CS3250 ASSIGNMENT 3 – PARALLELIZED WORD COUNT

In this assignment, you will implement a word counting program that computes word counts in parallel, merging the results in the end. Your program will break down its tasks into smaller pieces by working on different parts of a text file or text file(s), computing word counts for the smaller pieces and then merging the count of words at the end.

EXAMPLE COMMAND LINE INPUT

Example 1

Assuming in the current directory you have 3 files, file1, file2, and file3. You want to count words for files in the current directory:

Input: java WordCount . 5 2

- **First argument =** current directory
- **Second argument =** chunk size, meaning the size of each task defined by the number of lines. In this case, 5 lines per task. Each thread can handle a maximum of 5 lines at a time.
- **Third argument =** number of threads to use; in this case 2

Output: would contain a list of chunk files in the **output/** folder along with 1 **results.txt** file in the same output folder. Chunk files would contain word counts for the smaller pieces and results.txt would contain overall word counts based on all the chunks.

Example 2

Let's say you want to count all words within files in C:\Test\

Input: java WordCount C:\Test\ 100 10

- **First argument =** Directory
- **Second argument =** Chunk size. In this case, 100 lines per task. Each thread can handle a maximum of 100 lines at a time.
- **Third argument =** number of threads to use; in this case 10

Output: would contain a list of chunk files in the **output/** folder along with 1 **results.txt** file in the same output folder. Chunk files would contain word counts for the smaller pieces and results.txt would contain overall word counts based on all the chunks.

Example 3

Let's say you want to count all words within one file **C:\MyTestFile.txt**

Input: java WordCount C:\Test\MyTestFile.txt 10 10

- **First argument =** A specific file
- **Second argument =** Chunk size. In this case, 10 lines per task. Each thread can handle a maximum of 10 lines at a time.
- **Third argument =** number of threads to use; in this case 10

Output: would contain a list of chunk files in the **output** folder along with 1 **results.txt** file in the same output folder. Chunk files would contain word counts for the smaller pieces and results.txt would contain overall word counts based on all the chunks.

OUTPUT FOLDER

- Output folder should be named **output/**
- Output folder should be wiped out on every run (deleted)
- Output folder should be generated in the current directory only
- All your ***.chunk** files should reside in the output folder
- Your **results.txt** file should also reside in the output folder

CHUNK FILES AND RESULTS FILE

CHUNK FILES

- Chunk files should be named **originalfilename_chunkNum. chunk** (e.g. readme.txt_0.chunk, readme.txt_1.chunk, readme2.txt_0.chunk, readMe2.txt_1.chunk)
- No need for path information, just file name (LOWERCASED)
- Each chunk file should contain count of words for a specific chunk in descending order and lowercased
- Counts of words have to be unique. You cannot have the same word appear more than once in a chunk file or a results file.
- Number of chunk files depends on the chunk size specified as the input. **For example:** If you had 1 file with 100 lines, and your chunk size was 50. You would then have 2 chunks for the given file. Each chunk is to be processed in parallel. If you had 10 files, each with 100 lines and your chunk size was 20, you would have 50 chunks total and thus 50 chunk files.

- **Format of chunk files (tab separated not space):**
  word1 200
  word2 100
  word3 50

FINAL RESULTS FILE

- Final results file should be named **results.txt** and should be generated in the output folder
- Final results file should contain merged counts of all words across all chunks. For example, if "cat" had a count of 5 in chunk1 and 10 in chunk2, then total number of counts would be 15 for "cat". Counts should be in descending order and lowercased. No repeated words in this file – all words must be unique.
- **Format of result file (tab separated not space):**
  cat 2000
  mouse 1000
  food 100

INPUT ARGUMENT

Example Input: java WordCount C:\Test\ 10 10

- All input would be MANDATORY for this assignment. If wrong number of input arguments specified or invalid argument types, then print the following:
  - **Usage:** java WordCount <file|directory> <chunk size> <num of threads>
- **First argument =** A file or directory. You should be able to accept both (**hint:** File.isDirectory()). You only need to use the top level contents of a directory.
- **Second argument =** Chunk size. This means how many lines of text one thread should process. If value=10 and you use two threads, then first thread will process the first 10 lines second thread the next 10 lines. When first thread is done, it will move on to the next 10 lines. This can be any value from 10 to 5000. Anything else should be incorrect. See error messages below.
- **Third argument =** Number of threads to use; Values should be between 1- 100; Anything else should be incorrect. See error messages below.

ERROR MESSAGES

- If wrong number of input arguments specified or invalid arguments (wrong number of threads, wrong chunk size etc), then print the following:

- **Usage:** java WordCount <file|directory> <chunk size 10-5000> <num of threads 1-100>

- If file/directory not found or does not exist:
  - No such file/directory: <the directory that the user had specified>
- On Deadlock:
  - Redesign your program (no error messages)

TEST FILES

- For development you can test on some or all files within this zip file.

CHUNKING

- Chunking in this assignment refers to breaking your tasks down to number of lines a thread should work on at a given time. For example, thread 1 could work on lines 1-100 and thread 2 could work on line 101-200 in parallel.

IMPLEMENTATION DETAILS

- You should have one main class **WordCount.java** with NO PACKAGE.
- All arguments are to be read from the **String args[]** of your main method.
- You will have 3 arguments in total.
- Use thread pools to manage your threads.
- In addition to **WordCount.java** you can define other helper classes without any package definition. **Do not use external libraries or create a package as it will affect grading.**
- You can compare your output with peers to make sure your program is doing the right thing.
- Make sure you can read an input file from any location saved locally
- All text from the text file should be lowercased once its read in by your program

LOOPING

- There is no need for looping in this assignment
- We just need to re-run the program for every word count task.

WHAT TO SUBMIT?

- Put your source and class files into a **FOLDER** named **assignment3**.
- Zip assignment3 folder as <assignment3>_<firstandlastname>.zip; If your name is “Bruce Lee” your zip should be “assignment3_brucelee.zip”
- You will have one folder in the zip file named assignment3
- Your main program should be: WordCount.java as specified above

- Turn your zip file in on Canvas.

GRADING

10 points	Correctly submitted: zip file, WordCount class, WordCount.java
20 points	Correctness of chunk files produced in output/ folder including formatting, lowercasing and order
20 points	Correct error reporting on incorrect input (print messages as specified above)
30 points	Correct final results (results.txt) including format on various test cases
20 points	Correct operation on file or directory in the input argument
5 extra points	Submit 2 days before due date and program works seamlessly
105 points	

HINTS

- Implementing word count
  - See **hashmap** usage in the collections slides.
- Using threadpools
  - **See slides on threads**
- Splitting a line of text into token of words.
  - **String myStrArray[]=line**.split("\\s+");
- Removing non word characters around words for cleaning up words
  - **word**.replaceAll("\\W+");
- Remove white space around word
  - **word.trim()**;
- Check if a file exists:

```
File f = new File(filePathString);
if(f.exists() && !f.isDirectory()) { /* do something */ }
```