Project 2 - File System and I/O Operations

CSci 4061 : Introduction to Operating Systems
Due Date: Friday, 25th October, 2019

1 Instructions

- You may choose to complete this project in a group of up to two students.
- Each group should turn in one copy with the names of all group members on it.
- The code must be originally written by your group. No code from outside the course texts and slides may be used—your code cannot be copied or derived from the Web, from past offerings, other students, programmer friends, etc.
- All submissions must compile and run on any CSE Labs machine located in KH 4-250.
- A zip file should be submitted through Canvas by 11:59pm on Friday, Oct 25th.
- Note: Do not publicize this assignment or your answer to the Internet, e.g., public GitHub repo. To share code between team members, create a private repo in github.umn.edu.

2 Introduction

Jeffrey Dean and Sanjay Ghemawat from Google published a research paper [1] in 2004, on a new programming model that can process very large datasets in an efficient, distributed manner. This programming model is called the "MapReduce".
The MapReduce consists of two main phases, Map and Reduce. In the 'Map' phase, a user written map function is used to process input <key, value> pairs to intermediate <key, value> pairs. Then in the 'Reduce' phase, a reduce function combines the intermediate pairs based on the keys to give the final output. Since the dataset input can be very large, there will be multiple map and reduce jobs and it is essential to maintain a synchronized system.
Let us consider an example of counting the number of occurences of each word in a corpus of documents. The map function emits each word (key) it sees with a count '1' (value). The reduce function sums together all counts of emitted word by a particular word(key). The pseudocode for the same looks as below:

```
map(String key,String value):
        // key: document name
        // value:document contents
        for each word w in value:
                EmitIntermediate(w,"1");

reduce(String key,Iterator values):
        // key: a word
        // values: a list of counts
        int result= 0;
        for each v in values:
                result += ParseInt(v);
        Emit(AsString(result));
```

The above example is taken from [1].
In this project, we will mimic certain core functionalities of the MapReduce programming model using OS system calls.

2.1 Objectives

Concepts covered in this project.

- Process spawning
- Directory traversal
- File I/O operations
- Pipes
- Redirection

3 Project Overview

Given a folder with multi-level hierarchy, i.e., folders with multiple level of folders and text files. Each text file has a word per line. Your task is to count the number of words, per letter of the alphabet, i.e., compare the first letter of each word to increment the count corresponding to a letter, in the entire folder hierarchy. The result should be reported in the alphabetical order.
Key parties involved in the project:

- Master process - Parent process to all the spawned processes
- Mapper processes - Executes the map function on the partitioned data. The number of mapper processes should be specified at the execution time as input argument
- Reducer process - Executes the reduce function over the results from all the mapper process. For easiness, we have only a single reducer process for this project.

There are four phases in this project. First is the data partitioning phase where the 'master' will traverse through the folder hierarchy, identify all the files and split them equally among the mapper processes. During the second phase, the 'mappers' will process the files alloted to them by the 'master' and each of them will come up with a list of per letter word count. For the third phase, each 'mapper' will send their list to the 'reducer' process, who will combine them all to give a single list. In the last phase, the final list is then taken by the 'master' and reported.

4 Implementation details

Now let us have a look at the project implementation details. An example is provided for your understaning after the explanation of each phase.

Notice: You may use any algorithm that will help you to reach the final goals. For each phase there is an expected output, unless otherwise specified. They should be in the format specified, i.e., if the results are to be stored as a text file with a specific name format in a folder with a specific name, you should follow it.

4.1 Phase 1 - Data Partitioning Phase

This phase aims at traversing the folder hierarchy and partitioning the files equally among the 'mapper' processes.
The 'master' process creates a folder "MapperInput" in the current directory. It will then recursively traverses through the 'Sample' folder hierarchy (assuming 'Sample' is the relative or absolute, path of the folder that you passed to the executable) and divide the list of filepaths of text files with words, among 'm' files of name format 'Mapper_i.txt', where 'm' is the number of 'mappers' and i belongs to [0, 1, 2, . . . , m]. The 'Mapper_i.txt' should be created inside 'MapperInput'. You may use any partitoning logic that allows you to have almost same number of filepaths in each text file. For example, let there be 6 text files in 'Sample' hierarchy and 3 mapper processes. Then each of the three 'Mapper_i.txt' files created, will have

two file paths. In case the count of files is not a multiple of number of mappers, then add the extra files to 'Mapper_i.txt' in a round robin fashion.

Notice: Assume that the number of files is always greater than or equal to the number of mappers except for the case of empty folder.

The expected outputs from this Phase are the "Mapper_i.txt" files inside "MapperInput" folder

In Figure 1, 'Sample' is the folder passed as input to your executable. F1, F2, ...are the folders and Tfile*.txt are the text files with words.

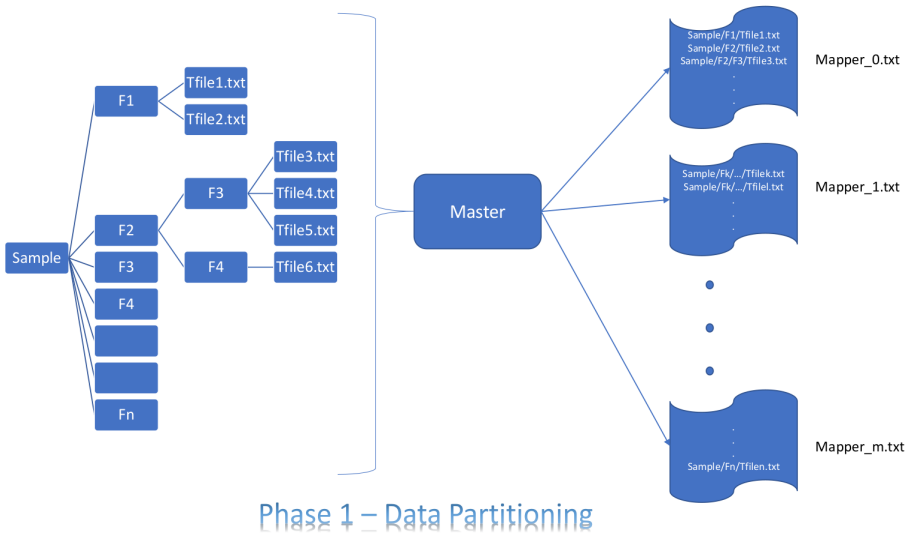


Figure 1: Data Partitioning

4.2 Phase 2 - Map Function

The master process creates 'm' pipes, one per mapper to communicate with the reducer process. The master then spawns the mapper processes. Each mapper process will pick one file from the "MapperInput", open each filepaths in the file and find the number of words corresponding to each letter of the alphabet. This information is then send to the reducer process via pipe by each mapper. Note to process the words as case-insensitive, i.e., take 'a' and 'A' as 'A'.

No output is expected from this phase. The grading will be carried out based on the per letter word count algorithm, pipe setup and usage.

In Figure 2, assume there are two file paths per 'Mapper_i.txt'. The 'Master' process forks the 'mappers'. Mapper1 processes 'Mapper_0.txt' and Mapper2 processes 'Mapper_1.txt'. Each mapper has a list with that keeps track of the per letter word count.

Notice: Please do not assume that the process ids of mappers are [0, 1, 2, ...]. It is upto to the OS to decide the process id. So there won't be a one-to-one mapping between the names of text files and mapper process ids.

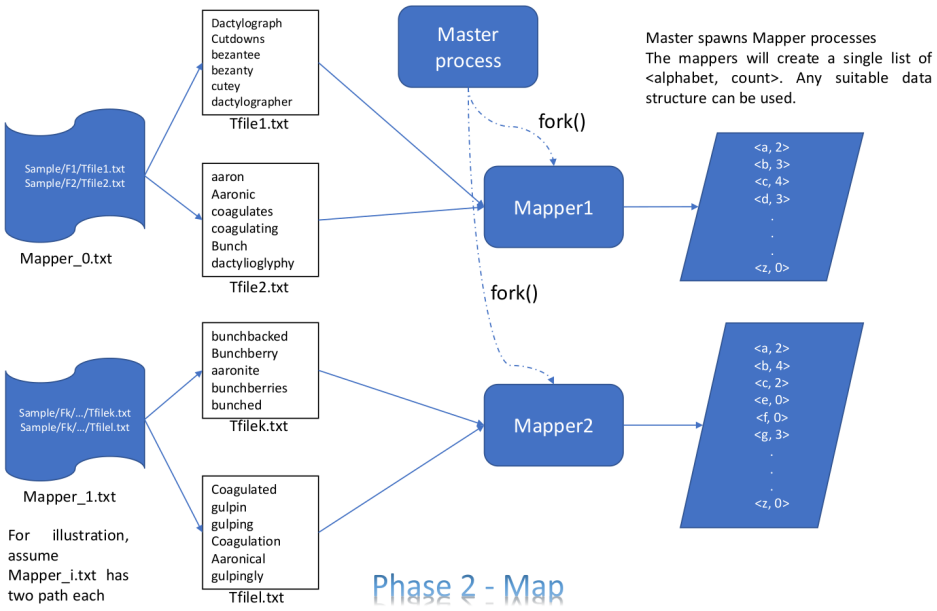


Figure 2: Map

4.3 Phase 3 - Reduce Function

The reducer process will receive the lists from each of the mapper processes via pipes and combine them to create a single list. The list is then written into a text file "ReducerResult.txt" in the current folder. Each line in the "ReducerResult.txt" will have the format as given below. There is only one space between 'letter_of_alphabet' and 'wordcount'.

```
letter_of_alphabet wordcount
```

The expected output from this phase is the "ReducerResult.txt"

In Figure 3, the 'reducer' receives lists from Mapper1 and Mapper2 via two pipes. This list is then combined and written to ReducerResult.txt in the current folder.

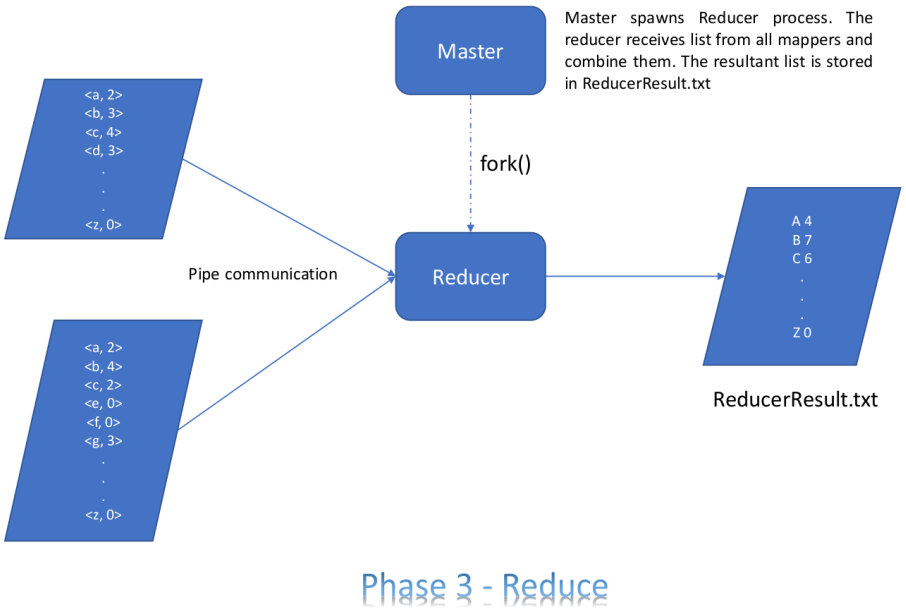


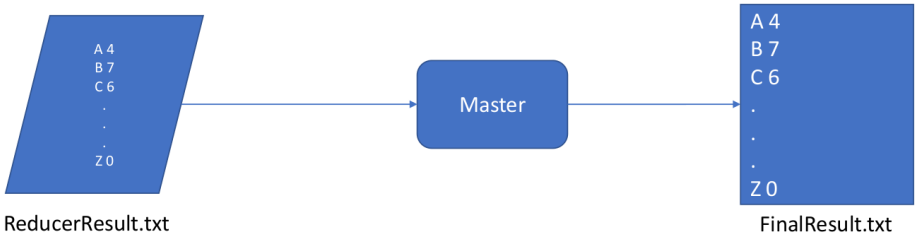
Figure 3: Reduce

4.4 Phase 4 -Final Result

The mapper processes will have exited by now. The 'master' process will wait for the reducer to finish. It will then read the results from 'ReducerResult.txt and report it to standard output. But the catch here is that, the standard output is redirected to a file "FinalResult.txt" in the current folder. In MapReduce the 'master' process exits towards the end after all the processes have completed. We are emulating that in this phase.

The expected output from this phase is the FinalResult.txt which is having the same format as ReducerResult.txt

In Figure 4, the 'master' will redirect the results from standard output to 'FinalResult.txt'.



Phase 4 – Final Result

Figure 4: Final Result

5 Execution

The executable 'mapreduce', for the project will accept 2 parameters

Command Line

```
$ ./mapreduce folderName #mappers
```

folderName is the name of the root folder to be traversed
#mappers is the number of mapper processes

6 Expected output

- The Sample folder is empty or there are no files in the Sample hierarchy

  Command Line

  ```
  $ ./mapreduce Sample 4
  The Sample folder is empty
  ```

- The Sample folder has files in its hierarchy

  Command Line

  ```
  $ ./mapreduce Sample 4
  ```

  The result is present in FinalResult.txt (below format) in the current folder

  ```
  A 5
  B 10
  C 3
  .
  .
  .
  Z 12
  ```

7 Testing

The results from the executable, 'mapreduce', generated out of your program will be tested on folders of multiple levels. The number of files that will be present in the folders will range from 'm' to 500, where 'm' is the number of mappers. Note in case of empty folder, the file count will be zero.

Notice: There will be an exact pattern matching algorithm used for grading the results obtained. So please be sure to adhere to the format.

To test your code with the sample test folders, run

Command Line

```
$ make test-mapreduce
```

Note that if you want to run explicitly on a test case, you will have to do it without the test-mapreduce target. Also, there is no auto-checking enabled for the correctness of result. The Expected_Output folder in Testcases have the expected output for each of the given test folders.

8 Assumptions

- Number of masters $= 1$, $0 <$ Number of mappers $<= 32$ and Number of reducers $= 1$.
- File count $>=$ number of mappers except for the case of empty folder.
- There will not be any space in the folder or file names.
- Only consider the file count for splitting data equally among mapper processes. Do not look at the size of the file.
- Use C library file operations to read and write (FILE *, fgets, fputs, fscanf, fprintf and so on) to and from a file in Phase 1, Phase 2 and Phase 3.
- In phase 4, use OS system calls to do redirection from standard output to file.
- Use pipes to communicate between Phase 2 and Phase 3.
- The executable name should be 'mapreduce', as specified in the makefile.
- The template code consists of 4 phase*.c files. Add code of each phase as functions to corresponding files and call them from main.c.
- Ensure to add the function prototypes to corresponding phase*.h files in 'include' folder.
- You are expected to provide proper guard mechanisms for header files.
- Ensure proper error handling mechanisms for file operations and pipe handling.
- Expect empty and multilevel folder hierarchies.

9 Extra credit

Symbolic link or soft link is a kind of file that points to another file name. This can be extended to directories as well. Refer to [4] for more details.

Consider the following example

```
Folder1/file1.txt ------> Folder5/file6.txt ====> physical data
```

Here file1.txt in Folder1 is a symbolic link for file6.txt in Folder5. One can interchangeably use file1.txt and file6.txt to read and write from the same physical location in the memory. The same can apply to directories.
In Phase 1, there can be folders or files that are symbolic links. Your task is to identify such folders/files and avoid reprocessing them. You may assume that all the symbolic links are valid.

To test this scenario with your code run

Command Line

```
$ make test-extracredits
```

If you are attempting extra credits, mention it in the README file

10 Deliverables

One student from each group should upload to Canvas, a zip file containing their C source codefiles, a makefile, and a README that includes the following details:

- The purpose of your program
- How to compile the program
- What exactly your program does
- Any assumptions outside this document
- Team names and x500
- Your and your partners individual contributions
- If you have attempted extra credit

The README file does not have to be long, but must properly describe the above points. Proper in this case refers to – first-time user can answer the above questions without any confusion. Within your code you should use one or two comments to describe each function that you write. You do not need to comment every line of your code. However, you might want to comment portions of your code to answer 'why', rather than 'how' you implement the said code. At the top of your README file and main C source file please include the following comment:

```
/*test machine: CSELAB_machine_name
* date: mm/dd/yy
* name: full_name1, [full_name2]
* x500: id_for_first_name, [id_for_second_name]
*/
```

11 Grading Rubric

1. 10% Correct README contents
2. 10% Code quality such as using descriptive variable names, modularity, comments, indentation. You must stick to one style throughout the project. If you do not already have a style for C programming, we suggest adopting K&R style [5].
3. 15% Proper error handling and data structure usage
4. 15% Data partitioning and file operations
5. 20% Process handling
6. 30% Pipe communication and redirection
7. 10% Extra credit

If your code passes the test cases, you will get 75% of the credit. The remaining 25% will be 1, 2 and proper data sructure usage.

References

[1] Dean, J. and Ghemawat, S., 2008. *MapReduce: simplified data processing on large clusters.* Communications of the ACM, 51(1), pp.107-113.

[2] Kay, A. R.,Robbins, S., 2004. *Chapter 6 - Unix Systems Programming: Communication, Concurrency And Threads, 2/E*. Pearson Education India.

[3] Kay, A. R.,Robbins, S., 2004. *Chapter 4 - Unix Systems Programming: Communication, Concurrency And Threads, 2/E*. Pearson Education India.

[4] Symbolic links
`https://www.gnu.org/software/libc/manual/html_node/Symbolic-Links.html`

[5] K&R Style
`https://en.wikipedia.org/wiki/Indentation_style#K&R_style`