Java Concurrent Exam Project

Simon Munch D2
simun15@student.sdu.dk
DM519

April 24, 2016

Contents

1 Methodology

Given the fact that i expected to be working with a large amount of threads, I quickly decided upon using a ExecutorService, as this would allow me to easily and effectivly keep track of many threads at the same time. Setting a fixed thread pool would also allow me to account for diferent amount of cores available to the code. As locks are a necessity when working with concurrent programing, I also decided to work with synchronized lists, futurelists and concurrent hash maps.

When it came to how i would split the work up, I considered creating a thread for every directory found by a recursive search to be given to a thread which then split up the work in this directory among other threads. I quickly ran into problems however, as i ended i deadlocks and had trouble finding a way the program would know when all directories have been visited, and when all the work in the directories have been sucessfully split up. Due to theese problems, I instead chose to have the program visit all txt files one at a time, split up the work in the text file and send this work as threads to the executor. This avoided deadlocks, made coding easier (fewer lines of code) and made me able to more seemlesly use the executorservice shutdown function to identify when all work had been done.

2 Advantages

2.1 findAll

To avoid waiting for locks on the synchronizedList, i use a futurelist for the results, and then add the results to the final list once the threads have done their work. Thus avoiding excessive wait on locks and enabling the executor to more swiftly move on to the next threads in the queue.

2.2 findAny

Due to the very similar nature of this method to findAll, i could reuse most of my code. I was worried what would happen should two threads find the word bieng looked for at the same time, so i decided upon having a synchronized list in common that the thread would insert the found result in, so should two threads find a result and insert them into the list before the executor terminated all threads, then it would not mess up the program.

2.3 Stats

Using a single concurrent hash map between the many threads, enabled me to quickly count the number of times a word appeard in a text. This made my code both fast and easy on memory consumption.

3 Limitations

3.1 findAll

As the future list 'get()' waits for all threads to be completed before moving on, should a single thread take an unexpected long amount of time (a single line bieng massive), then this will bottleneck the code, and will mean that all the other threads will be waiting for this work to be completed before moving on to other tasks.

3.2 Stats

I first considered to have the Hash map in Stats contain the word and a list of results as seen below:

```
Map< String , List < Result > >
```

As a list contains an index for the number of elements contained on the list, i would easily and quickly be able to find the number of times a word was found in a text. As i wanted to avoid looking through all the text files in the directories twice, i thougth that having all the informaiton i would need in a single Stats would be faster then just counting, and then calling findAll when *foundIn* would be called. However, this not only turned out to be so memory heavy that i got garbage collector and heap space errors, it also turned out to be slower even on smaller directories. So i decided against doing it this way. The code i did end up with ran faster, but as the concurrent hash map has locks for segments of it, should the same word appear a large amount of times compared to other words, the hashmaps locks might end up bieng a bottleneck for the code, as many threads would be waiting for the same lock.