Project 1

Write a GoLang multithreaded (goroutine, concurrency) applications to compute the average of the integers stored in a file.

Assignment

Write a GoLang multithreaded applications to compute the average of the integers stored in a file. Your program will take two **command-line input parameters**: M and fname. Here M is an integer and fname is the pathname (relative or absolute) to the input data file.

1. The format of the input data file is: a sequence of integers separated by white space, written in ASCII decimal text notation; you may assume that each integer is the ASCII representation of a 64-bit signed integer (and has ~20 digits).
2. Your program should spawn M workers threads and one coordinator thread. The main() of your program simply spawns the coordinator.
3. The workers and the coordinator are to be implemented as goroutines in the GoLang. Workers can communicate with the coordinator but do not communicate among themselves. Use GoLang channels for communication between the workers and the coordinators.
4. The coordinator partitions the data file in M equal-size contiguous fragments; each kth fragment will be given to the kth worker via a JSON message of the form that includes the datafile's filename and the start and end byte position of the kth fragment, eg "{datafile: fname, start: pos1 , end: pos2}" for a fragment with the bytes in the interval [pos1, pos2).
5. Each worker upon receiving its assignment via a JSON message it computes a partial sum and count which is the sum and count of all the integers which are fully contained within its assigned fragment. **It also identifies a prefix or suffix of its fragment that could be part of the two integers that may have been split between its two adjacent (neighboring) fragments.** Upon completion, the worker communicates its response to the coordinator via a JSON message with the **partial sum** and **count**, suffix, and prefix of its fragment, as well as it's fragments start and end eg: a worker whose assigned fragment that starts at 40, ends by 55, and contains "1224 5 8 10 678" will respond with the message "{psum: 23, pcount: 3, prefix: '1224 ', suffix: ' 678', start:40, end:55}".
6. The coordinator, upon receving a response from each worker, accumulates all the workers's partial sums and counts, as well as the sums and counts of the couple of integers in the concatenation of kth suffix and (k+1)th prefix (received by the the workers assigned the kth and (k+1)th fragments respectively. Upon receiving responses from all the workers, the coordinator prints and returns the average of the numbers in the datafile.

Your implementation should be correct and efficient by ensuring that

- the coordinator, upon completion, computes the correct sum of the integers in the datafile.
- there are no race conditions.
- has least amount of information (#bytes) communicated between the coordinator and the workers.
- least synchronization delay (elapsed time of syncrhonization between workers and coordinator).
- has least latency time for the coordinator (time difference between the last response received and the coordinator finishing).

- has least response time for the coordinator (time difference between the coordinator starting and the receiving its 1st response from a worker).
- has least elapsed time for the "slowest" worker (the time from when a worker starts and when it finishes).
- has least elapsed time for the coordinator (the time from when the coordinator starts and finishes).

In order to control for distortions (delays due other running processes and the filesystem), 'time' here refer to CPU time rather than wall time (real physical clock time).

Assignment HTML, PDF

Run

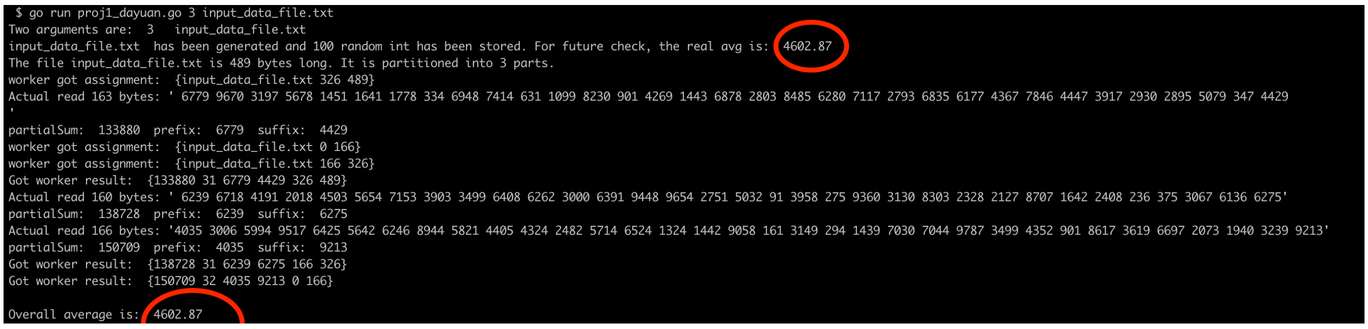
To run it, use below commands: (here *M=3, fname="input_data_file.txt"*)

```
cd ...path to.../proj1
go run proj1_dayuan.go 3 input_data_file.txt
```

To check out test:

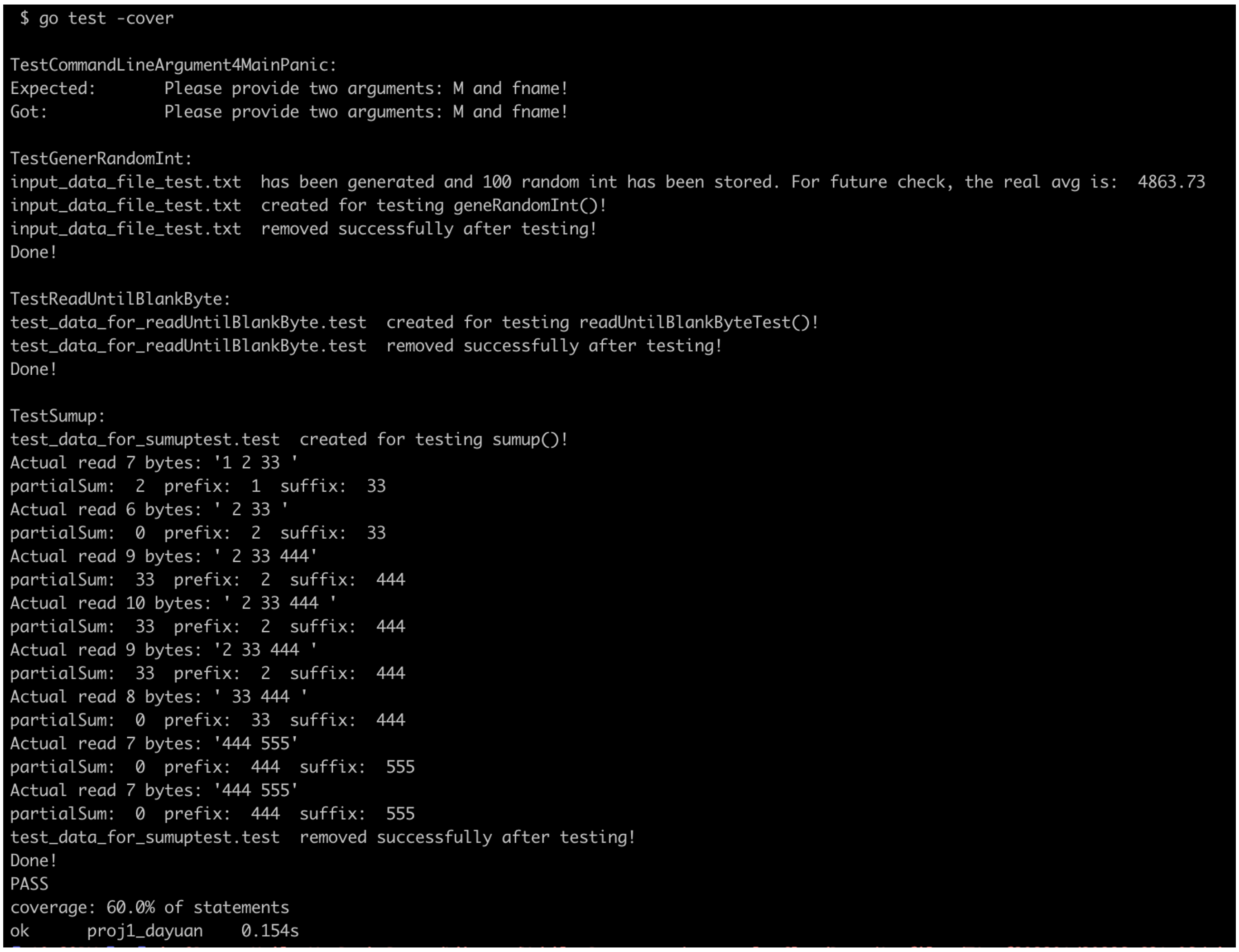
```
cd ...path to.../proj1
go test -cover
```

Screenshots:



```
 $ go run proj1_dayuan.go 3 input_data_file.txt
Two arguments are:  3   input_data_file.txt
input_data_file.txt  has been generated and 100 random int has been stored. For future check, the real avg is: 4602.87
The file input_data_file.txt is 489 bytes long. It is partitioned into 3 parts.
worker got assignment:  {input_data_file.txt 326 489}
Actual read 163 bytes: ' 6779 9670 3197 5678 1451 1641 1778 334 6948 7414 631 1099 8230 901 4269 1443 6878 2803 8485 6280 7117 2793 6835 6177 4367 7846 4447 3917 2930 2895 5079 347 4429
'
partialSum:  133880  prefix:  6779  suffix:  4429
worker got assignment:  {input_data_file.txt 0 166}
worker got assignment:  {input_data_file.txt 166 326}
Got worker result:  {133880 31 6779 4429 326 489}
Actual read 160 bytes: ' 6239 6718 4191 2018 4503 5654 7153 3903 3499 6408 6262 3000 6391 9448 9654 2751 5032 91 3958 275 9360 3130 8303 2328 2127 8707 1642 2408 236 375 3067 6136 6275'
partialSum:  138728  prefix:  6239  suffix:  6275
Actual read 166 bytes: '4035 3006 5994 9517 6425 5642 6246 8944 5821 4405 4324 2482 5714 6524 1324 1442 9058 161 3149 294 1439 7030 7044 9787 3499 4352 901 8617 3619 6697 2073 1940 3239 9213'
partialSum:  150709  prefix:  4035  suffix:  9213
Got worker result:  {138728 31 6239 6275 166 326}
Got worker result:  {150709 32 4035 9213 0 166}

Overall average is: 4602.87
```



```
 $ go test -cover

TestCommandLineArgument4MainPanic:
Expected:       Please provide two arguments: M and fname!
Got:            Please provide two arguments: M and fname!

TestGenerRandomInt:
input_data_file_test.txt  has been generated and 100 random int has been stored. For future check, the real avg is:  4863.73
input_data_file_test.txt  created for testing geneRandomInt()!
input_data_file_test.txt  removed successfully after testing!
Done!

TestReadUntilBlankByte:
test_data_for_readUntilBlankByte.test  created for testing readUntilBlankByteTest()!
test_data_for_readUntilBlankByte.test  removed successfully after testing!
Done!

TestSumup:
test_data_for_sumuptest.test  created for testing sumup()!
Actual read 7 bytes: '1 2 33 '
partialSum:  2  prefix:  1  suffix:  33
Actual read 6 bytes: ' 2 33 '
partialSum:  0  prefix:  2  suffix:  33
Actual read 9 bytes: ' 2 33 444'
partialSum:  33  prefix:  2  suffix:  444
Actual read 10 bytes: ' 2 33 444 '
partialSum:  33  prefix:  2  suffix:  444
Actual read 9 bytes: '2 33 444 '
partialSum:  33  prefix:  2  suffix:  444
Actual read 8 bytes: ' 33 444 '
partialSum:  0  prefix:  33  suffix:  444
Actual read 7 bytes: '444 555'
partialSum:  0  prefix:  444  suffix:  555
Actual read 7 bytes: '444 555'
partialSum:  0  prefix:  444  suffix:  555
test_data_for_sumuptest.test  removed successfully after testing!
Done!
PASS
coverage: 60.0% of statements
ok      proj1_dayuan    0.154s
```

Note

- position 0 refers to the first byte. So in the example of "1 2 33 444 555", for [0, 1) you need to read the first byte "1"; [1,5) gets " 2 3"; [2,6) gets "2 33"; [2,7) gets "2 33 ".

Reference:

- Golang command line argument https://stackoverflow.com/a/2707480
- main test https://stackoverflow.com/a/33723649
- write into file https://gobyexample.com/writing-files
- check whether a file exist https://stackoverflow.com/a/66405130
- go test -cover and -coverprofile https://stackoverflow.com/a/65454318
- concurrency vs parallelism https://stackoverflow.com/a/1050257
- concurrency vs parallelism vs asynchronization https://stackoverflow.com/questions/4844637/what-is-the-difference-between-concurrency-parallelism-and-asynchronous-methods
- concurrency vs multithreading https://stackoverflow.com/questions/35100102/what-is-the-differene-between-concurrency-and-multithreading
- concurrency in go (goroutine, channel, buffered channel) https://www.youtube.com/watch?v=LvgVSSpwND8
- Go routines (A tour of go) https://go.dev/tour/concurrency/1
- Channel range usage https://gist.github.com/DayuanTan/b01a67b1295d160e63c0a14b60c849bf
- to know how many bytes a file has https://stackoverflow.com/a/17133613

- Json in go: json.Marshal() json.Unmarshal() https://go.dev/blog/json