CSci 4061: Introduction to Operating Systems
Spring 2021
Project Assignment #1: Basic Map Reduce

Instructor: Abhishek Chandra

Due: 11:59 pm, Feb 24, 2021

1 Purpose

MapReduce [1, 2] is a programming model that allows processing on large datasets using two functions: `map` and `reduce`. It allows automatic parallelization of computation across multiple machines using multiple processes. Mapreduce model is widely used in industry for Big Data Analytics and is the de-facto standard for Big Data computing! In this project we will explore a simple verison of mapreduce using operating system primitives on a single machine which will use `fork`, `exec` and `wait` to run map and reduce functions. Utility functions that will help you with building the map and reduce tasks are provided with the project template. You will be given binaries for utilities that will run on the CSE-IT lab machines, we are **NOT** giving you source for the utilities since you may be asked to write them in future projects. You should work in your groups. **Please adhere to the output formats provided in each section.**

2 Problem Statement

The mapreduce programming model consist of two functions: map and reduce. The map function takes in <key, value> pairs, processes them and produces a set of intermediate <key, value> pairs. The key and value(s) are determined from input files. The intermediate pairs are then grouped based on the key. The reduce function will then reduce/merge the grouped intermediate values based on the key to produce the final result. In this assignment, you will use map and reduce *logic* paradigm for counting the number of **words of different lengths** in a large collection of documents.

You can refer to the Word Count Example [1, 2], where the objective is to calculate the frequency of each word in the list of documents. In the map phase, each mapper is responsible for a subset of input files and a set create intermediate files $word.txt$ that contains the frequency of each word calculated by a specific mapper. Master then shuffles the data in such a way that one reducer receives all the files related to a specific word. And, in the reduce phase each reducer calculates the final count of each word and store the result in the file.

Algorithm 1: `map`

Input: $(String\ \ fileName, String\ \ value)$, fileName: document name, value: document content
Result: $(len, count)$, where len is the word length and $count$ is the number of occurrences of len in $fileName$

for *each line* l *in* $value$ **do**
  EmitIntermediate(l) ;
end

Algorithm 2: `reduce`

Input: $(Integer\ \ wordLen, Iterator\ \ values)$, wordLen: a word length, values: list of counts
Result: $result$, where result is the occurrence count of wordLen

$result \leftarrow 0$;
for *each* v *in* $values$ **do**
  $result\ +\!= v$;
end
return $(result)$;

In algorithm 1, the map function simply emits the count associated with a word length. In algorithm 2, the reduce function sums together all the counts associated with the same word length. **Note that the above algorithms are just a high level abstraction of the word length count example. You will be seeing the detailed algorithms in sections 3.2 and 3.3**.
In this project, we will design and implement a single machine map-reduce using system calls for the above word length count application.
There are three phases in this project: `Master, Map` and `Reduce`.

- In `Master` phase (Refer section 3.1), you will be provided with an input file directory consisting of text files. The master will split the files and share it uniformly with all the mapper processes. **Note: The division of input file and sharing it with the mappers are already present in the template code provided** . Once the mappers complete, the master will spawn the reducer processes. Then the reducer processes will carry out the final word length count in the `Reduce` phase.
- In `Map` phase (Refer section 3.2), your mapper code will be provided with text files. You will have to read the text using the utility function `getMapperTasks()` provided and emit the count of `wordlength` 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. This file be created in folders for all the different word lengths. (Refer section 3.2)
- In the `Reduce` phase (Refer section 3.3), 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. The code for partitioning of data is given in the utility `getReducersTasks()`.(Refer section 3.3). The reducers will read the `m_mapperID.txt` files and compute the total count corresponding to the word length.(Refer section 3.3)

Objective: You will have to design and implement the `Master, Map` and `Reduce` phase.

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 text file relative to the provided Makefile location. The algorithm 3, provides a brief overview of the flow of control in the master process. This is your main control program. The code assumes the mapper and reducer executable are named `mapper` and `reducer`.
File: src/mapreduce.c

Algorithm 3: `master:mapreduce`

Input: $(Integer\ \ nMappers, Integer\ \ nReducers, String\ \ inputFile)$, nMappers: #mappers, nReducers: #reducers, inputFile: text file to be processed

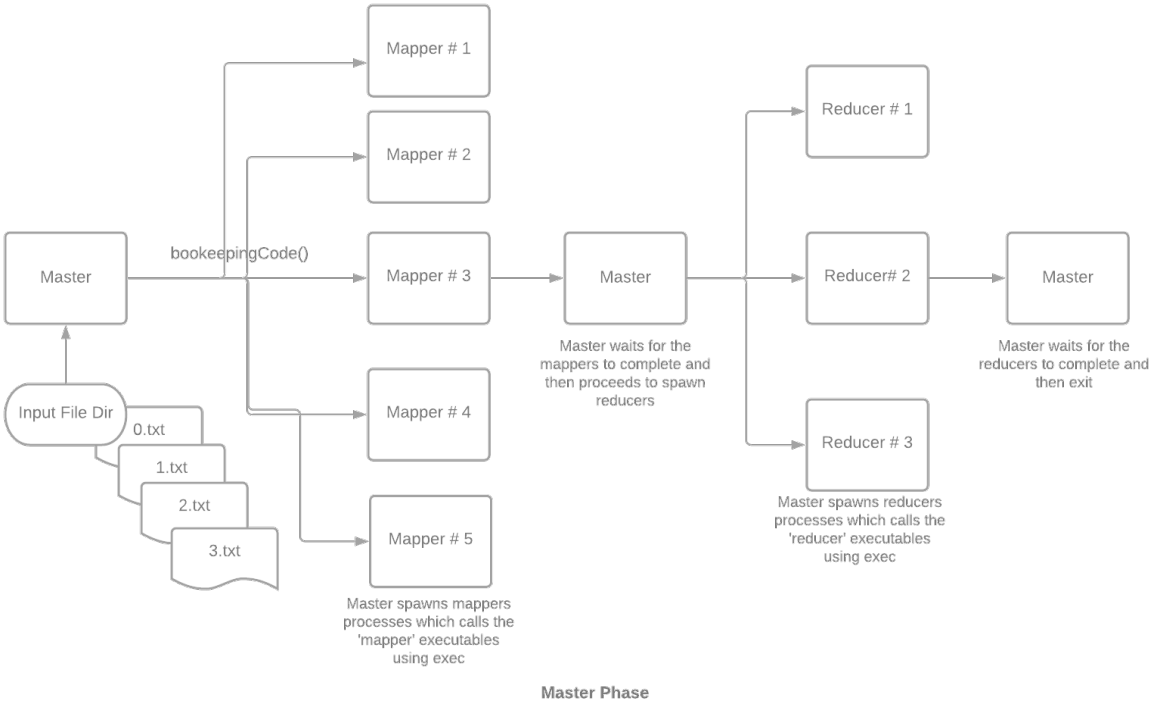
// directory creation and removal
$bookeepingCode()*$;

// 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.o object file. Please do not remove the function call.

First, the master calls a `bookeepingCode()`, which takes care of the creation of `output`, `output/IntermediateData`, `output/FinalData` . 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. Here is a picture!



3.2 Phase 2: Map phase

The mapper takes in three inputs. The mapper's id (i.e. 1, 2, ...). This will be assigned by the master when it calls `exec` on the mapper executable (i.e. it must be passed to `exec` as a command-line argument). Similarly, total number of mappers and path to the input file directory will be passed. The flow of control in mapper is defined in the algorithm 4.
File: src/mapper.c

Algorithm 4: `mapper`

Input: ($Integer\ mapperID$, $Integer\ nMappers$, $String\ inputFileDir$)
Result: ($m\_mapperID.txt$),

// variable to store input file names
$var \quad myTasks \leftarrow NULL$;

// get list of input files to be processed by this mapper
$nTasks \leftarrow getMapperTasks(nMappers, mapperID, inputFileDir, myTasks)$;

//process each task
for *i=0; i* $<$ $nTasks$*; i++* **do**
  $map(myTasks[i])$;
end

$writeIntermediateDS()$;

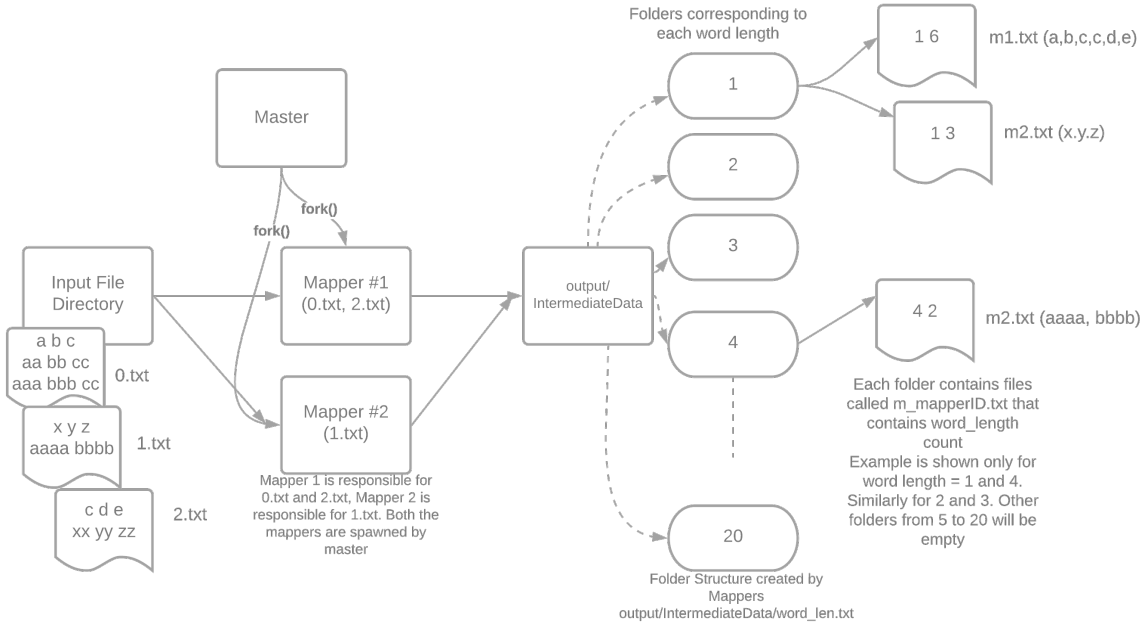
Notice: `getMapperTasks()` is defined in the provided utils.o object file. Please do not remove the function calls.

First, the bookeeping code will create `output/IntermediateData/` folder. Mappers should create folders inside this IntermediateData folder for all possible word lengths starting from 1 to 20 (`MaxWordLength`) where the generated `m_mapperID.txt` will be stored. For example in case of word length=1 and 2 mappers, the structure will be as follows:
In folder `output/IntermediateData/1` ,there will be two files generated by mapper → m_1.txt, m_2.txt.

The intermediate data structure can be of your choice. It can be a simple array where you store word length as indices of array and value be the count of that word length. Or, it can be a Linked List or any other data structure. The contents of the `intermediate data structure` is written to `m_mapperID.txt` files. This file be created in folders for all the different word lengths. For eg: If a mapper with mapperID=2 finds words of length 3 and 4. It will create m_2.txt in folder named 3 and folder named 4. The content of the text file will be just the word length<space>count.

The utility `getMapperTasks()` will be used to retrieve the input files that a particular mapper should process. The list of input file names will be stored in $myTasks$ array. So, if a mapper processes n files and each file contains some words so the mapper will count all the occurrences of a particular word length that was encountered in those n files. There are some other utility functions provided $getFilePointer()$, $getLineFromFile()$, $writeLineToFile()$ that can be used to read file, read line and write line to a file respectively. Read more about it in $utils.h$. These utility functions will be used in Reduce phase too.

Notice: A word should be composed of consecutive characters "c", where "c" ∈ {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.



3.3 Phase 3: Reduce phase

The reducer takes in three inputs. The reducer's id (i.e., 1, 2,..). This will be assigned by the master when it calls the `exec` on the reducer executable similar to the mapper. The other parameters are total number of reducers and intermediate file directory path. The flow of control in the reducer is given in algorithm 5.

File: src/reducer.c

Algorithm 5: `reducer`

Input: $(Integer\ reducerID, Integer\ nReducers, String\ intermediateDir$
Result: $FinalData/wordLength.txt$

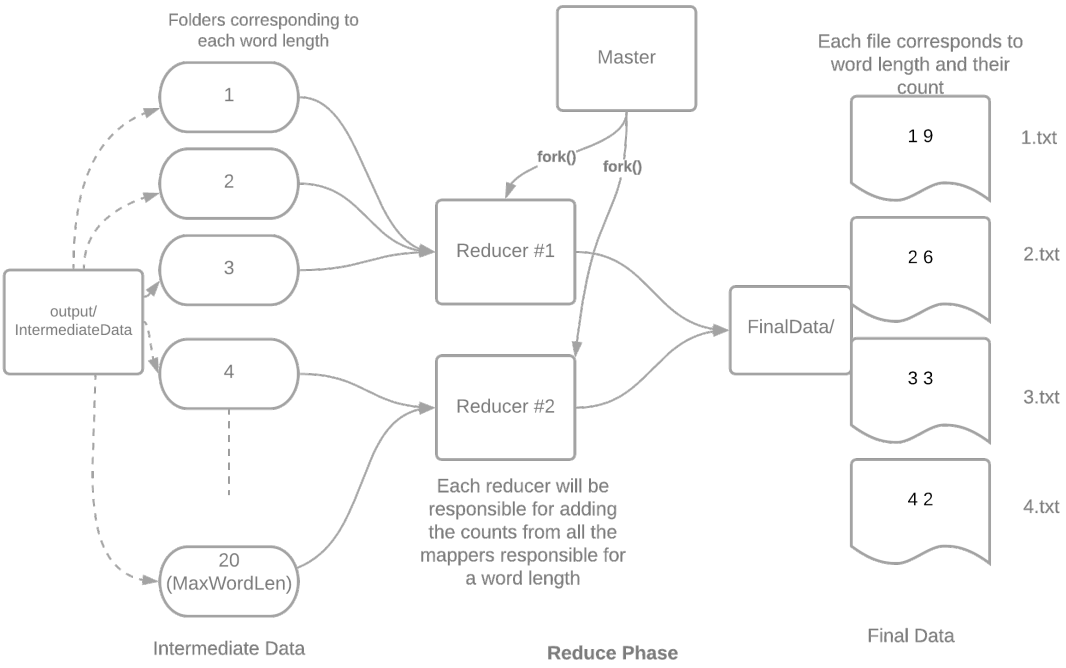
// character array to receive the intermediate file names
$var \quad myTasks[MaxNumIntermediateFiles] \rightarrow NULL;$
// $nTasks \leftarrow getReducerTasks(nReducers, reducerID, intermediateDir, myTasks);$

//process each task
for i=0; $i < nTasks$; i++ **do**
  $reduce(myTasks[i]);$
end

$writeFinalDS();$

Notice: `*getReducerTasks()` is defined in the provided utils.o object file. Please do not remove the function call.

A Reducer after calling getReduceTasks will be receiving the paths of `m_mapperID.txt` that it needs to process. This means all the mapper files corresponding to same word length will be going to the same reducer. Once the reducer receives the file path which is the key, it passes it to the `reduce()`. The `reduce()` calculates the total count for a particular word length from the file contents and stores it in an intermediate structure. The same process is repeated for all the `m_mapperID.txt` files shared. Once all the files are



processed, the reducer will then emit the "wordlength count" results to a file `wordlength.txt` for each word length.
The final data structure can be similar to the intermediate data structure that 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. Read more about it in $utils.h$.

4 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.

Command Line

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

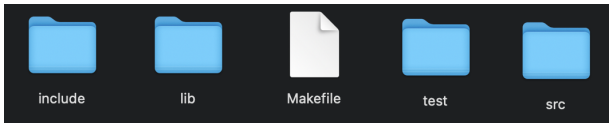
Notice: The final executable name should be `mapreduce`.

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.

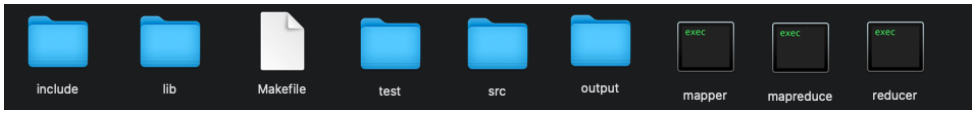
5 Expected Output

Please ensure to follow the guidelines listed below:

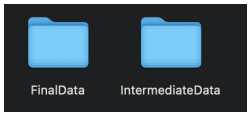
- Do not alter the folder structure. The structure should look as below before compiling via $make$:



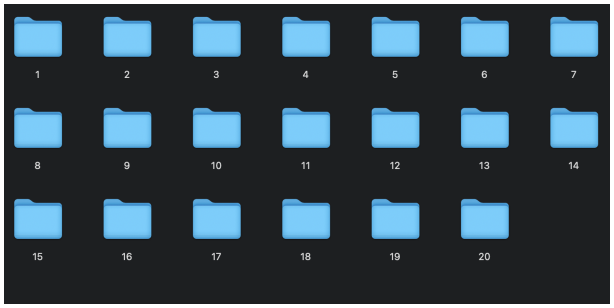
- After compilation, the folder structure will look as below. The output folder is auto-created:



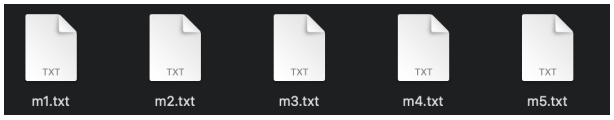
- The output folder content (auto-created) will be as follows:



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



- One of the Word length folders will contain files as follows. :



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

6 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
```

7 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.**

8 Deliverables

One student from each group should upload 2 items to Canvas

1) An Interim Submission. (Due by 02/17/2021, 11:59 PM)
Interim Submission should just contain the src file of mapreduce.c (master) spawning mappers and reducers and waiting for them to finish.

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, ...]
```

9 Getting started

Processes and exec
Start by experimenting with process creation, waiting and termination on simple code. Look at the man pages for `fork, wait`. Next create a simple hello world program, say `hello.c`. Create an executable `hello`. Now create another program called `driver.c`, that will be using different variants of `exec` calls to execute the `hello` executable.

String manipulation
Since the project is about text data, visit various string manipulation functions available in `string.h`. Some of the important functions are `strcpy, strcat, strtok, strcmp, strtol`. Also have a look at `sprintf`, `makeargv`. Read the man pages

Data structures and Dynamic memory allocation
One of the important parts of this project is to create the intermediate and final data structure. As mentioned in the map phase you have various options to build these data structure.

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.

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.

References

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

[2] Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters.