Advanced Programming
Programming Assignment 2

Simone Pignotti

December 6, 2015

The three main classes, implementing the requested features, are FineConcurrentDictionary, LazyConcurrentDictionary and LockFreeConcurrentDictionary, the three of them implementing the interface MyDictionary. The class Test contains the `main()` method and calls instances of the classes PutTestThread, RemoveTestThread and ReplaceAndGetTestThread. Even though the dictionary entries have similar structure for the first two versions, I decided to implement them separately since in the lock-free version, because of the absence of multiple inheritance in Java, it would have been impossible to extend both AtomicMarkableReference and the abstract DictionaryEntry class. I have added some exceptions to the `put()` and `replace()` methods' signatures, in order to avoid the insertion of null keys and values into the dictionary (both for a logical reason and not to confuse them with the sentinels which I have implemented as null key entries) and the exceeding of the capacity. This last component is fixed and instanciated by the constructors. I have not specified proper signatures for the methods, nor I have commented the code since it seems quite straightforward and very similar to the pseudocode seen during the lectures. Also the linearization properties are the same as the ones defined for the set data structure we used as example. On the other hand, I have focused on the correctness of the methods and on their optimization. In the following paragraphs some details are provided about the three versions of the dictionary and about the results obtained.

Fine-grained Synchronization In this version, entries contain a ReentrantLock which is used to lock the nodes while scanning the list. The size is implemented as an AtomicInteger, in order to provide the increment feature atomically, and in the put method an ad-hoc private method (`incrementSize()`) is called which tries to increment the size but it decrements it and throws an exception in case the dictionary is full. This strategy has also been used for the other two versions. The `search(K key)` private method scans the list till it reaches an element with a key greater or equal than its parameter, returning it still locked. Also its successor will still be locked after it returns.

Lazy Synchronization In the lazy version, entries have the same exact structure, apart from the variable mark which states if the entry has been logically removed. In this case the search method does not lock any entry, but this is done after it returns. The `validate(key,pre,cur)` private method, together with checking that the predecessor is not marked and it still points to the current node, checks that no other entry has been inserted after the precedent node before the instruction `cur = pre.getNext()` performed when the search returns.

Lock-free Synchronization The DictionaryEntry class implemented in this method extends the `AtomicMarkableReference` class contained in the `java.util.concurrent` package. The Reentrant lock of the previous implementations disappears, together with the next field which is replaced by the reference field of the extended class. There are two variations of the search method: `searchAndClean(key)` used by the put and remove methods (whose duty is also to remove the marked entries they run into) and the simpler `search(key)` which just scans the list.

Testing and Evaluation After having checked the sequential behaviour of every method, I created a Test class which stresses the three implementations by calling each method repeatedly from a certain number of threads. The parameters I used for my tests are 50 threads working on a 10000 entries capacity dictionary which is completely filled by put method calls, but they can be changed at any time in the Test class. After a stress test on the most "dangerous" operations, which are simultaneous (made more probable thanks to a barrier right before the operations) adds and removes on adjacent nodes, all the kinds of operations are executed for the same number of entries, and the average results are shown in the table below. While the very significant difference between the fine-grained and the lazy versions is not surprising at all, I was quite disappointed by the performances of the lock-free version: compared to the lazy one, in facts, it turns out to be often less efficient, although in average it is slightly faster. I thought it can be due to the machine I am using, which has just two cores and maybe cannot exploit all the advantages of the lock-free implementation which are higher as the number of thread concurrently working on the data structure increases. Furthermore, it sporadically happens that with big instances it misses one or two removes during the functionality test, but unfortunately I did not manage to fix this bug on time.

	Fine-grained	Lazy	Lock-free
Put	2043	95	136
Remove	907	179	147
Replace + Get	1183	192	173

Methods completion time in milliseconds