CSC209 Assignment 3: MapReduce

Due: 3:00 p.m. Wednesday March 23

Groups: You may work on this assignment individually or with a partner. Your partner can be from any section. Login to MarkUs to form a group and get a repo URL.

Starter code: a3_starter.tar

Introduction

> MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.
>
> Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication.
>
> – Jeffrey Dean and Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*

In this assignment, you will implement a simplified engine to run programs written using a MapReduce model. Since we have not yet studied how to communicate between processes on different machines, we will focus on using different processes communicating through pipes on a single machine.

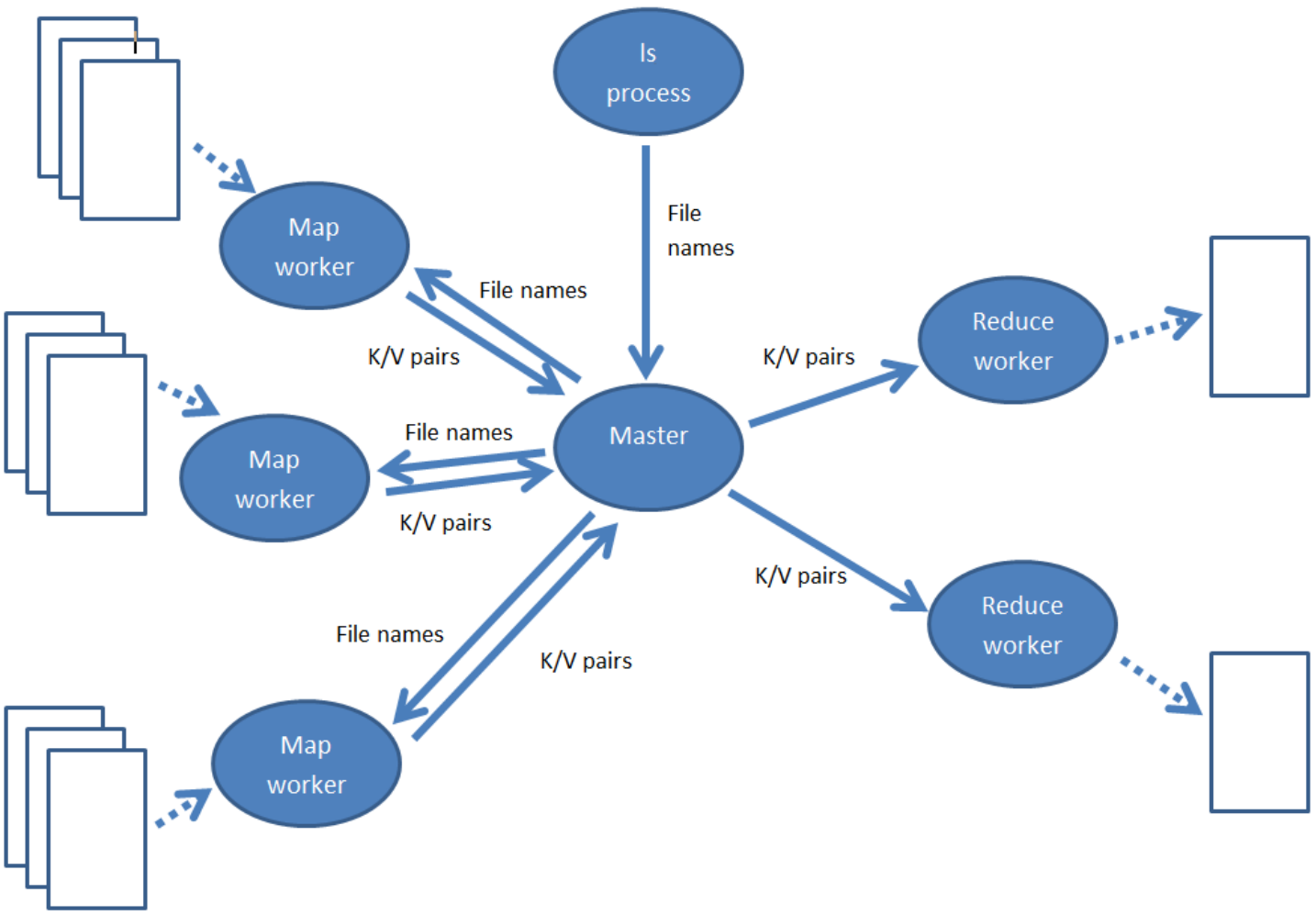
You will be writing the engine but not the map and reduce functions normally supplied by the programmer who is making use of the engine. We have provided these functions for the word frequency counting example described in the paper.

Algorithm

This section describes the MapReduce framework which you will implement. Read through this section carefully before writing any code. There will be four types of processes created for the execution of your program:

1. **master process**: the original process
2. **ls process**: process which performs an `ls` to get input filenames
3. **map worker**: process which performs map operations on input files
4. **reduce worker**: process which performs reduce operations on computed values

The diagram below gives an overview of the communication structure of the program. The dashed lines indicate file I/O operations, and the solid coloured lines indicate pipes between processes.



Master process

The master process creates the processes to perform map and reduce operations, assigns input files to the map processes, and facilitates communication between the different processes using pipes. While in sophisticated MapReduce networks the map and reduce nodes would communicate with each other using some protocol, to keep things simple for this assignment the other processes will only communicate with the master process directly.

The master process should proceed in four phases:

1. Create the necessary processes and pipes.
2. Read in the input filenames and assign them *evenly* to the map workers.

3. Read output key/value pairs from the map workers and send them *evenly* to reduce workers.
4. Wait for all of its child processes to exit.

Each map worker is assigned roughly the same *number of files* to handle, and that each file is assigned to only one worker.

When assigning key/value pairs to reduce workers, all pairs with the same key must be sent to the same worker - i.e., the master process partitions the pairs by groups of keys, and sends each partition to a different reduce worker. Note that each reduce worker can be responsible for more than one key, depending on the number of keys.

It is a little challenging to assign the same number of *pairs* to each reduce worker, and this is not required for this assignment. However, your implementation should assign approximately the same number of *keys* to each worker.

Note 1: all communication between the master and other processes must be done through pipes. However, **you must use `dup2` to make the master process read the output of the ls process from standard input.** For example, the master process could use `scanf` in a loop to read in file names one at a time.

Note 2: you are not required to use `select` (or something else) to choose which pipe to read from at any given time. Instead, you may (for example) use a simple loop to read from one map worker at a time, even completely exhausting one pipe before moving on to the next.

Map worker

A map worker process reads in input filenames sent by the master process. For each input file, it opens the file and reads in chunks into a buffer, calling a `map` function on each chunk.

It is the responsibility of the user of the framework to provide a valid `map` function that has the following prototype:

```
void map(const char *chunk, int outfd)
```

This function performs some computation on `chunk`, then writes zero or more key/value pairs to `outfd`, which must be the write-end of a pipe back to the master process. You can see an example of a `map` function in `word_freq.c`: this is a function that emits one pair per word in the file.

The size of the chunks is defined in the starter code (`mapreduce.h`); please do not change this, but make sure to add a final null-terminator to each chunk. Don't worry about what happens if a word is split across two chunks -- you are not expected to handle this case specially.

Reduce worker

A reduce worker process reads a sequence of key/value pairs from a pipe. It builds up a list of pairs for each distinct key, and calls the `reduce` function once for each key.

As with `map`, users are expected to provide a `reduce` function which conforms to the following prototype:

```
Pair reduce(const char *key, const LLValues *values)
```

Given a key and a list of values corresponding to that key, this function returns a new key/value pair. For example, the `reduce` in `word_freq.c` simply converts the values to numbers and adds them up.

Finally, the reduce worker takes each returned pair and writes it (in binary) to a file named `[pid].out`, where `[pid]` is replaced by the PID of the worker.

Note: a reduce worker should not call the `reduce` function until it has read in *all* pairs. You will need to make sure to close pipes appropriately so that the reduce function is called at the right time.

Your task

This assignment is more open-ended than the first two. You are responsible for creating your own source code files which implement the functionality described in the previous section. You are also responsible for creating your own Makefile; running `make` on CDF with your code must produce an executable called `mapreduce`, which takes the following command line option arguments:

1. `-m numprocs`, where `numprocs` is the number of map workers. If not provided, default is 2.
2. `-r numprocs`, where `numprocs` is the number of reduce workers. Default is 2.
3. `-d dirname` where `dirname` is the absolute or relative path to a directory containing input files. (You should not assume that `dirname` provided by the user will end in a slash.) This argument is **required**.

Your Makefile should use `word_freq.c` in the compiled executable, so that you actually have a standalone program which executes a MapReduce algorithm to compute word frequencies across a set of files. (In your spare time, you might want to experiment with different `map` and `reduce` implementations for different algorithms -- try searching online for some ideas!)

To help the readability of your code, please separate your code for the master process, map workers, and reduce workers into different files (e.g., `master.c`, `mapworker.c`, `reduceworker.c`). It will be tempting to make your master process code one very long function - use helper functions appropriately to keep your `main` function short.

We have provided some starter code files - download and do not modify these files. `mapreduce.h` defines useful constants, data structures, and documentation for the external interface for `map` and `reduce`. `linkedlist.h` and `linkedlist.c` provide a specialized linked list implementation; we strongly recommend taking advantage of these

functions so that you don't need to worry about storing and manipulating the key-pair data. Finally, `word_freq.c` contains a sample implementation of `map` and `reduce`. We have provided some sample data in the folder `texts`.

Requirements

You must use `getopt` to read the arguments. For example the program may be run as `mapreduce -m 5 -r 3 -d texts`. The order of the arguments should not matter, so `mapreduce -d texts -r 3 -m 5` is an equivalent call. If `-d` is not specified, or the arguments for `-m` or `-r` are either missing or cannot be fully parsed as a positive integer, print a usage message and terminate the program with a non-zero exit status.

You must fork a process to run an `ls` command on the given directory, and use pipes and `dup2` to redirect the stdout of that process to the stdin of the master process.

You may not change any of the provided data structures which are relevant to the `map` and `reduce` interface - we will be assuming your program follows this interface when testing your code.

You must perform error-checking for all system calls (and functions which use system calls), in the same style as in the video lectures. We recommend defining a set of helper functions to wrap these calls to reduce duplication.

You must explicitly free all dynamically-allocated memory, for all processes. This includes linked lists - see the documentation in `linkedlist.h`. You must also close all file descriptors for all processes, except stdin, stdout, and stderr.

Your Makefile must use the gcc flags `-std=c99`, `-Wall`, and `-Werror`.

Advice

Unfortunately, debuggers are of limited value when working with programs that spawn other processes.

Make sure you can create all processes, send dummy messages through pipes, and close all pipes properly *before* trying to use the provided `map` and `reduce` functions. The algorithm for this assignment is actually quite simple; you'll spend most of your time fiddling with pipes.

There are two common problems that show up when programming pipes. The first is that you will sometimes see a process block on a read from a pipe even when you are quite sure that nobody will be writing to it. This happens when all of the write file descriptors on the pipe are not closed. Remember that file descriptors are duplicated when you call `fork`.

The second common problem is to see a "Broken pipe" message. This happens when a process writes to a pipe that has already been closed. In the case of this assignment, that would be an error, because the read side of the pipe should not be closed until all

of the pairs have been read.

Submission instructions

Make sure you have **logged into MarkUs and created a group or selected "Work Alone."** Once you do so, you will be told what svn repository to use.

Commit all of your source and header files to your A3 repository. Also commit your Makefile. Make sure your code compiles on CDF with the required flags before your final submission. Also make sure to check your repo URL *on MarkUs* before you submit, even if you are working alone (see above paragraph). If you don't do so, your work will not be graded!

Coding style and comments (worth roughly 10%) are just as important in CSC209 as they were in previous courses. Use good variable names, appropriate functions, descriptive comments, and blank lines. Remember that someone needs to read your code. You may write extra helper functions.

Please remember that if you submit code that **does not compile, or produces any warning messages from -Wall, it will receive a grade of 0**. The best way to avoid this potential problem is to write your code incrementally. The starter code compiles without warnings and runs. As you solve each small piece of the problem make sure that your new version continues to compile without warnings and run. Get a small piece working, commit it, and then move on to the next piece. This is a much better approach than writing a whole bunch of code and then spending a lot of time debugging it.