Part I: Map/Reduce input and output

fix the sequential implementation

1. Task: the code we give you is missing two crucial pieces:
(1) the function that divides up the output of a map task,
`doMap()` function in `common_map.go`
(2) the function that gathers all the inputs for a reduce task
`doReduce()` function in `common_reduce.go`

2. Test: checks the correctness of your implementation, these tests are implemented in the file `test_test.go`

```
$ cd 6.824
$ export "GOPATH=$PWD"    # go needs $GOPATH to be set to
the project's working directory
$ cd "$GOPATH/src/mapreduce"
$ go test -run Sequential
ok      mapreduce 2.694s
```

3. Bug: If the output did not show *ok* next to the tests, your implementation has a bug in it.
Debug: set `debugEnabled = true` in `common.go`, and add `-v` to the test command above

```
$ env "GOPATH=$PWD/../../" go test -v -run Sequential
=== RUN     TestSequentialSingle
master: Starting Map/Reduce task test
Merge: read mrtmp.test-res-0
master: Map/Reduce task completed
--- PASS: TestSequentialSingle (1.34s)
=== RUN     TestSequentialMany
master: Starting Map/Reduce task test
Merge: read mrtmp.test-res-0
Merge: read mrtmp.test-res-1
Merge: read mrtmp.test-res-2
master: Map/Reduce task completed
--- PASS: TestSequentialMany (1.33s)
PASS
ok      mapreduce 2.672s
```

Part II: Single-worker word count

1. Task: find empty `mapF()` and `reduceF()` functions. Your job is to insert code so that `wc.go` reports the number of occurrences of each word in its input.

2. Test I: There are some input files with pathnames of the form `pg-*.txt` in ~/6.824/src/main, downloaded from Project Gutenberg. Here's how to run `wc` with the input files:

```
$ cd 6.824
$ export "GOPATH=$PWD"
$ cd "$GOPATH/src/main"
$ go run wc.go master sequential pg-*.txt
# command-line-arguments
./wc.go:14: missing return at end of function
./wc.go:21: missing return at end of function
```

3. Hint

(1) Your `mapF()` will be passed the name of a file, as well as that file's contents; it should split the contents into words, and return a Go slice of `mapreduce.KeyValue`. While you can choose what to put in the keys and values for the `mapF` output, for word count it only makes sense to use words as the keys.

(2) Your `reduceF()` will be called once for each key, with a slice of all the values generated by `mapF()` for that key. It must return a string containing the total number of occurences of the key.

(3) Hint URL

- a good read on what strings are in Go is the Go Blog on strings.
- you can use `strings.FieldsFunc` to split a string into components.
- the strconv package (http://golang.org/pkg/strconv/) is handy to convert strings to integers etc.

4. Test II: You can test your solution using:

```
$ cd "$GOPATH/src/main"
$ time go run wc.go master sequential pg-*.txt
master: Starting Map/Reduce task wcseq
Merge: read mrtmp.wcseq-res-0
Merge: read mrtmp.wcseq-res-1
Merge: read mrtmp.wcseq-res-2
master: Map/Reduce task completed
14.59user 3.78system 0:14.81elapsed
```

Correct Answer:

The output will be in the file "mrtmp.wcseq". Your implementation is correct if the following command produces the output shown here:

```
$ sort -n -k2 mrtmp.wcseq | tail -10
he: 34077
was: 37044
that: 37495
I: 44502
in: 46092
```

```
a: 60558
to: 74357
of: 79727
and: 93990
the: 154024
```

Remove Output:

You can remove the output file and all intermediate files with:

```
$ rm mrtmp.*
```

Easy Test:

To make testing easy for you, run:

```
$ bash ./test-wc.sh
```

Part III: Distributing MapReduce tasks

you will complete a version of MapReduce that splits the work over a set of worker threads that run in parallel on multiple cores. Your implementation will use RPC to simulate distributed computation.

code in `mapreduce/master.go` does most of the work of managing a MapReduce job
code for a worker thread, in `mapreduce/worker.go`
code to deal with RPC in `mapreduce/common_rpc.go`

Your solution to Part III should only involve modifications to `schedule.go`.

1. Task

implement `schedule()` in `mapreduce/schedule.go`

(1) The master calls `schedule()` twice during a MapReduce job, once for the Map phase, and once for the Reduce phase.

(2) `schedule()`'s job is to hand out tasks to the available workers.

(3) `schedule()` must give each worker a sequence of tasks, one at a time. `schedule()` should wait until all tasks have completed, and then return.

(4) `schedule()` tells a worker to execute a task by sending a `Worker.DoTask` RPC to the worker.

(5) Use the `call()` function in `mapreduce/common_rpc.go` to send an RPC to a worker. The first argument is the the worker's address, as read from `registerChan`. The second argument should be `"Worker.DoTask"`. The third argument should be the `DoTaskArgs` structure, and the last argument should be `nil`.

2. Test: To test your solution, you should use the same Go test suite as you did in Part I, but replace `-run Sequential` with `-run TestBasic`.

```
$ go test -run TestBasic
```

3. Hint

- RPC package documents the Go RPC package.
- `schedule()` should send RPCs to the workers in parallel so that the workers can work on tasks

concurrently. You will find the `go` statement useful for this purpose; see Concurrency in Go.

- `schedule()` must wait for a worker to finish before it can give it another task. You may find Go's channels useful.
- You may find sync.WaitGroup useful.
- The easiest way to track down bugs is to insert print state statements (perhaps calling `debug()` in `common.go`), collect the output in a file with `go test -run TestBasic > out`, and then think about whether the output matches your understanding of how your code should behave. The last step (thinking) is the most important.
- To check if your code has race conditions, run Go's race detectorwith your test: `go test -race -run TestBasic > out`.
-

Part IV: Handling worker failures

1. Task

If the master's RPC to the worker fails, the master should re-assign the task given to the failed worker to another worker. It may happen that two workers receive the same task, compute it, and generate output.

2. Test

Your implementation must pass the two remaining test cases in `test_test.go`.

```
$ go test -run Failure
```