Your Job

Your job is to implement a distributed MapReduce, consisting of two programs, the master and the worker. There will be just one master process, and one or more worker processes executing in parallel. In a real system the workers would run on a bunch of different machines, but for this lab you'll run them all on a single machine. The workers will talk to the master via RPC. Each worker process will ask the master for a task, read the task's input from one or more files, execute the task, and write the task's output to one or more files. The master should notice if a worker hasn't completed its task in a reasonable amount of time (for this lab, use ten seconds), and give the same task to a different worker.

We have given you a little code to start you off. The "main" routines for the master and worker are in `main/mrmaster.go` and `main/mrworker.go`; don't change these files. You should put your implementation in `mr/master.go`, `mr/worker.go`, and `mr/rpc.go`.

Here's how to run your code on the word-count MapReduce application. First, make sure the word-count plugin is freshly built:

```
$ go build -buildmode=plugin ../mrapps/wc.go
```

In the `main` directory, run the master.

```
$ rm mr-out*
$ go run mrmaster.go pg-*.txt
```

The `pg-*.txt` arguments to `mrmaster.go` are the input files; each file corresponds to one "split", and is the input to one Map task.

In one or more other windows, run some workers:

```
$ go run mrworker.go wc.so
```

When the workers and master have finished, look at the output in `mr-out-*`. When you've completed the lab, the sorted union of the output files should match the sequential output, like this:

```
$ cat mr-out-* | sort | more
A 509
ABOUT 2
```

```
ACT 8
...
```

We supply you with a test script in `main/test-mr.sh`. The tests check that the `wc` and `indexer` MapReduce applications produce the correct output when given the `pg-xxx.txt` files as input. The tests also check that your implementation runs the Map and Reduce tasks in parallel, and that your implementation recovers from workers that crash while running tasks.

If you run the test script now, it will hang because the master never finishes:

```
$ cd ~/6.824/src/main
$ sh test-mr.sh
*** Starting wc test.
```

You can change `ret := false` to true in the Done function in `mr/master.go` so that the master exits immediately. Then:

```
$ sh ./test-mr.sh
*** Starting wc test.
sort: No such file or directory
cmp: EOF on mr-wc-all
--- wc output is not the same as mr-correct-wc.txt
--- wc test: FAIL
$
```

The test script expects to see output in files named `mr-out-X`, one for each reduce task. The empty implementations of `mr/master.go` and `mr/worker.go` don't produce those files (or do much of anything else), so the test fails.

When you've finished, the test script output should look like this:

```
$ sh ./test-mr.sh
*** Starting wc test.
--- wc test: PASS
*** Starting indexer test.
--- indexer test: PASS
*** Starting map parallelism test.
--- map parallelism test: PASS
*** Starting reduce parallelism test.
--- reduce parallelism test: PASS
*** Starting crash test.
```

```
--- crash test: PASS
*** PASSED ALL TESTS
$
```

You'll also see some errors from the Go RPC package that look like

```
2019/12/16 13:27:09 rpc.Register: method "Done" has 1 input parameters; needs exactly three
```

Ignore these messages.

A few rules:

- The map phase should divide the intermediate keys into buckets for `nReduce` reduce tasks, where `nReduce` is the argument that `main/mrmaster.go` passes to `MakeMaster()`.
- The worker implementation should put the output of the X'th reduce task in the file `mr-out-X`.
- A `mr-out-X` file should contain one line per Reduce function output. The line should be generated with the Go `"%v %v"` format, called with the key and value. Have a look in `main/mrsequential.go` for the line commented "this is the correct format". The test script will fail if your implementation deviates too much from this format.
- You can modify `mr/worker.go`, `mr/master.go`, and `mr/rpc.go`. You can temporarily modify other files for testing, but make sure your code works with the original versions; we'll test with the original versions.
- The worker should put intermediate Map output in files in the current directory, where your worker can later read them as input to Reduce tasks.
- `main/mrmaster.go` expects `mr/master.go` to implement a `Done()` method that returns true when the MapReduce job is completely finished; at that point, `mrmaster.go` will exit.
- When the job is completely finished, the worker processes should exit. A simple way to implement this is to use the return value from `call()`: if the worker fails to contact the master, it can assume that the master has exited because the job is done, and so the worker can terminate too. Depending on your design, you might also find it helpful to have a "please exit" pseudo-task that the master can give to workers.