Objectives

The goal of this project is to introduce you to a paradigm of parallel programming called "Map/Reduce".

You will do a simple Map Reduce implementation in C programming language to illustrate the basic functioning of a MapReduce framework. Your implementation will run on local machine and will fork the corresponding worker processes to simulate parallel processing in a cluster of machines. The Inter-Process Communication (IPC) among the workers processes and the parent process (the user) is achieved by using Unnamed Pipes.

MapReduce

In 2004, Google released a general framework for processing large data sets on clusters of computers. We recommend you read this paper that gives detailed information about MapReduce. However, we will explain everything you need to know below.

To demonstrate what MapReduce can do, we'll start with a small dataset–three lines of text:

```
Hello
there
class!
```

The goal of this MapReduce program will be to count the number of occurrences of each letter in the input.

MapReduce is designed to make it easy to process large data sets, spreading the work across many machines. We'll start by splitting our (not so large) data set into one chunk per line.

	Chunk #1	**Chunk #2**	**Chunk #3**
Input	"Hello"	"there"	"class!"

Map. Once the data is split into chunks, `map()` is used to convert the input into `(key, value)` pairs. In this example, our `map()` function will create a `(key, value)` pair for each letter in the input, where the key is the letter and the value is 1.

	Chunk #1	**Chunk #2**	**Chunk #3**
Input	"Hello"	"there"	"class!"
Output	`(h, 1)`	`(t, 1)`	`(c, 1)`
	`(e, 1)`	`(h, 1)`	`(l, 1)`
	`(l, 1)`	`(e, 1)`	`(a, 1)`
	`(l, 1)`	`(r, 1)`	`(s, 1)`

	Chunk #1	Chunk #2	Chunk #3
	`(o, 1)`	`(e, 1)`	`(s, 1)`

Reduce. Now that the data is organized into `(key, value)` pairs, the `reduce()` function is used to combine all the values for each key. In this example, it will "reduce" multiple values by adding up the counts for each letter. Note that only values for the same key are reduced. Each key is reduced independently, which makes it easy to process keys in parallel.

	Chunk #1	Chunk #2	Chunk #3
Input	`(h, 1)`	`(t, 1)`	`(c, 1)`
	`(e, 1)`	`(h, 1)`	`(l, 1)`
	`(l, 1)`	`(e, 1)`	`(a, 1)`
	`(l, 1)`	`(r, 1)`	`(s, 1)`
	`(o, 1)`	`(e, 1)`	`(s, 1)`
Output	`(a, 1)`		
	`(c, 1)`		
	`(e, 3)`		
	`(h, 2)`		
	`(l, 3)`		
	`(o, 1)`		
	`(r, 1)`		
	`(s, 2)`		
	`(t, 1)`		

Note: Here we gave an example of letter count, but you need to implement word count. The word count example will be given in discussion session.

Assignment

For this project you will build a simplified version of the MapReduce framework and WordCount application.

Your implementation will run multiple processes on one machine as independent processing units and use IPC mechanisms for communication. `map()` and `reduce()` will be programs that read from standard input and write to standard output. The input data for each mapper program will be lines of text.

You will spread the work across ***N*** instances of the mapper executable. The input file will be split into ***N*** chunks, with one chunk for each mapper process. The mapper processes will run in parallel.

Reduce invocations are distributed by partitioning the intermediate key space into ***M*** pieces. For simplicity, set the number of Reduce workers equal to ***M = N/4***.

Your program will:

- Split the input file into N parts and pipe the contents into N different mapper processes
- Pipe the output of the mapper processes into the M reducer processes
- Write the output of the reducer processes to the output file.
- Parallelize these tasks to achieve speedup

Hint: You might want to use pthread_mutex if you experience wrong results or deadlocks if you are planning to use multithreading and multiprocessing together.

Things we will be testing for:

- Inputs of varying size
- Different types of mapper and reducer tasks
- Both mapper and reducer generating accurate output to stdout file descriptor independently
- Splitter being used correctly to generate equally sized input data for each mapper
- Correctly logging output
- All mappers being run in parallel, resulting in **performance speedup if multi-cores are assigned.**
- **No memory leaks and memory errors** when running the application

Things that will not be tested for:

- Illegal inputs for map-reduce (Input data in a format other than as described above)

Restricted Functions

Since a learning objective of this assignment is to use the fork-exec-wait pattern and to learn inter-process communication, if you use `popen`, `system`, or related functions you will automatically fail this project. For this task, you can use any system you want for implementation, and testing will be done from GitHub.

You definitely cannot use Hadoop or any other exterior implementations.

Input Data

For an input file like below,

input.txt

```
Hello, we implement a WordCount application for operating systems course!

All the projects were fun but WordCount project is the most fun.

Make sure to have all characters in test file as ASCII.

have fun!!!
```

the expected output file will be (sorted):

```
a 1
all 2
application 1
as 1
ascii 1
but 1
characters 1
course 1
file 1
for 1
fun 3
have 2
hello 1
implement 1
in 1
is 1
make 1
most 1
operating 1
project 1
projects 1
sure 1
systems 1
test 1
the 2
to 1
we 1
were 1
wordcount 2
```

You can use the IDE we provided in GitHub repo (Replit) or any other text editor of your choice to implement the project. We will provide all test inputs and outputs for this task in your repo.

You can also download books from here for testing. We will discuss some of the input/output examples in discussion session.

We have provided a sample main.c for your reference which uses bruteforce to pass some of the tests. Sample code does not use map/reduce so you have to change it into map/reduce. **You should edit the Makefile in your repo in order GitHub tests to be working.**

Grading

Total number of the points for this project is 100.

Report (10 points):

See Submission Instructions document in Resources for details.

- DESIGNDOC.txt/pdf
- README.md

MapReduce Functionality (30 points):

All the functionalities described above will be checked, and if no map reduce is implemented, you will automatically get 0. You can read this link for Map Reduce functionality.

Tests (40 pts):

This time we will use GitHub for testing your code and you would be able to see your score immediately after pushing your code. There are 20 tests (very basic tests to hard ones), and each test is 2 points.

If you do not use map/reduce, you will automatically get 0 from all tests even if you pass them.

Free Response Questions (20 points)

FRQ will be posted separately to Piazza and Blackboard.