CS 343 Assignment 1 WordCount.go
Michelle Kim, Veronica Lin
Feb 18, 2023

Part 2

1. How many threads will you create? Does the number of threads need to be equal to the number of files?

We will create three threads in total (one thread per file), and the number of threads need to be equal to the number of files. We are doing this to *ensure mutual exclusion* among processes, so that they don't modify and disrupt the shared resources. For example, if we don't protect the resources for each thread using SafeCounter to lock each resource, the hash map "freq" that keeps track of the count of each word in this scenario will be accessed by all the threads.

The SafeCounter is a struct we customized to protect a field and to create resources. We used *mu sync.Mutex* to protect the v field, which is in this case *map[string]int* used to save the map between words and their count.

In the code below, we first initialize an empty map, and once the action is done, it allocates and returns the new map. This is used to count frequency of words in thread-safe manner to increase value and retrieve the count.

```
type SafeCounter struct { //custom datatype
   mu sync.Mutex
   v map[string]int
}

func (c *SafeCounter) Increase(key string) {
   //Method defined in Safecounter struct incrementing value with given key in map

   c.mu.Lock()
   defer c.mu.Unlock()
   c.v[key]++
}

func (c *SafeCounter) Value(key string) int {
   //Value() return value associated with a given key in the map
   c.mu.Lock()
   defer c.mu.Unlock()
   return c.v[key]
}

counter := SafeCounter{v: make(map[string]int)}
```

2. How to distribute the files over the different threads?

```
var wg sync.WaitGroup
for _, file := range files {
  wg.Add(1)
       readFile, err := ioutil.ReadFile("input/" + file) //return file object
       if err != nil {
           fmt.Println("Error reading File", err)
           return
       }

       go func(s string) {
           defer wg.Done()
           content := strings.ToLower(string(readFile))
           counting(content, &counter)
       }(file)
   }
   wg.Wait()
...
}
```

We are ensuring this by incrementing waitgroup everytime we read a file in files. This makes the ‘wg’ WaitGroup how many threads abd goroutines we are going to concurrently run. The “counter“ variable represents the thread that operates under SafeCounter and it will access specific content which is each file read from the files array. “counter” counts the frequency of words and updates the SafeCounter struct using the counting function.

3. How will you compile the output into a single file at the end of the computation?

We first created an array called “count” that will hold the words and their frequency as key-value pairs. Then we retrieved these pairs from each counter, which are the three threads. Since we are now writing to the same count, the threads’ result will all be put into “count”.

```
    count := make([]WordCount, 0, len(counter.v))

    for k, v := range counter.v {
        count = append(count, WordCount{k, v}) //k= string, v = int
    }
```

Then, we accessed each key and value from wordOutput to format and successfully wrote in a file “f” which is a final output text file we created in the beginning of the code.

```
for key, value := range wordOutput {
        final_string := fmt.Sprintf("%s %d \n", key, value)
        _, err := f.WriteString(final_string) // returns number of bytes written and
err
        if err != nil {
            log.Fatal(err)
            return
        }
    }
```

4. Do you need to use any data structure to ease the computation? If yes, was it a shared resource? How can you guarantee mutual exclusion?

The data structure we used was a hashmap to keep track of all the words in the input files and their occurrences. The hashmap was a shared resource because all three threads can access it, though each thread cannot look at others when one is accessing the hashmap and updating it.

We guaranteed this mutual exclusion by creating a Safecounter struct using “mu sync.Mutex,” in which we called lock() and unlock() to ensure that each process cannot edit the shared resource unless it has the “key” to do it, and protecting the shared resources from being accessed concurrently.

Part 3

1. Did you face any other challenges? How did you approach / solve them?

- We encountered a problem with cleaning the input text. In order to count all the words accurately, we needed to remove all the punctuation from the input files and count words in a string separated by space. To achieve this goal, we used the regexpr (regular expression) package, and we replaced all non-alphanumeric characters with an empty string, and eventually we ended up with a really long string with a count of 1. This happened because we replaced the undesired characters with an empty string instead of a space. We fixed it by changing it to a space and we were able to output the word count correctly.
- Another problem we faced was that in order to ensure mutual exclusion for the multi-threaded function, our initial idea was to create a for loop and make each piece of file run one by one so that they don't write at the same time. However, this slowed down the computation because we're not letting the threads run freely and grab the information they need at the same time. We researched this problem and went to the office hours. We eventually found a really helpful piece of code from one of the go exercises and used sync.Mutex to help us achieve mutual exclusion while threads are running at the same time.
- When dividing content between the threads, we initially gave each file to the threads. However, one of the threads received a very long file while the two others received a very short file. This was easier to implement, but very inefficient for the threads. We therefore decided to merge all the content first and then divide them into about equal amounts of chunks, with the number of chunks corresponding to the number of threads. Below graph is the final results we got from this new method of implementation.

2. Measure the running time within your code for part 2.

We can observe that between 3-6 number of threads, the threads efficiently divide the files. (marked by the red circle in the attached chart) With too many threads, however, it may be more inefficient because dividing content into many chunks may cost more time. For a more thorough comparison, we could run 5-10 trials each to average.

(Chart and graph on the next page)

1: Single-Threaded, Else: Multi-Threaded

Threads Num	Trial1	Trial2	Trial3	Average
1	0.268	0.409	0.213	0.297
2	0.178	0.328	0.184	0.23
3	0.171	0.167	0.184	0.174
4	0.193	0.188	0.215	0.199
5	0.162	0.180	0.173	0.172
6	0.267	0.177	0.190	0.19
7	0.213	0.171	0.206	0.297
8	0.183	0.172	0.288	0.214
9	0.280	0.176	0.237	0.231
10	0.228	0.212	0.202	0.214