Principles of Concurrency

Lecture 5
Concurrent Data Structures

Material adapted from Herlihy and Shavit, Art of Multiprocessor Programming, Chapters 9 and 10
The Problem

- A concurrent data structure:
  - has state manipulated by a number of methods
  - these methods can be invoked concurrently

- How should we coordinate access?
  - Coarse-grained synchronization:
    - protect all methods with a lock
    - even if acquiring an releasing a lock is efficient, the level of concurrency admitted is low
  - Alternatives:
    - fine-grained synchronization:
      - logically break-up the object into multiple pieces
      - induce coordination only when methods interfere with the accesses they perform
    - optimistic synchronization:
      - assume conflicts won’t occur (don’t use any kind of synchronization)
      - validate the assumption was correct post-facto
    - lazy synchronization:
      - split a method action into multiple parts, deferring expensive operations
    - non-blocking synchronization
      - eliminate locks entirely, relying on low-level atomic primitives (e.g., CAS)
Running Example

A set specification:

```kotlin
data class Set<T> = {
    val add : T -> Boolean
    val remove: T -> Boolean
    val contains: T -> Boolean
}
```

- `add(x)` adds `x` to the set and returns true only if the set did not contain `x` previously
- `remove(x)` removes `x` from the set and returns true only if `x` was in the set previously
- `contains(x)` returns true if the set contains `x`

Implementation:

```kotlin
data class Node<T> = {
    val item : T
    val key : Integer
    val next : Node
}
```

```kotlin
val set : List<Node>
```
The **Node** class keeps track of the item, the item’s key, and the next node in the list. Some algorithms require technical changes to this class.

**Figure 9.3** A sequential Set implementation: adding and removing nodes. In Part (a), a thread adding a node uses two variables: `curr` is the current node, and `pred` is its predecessor. The thread moves down the list comparing the keys for `curr` and `b`. If a match is found, the item is already present, so it returns false. If `curr` reaches a node with a higher key, the item is not in the set so Set `b`’s next field to `curr`, and `pred`’s next field to `b`. In Part (b), to delete `curr`, the thread sets `pred`’s next field to `curr`’s next field.

and key throughout any given example, which allows us to abuse notation and use the same symbol to refer to a node, its key, and its item. That is, node `a` may have key `a` and item `a`, and so on.

The list has two kinds of nodes. In addition to regular nodes that hold items in the set, we use two sentinel nodes, called `head` and `tail`, as the first and last list elements. Sentinel nodes are never added, removed, or searched for, and their keys are the minimum and maximum integer values.

Ignoring synchronization for the moment, the top part of Fig. 9.3 schematically describes how an item is added. All algorithms presented here work for any ordered set of keys that have maximum and minimum values and that are well-founded, that is, there are only finitely many keys smaller than any given key. For simplicity, we assume here that keys are integers.

**The key** is a hash of the node’s value used to order elements in the list.
Properties

Want the implementation to preserve useful properties expressed by the specification

Assumption:
Freedom from interference: only the set implementation’s methods have access to the list representation

Relate the abstraction (a set of values) with the implementation (an ordered list of nodes) using representation invariants:
- Constraints on the representation that allow it behaviorally act like a set
- Serves as a contract among the implementation’s methods
  - Sentinels are never added or removed
  - Every node (except the tail) in the list is reachable from the next field of another node
  - Distinct values always have distinct keys
  - ...

Safety properties:
Concurrent operations on a set are linearizable (i.e., methods appear to execute atomically): a set of concurrent method invocations always produces a state that represents a valid sequential execution
fun <T>insert(item : T) = {
    head.acquire()
    val pred = head
    val curr = pred.next
    curr.acquire()
    while (curr.key < key) {
        pred.release()
        pred = curr
        curr = curr.next
        curr.acquire()
    }
    if (curr.key == key) {
        return false
    }
    val newNode = new Node(item)
    newNode.next = curr
    curr,acquire()
    pred.next = curr
    curr.unlock();
    finally
    pred.unlock();
}

Exercise: implement remove()
Fine-grained locking is an improvement over coarse-grained locking, but:
- potentially involves long series of lock acquisitions and releases
- induces bottlenecks when a thread is manipulating the earlier part of the list

Alternative approach:
- optimistically traverse the list (without using locks)
- when ready to insert, validate that the predecessor and successor nodes are still reachable (use locks for this purpose)
- similar protocol for remove
Lazy Synchronization

- Optimistic synchronization works well if the cost of traversing a list twice (without locks) is faster than traversing it once with locks
- Can we improve this so that insert() and remove() traverse a list only once

Add an extra marked bit to every node indicating if it is reachable from the head

Invariant: every unmarked node is reachable

No need to lock target node
Insert locks predecessor
Remove: (1) marks the target node (logical removal)
(2) updates the predecessor to point to the target’s successor (physical removal)
Validation checks if the predecessor and current nodes are not marked and the current predecessor still points to the current node
Lazy Synchronization

Chapter 9
Linked Lists: The Role of Locking

After unsuccessful list depends only on whether it is marked.

Thread sees that insertion of the new node with key unsuccessful it finds the marked node the list. Linearizing thread the removed section of the list leading to

Notice that nodes with items traversing the list, a concurrent call disconnects the sublist referred to by a call to when it found the marked node a's call is linearized at the point when it contains() method at the point

A is traversing the list leading to marked node a is traversing the list leading to marked node. It would be wrong to linearize thread after unsuccessful a's call is linearized at the point when it contains() method at the point.

A is traversing the list, another thread adds a new node with key unsuccessful it finds the marked node the list. Linearizing thread the removed section of the list leading to

Now let us consider the scenario depicted in Part (b). While thread a adds a new node with key unsuccessful it finds the marked node the list. Linearizing thread the removed section of the list leading to

A is traversing the list, another thread adds a new node with key unsuccessful it finds the marked node the list. Linearizing thread the removed section of the list leading to

Reasoning about contains()
All previous approaches involve locks at some point in the implementation. Can we devise a solution that eliminates locks altogether?

Need a way to ensure a node’s fields cannot be updated after it has been logically or physically removed. Approach: treat a node’s next and marked fields as an atomic unit: attempting to update the next field if the marked bit is set will fail.

Use a variant of compare-and-swap() to achieve this behavior.

The need for atomic update of mark and next fields.
fun <T>add(item :T) = {
    val key = item.hash()
    while (true) {
        val (pred, curr) = find(head, key)
        if (curr.key == key) {
            return false
        } else {
            val node = new Node(item)
            node.next = AtomicMarkableReference(curr, false)
            if pred.next.CAS(curr, node, false, false) {
                return true
            }
        }
    }
}