**Assignment 2**

Due date: 11:59PM EST, 3/30/2015

This assignment focuses on Go programming and evaluation. The assignment demonstrates the use of Go channels for communication between threads of execution. If you have not gone through tour.golang.org, it is suggested you do so.

Go programs can often be highly parallelized without the need for explicit synchronization. Go channels provide a safe, simple, powerful means for goroutines to communicate with each other easily. The goal of this assignment is to parallelize reading data out of multiple files into a shared data structure using only channels for communication and safety, while, at the same time, running queries against the data that has been read out so far. Many running goroutines should be able to add words and run queries in parallel without corrupting any data. Queries can be of the basic **GetCount** to retrieve the number of times a word has been read across all files so far, or using a more complex **Reduce** call.

1. Implement the **EmergingMap** interface using only channels for communication
2. **Mutex**es and similar synchronization primitives are not allowed

NOTE:

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

**Write a Go shared map structure using only channels: cmap.go**

You are provided with the **emerging.go** driver file. You must create a **cmap.go** file with your implementation.

The **emering.go** driver takes as command line arguments:

(1) the number of readers,

(2) the number of askers,

(3) an “ask file” (for the askers to draw words from),

(4) an “reducedelay,” the delay in milliseconds for reducers,

(5) comma separated list of files to fill map with,

(6) an “askdelay”, the delay in milliseconds for askers.

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

go 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.

**Reduce to least frequent word: emerging.go**

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.

**PROGRAM OUTPUT FORMAT**

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. 4 readers, 2 askers
   3. 16 readers, 2 askers
   4. 2 readers, 4 askers
   5. 4 readers, 8 askers
   6. 16 readers, 32 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 -askfile=”ask.txt” -reducedelay=100 -infiles=”a.txt,b.txt”**

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. (Check with the CS admin for more information if you’d like to know more about the configurations of the CS machines.)
3. Briefly explain your findings

**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]\_2 for assignment 2, e.g. davidl\_2).
* 4. Tar the contents of this directory using the following command.   
  **tar –cvf [directory\_name].tar [directory\_name]**   
  e.g. tar -cvf davidl\_2.tar davidl\_2/
* 5. Upload your tarred file to Blackboard under “Assignment 2”
* 6. Each team only needs to submit one copy. It does not matter which member on the team submits