Multicore Programming
Projet
Mémoire transactionnelle logicielle

1 Introduction

En cours, nous avons vu deux façons génériques d'implémenter n'importe quel objet linéarisable. La première utilise des verrous : en empêchant physiquement des opérations de s'exécuter en concurrence, on est certain que la concurrence ne posera pas de problème. Le verrouillage à gros grain est certes très simple à mettre en œuvre mais réduit énormément le parallélisme, et donc les performances. Le verrouillage à grain fin est beaucoup plus difficile à mettre en œuvre. La seconde option se base sur le consensus ou des instructions spéciales, comme `compareAndSet`. Comme pour les solutions bloquantes, les constructions universelles non-bloquantes qui construisent un journal des opérations exécutées sont très inefficaces, et les algorithmes spécifiques à un objet donné, comme la pile, sont très complexes. Dans ce projet, nous explorons une troisième piste : les mémoires transactionnelles logicielles.

L'idée est la suivante : comme avec l'utilisation de verrous, le code à exécuter atomiquement doit être encadré par l'appel de deux fonctions, `begin` et `try_to_commit`. Entre les deux, il est possible de faire des lectures et des écritures sur des registres X en appelant les fonctions X.`read`() et X.`write`(v).

Les lectures, les écritures et l'invocation de la fonction `try_to_commit` peuvent *aborter*. Dans ce cas, on dit que la transaction complète aborte. Intuitivement, une transaction abortée n'a aucun effet et doit être redémarrée. Le critère de cohérence attendu sur les transactions est appelé *sérialisabilité (stricte)*[1]. Une exécution est sérialisable si elle est équivalente à une exécution séquentielle (c'est-à-dire dans laquelle il n'y a pas de concurrence entre les transactions) qui contient toutes (et ne contient que) les transactions qui n'ont pas aborté. Autrement dit, l'exécution dans laquelle toutes les transactions abortées ont été supprimées, est linéarisable.

La mémoire transactionnelle fournit les deux interfaces suivantes. Les fonctions `begin` et `try_to_commit` sont fournies comme des méthodes d'une classe implémentant `Transaction`. Cette interface fournit également une méthode `isCommited` retournant `true` si, et seulement si, `try_to_commit` a été appelée, s'est terminée sans lever d'exception, et `begin` n'a pas été appelée depuis. L'interface `Register<T>` fournit les méthodes permettant de lire et écrire dans un registre. Remarquez que les aborts sont ici gérés à l'aide d'exceptions.

[1] *Stricte* fait référence au temps réel, comme dans la linéarisabilité : si une transaction T_1 se termine avant que T_2 ne commence, T_1 devra être placée avant T_2 dans l'exécution séquentielle équivalente pour avoir le sérialisabilité stricte, mais ce n'est pas demandé pour la sérialisabilité.

```
interface Transaction {
  public void begin();
  public void try_to_commit() throws AbortException;
  public boolean isCommited();
}
interface Register<T> {
  public T read(Transaction t) throws AbortException;
  public void write(Transaction t, T v) throws AbortException;
}
```

Le code suivant présente un exemple d'utilisation de la mémoire transactionnelle. Il s'agit essentiellement d'une transaction contenant une lecture suivie d'une écriture, au sein d'une boucle réessayant d'exécuter la transaction jusqu'à ce qu'elle soit acceptée.

```
void increment (Register<Integer> X) {
  Transaction t = new STMTransaction();
  while (!t.isCommited()) {
    try {
      t.begin();
      X.write(t, X.read(t) + 1);
      t.try_to_commit();
    } catch (AbortException e) {}
  }
}
```

2 Algorithme *Transactional Locking 2*

Beaucoup d'algorithmes ont été proposés pour implémenter la mémoire transactionnelle, dont les plus simples sont expliqués par l'article de Raynal et Imbs [3]. La lecture de cet article, en ligne sur Madoc, est vivement recommandée. L'algorithme 1 présente Transactional Locking 2 (TL2). Il est basé sur une horloge logique globale, clock, incrémentée à chaque fois qu'une transaction est sur le point d'être acceptée (ligne 20), et qui décrit l'ordre (ou plutôt *un* ordre) dans laquelle les transactions acceptées sont sérialisées.

Chaque registre X accessible dans une transaction comporte deux champs : un champ de données X.`value` qui contient la dernière valeur écrite et un champ de contrôle X.`date` qui contient la date de la dernière transaction acceptée à avoir écrit dans X. De plus, un verrou est associé à chaque registre.

Chaque transaction T gère une variable locale lrs_T contenant l'ensemble des variables lues, une variable locale lws_T contenant l'ensemble des variables écrites au cours de T et une variable birthDate contenant la valeur de clock au début de la transaction.

Lors d'une écriture sur X, une copie locale de X est créée pour conserver la valeur écrite. De plus X est placée dans l'ensemble lws_T. Lors d'une lecture de X, si X a été précédemment écrite par la même transaction, la valeur locale est retournée. Sinon, X est placée dans l'ensemble lrs_T, puis il est vérifié que X n'a pas été modifiée depuis le début de la transaction en comparant la date de X à birthDate (auquel cas l'abort est nécessaire) et la valeur en mémoire partagée est retournée.

La fonction `try_to_commit` est bloquante. Lorsqu'elle est appelée, la transaction prend les verrous sur tous les registres puis vérifie que les valeurs lues sont toujours cohérentes entre elles, et cohérentes avec les nouvelles valeurs qu'elle a écrites localement. Cela est fait en comparant la date actuelle de chaque registre lu à la date birthDate du début de la transaction. Si l'une de ces dates est plus grande, un registre a été écrit par une transaction concurrente et un abort est nécessaire. Sinon, la transaction peut être acceptée : les écritures sont reportées sur la mémoire partagée, à une date obtenue en incrémentant l'horloge globale.

Algorithm 1: Algorithme TL2

```
1  operation T.begin()
2      re-initialize local variables;
3      birthDate ← clock;
4  operation X.write(T, v)
5      if there is no local copy lcx of X then
6          allocate local space lcx for a copy
7      lcx.value ← v; lws_T ← lws_T ∪ {X};
8  operation X.read(T)
9      if there is a local copy lcx of X then
10         return lcx.value;
11     else
12         lcx ← X.copy; lrs_T ← lrs_T ∪ {X};
13         if lcx.date > birthDate then abort;
14         else return lcx.value;
15 operation T.try_to_commit()
16     lock all the objects in (lrs_T ∪ lws_T);
17     foreach X ∈ lrs_T do
18         if X.date > birthDate then
19             release all the locks; abort;
20     commitDate ← clock.getAndIncrement();
21     foreach X ∈ lws_T do
22         X ← (lcx.value, commitDate);
23     release all the locks;
```

3 Travail demandé

Votre travail consiste à implémenter une mémoire transactionnelle logicielle fonctionnant selon l'algorithme TL2. Chaque fonction devra être starvation-free. Vous devrez rendre deux fichiers :

- un rapport succinct (au plus 4 pages) au format .pdf. Vous y discuterez en particulier de la propriété de vivacité sur les transactions garantie par l'algorithme TL2.
- une archive contenant votre code en Java, y compris :
  - une implémentation de TL2,
  - une implémentation d'objets partagés de votre choix utilisant des transactions,
  - une application multi-thread utilisant ces objets.

4 Références

Le concept de mémoire transactionnelle matérielle a été proposé par Herlihy et Moss en 1993 [1]. L'algorithme TL2 a été découvert par Dice, Shalev et Shavit en 2006 [2]. La présentation de l'algorithme et la formulation de ce sujet ont été largement inspirées d'un article de Imbs et Raynal de 2009 [3].

[1] Maurice Herlihy et J. Eliot B. Moss. *Transactional memory: architectural support for lock-free data structures.* International Symposium on Computer Architecture (1993)

[2] Dave Dice, Ori Shalev et Nir Shavit. *Transactional Locking II.* International Symposium on Distributed Computing (2006)

[3] Damien Imbs et Michel Raynal. *Software transactional memories: an approach for multicore programming.* International Conference on Parallel Computing Technologies (2009)

De nombreuses implémentations des mémoires transactionnelles logicielles sont disponibles en Java, même si aucune n'est universellement acceptée. Voici des liens vers quelques unes des plus connues.

AtomJava : `https://wasp.cs.washington.edu/wasp_atomjava.html`

JVSTM : `http://inesc-id-esw.github.io/jvstm`

Deuce STM : `https://sites.google.com/site/deucestm`

Multiverse : `https://github.com/pveentjer/Multiverse`

DSTM2 : `http://www.oracle.com/technetwork/indexes/downloads/index.html`

ObjectFabric : `https://github.com/cypof/objectfabric`

Étant données les limitations des mémoires transactionnelles logicielles, beaucoup de chercheurs militent pour l'introduction de mémoires transactionnelles matérielles, dans lesquelles les mêmes propriétés sont assurées par l'architecture physique de la mémoire partagée, au même titre que les caches.