CSci 4061: Introduction to Operating Systems
Spring 2022
Project #1: Basic Map Reduce

Instructor: Jon Weissman

Due: 12 pm Feb 17, 2022

1 Objective

In this project you will be learn about the use of *system calls for processes and I/O*. To do this, you will implement a simple version of MapReduce to the count number of occurrences of words in a text file. MapReduce model was developed by Google and is widely used in industry for Big Data Analytics and is the de-facto standard for Big Data computing! It creates multiple processes that run on many machines; we will create multiple processes on a single machine. The public version is called Hadoop. The writeup is long to explain the details of MapReduce, don't be intimidated 😊

Utility functions are provided with the project template as binaries. Source code for utilities are intentionally not provided as you may be required to implement these in the future. 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 intermediate pairs are then grouped based on the key. Thereduce function will then reduce/merge the grouped intermediate values based on the key to produce theresult. Consider the following example map and reduce *logic* for counting the number of occurrences of each word in a large collection of documents.

Algorithm 1: map

Input: (*String key, String value*), key: document name, value: document content
Result: (*w, count*), where w is the word and *count* is the number of occurrences of w in *key* (i.e the file)

```
for each word w in value do
    EmitIntermediate(w, 1) ;
end
```

Algorithm 2: reduce

Input: (*String key, Iterator values*), key: a word, values: list of counts
Result: *result*, where result is the occurrence count of key

```
result ← 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. In algorithm 2, the reduce function sums together all the counts associated with the same word. **You will be seeing the detailed algorithms in sections 3.2 and 3.4.**

In this project, we will design and implement a single machine map-reduce using system calls for the

above word count application. There are four phases in this project: Master, Map, Shuffle and Reduce.

Objective: You will have to design and implement the Master, Map and Reduce phase. The Shuffle phase will be provided to you as object code.

3 Phase Description

In this section, we will see the brief design details of different phases.

3.1 Phase 1: Master phase

The master process drives all the other phases in the project. It takes three inputs from the user: #mappers, #reducers and the path of the input text file. Algorithm 3,provides a brief overview of the master process.

File: src/mapreduce.c

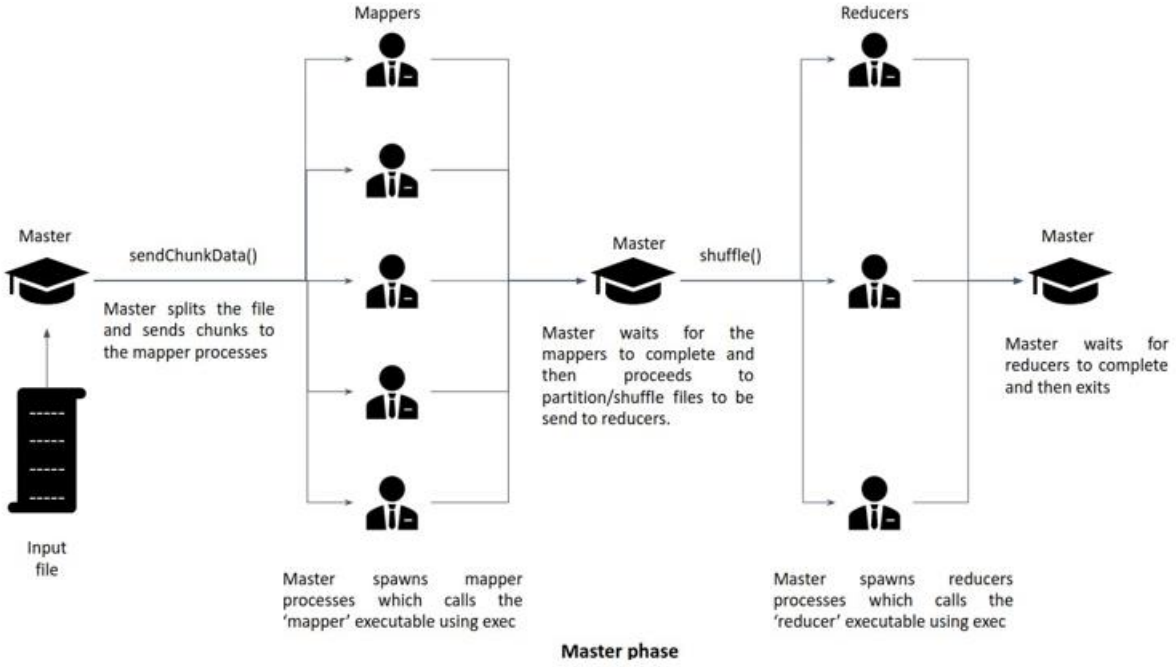
Algorithm 3: master:mapreduce

Input: (*Integer nMappers, Integer nReducers, String inputFile*)
[nMappers: #mappers,nReducers: #reducers, inputFile: text file to be processed]

```
// directory creation and removal
bookeepingCode();

// divides input file into 1024B segments and adds it to the queue
sendChunkData();

// spawn mapper processes with each calling exec on "mapper" executable
spawnMapper(nMappers);

// wait for all child processes to terminate
waitForAll();

// send token.txt files to reducers
shuffle();

// spawn nReducer processes with each calling exec on "reducer" executable
spawnReducers(nReducers);

// wait for all child processes to terminate
waitForAll();
```

Note: bookeepingCode(), sendChunkData() and shuffle() are defined in the provided utils.o object file. Please do not remove the function calls.

It first divides the input file into segments of 1024B and stores them in a queue, from where the mappers will retrieve them one by one, process them in parallel, until the queue is empty. It then spawns mapper processes (using fork() & exec()) and waits for them to complete before proceeding. It then moves to the Shuffle phase where the intermediate output data in the form of word.txt files are partitioned across the reducers.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

Map phase is responsible for counting occurrences of words in the 1024B segments that is assigned to it. The mapper takes in one input, the mapper's id (i.e. 1, 2, ...). This will be passed as an argument by the master whenit calls exec on the mapper executable.

File: src/mapper.c

Algorithm 4: mapper

Input: (*Integer mapperID*), mapperID: mapper's id assigned by master $\in$ {1, 2, ..., nMappers}
Result: (*word.txt*), text files containing the word and list of "1"s (word 1 1 1 1 ...)

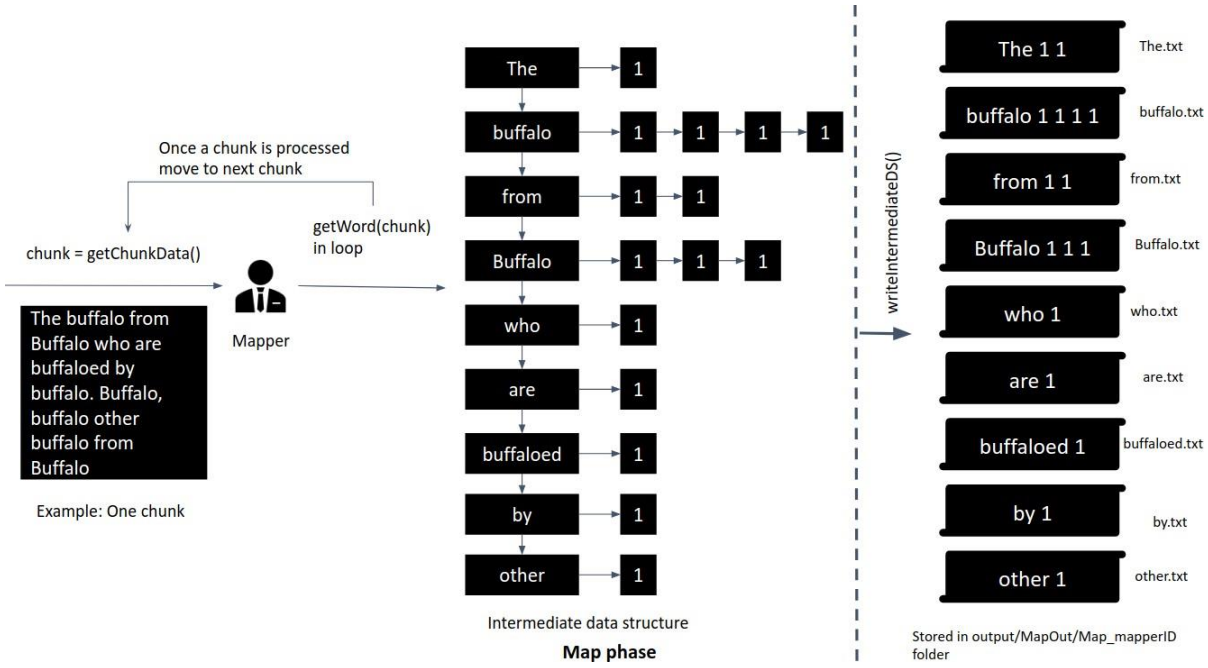
```
// create mapper output directory
mapOutDir ← createMapDir(mapperID);
while master send chunks do
    chunk ← getChunkData(mapperID);
    map(chunk);
end
// write the intermediate structure contents to corresponding words.txt files
writeIntermediateDS();
```

Note: createMapDir() and getChunkData() are defined in the provided utils.o object file. Please do not remove the function calls.

First, the mapper calls createMapDir() to create output/MapOut/Map_mapperID folder where the generated word.txt will be stored. In Master phase we saw that the master will be storing chunks of data into a queue. The mapper will use the getChunkData() (**Provided in utils**) to retrieve these chunks one by one. The received chunk is then passed to the map() to tokenize and to store the value "1" in an intermediate data structure. Note that a word can occur multiple times in a chunk, which means you will have to store a value list of "1"s associated with a word. The definition of word is given below:

Note: A word should be composed of consecutive characters "c", where "c" $\in$ {A...Z, a...z, 0...9}
Example: Thi's is. a te_xt* 0h gr8!!!
The words in this sentence are {Thi, s, is, a, te, xt, 0h, gr8}
Words are case sensitive, which means "text" and "Text" are different.
The getWord() utility allows you to extract out words from a chunk. Refer to utils.h for sample code.

A sample intermediate data structure you can use is provided in mapper.h along with the associated helper functions in mapper.c. It is a two-level nested linked list. The first level is used to store the word and the second level associated to each word is used to store "1"s. Once all the chunks are processed, the mapper will create a word.txt file associated with each word in the intermediate structure. The file content will look like "word 1 1 1 ...".



3.3 Phase 3: Shuffle phase

Note: This phase is not meant to be implemented. It is already provided to you in the mapreduce.c file. Please do not remove the function call.

Once all the mapper processes complete and terminate, the master process will call the shuffle(). The shuffle function will divide the word.txt files in output/MapOut/Map_mapperID folders across nReducers and send the file paths to each reducer based on a hash function.

3.4 Phase 4: Reduce phase

Reduce phase is responsible for producing the final word count. The reducer takes in one input, the reducer's id. This will be assigned by the master when it calls the execon the reducer executable. The flow of control in the reducer is given in algorithm5.

In this phase the master process will be sending the paths of word.txt to the reducers based ona hash function. This means files with same names across different Map_mapperID folders will be going to the same reducer. Once the reducer receives the file path, it passes it to the reduce(). Thereduce() function calculates the total count for the word from the file contents and stores it in an intermediate structure provided to you in reducer.h and reducer.c. The same process is repeated for all the word.txt files. Once all the files are processed, the reducer will then emit the "word count" results to a single file Reduce_reducerID.txt.

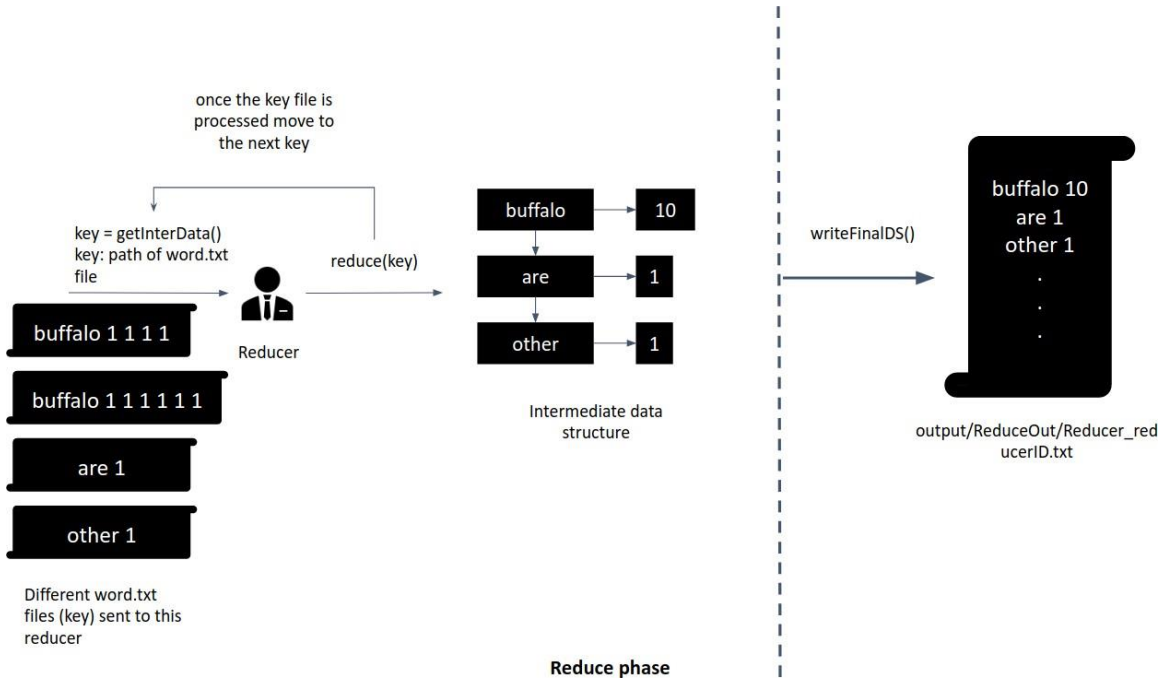
File: src/reducer.c

Algorithm 5: reducer

Input: (*Integer reducerID*), reducerID: reducer's id assigned by master $\in$ {1, 2, …, nReducers}
Result: *Reducer_reducerID.txt*, The text file will consist of the final count corresponding to each word sent to (i.e. assigned to) the reducer by the master

```
// character array to receive the word.txt path (i.e. the file containing the intermediate pairs for a
  particular key)
var   key[KEY SZ];
while master sends key do
    getInterData(key, reducerID);
    reduce(key);

end
// This is an optional function to write the final intermediate
// structure you may use to store the final <word, count> per reducer, to file
  Reduce_reducerID.txt
// Instead, you can add logic of your own to write the <word,count> data
// to Reduce_reducerID.txt in the reduce() function itself
writeFinalDS();
```

Note: getInterData() is defined in the provided utils.o object file. Please do not remove the function call.



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 use #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
```

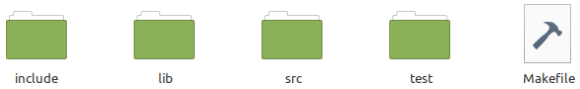
Note: The final executable name should be mapreduce.

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

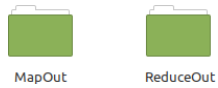
Please follow the guidelines listed below:

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

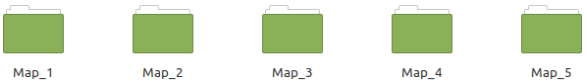


- After compilation, the folder structure will look as below. The output folder is auto-created:

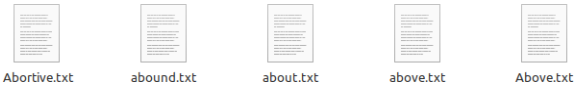
- The output folder content (auto-created) will be as follows:



- The MapOut folder content (auto-created) will be as follows for 5 mappers:



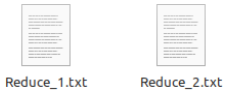
- The Map_mapperID folder content will be as follows. The files should be created by your code:



- A sample word.txt file should look as follows. Here the list of "1"s emitted are from the value list associated with the word in the intermediate structure of Map phase. In this case, the word above occurred 8 times in the chunks retrieved by the corresponding mapper:

```
above 1 1 1 1 1 1 1 1
```

- The ReduceOut folder content will be as follows for 2 reducers. The files should be created by your code:



- A sample Reduce_reducerID.txt file should look as follows:

```
Periods 1
veil 8
raise 15
erspread 1
Exposed 1
Son 55
unmoved 7
righteous 5
anon 7
whence 30
```

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 to see if your solution matches our solution. You can navigate to the folder and run the following command to see the expected output. During the execution if there areany issues, please let us know as soon as possible.

```
Command Line
$ cd solutionexe
$ ./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.
- 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 50 bytes.
- Follow the expected output information provided in the previous section.
- The chunk size will be at most 1024 bytes as there is a chance that some of the 1024th byte in input File is the middle of a word.
- 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

There will be 2 submissions, one intermediate submission due 1 week after the release of the project and a final submission due 2 weeks after the release.

Intermediate Submission :

Complete mapreduce.c file. [Look at the TODOs in the template code provided for mapreduce.c]. Ensure that correct number of mapper and reducer prints are output to the console. The print statements are already added in the template code for mapper.c and reducer.c.

Both intermediate and final submissions should contain the following.

One student from each group should upload to Canvas , 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
- 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 must properly describe the above points. The code should be well commented (Need not explain every line). 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. (fork(), exec() and wait())

File system calls
Use file system calls open, write, read, close to create a file, write some contents, read the contents and to close the file respectively. Have a look at the man pages of the function to understand them in detail. You can also use C library filecalls like fopen, fread, fwrite, fclose or any other high-level I/O calls if you wish.

String manipulation
Since the project is about text data, go over various string manipulation functions. Someof the important functions are strcpy, strcat, strtok, strcmp, strtol, sprintf.

10 Rubric: Subject to change

- 5% README
- 15% Intermediate submission [Including Readme, code]
- 15% Documentation within code, coding, and style: indentations, readability of code, use of defined constants rather than numbers
- 65% 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.
- A sample test case is provide to you upfront. You may change the value of #mappers and #reducers to test out your code. Think about other corner cases that may occur in the code, for example, an empty input file. Your code should be able to handle such cases. 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 will use the GCC version installed on the CSELabs machines 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 map/MapOut/Map_mapperID and map/ReduceOut/Reduce_reducerID.txt with the one that we have generated. The TAs will be 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, without including the ones generated by the template code. 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.