Karim Rhoualem
Student #26603157

Due: Sunday, October 4, 2020
Section: FK-X

Programming Assignment #1 – Lab Report

The program is comprised of three classes. An analysis is done of each class, and as progression is made, the correlation between classes is further explained. The main goal of the assignment is to conduct a multi-threaded search through an array of input values to find defective lightbulbs. That is, the lightbulbs with value 0 are deemed to be defective and are inserted into a defective lightbulb array, and the ones with value 1 are deemed to be functional and ignored. A recursive method called FindDefective() is used. Each call to this method splits the array into two sub-arrays and creates a new thread for each half to further recurse through the sub-arrays. This recursion is repeated until all the defective lightbulbs are found. When the multi-threaded recursive search terminates, the program outputs the total number of defective bulbs at their respective original indices in the array to the console.

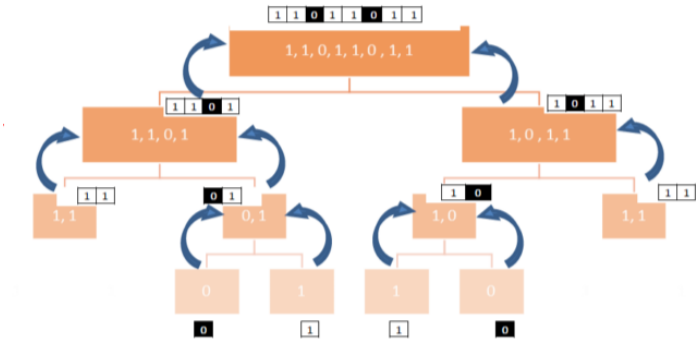
First and foremost, this program is a command line program which receives only one argument – the input text (.txt) file that is saved in a local directory. The commands used to compile and run the code are *javac Program.cs* and *java Program <srcfile>*, respectively.

The main method and entry point of the program is through Program.java – the first class. The main method receives the command line argument that was specified and does a series of checks prior to beginning the multi-threaded search. The argument is first checked to determine whether the command line argument is missing; if it is provided, the program continues, otherwise it will end. The next step is to make a call to the GetBulbArray() method from the FileReader class – the second class. This method creates the appropriate file reading objects and loops through the provided input file. The first iteration through the file saves the first value as the size of the array that is to be created. Each subsequent iteration stores the bulb values into the newly created array. A variety of methods and exception handling code is used to ensure that no erroneous information is provided in the input .txt file. Finally, the GetBulbArray() method returns the integer array of bulb values.

Next, back in Program.java, a DefectiveThread object is created. DefectiveThread extends the Thread class, and as such, the first thread is created to begin the search for the defective bulbs. The .start() method is called on the object, which in turn makes a call to the override .run() method that is specified in DefectiveThread.java – the third and final class. Additionally, the main thread makes a call to the .join() method on the DefectiveThread object, which will wait until the sub-threads return before continuing in the main thread. Keep in mind that this is a recursive program. As such, every time that .start() is called, .run() is subsequently called, which ultimately calls the FindDefective() method. The FindDefective() method simply takes an array of integers as a parameter. The method uses a typical approach for a recursive-style program. The method has a base case that checks if the array is of size 1 to break out from the recursive calls. If this is not the case, the recursion continues to break the supplied array into two equal halves. Each sub-array is then recursively used to continue the search for the defective bulbs. With each new recursive call, a new thread is created by creating a new object of the DefectiveThread class and calling its .start() method (which in turn calls .run() and then .FindDefective(), etc.). After the .start() method is called, a .join() method follows to ensure that the program pauses correctly and doesn't bubble back up to the main program prematurely.

To ensure that there are no race conditions present within this multi-threaded model, synchronization is introduced by via static objects. The main points of concern for this multi-threaded program lie in the incrementation of the defective lightbulb count and the incrementation of the thread count. Given that the operations of the various threads are unpredictable, it is important to synchronize the variables so that the correct values are being incremented. In other words, it is important to ensure that none of the other threads are manipulating these static variables while a thread is in the process of manipulating the given static variable. This is done by creating object locks on the variables and passing those objects to the *synchronized* statement. This way, whenever the static variables are being operated on, the threads synchronize amongst each other so that nobody else has access to the variables while that code block is running. Additionally, synchronization is introduced when a defective bulb is found and must be added to the ArrayList of defective bulbs. A *synchronized* statement is used with a lock object to ensure that there are no two defective bulbs being added to the list at the same time. Although the risk is smaller for this operation than the incrementation of the static variables, there is still the possibility that the links between the items in the list be overwritten if two threads are trying to add a defective bulb to the list at the same time.

The above process continues until each sub-array either no longer consists of any defective bulbs, or if the sub-array is reduced to an array of size 1. This process traces out a tree, which is commonly referred to as a binary tree traversal. An example of the process can be seen below:



When each thread completes as per the above specifications, the threads bubble back up to the main calling thread in Program.java and continue their remaining operations. At this point, the total number of threads that were synchronized between the created threads are output to the console. Additionally, the indices of each defective bulb from the defective array is displayed.

The process is now summarized in a concluding statement. Program.java checks the command line to ensure that the input file has been provided as an argument. Then, the FileReader.java class calls its static GetBulbArray() method to fetch the inputs of the bulb array and store them into a local array. Finally, the first thread is created by instantiating a DefectiveThread object. This thread is started and recursively sub-divides the array into equal halves by creating new threads to search for the defective bulbs. Along the way, a series of .join() methods are called on the new threads in order to allow for correct program flow. To ensure correct functionality of the program, several defective bulb counting variables are synchronized in order to bypass the race condition. When all the defective bulbs are found, functionality returns to Program.java, where the total number of defective bulbs and their respective indices are output to the console.