Lab 1: Map Reduce

Summary

This lab is my first experience with Golang. Compared to C/C++, I found it immensely helpful that Golang has automatic garbage collections and race detectors. Also, it's been a while for me to write concurrent programs using multithreading, locks, and condition variables and in my implementations I actually used all of these to support a solution with parallelism.

Features

Stateless Worker

- In this lab, I found out we actually don't need to know the state of the workers. Workers could be alive, dead, or slow, and as long as tasks are finished in time, then every worker is the same. To keep track of the task status, I created a Task struct like the following:

```
type Task struct {
  start time.Time
  status TaskStatus
  file string
  id int
}
```

- I used `start` to keep track of when a task was assigned and if it exceeds 10 seconds, then we should reassign the task to other worker.

GoRoutine Monitoring

- I found GoRoutine immensely helpful
- To check the 10 seconds after a task has been assigned, in `GetTasksForWorker` function, I spawn a goroutine monitoring the task that got assigned, and the goroutine will flip the `status` of task to `NotAssigned` if the task is still `inProgress` after 10 seconds. If the task is `Completed`, the Goroutine will simply exit

Lazy Task

- Instead of creating GoRoutine monitoring all tasks once the Coordinator is up, we only need to check the task's status when we want to get tasks for a specific worker. That's why we spawned the go routine in `GetTasksForWorker` instead of in `MakeCoordinator`

Interesting Mistakes I made

DeadLocks

- I ran into dead locks a couple of times when implementing my solution
- First, I was not clear about the `defer` keyword in Golang, and in my solution I used `defer c.mu.Unlock()` almost all the times and thinking it is going to unlock the lock before the function returns.
- Second, when monitoring the task, I thought of using a while-loop to continuously monitoring the task and use time.Sleep(10 * Seconds) to run the iteration once in 10 seconds. However, wrapping a lock outside a while-loop clearly is not a good idea as `HandleFinished` task and `GetTasksForWorkers` both required this lock to work
    - So I ended up only locking the flip action

Condition Variable

- I was really dumb to think that condition variable requires no initialization steps. However, it turns out

`c.cond = *sync.NewCond(&c.mu)` is essential when using condition variable

Pointers

- The expression `for idx, task := range tasks` , task is actually a value and makes a copy of `tasks[idx]`, this took me some time to realize and after that i started using more of a traditional for-loop