Software Engineering Concepts: Assignment 1 Report

Name:

Lakshan Martin

ID:

13983521

Submission Date:

16th September 2021

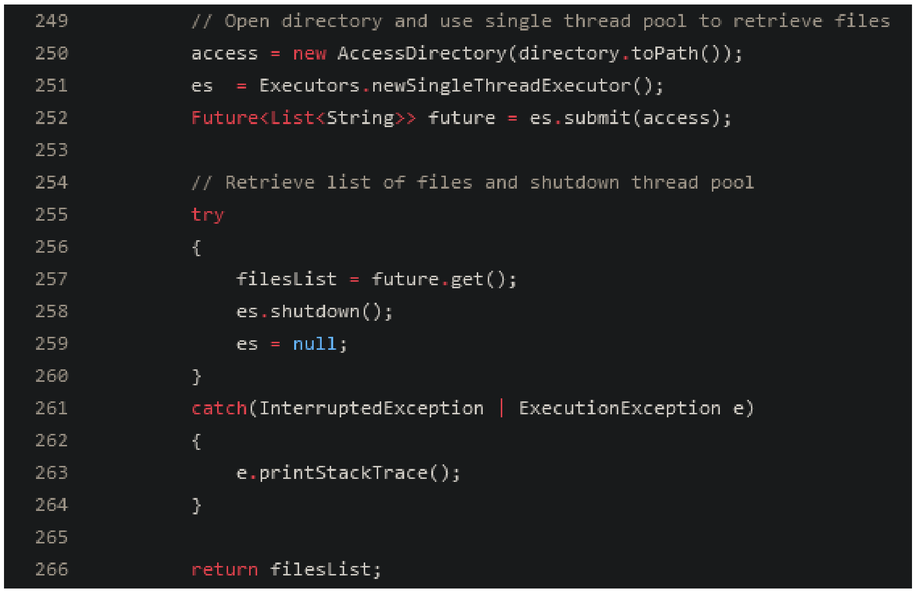
Contents

Multithreading Design Choices

During the planning stage for this assignment, I recognised that there were several clear and distinct steps involved with the file comparison process. The collection of files, sorting of files, and finally the comparison itself. This realization formed the foundation of my subsequent design choices, as to be discussed within this report.

Starting threads & their purpose

The first action taken after the user selects a directory, containing files to be compared, is to analyse and input each valid file into a list. Given that I couldn't think of a way to utilise multithreading in this process, I opted to create a single thread pool using an ExecutorService. This was started from the UserInterface class within the getFilesList() method. An object of the AccessDirectory class is then used through a callable future to retrieve the list of files to be compared.



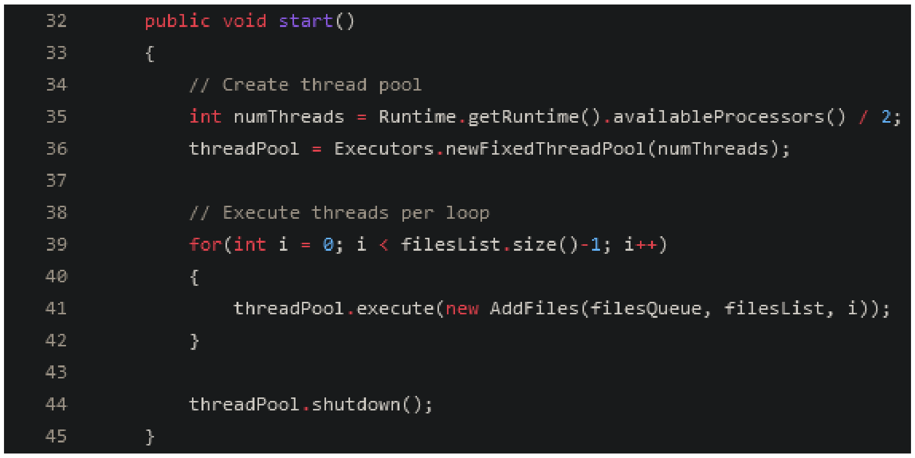
```
        // Open directory and use single thread pool to retrieve files
        access = new AccessDirectory(directory.toPath());
        es  = Executors.newSingleThreadExecutor();
        Future<List<String>> future = es.submit(access);

        // Retrieve list of files and shutdown thread pool
        try
        {
            filesList = future.get();
            es.shutdown();
            es = null;
        }
        catch(InterruptedException | ExecutionException e)
        {
            e.printStackTrace();
        }

        return filesList;
```

Figure 1: Portion of getFilesList() method from UserInterface class

The second major action, is collating the pairs of files to be compared. I have implemented a separate class, called 'CollectionPool', to manage this process. This class is responsible for the set up of the first thread pool I've employed to increase the overall file comparison speed. This pool will execute new threads based on the number of items in the List of files. These threads created, then go on to be used to add files to be compared to a BlockingQueue in a concurrent fashion. This represents the first performance improvement over single threaded processing.



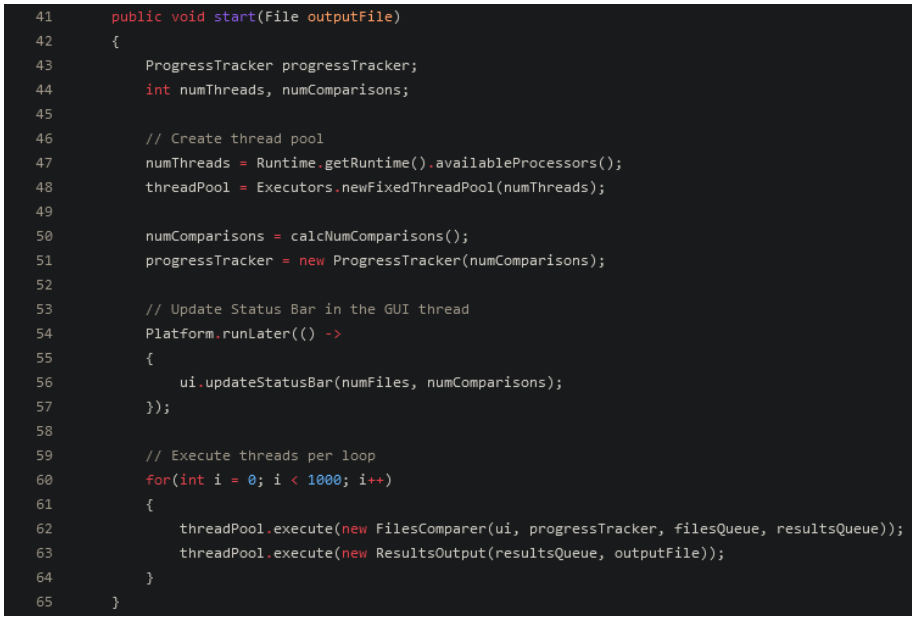
```java
    public void start()
    {
        // Create thread pool
        int numThreads = Runtime.getRuntime().availableProcessors() / 2;
        threadPool = Executors.newFixedThreadPool(numThreads);

        // Execute threads per loop
        for(int i = 0; i < filesList.size()-1; i++)
        {
            threadPool.execute(new AddFiles(filesQueue, filesList, i));
        }

        threadPool.shutdown();
    }
```

Figure 2: Thread pool start() from the CollectionPool class

The third major action, is the actual comparison and output of results. Again, this presented another opportunity to utilise multithreading to vastly improve the efficiency and speed of this process. I created another class, called 'CompareResultsPool', to start another thread pool. The threads within this pool are used to execute the actual file comparison and output the results to a CSV file. Again, this is done in a concurrent fashion with observable improvements over a single threaded application.



```java
    public void start(File outputFile)
    {
        ProgressTracker progressTracker;
        int numThreads, numComparisons;

        // Create thread pool
        numThreads = Runtime.getRuntime().availableProcessors();
        threadPool = Executors.newFixedThreadPool(numThreads);

        numComparisons = calcNumComparisons();
        progressTracker = new ProgressTracker(numComparisons);

        // Update Status Bar in the GUI thread
        Platform.runLater(() ->
        {
            ui.updateStatusBar(numFiles, numComparisons);
        });

        // Execute threads per loop
        for(int i = 0; i < 1000; i++)
        {
            threadPool.execute(new FilesComparer(ui, progressTracker, filesQueue, resultsQueue));
            threadPool.execute(new ResultsOutput(resultsQueue, outputFile));
        }
    }
```

Figure 3: Thread pool start() from the CompareResultsPool class

Thread communication & shared resources

I have employed a variety of techniques to enable each of the required threads to communicate and share resources. All data from the non-GUI threads, is relayed to the GUI thread using 'Platform.runLater()'. The updateResultsTable() and updateProgressBar() from the GUI thread is called from the FilesComparer class, which run on a threads executed from the thread pool created from the CompareResultsPool class.

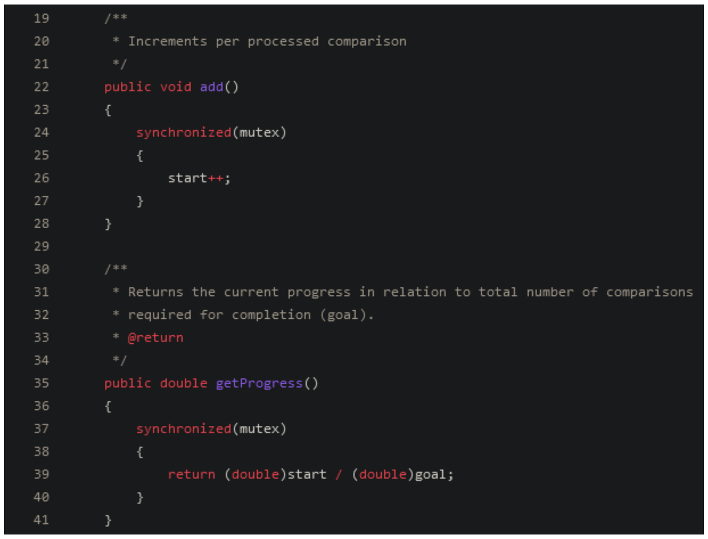
Also, as previously mentioned, I have used a Future object to retrieve the List<String> of files from the end of thread process for the AccessDirectory class.

I have also utilised two separate BlockingQueues to allow the different threads to share common required resources. One BlockingQueue (filesQueue) is used by the AddFiles class, as created by the CollectionPool class thread pool, to store the files to be compared (see Figure 2: Line 41). This queue is then used by the FilesComparer class, as created by the CompareResultsPool class thread pool, to extract the data for comparison. The second BlockingQueue (resultsQueue) is also used by the FilesComparer class, but instead for purpose of storing the results of the file comparisons. This queue is then subsequently used by the ResultsOutput class (see Figure 3: Line 62, 63), from yet another thread, to extract the data for outputting to the CSV file.

Dealing with race conditions & deadlocks

BlockingQueues were used to prevent the threads from interfering with one another and corrupting data. This provided a way for several threads to add and take data from the queues. Essentially, any threads that attempted to add comparison results data to the resultsQueue (ArrayBlockingQueue<>(100)) whilst it was at full capacity of 100, would be required to wait, until another part of the application withdrew data from the queue first. This worked vice-versa for when the queue was empty during a data withdrawal action too.

The ProgressTracker class was used to track the number of comparisons being completed in real-time in relation to the total number of comparisons to be done. This required several different threads calling it's add() method, to notify when a comparison has been done, and its getProgress() method, to check when the total comparisons have been completed. This data was in turn used to update the GUI's progress bar. Each of these threads were able to safely interact with this single object through the use of a mutex Object, which locked and unlocked actions within each method.



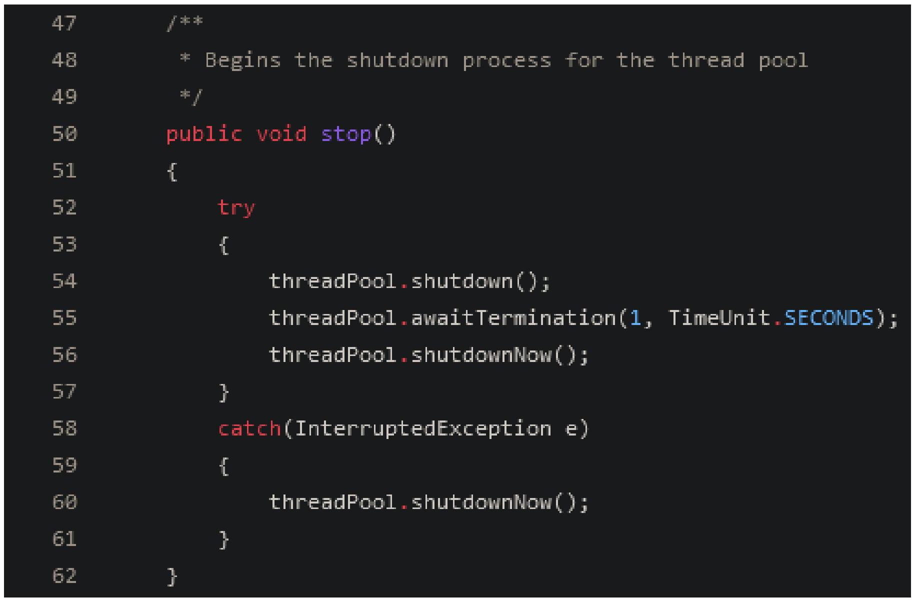
```
19      /**
20       * Increments per processed comparison
21       */
22      public void add()
23      {
24          synchronized(mutex)
25          {
26              start++;
27          }
28      }
29
30      /**
31       * Returns the current progress in relation to total number of comparisons
32       * required for completion (goal).
33       * @return
34       */
35      public double getProgress()
36      {
37          synchronized(mutex)
38          {
39              return (double)start / (double)goal;
40          }
41      }
```

Figure 4: Example of mutex Object

Ending threads

The first single thread pool is shutdown immediately after the Future object receives it's promised List<String> (see Figure 1: Line 258). This is to immediately free up resources once this task is completed.

The rest of the threads used are created from the two other thread pool previously mentioned. Each respective class, that created the thread pools, also have methods to systematically shutdown the thread pools. Please see Figure 5, for an example of how these two larger thread pools are shut down. This stop() method for each thread pool is called whenever the 'Stop' button from the GUI is pressed, or when the 'Compare…' button is pressed after the first time.



```
/**
 * Begins the shutdown process for the thread pool
 */
public void stop()
{
    try
    {
        threadPool.shutdown();
        threadPool.awaitTermination(1, TimeUnit.SECONDS);
        threadPool.shutdownNow();
    }
    catch(InterruptedException e)
    {
        threadPool.shutdownNow();
    }
}
```

Figure 5: Thread pool shutdown procedure

Scalability Issues

Non-functional requirements & large input data problems

With the increase of input data comes the requirement for greater processing power to handle such data. An application such as this, is currently bound by the processing speed of the computer from which it is executed.

Based on this, a general non-functional requirement, would be that the application should be able to handle large amounts of data without hindering the Operation System's performance by draining too many resources.

We should also take into consideration the potential event for failure whilst performing the comparison and outputting to file. For example, if the application was nearing the end of a comparison of 100,000,000 files, only to have the application/operating system crash or a power failure, leading to the complete corruption of the output file would be devastating. Rendering the time taken to process all that data wasted. With this in mind, we should carefully consider the recoverability and maintainability of the applications output data.

Potential ways to address these problems

If this application was to ever be designed to accommodate extremely large data sets, I would be inclined to shift from using a Personal Computer to run the application, to a dedicated computer comprised of high-end hardware to manage increased load.

This would allow all available processing resources to be optimized for the prime use of the application as it executes comparisons of files on a mass scale. Background applications and any other concurrently running applications should be kept to a minimum so as to not draw processing resources away from the File Comparison application.

Whilst running the application on a dedicated computer would address it’s processing and performance requirements, it still would not improve its recoverability or maintainability as described in the previous section. I would seek to re-engineer the application to output to remote data centers designed to store and recover large amounts of data. I would also consider outputting results in batches, rather than a single extremely large file. This would alleviate the disastrous outcome of the application crashing and corrupting the output file at 99% completion. For example: If the output file was generated in batches of 25% of the total required comparisons, then only the last 25% of data results would be lost in the event of catastrophic failure at 99%. Preserving the first three batches of data totaling 75% of the overall requested data. In terms of millions of files, I believe this to be a great benefit and better than losing 100% of the data.