**Assignment 1**

Due date: 11:59PM EST, 2/21/2017

*You may work alone or in a team of up to 2 students at most.*

*One of your three Assignments throughout the semester is skippable.*

The goal of this assignment is to parallelize reading data out of multiple files into a shared data structure using only channels for communication, while at the same time running queries against the data that has been read out so far. The driver file, **emerging.go,** has been provided in the assignment package. It depends on a file, **cmap.go,** which you must provide in this assignment. Queries can be of the basic **GetCount** to retrie v ve the number of times a word has been read across all files so far, or using a more complex **Reduce** call.

Detailed guidelines are provided below. Please note that mutexes and similar synchronization primitives are NOT allowed.

Detailed Guidelines

The **emering.go** driver takes as command line arguments: the number of readers, the number of askers, an “ask file” (for the askers to draw words from), an “askdelay,” for the time between asks, a directory containing the files for the readers to use, and a “reducedelay” for the time between reduce calls. These flags are described in **emerging.go**. The details of how **emerging.go** works can be largely glanced over.

The reader goroutines will read individual words out of a file and send each word to the shared map using **AddWord** after it is read. The shared map must keep a count of how many times it has seen each word in a thread-safe manner. The asker goroutines will periodically ask the shared structure for the number of occurrences of a random word in the askfile using **GetCount**. The structure cannot add words at the same time it is searching for a word's count. The reducer goroutines will infrequently request a functional reduce from the shared structure. It will provide a function of type **ReduceFunc** when doing so.

The EmeringMap interface is defined as:

type EmergingMap interface {

Listen() // start listening for requests

Stop() // stop listening, exit goroutine

AddWord(word string) // increase count for given word

GetCount(word string) int // retrieve count for given word

// reduce over current words

Reduce(functor ReduceFunc, accum\_str string, accum\_int int) (string, int)

}

ReduceFunc is defined in **emerging.go** as:

type ReduceFunc func(key1 string, val1 int, key2 string, val2 int) (string, int)

Create your implementation in **package main**, in the file **cmap.go**. The structure will consist of several channels for asking, adding, and reducing, along with **map[string]int**. Each of these channels must be buffered to the constant buffer sizes defined in **emerging.go**. There will also likely need to be a channel for terminating the **Listen** mainloop. The functions associated with this structure should write requests into the channels, and the **Listen**() function will decided what action to take based on what channel has data in it using a **select** block. **Listen** should loop infinitely until being told to stop by the **Stop** function. Each **case** in **Listen**’s mainloop’s **select** should perform one task using the information it has read from the channel and possibly provide a response.

Your **cmap.go** file must include a function **NewChannelMap()** which returns a pointer to an object of your implemented structure.

You may add extra types as needed but you must not change the interface.

As part of your programming tasks, you are asked to add one more query to **emerging.go** using Reduce to find the word that has *appeared least frequently so far*. The steps to do this will be nearly identical to those that find the word of highest occurrence.

Output will be handled by **emerging.go**, you need not add any additional prints.

EVALUATE YOUR GO PROGRAMS

Perform the following tests:

1. Time the following runs of your program (along with the two reducers) with an askdelay of 10ms:
   1. 1 reader, 1 asker
   2. 16 readers, 2 askers
   3. 4 readers, 8 askers
   4. 16 readers, 32 askers
   5. 64 readers, 64 askers

An example of running with 2 readers, 2 askers, with askdelay of 10 (and other flags using defaults):

$ **./emerging -readers=2 -askers=2 -askdelay=10**

1. Describe the results and how your implementation scales with more readers and more askers. Briefly discuss your results.
2. Note down the type of CPU on the system (2-, 4-, 6-, 8-core, etc.). You are strongly encouraged to perform the experiments on machines with more cores.

**SUBMISSION INSTRUCTION**

* 1. Include your 2 go source files, together with extra file(s) describing the evaluations of these programs
* 2. Write a README file (text file, do not submit a .doc file) which contains
  + - You name(s) and email address(es). PLEASE list your team members if any.
    - Whether your code was tested, and if so, under what computer software/hardware setting.
    - Briefly describe anything special about your submission that the TA should take note of.
* 3. Place all files under one directory with a unique name (such as [userid]\_1 for assignment 1, e.g. davidl\_1).
* 4. Tar the contents of this directory using the following command.   
  **tar –cvf [directory\_name].tar [directory\_name]**   
  e.g. tar -cvf davidl\_1.tar davidl\_1/
* 5. Upload your tarred file to Blackboard under “Assignment 1”
* 6. Each team only needs to submit one copy. It does not matter which member on the team submits