Concurrent Building Blocks (Part 2)

Concurrent and Parallel Programming

15 marzo 2022

Insight of Concurrent Collections

- To allow for good performances, all the concurrent building blocks have been developed by using advanced concurrent programming techniques.
- In particular, non-blocking algorithms are exploited.

Non-blocking algorithms

- Non-blocking algorithms allow more than one thread to compete for a shared resource, without indefinitely postponing the execution with mutual exclusion (locks).
- A non-blocking algorithm is:
  - lock-free if global advancement is granted (but it could not be the case for single thread advancement).
  - wait-free if advancement is granted even at the single thread level.

- Warning: wait-free algorithms are very complex to develop and are therefore rare. They can also be slow because of more required code.

Lock-free algorithms

- Lock-free algorithms are mainly based on optimistic locking (by taking advantage of CAS operations).
- As a consequence:
  - provide excellent scalability - performance degrades slowly as threads increase.
  - have good liveness properties - in general, each thread is able to progress regularly.
  - are immune to deadlocks.
- Lock-free algorithms are particularly useful when many threads are competing for the same resources.

SUPSI

Dipartimento tecnologie innovative

Example

```java
public class ConcurrentStack<E> {
    AtomicReference<Node<E>> top = new AtomicReference<>();

    public void push(Node<E> node) {
        Node<E> newHead = node;
        Node<E> oldHead;
        do {
            oldHead = top.get();
            newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead, newHead));
    }

    public Node<E> pop() {
        Node<E> oldHead;
        Node<E> newHead;
        do {
            oldHead = top.get();
            if (oldHead == null)
                return null;
            newHead = oldHead.next;
        } while (!top.compareAndSet(oldHead, newHead));
        return oldHead;
    }
}
```

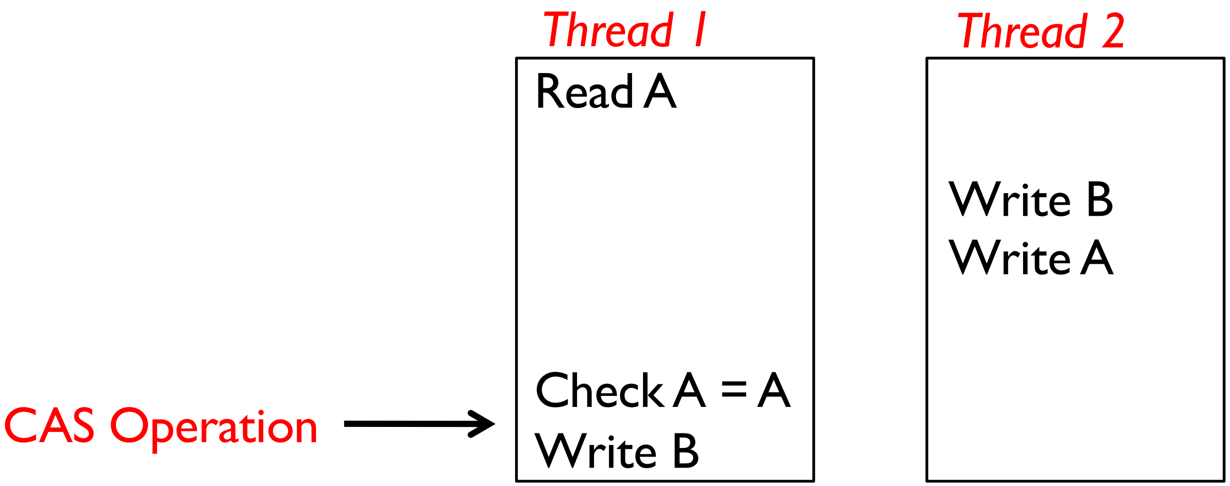
But pay attention to the ABA problem!

```java
public class Node<E> {
    public final E item;
    public Node<E> next;

    public Node(E item) {
        this.item = item;
    }
}
```

ABA problem

- If the value of a variable V is A before V is modified, the algorithm might fall into confusion, if at the same time, the value is modified from A to B and then back to A. In this situation, the initial modification is successful, but the result might not be the desired one.



ABA problem

Thread 1	Thread 2	ConcurrentStack
		A → B → C
Start pop(A) oldHead = A newHead = B		
	Start and complete pop(A)	B → C
	Start and complete pop(B)	C
	Start and complete push(A)	A → C
Complete pop(A) CAS OK		B ConcurrentStack corrupt!

Example

```java
public class ConcurrentStack<E> {
    AtomicReference<Node<E>> top = new AtomicReference<>();

    public void push(E item) {
        Node<E> newHead = new Node<>(item);
        Node<E> oldHead;
        do {
            oldHead = top.get();
            newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead, newHead));
    }

    public E pop() {
        Node<E> oldHead;
        Node<E> newHead;
        do {
            oldHead = top.get();
            if (oldHead == null)
                return null;
            newHead = oldHead.next;
        } while (!top.compareAndSet(oldHead, newHead));
        return oldHead.item;
    }

    private static class Node<E> {
        public final E item;
        public Node<E> next;

        public Node(E item) {
            this.item = item;
        }
    }
}
```

Solution

SUPSI

Dipartimento tecnologie innovative

Thread-safety delegation

- In some situations, an object composed of thread-safe objects is also automatically thread-safe. In other situations, it is just a good starting point.
- Thread-safety delegation, performed by a class to the contained objects might require additional protection.

Example (1/4)

```
public class VehicleTracker {
    private final Map<String, Point> locations;

    public VehicleTracker(Map<String, Point> points) {
        locations = points;
    }

    public Map<String, Point> getLocations() {
        return locations;
    }

    public Point getLocation(String id) {
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y) {
        if (!locations.containsKey(id))
            throw new IllegalArgumentException("invalid vehicle name: " + id);
        locations.get(id).set(x, y);
    }
}
```

```
class Point {
    private int x, y;

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void set(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Example (2/4)

```java
public class VehicleTrackerV2 {
    private final ConcurrentMap<String, Point> locations;

    public VehicleTrackerV2(Map<String, Point> points) {
        locations = new ConcurrentHashMap<>(points);
    }

    public Map<String, Point> getLocations() {
        return locations;
    }

    public Point getLocation(String id) {
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y) {
        if (!locations.containsKey(id))
            throw new IllegalArgumentException("invalid vehicle name: " + id);
        locations.get(id).set(x, y);
    }
}
```

```java
class Point {
    private int x, y;

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void set(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Example (3/4)

```java
public class VehicleTrackerV3 {
    private final ConcurrentMap<String, ImmPoint> locations;

    public VehicleTrackerV3(Map<String, ImmPoint> points) {
        locations = new ConcurrentHashMap<>(points);
    }

    public Map<String, ImmPoint> getLocations() {
        return locations;
    }

    public ImmPoint getLocation(String id) {
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y) {
        if (locations.replace(id, new ImmPoint(x, y)) == null)
            throw new IllegalArgumentException("invalid vehicle name: " + id);
    }
}
```

```java
final class ImmPoint {
    private final int x, y;

    public ImmPoint(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}
```

Example (4/4)

```
public class DelegatingVehicleTracker {
    private final ConcurrentMap<String, ImmPoint> locations;

    public DelegatingVehicleTracker(Map<String, ImmPoint> points) {
        locations = new ConcurrentHashMap<>(points);
    }

    public Map<String, ImmPoint> getLocations() {
        return Collections.
                unmodifiableMap(new HashMap<>(locations));
    }

    public ImmPoint getLocation(String id) {
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y) {
        if (locations.replace(id, new ImmPoint(x, y)) == null)
            throw new IllegalArgumentException("invalid vehicle name: " + id);
    }
}
```

```
final class ImmPoint {
    private final int x, y;

    public ImmPoint(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Getters …
}
```

ConcurrentHashMap +
Immutable Point +
Returns Unmodifiable Map

Thread-safety delegation

- It is possible to delegate the thread-safety to the objects contained in the class, if these objects are independent from one another (there are no invariants that simultaneously involves the objects).
- Instead, if a class implements compound actions, delegation to the contained objects is not sufficient to grant thread-safety.

Example

```java
class ListHelper<E> {
    public List<E> list = Collections.synchronizedList(new ArrayList<E>());

    public boolean putIfAbsent(E x) {
        boolean absent = !list.contains(x);
        if (absent)
            list.add(x);
        return absent;
    }
}
```

Extension of thread-safe classes

- The safest way to add atomic operations to a class is by directly modifying the class.
- ... but this solution is not always possible ...
- As alternatives, it is possible to:
  - extend the class by inheritance: not robust because the synchronization policy is distributed over several classes. In addition, inheritance is not always allowed.
  - develop a helper class using client-side locking: even less robust, because the relationship between the classes is weaker than inheritance.
  - develop a wrapper class with composition and delegation.

Client-side locking

```java
class GoodListHelper<E> {
    public List<E> list = Collections.synchronizedList(new ArrayList<E>());

    public boolean putIfAbsent(E x) {
        synchronized (list) {
            boolean absent = !list.contains(x);
            if (absent)
                list.add(x);
            return absent;
        }
    }
}
```

- It has to be known, which is the internally used lock.
- In the case of concurrent collections, this approach is not allowed.

Composition and Delegation

- The most robust approach to add functionality to a thread-safe class is by using composition and delegation.
- At the practical level, this solution is very similar to the approach used by synchronized collections. Performances might get compromised.
- The class that has to be extended has to be included in a wrapper class by composition and delegation. The wrapper class has to manage all the synchronization needs (with locks, atomic variables and other tools).
- With this technique, compound actions can be added to any type of collection, including standard concurrent collections.

Composition and Delegation

```
public class ImprovedList<T> implements List<T> {
    private final List<T> list;

    public ImprovedList(List<T> list) {
        this.list = list;
    }

    public synchronized boolean putIfAbsent(T x) {
        boolean contains = list.contains(x);
        if (contains)
            list.add(x);
        return !contains;
    }

    // Plain vanilla delegation for List methods.
    public synchronized int size() {
        return list.size();
    }

    public synchronized boolean add(T e) {
        return list.add(e);
    }

    // ...
}
```

Could also use other techniques

Summary of topics

- Non-blocking algorithms and the ABA problem
- Thread-safety delegation
- Extension of thread-safe classes