CSci 4061: Introduction to Operating Systems
Spring 2021
Project Assignment #2: IPC based Map Reduce

Instructor: Abhishek Chandra

Due: 11:59 pm, March 17, 2021

1 Purpose

In project 1, we built a simple version of mapreduce using operating system primitives such as fork, exec and wait. While doing so, several utility functions were provided which helped you implement the map and reduce tasks. In this project, you will be required to implement these utility functions to perform File IO and perform inter process communiation (IPC) using pipes to send the data to mappers. You should work in groups as in Project 1. Please adhere to the output formats provided in each section.

2 Problem Statement

In this project, we will revisit the single machine map-reduce designed for the word length count application as in Project 1. You can use the code from Project 1. We have added another phase called stream phase. So, there are four phases: **Master, Stream, Map and Reduce**.
Given an input file directory with multi level hierarchy, i.e., folders with multiple level of folders and text files. Each text file will contain words as in Project 1 and you will have to count the number of words of different lengths. The final output will be same as of Project 1.

- In `Master` phase, the input file directory is taken as input from the command line. The master will traverse the input file directory and search for the text files and split the files equally among the mapper process.
- In `Stream` Phase, the stream processes will read the text files from the file directories and send the data to mappers process. There will 1:1 relationship between stream and mapper process. So number of stream processes will be equal to the mapper processes.
- In `Map` phase, each mapper will read the text received from the stream process and emit the count of word length into an intermediate data structure. Once the Map phase is complete, the contents of the `intermediate data structure` is written to `m_mapperID.txt` files (Same as Project 1). This file be created in folders for all the different word lengths.
- In `Reduce` phase (Refer section 3.4), the generated `m_mapperID.txt` files are accessed per folder across different reducers. All the files belonging to a particular folder will be accessed by the same reducer. One reducer can access more than one folders. **In Project 1, you were given the getReducerTasks() utility, in Project2 you will have to implement getReducerTasks()** and rest of the code of reduce will be same as Project 1. The reducers will read the `m_mapperID.txt` files and compute the total count corresponding to the word length.(Refer section 3.4)

Summary: You will have to implement the File IO in `Master` phase, File IO in `Stream` phase, IPC between Stream processes and Mapper processes and getReducersTasks() in `Reduce` phase. Rest should be same as Project 1.

3 Phase Description

In this section, we will see the brief design details of different phases that will help you get started.

3.1 Phase 1: Master phase

The master process drives all the other phases in the project. It takes three inputs from the user: number of mappers, number of reducers and the path of the input file directory relative to the provided Makefile location. The algorithm 1 provides a brief overview of the master process. This is your main control program. The code assumes the mapper and reducer executable are named `mapper` and `reducer` and stream executable is named as `stream`.
File: src/mapreduce.c

Algorithm 1: `master:mapreduce`

Input: $(Integer\ \ nMappers, Integer\ \ nReducers, String\ \ inputFileDir)$

// output directory creation and removal
$bookeepingCode()*$;

// directory traversal
$traverseInputFileDirectory()$;

// open pipes
$openpipes()$;

// spawn stream processes with each calling exec on "stream" executable
$spawnStream(nMappers)$;

// spawn mapper processes with each calling exec on "mapper" executable
$spawnMapper(nMappers)$;
// wait for all child processes to terminate
$waitForAll()$;

// spawn nReducer processes with each calling exec on "reducer" executable
$spawnReducers(nReducers)$;
// wait for all child processes to terminate
$waitForAll()$;

Notice: `*bookeepingCode()` is defined in the provided utils.c file. Please do not remove the function call. Make sure you understand the difference with mapreduce.c in Project 1.

First, the master calls a `bookeepingCode()`, which takes care of the creation of `output, output/IntermediateData, output/FinalData` . The master will traverse the input file directory and identify all the text files and create the folder `MapperInput`, inside the folder will be text files containing filepaths that stream process will read. See the illustration for better understanding. So, master will create nMapper files inside `MapperInput` folder. For example, if folder is Sample as shown in 3.2 and there are 3 mappers, then `MapperInput` will contain `Mapper1.txt, Mapper2.txt, Mapper3.txt.Mapper1.txt` will contain file paths (Sample/F1/Tfile1.txt) in each line as shown in 3.2.

The master will then open nMappers pipes for Inter process communication between stream processes and map processes.
The stream processes are spawned using `fork` which in turn calls `exec` family functions for executing the `stream` executable. Then the mapper processes are spawned using `fork` which in turn calls `exec` family functions for executing the `mapper` executable. The master process will wait until all the mappers have completed their task. Following this, the master process will spawn the reducers which will call `exec` to execute the `reducer` executable. Again the master will wait for all the reducer processes to complete execution before exiting the code.

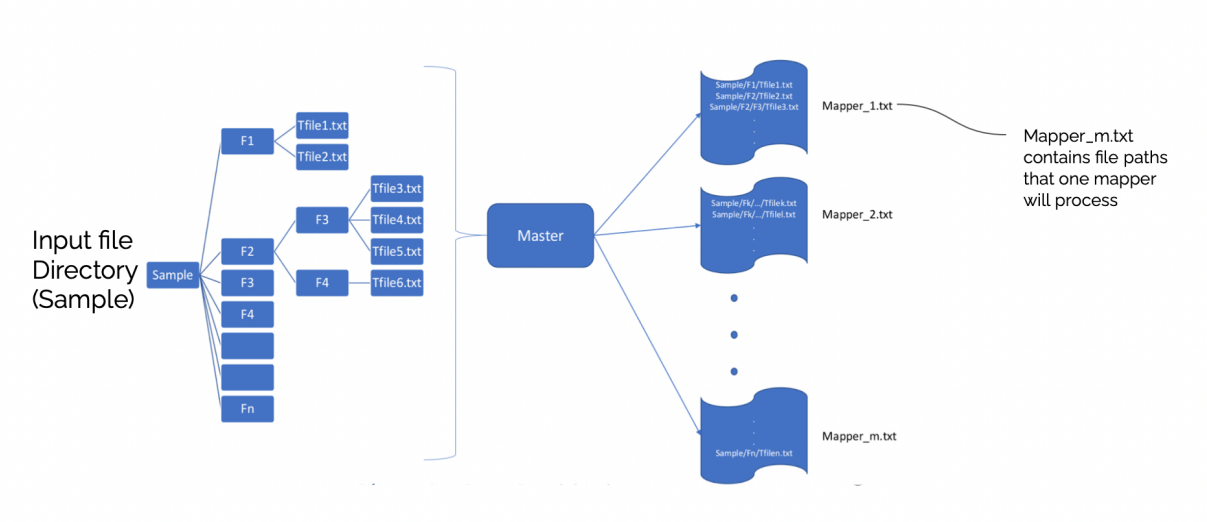


Figure 1: Traversal File Folder and creating MapperInput Folder

3.2 Phase 2: Stream Phase

The objective of the stream process is to read the words from text files and send the words to mappers via pipe. So it will read the MapperInput/MapperID.txt file that is associated with stream process (ID is taken as input from the master process) and start the reading the file and write the contents to the pipe. Use `STDOUT` redirection to write the data into pipes. In the illustration, `Stream1` process will read the `Mapper1.txt` and go into the file paths present in the `Mapper1.txt` so it will go to `Tfiles` and read the content and **send the words to the pipes**. Similary `Stream2` will do it for `Mapper2.txt` and so on.

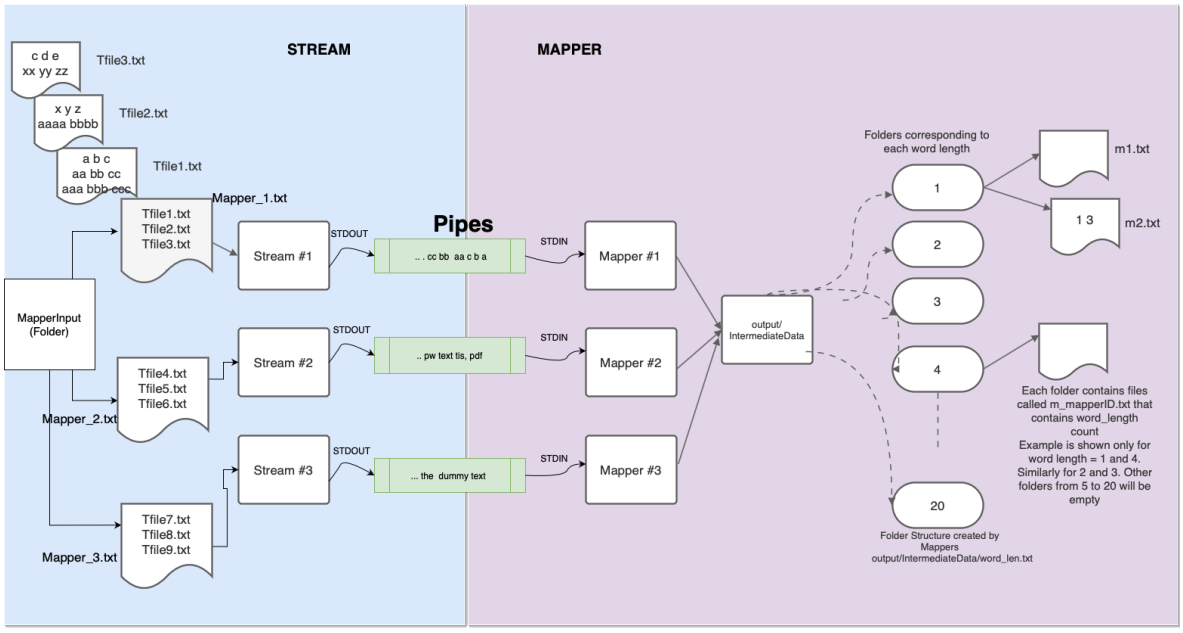


Figure 2: Stream Phase and Mapper Phase

3.3 Phase 3: Map phase

The mappers will read the data (words) from the pipe **and not from any file** to perform word length count. Mapper will create the Intermediate Data Structure and File folders as in Project 1. **Use STDIN redirection to read the data from pipes.** . Mapper processes and Stream processes will be in 1:1 relationship as shown in the illustration.

Notice: A word should be composed of consecutive characters "c", where "c" $\in$ {A...Z, a...z, 0...9}
Words are separated by whitespaces as delimeters.
Example: Thi's is. a text* 0h gr8
The words in this sentence are {Thi's, is., a, text*, 0h, gr8}
Words are case sensitive, which means "text" and "Text" are different.

3.4 Phase 4: Reduce phase

This phase is same as reduce phase in Project 1. It will read data from the files created by Mappers. The algorithm of the reducer can be referred from Project 1. The only change is you will have to implement the getReducerTasks() function. This function was given to you in Project 1.
You can implement getReducerTasks as per your liking. The idea is that reducer process will traverse into the intermediateDir folder and get the file names that it will process. Each reducer is responsible for some files in intermediateDir folder.
Final Output is exactly same as the output of Project 1. The final data structure stores the final count of each word length. The final data structure can be written to files using utility functions $writeLineToFile()$. Utility functions provided to you are $getFilePointer()$, $getLineFromFile()$, $writeLineToFile()$ that can be used to read file, read line and write line to a file respectively.

4 Extra Credit

One of the test cases (Input file directory) will contain Symbolic links and Hard links. You will have to implement handling of the links while traversing the input file directory. If you successfully pass the test case, we will provide Extra credit (10%).

5 Compile and Execute

Compile

The current structure of the `Template` folder should be maintained. If you want to add extra source(.c) files, add it to `src` folder and for headers user `include`. The current `Makefile` should be sufficient to execute the code, but if you are adding extra files, modify the `Makefile` accordingly. For compiling the code, the following steps should be taken:

Command Line

```
$ cd Template
$ make
```

The template code will not error out on compiling.

Execute

Once the $make$ is successful, run the mapreduce code with the required mapper count, reducer count and input file directory.

Command Line

```
$ ./mapreduce #mappers #reducers inputFileDirectory
```

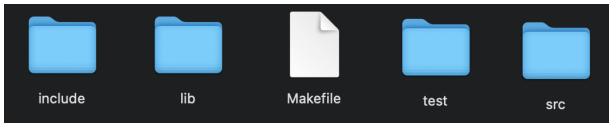
Notice: The final executable name should be `mapreduce`.

Note that number of mappers is greater than or equal to number of reducers. The inputFile path should be relative to the `Makefile` location.
On running the `mapreduce` executable without any modifications to template code will result in error.

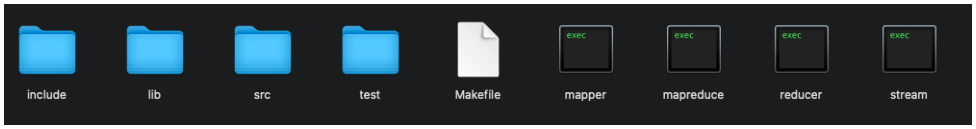
6 Expected Output

Please ensure to follow the guidelines listed below:

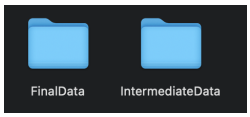
- Do not alter the folder structure. The structure should look as below before compiling via $make$:



- After compilation, the folder structure will look as below. The output folder is auto-created:

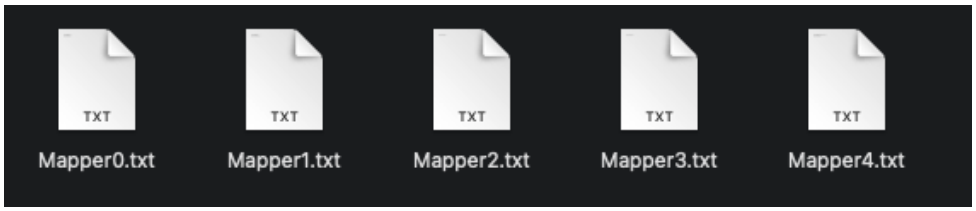


- The output folder content (auto-created) will be as follows:

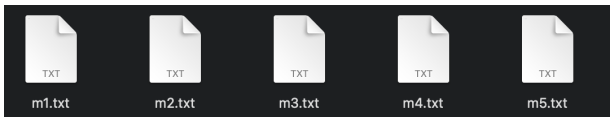


- Another important folder that will be checked is MapperInput, created by your code

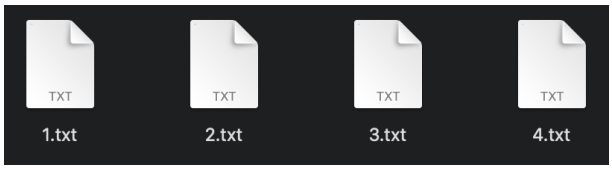
- The files in the MapperInput in the case of 5 Mappers
- The IntermediateData folder content (auto-created) will be as follows



- Word length folders will contain files as follows in case of 5 mappers :



- Example of the FinalData folder content will be as follows. The files should be created by your code:



7 Testing

A test folder is added to the template with one test case. You can run the testcase using the following command

```
Command Line
  $ make t1
```

The working solution for the code is provided to you in the `solutionexe` folder. You can run navigate to the folder and run the following command to see the expected output. During the execution if there are any issues, please let us know as soon as possible.

```
Command Line
  $ cd solutionexe
  $ make clean
  $ ./mapreduce #mappers #reducers test/T1/F1.txt
```

8 Assumptions / Points to Note

The following points should be kept in mind when you design and code:

- The input file sizes can vary, there is no limit.
- Number of mappers will be greater than or equal to number of reducers, other cases should error out. Maximum number of mappers or reducers will be limited to 20.
- The system calls that will be used for the project are fork, exec and wait.
- Add error handling checks for all the system calls you use.
- Do not use the system call "system" to execute any command line executables.
- You can assume the maximum size of a file path to be 200 bytes.
- You can assume the maximum length of a word to be 20
- You can assume the maximum number of input files to be 100
- You can assume the maximum number of intermediate files to be 400
- Follow the expected output information provided in the previous section.
- The chunk size to read lines will be atmost 1024 bytes
- If you are using dynamic memory allocation in your code, ensure to free the memory after usage.
- **The provided lib/utils.o file will not run on Mac machines. ssh into Linux machines for using the object file.**

9 Deliverables

One student from each group should upload 2 items to Canvas

1) An Interim Submission. (Due by 03/10/2021, 11:59 PM)
Interim Submission should just contain the src file stream.c that will nMappers pipes and redirect STDOUT to write into the pipes.

2) A zip file containing the source code, 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 member names and x500
- Contribution by each member of the team

The README file does not have to be long, but mus properly describe the above points. The code should be well commented, it doesn't mean each and every line. When a TA looks at your code he/she/they should be able to understand the jist. You might want to focus on the "why" part, rather than the "how", when you add comments. At the top of the README file, please include the following:

README.md

```
test machine: CSELAB_machine_name
date: mm/dd/yy
name: full_name_1, [full_name_2, ...]
x500: id_first_name, [id_second_name, ...]
```

10 Rubric: Subject to change

- 5% README
- 10% Interim Subimmsion
- 10% Documentation within code, coding, and style: indentations, readability of code, use of defined constants rather than numbers
- 75% Test cases: correctness, error handling, meeting the specifications
- Please make sure to pay attention to documentation and coding style. A perfectly working program will not receive full credit if it is undocumented and very difficult to read.
- Sample test cases are provide to you upfront. Please make sure that you read the specifications very carefully. If there is anything that is not clear to you, you should ask for a clarification. We also check for some other test cases not given to you. You can refer to the Section "Testing Strategy" for more information.
- We will use the GCC version installed on the CSELabs machines(i.e. 9.3.0) to compile your code. Make sure your code compiles and run on CSELabs.
- **Please make sure that your program works on the CSELabs machines** e.g., KH 4-250 (csel-kh4250-xx.cselabs.umn.edu). You will be graded on one of these machines.

11 Testing strategy

We will be comparing the results of your `output/FinalData/` with the one that we have generated. There are three sample test cases for your own testing. You can use `make t0`, `make t1`, `make t2` to run each test case separately, or use `make test` to run all test cases at one time.
We will also check if your `MapperInput` folder is built correctly after traversing the input file folder.

The TAs will be running more test cases for grading, and going through your code to see if system calls are used correctly. Error handling is a must for all the system calls. Ensure that the total number of processes created by the master is #mappers + #reducers. Proper creation of intermediate data structure along with freeing the memory after usage will be checked.