return épargne.get();  
    }  
}
```

Comment garder un solde positif ?

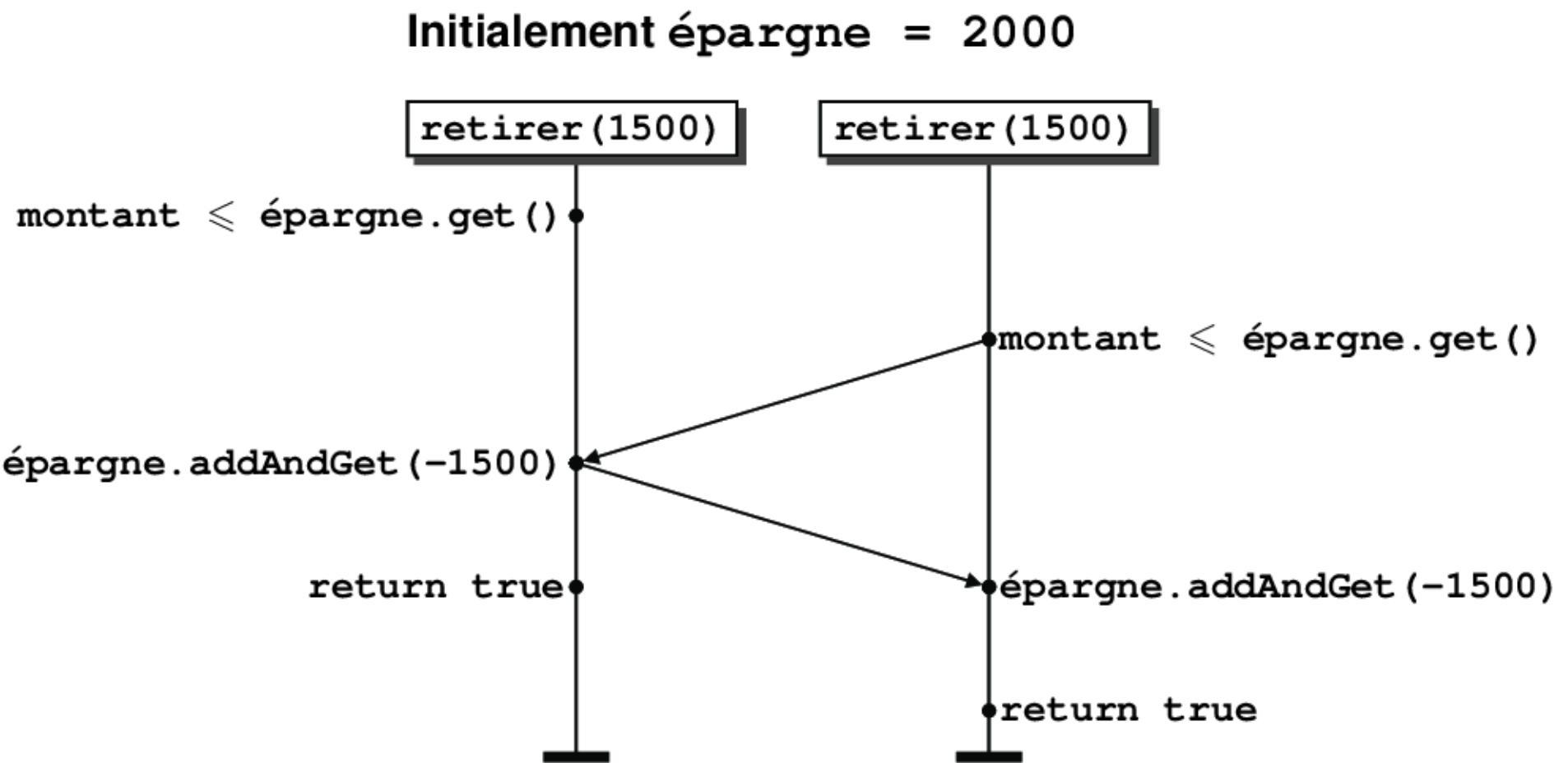
La maison ne fait pas crédit !

```
public class CompteBancaire {  
    private final AtomicLong épargne;  
  
    ...  
  
    public boolean retirer(long montant) {  
        if (montant > épargne.get()) return false;  
        épargne.addAndGet(-montant);  
        return true;  
    }  
  
    ...  
}
```



Quel est le souci ?

Un scenario extrême



Quel est le nouveau solde ?

Une opération atomique fondamentale, mais un peu curieuse

```
boolean compareAndSet(valeurAttendue, valeurNouvelle)
```

- ~~> Affecte **atomiquement** la valeur **valeurNouvelle** dans l'objet atomique à **condition que** la valeur courante de cet objet soit effectivement égale à **valeurAttendue**.

Il faut deviner la valeur courante pour la modifier !

- ~~> Retourne **true** si l'affectation a eu lieu, **false** si la valeur courante de l'objet atomique est différente de **valeurAttendue** au moment de l'appel.

Un appel à **compareAndSet()** sera remplacé par la machine virtuelle Java par l'opération assembleur correspondante dans la machine : par exemple, les instructions de comparaison et échange CMPXCHG8B ou CMPXCHG16B des processeurs Intel.

Recette pour programmer avec un objet atomique

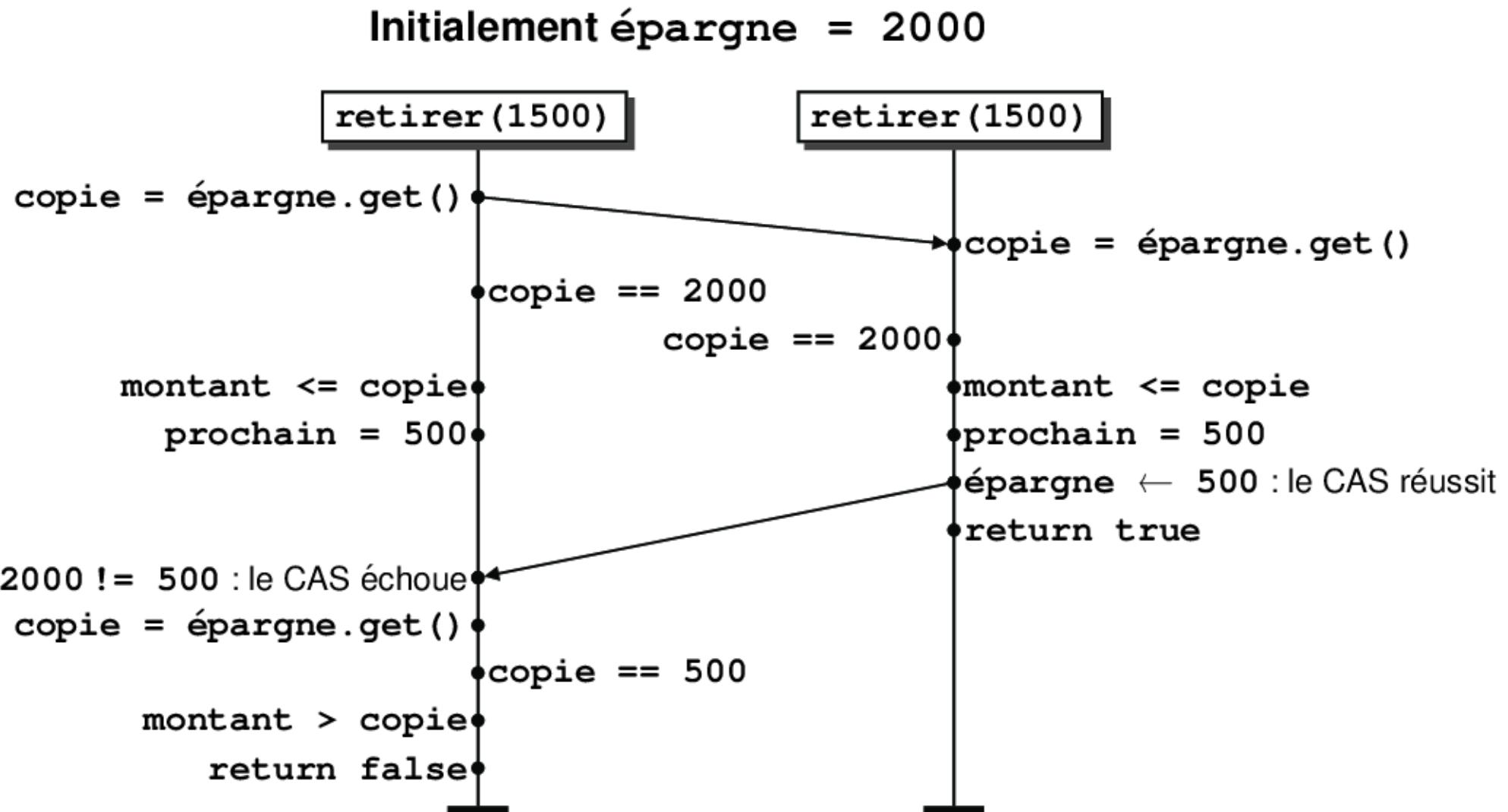
Pour modifier correctement un objet atomique, il suffit en général de

- ① fabriquer une copie locale de la valeur courante de l'objet atomique ;
- ② préparer une modification de l'objet *à partir de la copie obtenue* ;
- ③ appliquer une mise-à-jour de l'objet atomique conformément à l'étape 2, si sa valeur courante correspond encore à celle copiée à l'étape 1 (et sinon, recommencer).

```
public boolean retirer(long montant) {  
    for (;;) {  
        int copie = épargne.get();  
        if (montant > copie) return false;  
        int prochain = copie - montant;  
        if (épargne.compareAndSet(copie, prochain)) return true;  
    }  
}
```

Cette méthode agit elle de manière atomique ?

Fonctionnement formellement non-atomique de `retirer()`



Le thread à droite réussit le `compareAndSet()` en premier : il renvoie `true` ; l'autre thread exécute un *tour de boucle supplémentaire* et renvoie `false`.

Atomicité de la méthode `retirer()`

La méthode `retirer()` modifie potentiellement la valeur de l'entier atomique **épargne**.

Pour le reste, elle ne travaille que sur des *variables locales*.

Une exécution de la méthode `retirer()` conduira

- ou bien à aucun changement pour l'entier atomique **épargne**, si le montant est trop élevé ;
- ou bien à une décrémentation atomique de l'entier atomique **épargne**, du montant indiqué, lors de l'appel réussi à `compareAndSet()`, précédé éventuellement d'une phase d'attente qui ne produit aucun effet.

On peut donc considérer que la méthode `retirer()` agit de manière atomique, dès lors qu'on l'identifie à l'opération `compareAndSet()` réalisée.

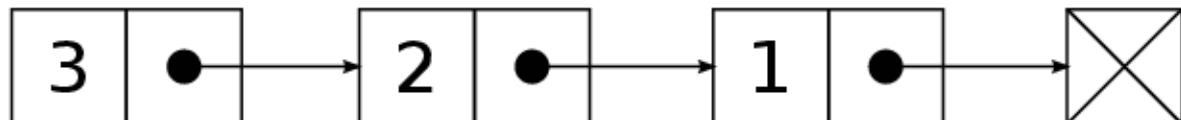
Les variations du compte bancaire peuvent alors s'analyser comme une succession d'opérations atomiques formée d'appels à `déposer()`, `retirer()` ou `soldé()`.

✓ *Exemple du compte bancaire*

👉 *Exemple d'une pile concorrente*

La brique de base : le noeud

```
class Noeud {          // On s'intéresse à une pile d'entiers  
    final Integer valeur;  
    Noeud suivant;      // Référence vers le noeud suivant  
  
    public Noeud(Integer valeur) {           // Constructeur  
        this.valeur = valeur;  
    }  
}
```

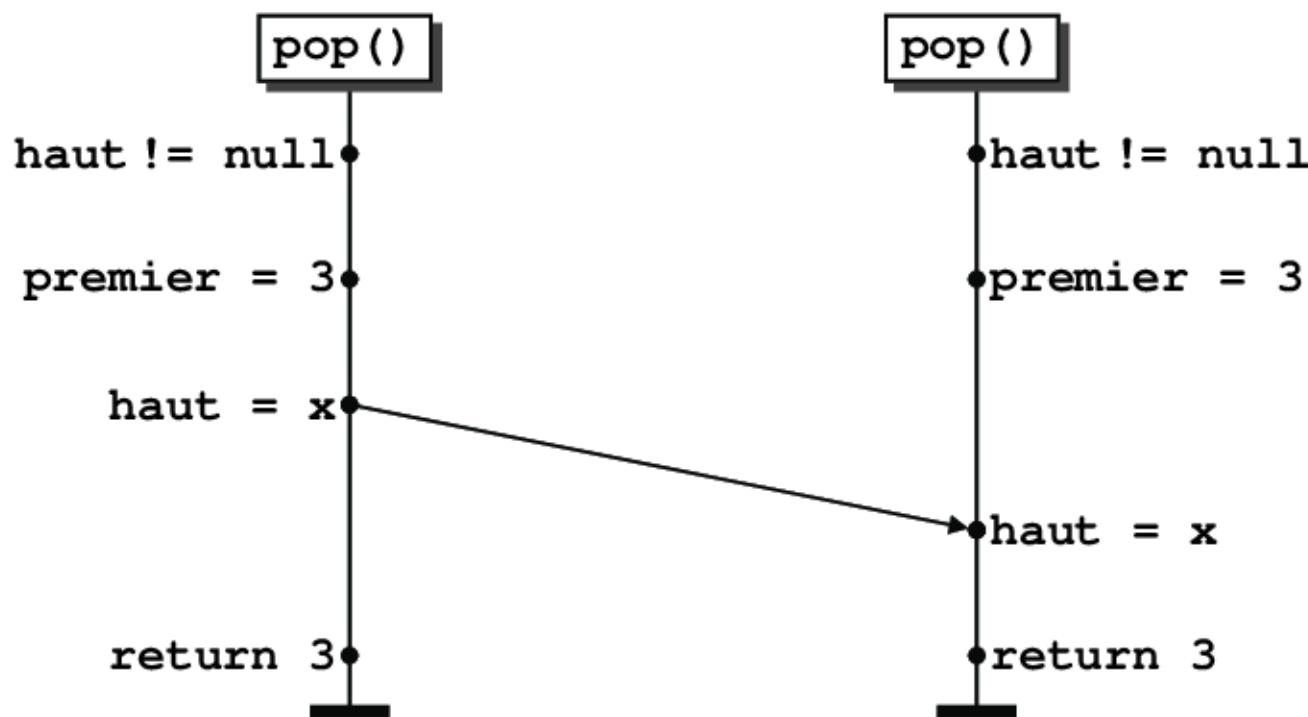
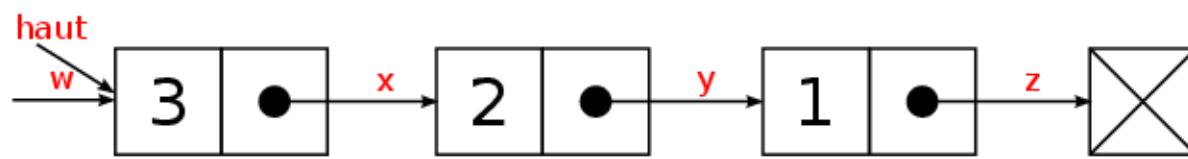


Une pile synchronisée (avec un verrou, donc)

```
class PileSynchronisee { // C'est une espèce de liste de Noeuds
    private volatile Noeud haut;
    // haut est (une référence vers) le premier noeud de la pile
    public synchronized void push(Integer valeur) {
        Noeud prochain = new Noeud(valeur);
        prochain.suivant = haut;
        haut = prochain;           // Modification du haut de la pile
    }
    public synchronized Integer pop() {
        if (haut == null) return null;      // La pile peut être vide
        Integer premier = haut.valeur;
        haut = haut.suivant;             // Modification du haut de la pile
        return premier;
    }
}
```

La même chose sans verrou ?

Évidemment, on ne doit pas supprimer brutalement le verrou



À la fin, les deux threads renvoient la valeur 3 et la pile contient encore deux éléments : «2» suivi de «1».

Seconde recette : programmer avec des références atomiques

Les objets atomiques sont précieux pour programmer sans verrou. Mais il en existe de peu de sortes... Pour manipuler des données un peu complexes de manière atomique, comme ici un noeud, il faut souvent encapsuler les données dans un objet et utiliser une **référence atomique** vers cet objet.

Pour modifier une référence atomique, il suffit en général

- ① d'obtenir une *copie locale X* de la valeur courante de la référence atomique ;
- ② de construire, *à l'aide de la référence X*, un **nouvel objet Y** qui met en oeuvre la modification sollicitée : Y est un candidat pour remplacer l'objet référencé par X ;
- ③ d'appliquer une mise-à-jour de la référence atomique vers la référence de Y obtenue à l'étape 2, à condition que la valeur courante de cette référence atomique corresponde encore à la copie X obtenue l'étape 1 (et sinon, recommencer).

Une approche pratique : les objets immuables

Cette recette est particulièrement sûre lorsque les objets référencés sont **immuables** :

- lors de l'étape 3, si la référence apparaît inchangée lors du test, alors l'objet lui-même est aussi inchangé depuis la copie X de l'étape 1.
- par conséquent, la consistance de la modification réalisée est assurée par la construction de l'objet Y à partir de la copie X.

Nous appliquerons cette stratégie lors du prochain TD.

Dans les exemples de code sans verrou qui suivent, nous n'adopterons pas cette contrainte car

- dans le premier cas, c'est inutile et ça réduirait les performances obtenues, alors que la question des performances sera abordée ;
- dans le second cas, nous voulons justement exhiber un bug qui ne pourrait pas se produire si l'on adoptait cette voie.

Algorithme de Treiber (1/2)

```
public class PileConcurrente {      // Code emprunté au livre JCIP
    AtomicReference<Noeud> haut = new AtomicReference<Noeud>();
    // haut est une référence à un noeud, manipulable atomiquement
    // Il désigne le noeud en haut de la pile (le premier noeud)
    public void push(Integer valeur) {
        Noeud prochain = new Noeud(valeur);
        Noeud copie ;
        do {
            copie = haut.get(); // Copie la référence vers le 1er noeud
            prochain.suivant = copie;
        } while ( ! haut.compareAndSet(copie, prochain) );
    }
}
```

Si cette méthode s'exécute de manière atomique, alors les références `haut.get()` et `copie` restent égales : le CAS s'applique et `haut` prend la valeur de `prochain`, c'est-à-dire une référence vers le nouveau noeud. Sinon, il faut recommencer !

Algorithme de Treiber (2/2)

```
public Integer pop() {  
    Noeud copie;  
    Noeud prochain;  
    do {  
        copie = haut.get();  
        if (copie == null) return null;  
        prochain = copie.suivant;  
    } while ( ! haut.compareAndSet(copie, prochain) );  
    return copie.valeur;  
}  
}
```

Si cette méthode s'exécute de manière atomique, alors les références `haut.get()` et `copie` restent égales : CAS s'applique et `haut` prend la valeur de `prochain`, c'est-à-dire une référence vers le noeud en seconde position. Sinon, il faut recommencer !

Précaution oubliée (un peu exprès)

Le champ **haut** devrait être **private**, comme pour la pile synchronisée.

Sinon

- il est possible, connaissant la pile, d'accéder aux noeuds fabriqués au fur et à mesure par la méthode **push ()** ;
- il devient possible de modifier la pile sans passer par les méthodes de la pile !

Bémol à propos de cet exemple

Une pile n'est vraiment pas la structure de données la plus pertinente pour illustrer la conception de structures de données non bloquantes (sans verrou) :

- Une file ou une liste triée, avec des opérations plus complexes, serait plus appropriée.
- Le gain de performance, s'il y en a un, sera faible, car les opérations sur une pile sont intrinsèquement séquentielles ;

Néanmoins, l'étude de cette pile permet :

- d'approfondir l'étude des *techniques de programmation sans verrou* à l'aide de références atomiques ;
- de se convaincre que **l'éventuel surcoût de performance** induit par la suppression des verrous devrait être nul, ou faible ;
- d'aborder la question de la **gestion de la contention** par les verrous ;
- d'introduire également **le problème ABA**.

- ✓ *Exemple du compte bancaire*
- ✓ *Exemple d'une pile concorrente*



Un mot sur les performances et la contention

Un mot sur les performances

Dans une situation de contention faible, c'est-à-dire lorsqu'il y a peu de threads agissant en concurrence sur la structure de donnée, les structures sans verrou seront en général plus performantes que les structures << synchronisées >> parce que

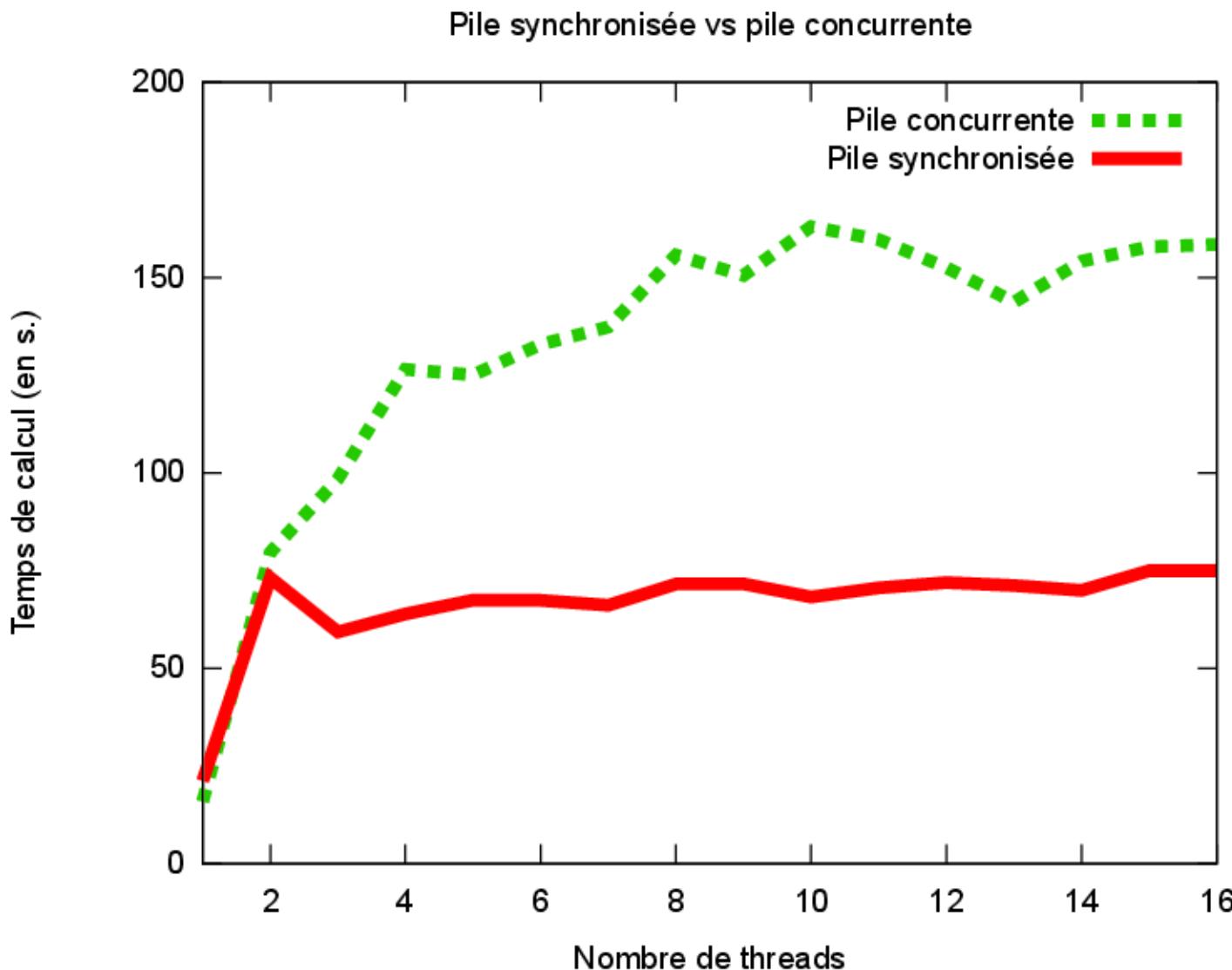
- l'opération CAS réussira la plupart du temps ;
- *la prise d'un verrou coûte au moins aussi chère qu'une opération CAS, et souvent même plus* ;
- la boucle d'itération, en cas d'échec fugace, évite un changement d'état interne du thread et la perte de l'accès au processeur (changement de contexte).

En cas de forte contention, c'est-à-dire lorsque de nombreux threads tentent d'accéder aux mêmes données, les verrous (notamment les verrous intrinsèques) offrent souvent de meilleures performances que l'attente active ; mais ces situations sont assez rares (et parfois artificielles). **Elles peuvent être aussi le fruit d'une mauvaise conception.** Enfin, les techniques utilisées par les verrous pour la *gestion de la contention* peuvent être appliquées aux structures de données sans verrou.

Benchmark adopté : empilage et dépilage aléatoires et répétitifs

```
public void run() {  
    for (int i = 1; i <= part; i++) {  
        if (aléa.nextInt(100) < pourcentage) {  
            pile.push(aléa.nextBoolean()) ;  
            // EMPILEMENT D'UN BOOLÉEN ALÉATOIRE  
        }  
        else {  
            pile.pop() ;  
            // DÉPILEMENT D'UN ÉLÉMENT  
        }  
    }  
}
```

Performances décevantes sur une machine à 8 processeurs



En cas de forte contention, pour la pile concurrente, le temps de calcul est grosso-modo proportionnel au nombre de threads actifs : **haut** est un << hotspot. >>

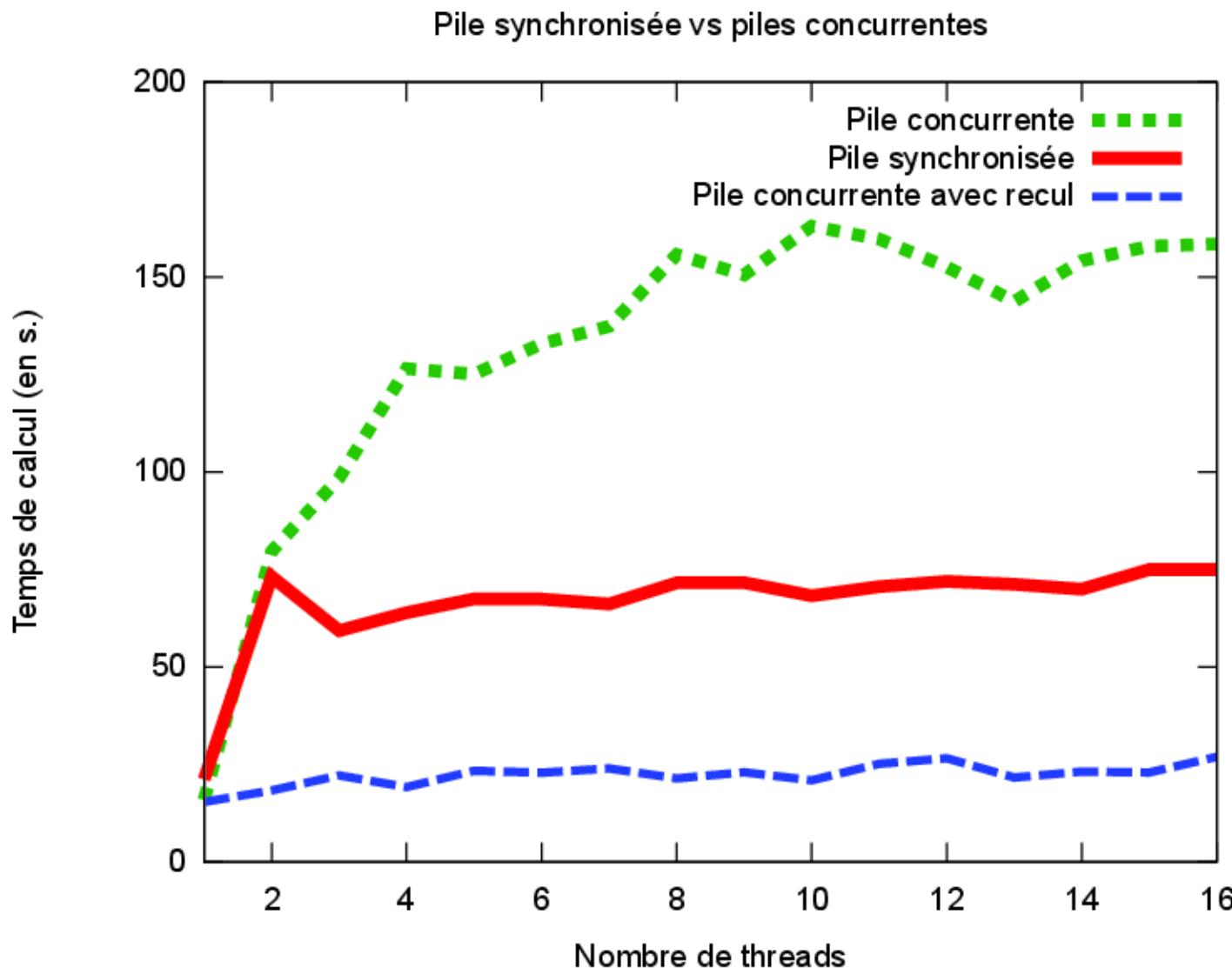
Ajout du recul en cas d'observation d'une contention (1/2)

```
class PileConcurrenteAvecRecul {  
    AtomicReference<Noeud> haut = new AtomicReference<Noeud>();  
    int DELAI_MIN = 8, DELAI_MAX = 4096;  
  
    public void push(Boolean valeur) {  
        int délai = DELAI_MIN ;  
        Noeud prochain = new Noeud(valeur) ;  
        Noeud copie ;  
        for (;;) {  
            copie = haut.get() ;  
            prochain.suivant = copie ;  
            if ( haut.compareAndSet(copie, prochain) ) return ;  
            // Il y a contention sur la variable haut: reculons !  
            Thread.sleep( aléa.nextInt(délai) ) ;  
            if (délai <= DELAI_MAX) délai += délai ;  
        }  
    }  
}
```

Ajout du recul en cas d'observation d'une contention (2/2)

```
public Boolean pop() {  
    Noeud copie;  
    Noeud prochain;  
    for(;;) {  
        copie = haut.get();  
        if (copie == null) return null;  
        prochain = copie.suivant;  
        if (haut.compareAndSet(copie,prochain)) return copie.valeur;  
        // Il y a contention sur la variable haut: reculons !  
        Thread.sleep( aléa.nextInt(délai) ) ;  
        if (délai <= DELAI_MAX) délai += délai ;  
    }  
}
```

Performances sur une machine à 8 processeurs



L'ajout d'un recul (analogique au protocole CSMA/CD dans les réseaux) lors d'une contention permet d'obtenir de bien meilleures performances.

- ✓ *Exemple du compte bancaire*
- ✓ *Exemple d'une pile concurrente*
- ✓ *Un mot sur les performances et la contention*
- 👉 *Le célèbre problème ABA*

Problème ABA de la programmation sans verrou

Le problème connu sous le nom de « ABA » correspond au scenario suivant :

- Le Thread 1 fabrique une copie de la variable partagée dont la valeur est **A** ;
- Le Thread 1 prépare une modification atomique de cette variable en s'appuyant sur la valeur de la copie obtenue ;
- Le Thread 2 modifie la variable partagée et lui attribue la valeur **B** ;

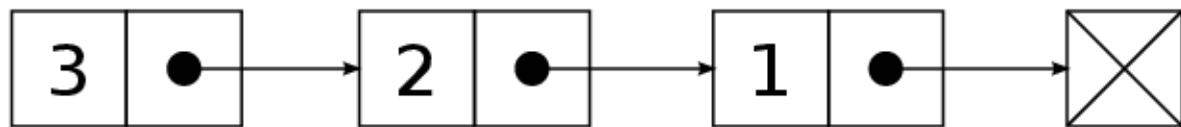
Rmq : Si le Thread 1 reprend la main ici et applique CAS, il observera le changement de valeur et le CAS renverra false ; le Thread 1 devra recommencer : tout va bien !

- Le Thread 2 modifie une seconde fois la valeur de la variable partagée et lui attribue à nouveau la valeur **A** ;
- Le Thread 1 reprend la main et applique CAS : la valeur trouvée étant celle attendue, le CAS applique la mise-à-jour, mais il ne devrait peut-être pas :

Si la variable partagée **A** est simplement une *donnée* d'un objet atomique (Boolean, Integer, ou Long), tout va bien : la modification préparée reste cohérente.

En revanche, si la variable partagée **A** est une **référence atomique** (comme **haut**) et si la modification préparée (ici : **prochain**) s'appuie sur l'objet référencé, cette modification n'est plus valide car l'objet référencé par **A** peut avoir changé, même si **A** apparaît inchangée.

À votre avis ?

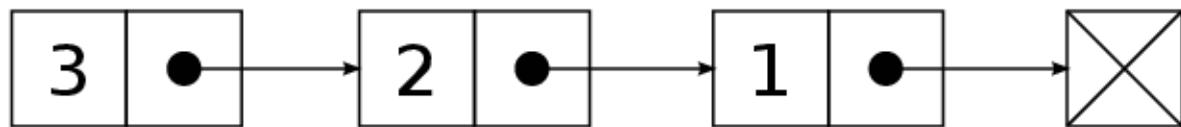


Un thread appelle **pop ()**.

En parallèle, un autre thread exécute **pop () ; pop () ; push (4)**.

Combien de noeuds contiendra la pile à la fin ?

À votre avis ?



Un thread appelle **pop ()**.

En parallèle, un autre thread exécute **pop () ; pop () ; push (4)**.

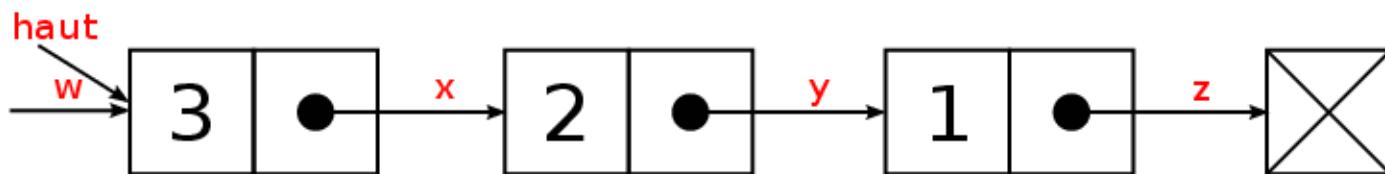
Combien de noeuds contiendra la pile à la fin ?

De deux choses l'une :

- tous les **pop ()** ont lieu avant le **push (4)** : la pile contient «4».
- le **pop ()** du premier thread intervient après le **push (4)** : la pile contient «1».

La pile contiendra un seul noeud si les deux méthodes **pop ()** et **push ()** agissent de manière atomique.

Scenario catastrophe hypothétique



Le Thread 1 commence à exécuter un **pop()** :

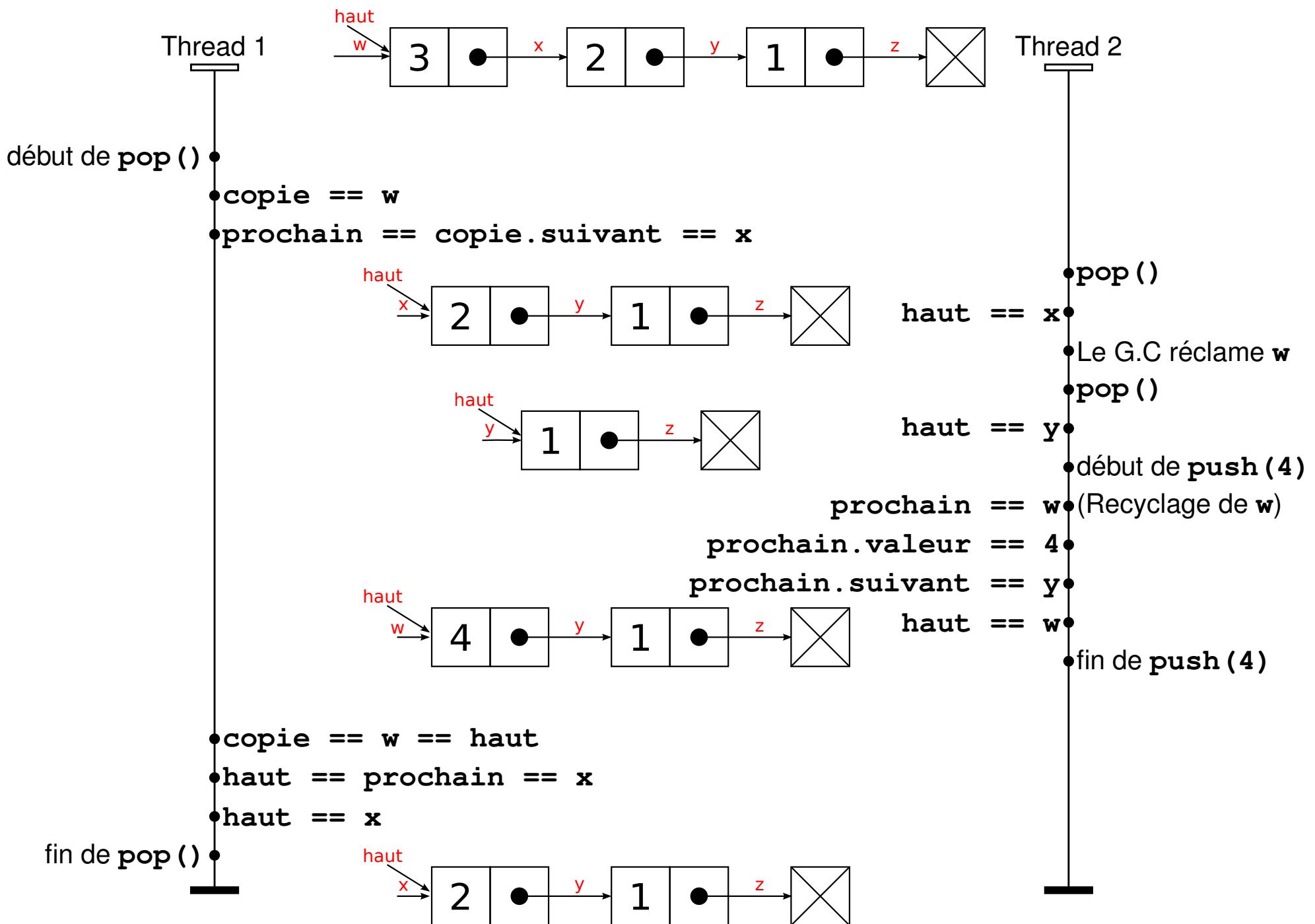
- Il fabrique une copie locale : **copie = haut.get() = w**
- Il prépare la nouvelle valeur : **prochain = copie.suivant = x**

À ce stade, le Thread 2 prend la main pour exécuter **pop(); pop(); push(4)**.

Après le premier **pop()**, le ramasse-miette réclame le noeud "3" puis lors du **push(4)** le nouveau noeud est créé avec pour référence w : la référence du haut de la pile apparaitra donc inchangée pour le Thread 1.

Le Thread 1 reprend la main pour finir le **pop()** en appliquant CAS : puisque la référence du haut de la pile est inchangée, le haut de la pile vaudra x et la pile contiendra 2 noeuds !

Scenario catastrophe hypothétique via un schéma



Absence de problème ABA en Java (avec ce code)

En réalité, ce scenario catastrophe ne peut pas avoir lieu en Java, car

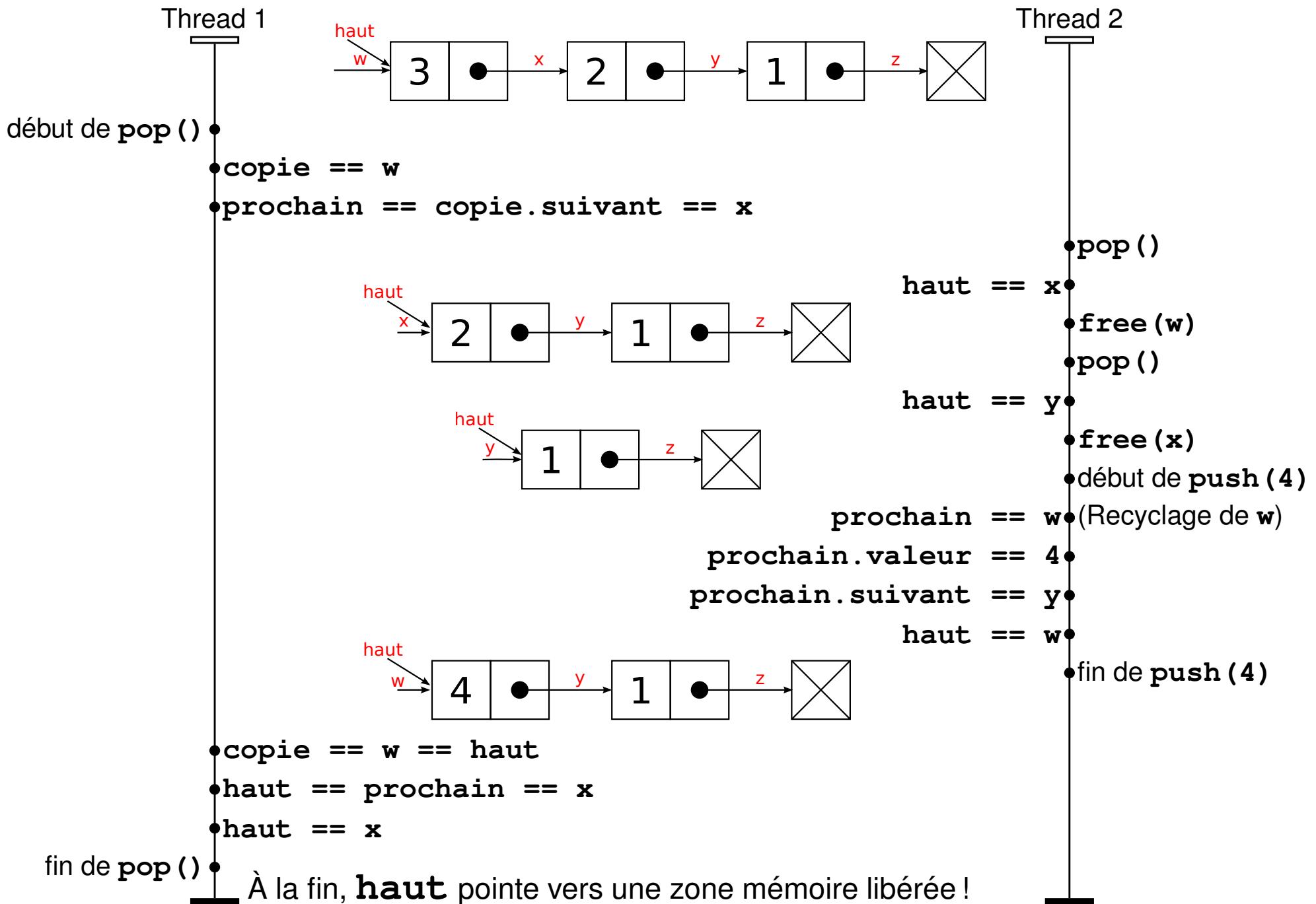
- Lors du `pop()` du Thread 1, `copie` devient une nouvelle référence vers le haut de la pile : `copie == w`.
- **Le ramasse-miette (GC) ne peut pas réclamer ce noeud, même s'il est supprimé de la pile par le Thread 2, car une référence active vers cet objet demeure pour le Thread 1.**
- Le noeud construit pour insérer 4 aura une référence différente de `w`.
- Le nouveau haut de la pile correspondra à une référence différente de `w`.
- Le Thread 1 constatera que le haut de la pile a été modifié : le CAS ne s'appliquera pas ; le Thread 1 devra recommencer un tour de boucle pour effectuer le `pop()` réclamé.

- ✓ *Exemple du compte bancaire*
- ✓ *Exemple d'une pile concurrente*
- ✓ *Un mot sur les performances et la contention*
- ✓ *Le célèbre problème ABA*



Illustration du problème ABA en C

Scenario ABA si on code en C (c'est-à-dire sans GC)



Solutions au problème ABA en C

Il existe plusieurs techniques, en C, pour pallier au problème ABA lorsqu'il est susceptible d'apparaître.

Les plus courantes sont de

1. *simuler* une espèce de ramasse-miette afin de retarder la libération de la mémoire ;
2. *numéroter* les éléments ou les mises-à-jour afin de distinguer le premier A vu du second A observé. Ceci nécessite l'emploi de primitives atomiques spéciales.

- ✓ *Exemple du compte bancaire*
- ✓ *Exemple d'une pile concurrente*
- ✓ *Un mot sur les performances et la contention*
- ✓ *Le célèbre problème ABA*
- ✓ *Illustration du problème ABA en C*
- 👉 *Illustration du problème ABA en Java*

Une pile de noeuds (1/2)

Le ramasse-miette est un atout important, mais subtile, pour éviter le problème ABA en Java. Ce n'est cependant pas une panacée. Dans certains cas, les références utilisées par une structure fuient en dehors de cette structure, ce qui ramène au problème ABA.

```
public class PileDeNoeuds {  
    private AtomicReference<Noeud> haut=new AtomicReference<Noeud>();  
    // haut est une référence à un noeud, manipulée atomiquement  
  
    public void push(Noeud entrée) {  
        Noeud copie;  
        do {  
            copie = haut.get(); // Référence du premier noeud  
            entrée.suivant = copie;  
        } while ( ! haut.compareAndSet(copie, entrée) );  
    }  
}
```

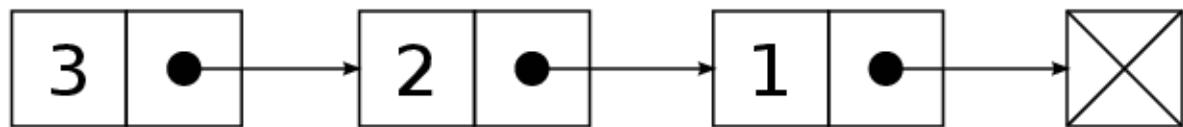


Une pile de noeuds (2/2)

```
public Noeud pop() {  
    Noeud copie;  
    Noeud prochain;  
    do {  
        copie = haut.get();  
        if (copie == null) return null;  
        prochain = copie.suivant;  
    } while ( ! haut.compareAndSet(copie, prochain) );  
    return copie;  
}  
}
```



À votre avis ?

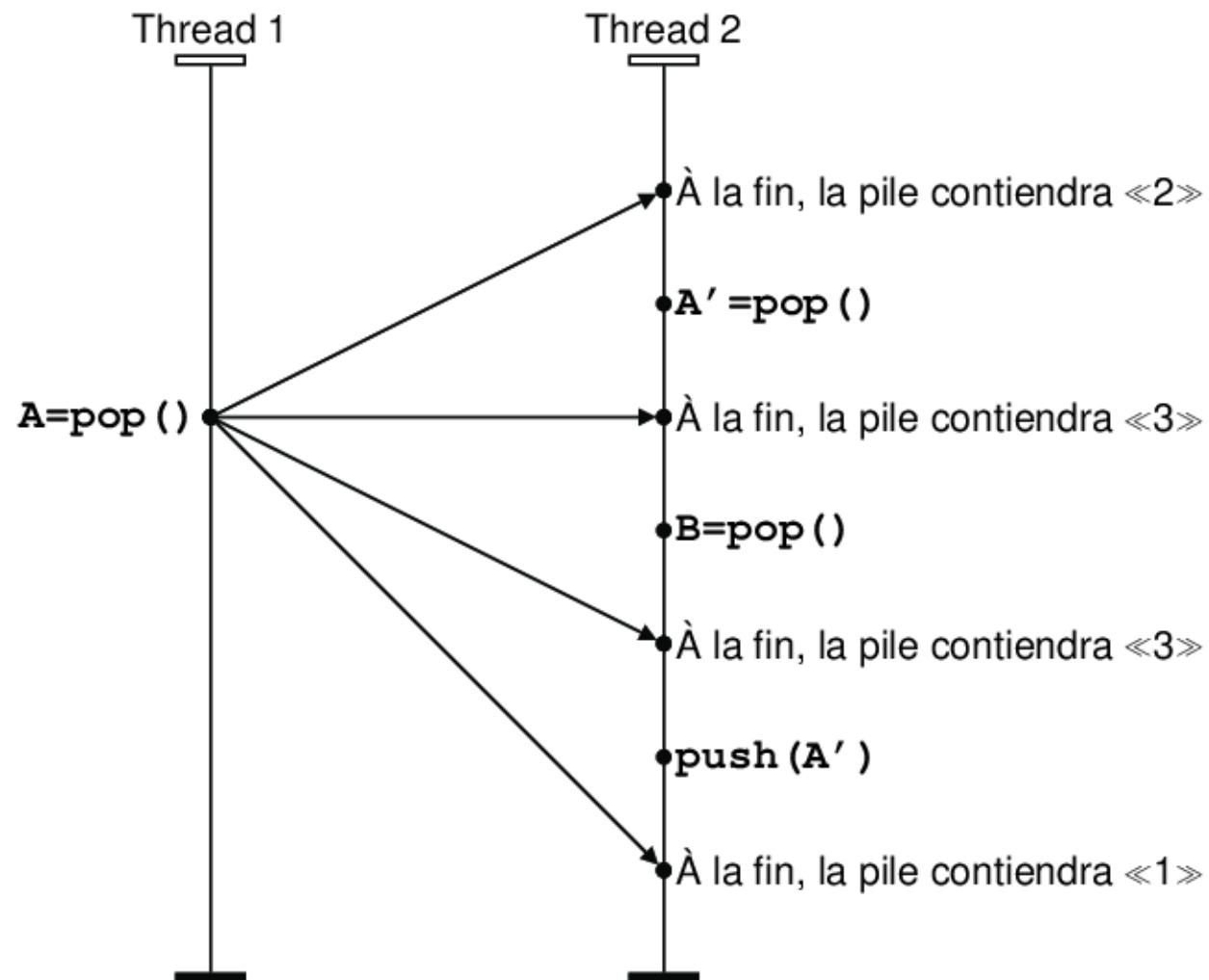
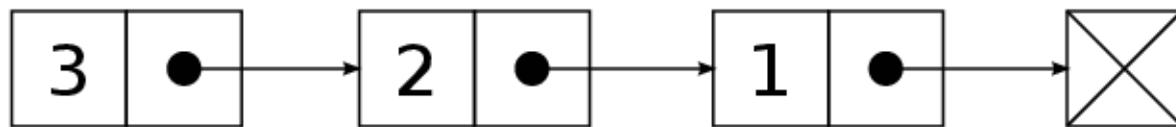


Un thread appelle **A=pop ()**.

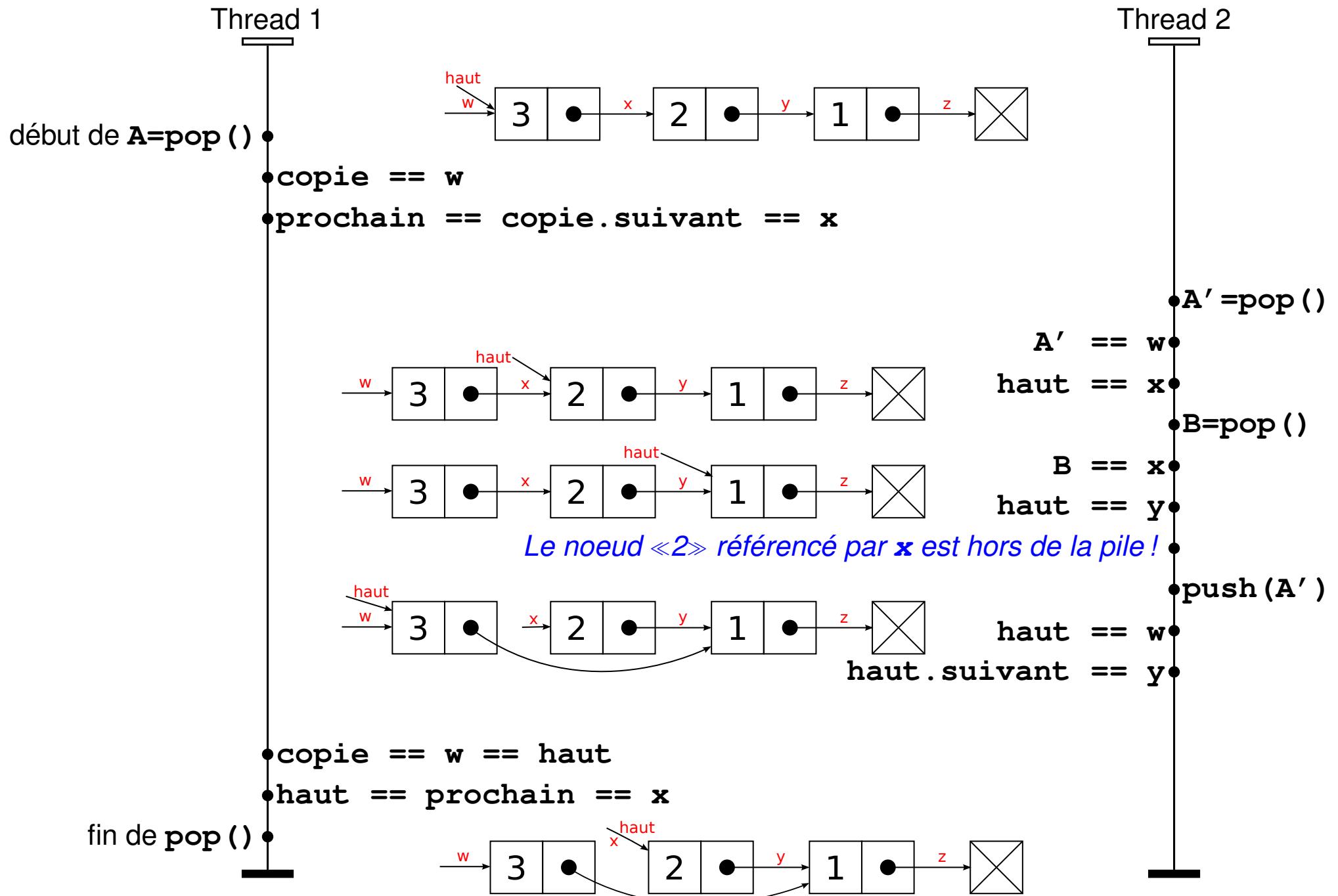
Un autre thread exécute **A' =pop () ; B=pop () ; push (A')**.

Combien de noeuds contiendra la pile à la fin ?

En considérant que les deux méthodes agissent atomiquement



Problème ABA (en Java) avec cette pile



- ✓ *Exemple du compte bancaire*
- ✓ *Exemple d'une pile concorrente*
- ✓ *Un mot sur les performances et la contention*
- ✓ *Le célèbre problème ABA*
- ✓ *Illustration du problème ABA en C*
- ✓ *Illustration du problème ABA en Java*
- 👉 *Quelques solutions au problème ABA*

Avec des noeuds immuables (1/2)

Avec des objets immuables, si la référence n'a pas changé, alors l'objet n'a pas changé pas non plus !

```
final class Noeud {  
    final Integer valeur ;  
    final Noeud suivant ;  
  
    public Noeud(Integer valeur, Noeud suivant) {  
        this.valeur = valeur;  
        this.suivant = suivant;  
    }  
}
```

Une fois construit, un noeud ne peut plus être modifié : il est **immutable**.

Avec des noeuds immuables (2/2)

```
public void push(Noeud entrée) {  
    Noeud copie;  
    Noeud nouveau;  
    do {  
        copie = haut.get(); // Référence du premier noeud  
        nouveau = new Noeud (entrée.valeur, copie);  
    } while ( ! haut.compareAndSet(copie, nouveau) );  
}
```

Le noeud **entrée** offert lors d'un **push()** ne peut pas servir lui-même à la construction de la pile, puisque l'attribut **suivant** doit être modifié. Un nouveau noeud doit donc être construit *à chaque tour de boucle*.

En cas de contention, les performances seront dégradées !

Cela suffit néanmoins à se prémunir en général contre le problème ABA.

Alternative : copie profonde en entrée

```
public void push(Noeud entrée) {  
    Noeud copie;  
  
    Noeud nouveau = new Noeud (entrée.valeur, null);  
    do {  
        copie = haut.get(); // Référence du premier noeud  
        nouveau.suivant = copie;  
    } while ( ! haut.compareAndSet(copie, nouveau) );  
}
```

Le noeud **entrée** offert lors d'un **push()** ne sert plus lui-même à la construction de la pile : on choisit au contraire de construire un nouveau noeud avec la même valeur.

Cela suffit à proscrire le problème ABA dans ce cas précis. Mais ça ne règle pas tous les problèmes potentiels, car les noeuds extraits de la pile lors d'un **pop()** permettent d'accéder à la structure de la pile restante.

Copie partielle en sortie, pour plus de sécurité

```
public Noeud pop() {  
    Noeud copie;  
    Noeud nouveau;  
    Noeud prochain;  
    do {  
        copie = haut.get();  
        if (copie == null) return null;  
        prochain = copie.suivant;  
        nouveau = new Noeud (copie.valeur, null);  
    } while ( ! haut.compareAndSet(copie, prochain) );  
    return nouveau;  
}  
}
```

Les noeuds de la pile ne sont plus publiés : seule la pile a accès aux divers noeuds de la pile...

Ce qu'il faut retenir

La *lecture optimiste* à l'aide d'un « stamped lock » produit assez souvent des résultats meilleurs que ceux observés avec les verrous de lecture-écriture.

Il est souvent souhaitable de *programmer sans utiliser de verrou*. Pour cela il existe des recettes à base d'*objets atomiques* qui reposent essentiellement sur l'instruction **compareAndSet ()** et une boucle d'itération en cas d'échec (qui s'apparente à une attente active).

Pour les données complexes, on utilise une *référence atomique* vers un objet. Les structures obtenues sont souvent aussi performantes que les structures synchronisées.

Néanmoins, il faut veiller à proscrire le *problème ABA*, s'il est susceptible d'apparaître. Les *objets immuables* sont un moyen efficace, mais parfois coûteux, pour s'en prémunir.