Problem Solving in Computer Science

Scribe on duty: Vitaly Chipounov

13.11.2008

Concurrent Reasoning for Linked Lists

In this lecture Verena presented and explained different implementations of concurrent lists in Java. These implementations differ in the way they use locking and have different properties. These different examples are taken from the book *The Art of Multiprocessor Programming*, by Maurice Herlihy and Nir Shavit [1].

Coarse-grained Synchronization

This algorithm locks the whole list during the `add()` and `remove()` operations. This means that only one of these operations can be carried out at a time. The algorithm inherits progress conditions from the `lock()` operation.

Fine-grained Synchronization

Contrary to the coarse-grained algorithm, this method traverses the list and locks only those nodes currently used. The main drawback is that concurrent accesses to the list cannot go beyond the nodes locked by the slowest process. This algorithm also inherits progress conditions from the `lock()` operation.

Optimistic Synchronization

The optimistic synchronization, contrary to the previous algorithm, does not lock every traversed node, but only the node of interest. It has the following steps:

- Search without locking
- Lock the nodes of interest and check correctness by traversing the list once more
- Repeat if validation fails

The algorithm works well if traversing the list is cheap. However, it is not wait-free even if the lock is so. Indeed, the algorithm could keep validating the list, being stuck in the while loop.

Lazy Synchronization

This algorithm extends the node data structure by adding to it a boolean variable "marked"which marks the nodes for deletion. The algorithm preserves the following invariant: if "marked" is false then the node is reachable from the head.

- `contains()` needs no locks.
- `add()` traverses the list only once.
- `remove()` marks the target before removing.
- `add()` and `remove()` are not wait-free.

Non-Blocking Synchronization

The algorithm is organized as follows:

- First idea: mark nodes without locking. It does not work because the next pointer of marked nodes could be changed meanwhile.
- Solution: treat "next" and "marked" as a single atomic unit: "next" can only be changed if "marked" is false.
- Problem in `remove()`: if thread A wants to remove $curr_A$ but $pred_A$ (or even more predecessors) is marked, it has to remove $pred_A$ physically before $curr_A$ can be removed physically.
- Idea: all threads doing `add()` and `remove()` "clean" the list by physically removing marked nodes.

The non-blocking synchronization method uses atomic operations to guarantee its safety. The Java implementation of the algorithm can use the class `AtomicMarkableReference<T>` from the package `java.util.concurrent.atomic`. This class contains the following important methods:

- `boolean compareAndSet(V expectedReference, V newReference, boolean expectedMark, boolean newMark)`
- `T get(boolean[] markHolder)`

References

[1] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann, 2008.