Міністерство освіти і науки України

Національний технічний університет «Дніпровська політехніка»

ЗВІТ
з лабораторної роботи №2
дисципліни «Розробка мікросервісних систем на мові Golang»

Виконав: ст. гр. 123-20ск-1

Гладкий Сергій Сергійович

Прийняв:
Реута О.В.

Дніпро
2022

Task 2

“CSV Concurrent Sorter” is a CLI application that allows sorting of its input presented as CSV-text.

Technical details

Using the “CSV Sorter” from the Task 1, extend it with the following required features:

1. The application has additional option **-d dir-name** that specifies a directory where it must read input files from. All files in the directory must have the same format. The output stays the same, it is a one file or a standard output with sorted content from all input files.
2. Processing must be implemented concurrently based on pipeline. The pipeline includes three stages:
   - Directory Reading:
     - read the directory content and all its subdirectories
     - send all found file names to a channel for the future processing
   - File Reading – for each file name received from the input channel
     - read the file from the disk
     - send its content line by line to the output channel.
   - Sorting:
     - read lines from the input channel
     - collect them into a buffer
     - when the input channel is exhausted, sort lines in the buffer
     - write the result to the output file (if option **-o** is set) or to the standard output
3. The application must print an error message, if two options **-d** and **-i** are set at the same time.
4. If option **-i** is set the application must read only the file defined by this option and then must use the pipeline to process file content.

Program code:

functions.go:

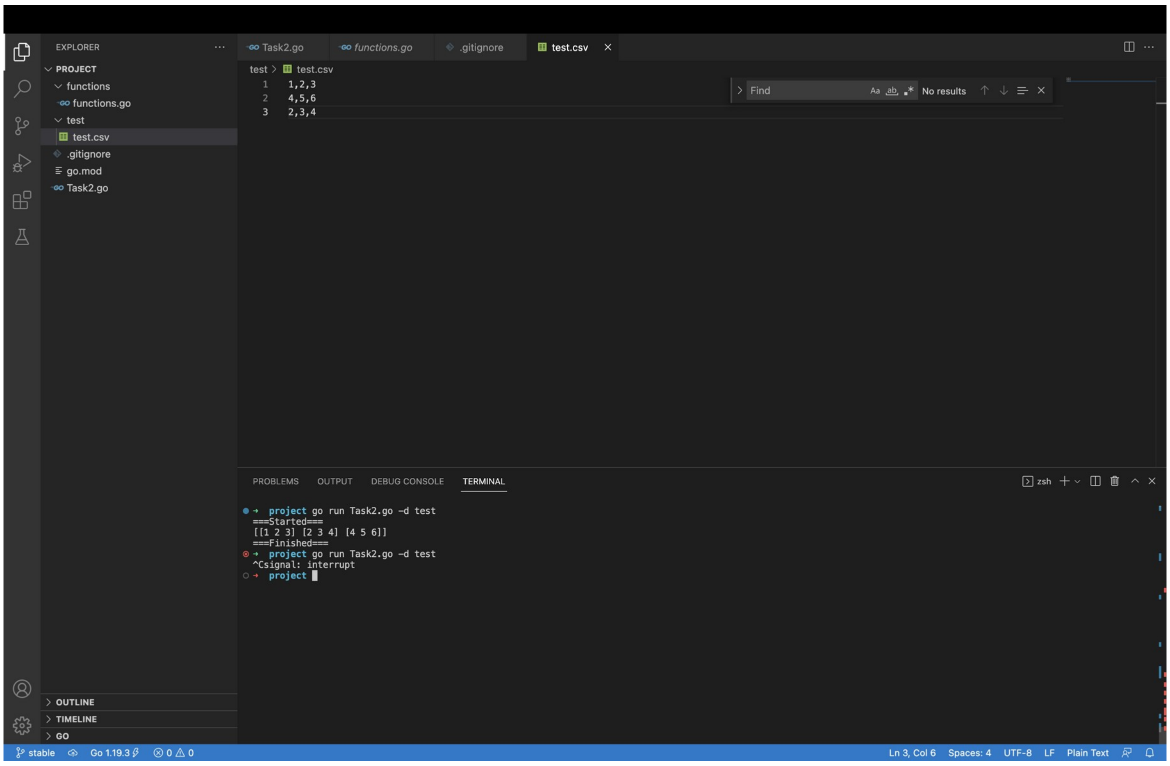
```
package functions

import (
   "bufio"
   "encoding/csv"
   "fmt"
```

```go
	"os"
	"sort"
	"strings"
)
func ReadCsvFile(filePath string) [][]string {
	f, err := os.Open(filePath)
	if err != nil {
		fmt.Println(err)
		os.Exit(1)
	}
	defer f.Close()
	content := [][]string{}
	csvReader := csv.NewReader(f)
	records, err := csvReader.Read()
	for records != nil {
		if err != nil {
			fmt.Println("", err)
			os.Exit(1)
		}
		content = append(content, records)
		records, err = csvReader.Read()
	}
	return content
}
func SortCsvData(content [][]string, ignoreHeader, reverse bool, field int) {
	if field > (len(content[0]) - 1) {
		fmt.Printf("Error: only %d column in this file.\n", len(content[0]))
		os.Exit(1)
	}
	if reverse {
		if ignoreHeader {
			sort.Slice(content[1:], func(i, j int) bool {
				return content[1:][i][field] > content[1:][j][field]
			})
		} else {
			sort.Slice(content, func(i, j int) bool {
				return content[i][field] > content[j][field]
			})
		}
	} else {
		if ignoreHeader {
			sort.Slice(content[1:], func(i, j int) bool {
				return content[1:][i][field] < content[1:][j][field]
			})
		} else {
			sort.Slice(content, func(i, j int) bool {
				return content[i][field] < content[j][field]
			})
		}
	}
}
func WriteCsvFile(name string, data [][]string) {
    file, err := os.Create(name)
    if err != nil {
        fmt.Println("Unable to create file:", err)
        os.Exit(1)
    }
    defer file.Close()
    csvWriter := csv.NewWriter(file)
    err = csvWriter.WriteAll(data)
    if err != nil {
        fmt.Println("Unable to create file:", err)
        os.Exit(1)
    }
    fmt.Println("File created.")
}
func WriteRecords() [][]string {
    s := bufio.NewScanner(os.Stdin)
    records := [][]string{}
    n := 0
    for s.Scan() {
        line := s.Text()
        if line == "" {
            break
        }
        row := strings.Split(line, ",")
        if n == 0 {
            n = len(row)
        }
        if n != len(row) {
            fmt.Printf("Error: row has %d column, but must have %d\n", len(row), n)
            os.Exit(1)
        }
        records = append(records, row)
    }
    return records
}
```

Task2.go:

```
package main

import (
    "flag"
    "fmt"
    "log"
    "os"
    "os/signal"
    "path/filepath"
    "sync"
    "syscall"
    "task2/functions"
)

func main() {
    var (
        inputFileName  = flag.String("i", "", "Use a file with the name file-name as an in-
put.")
        outputFileName = flag.String("o", "", "Use a file with the name file-name as an
output.")
        ignoreHeader   = flag.Bool("h", false, "The first line is a header that must be ig-
nored during sorting but included in the output.")
        sortingField   = flag.Int("f", 0, "Sort input lines by value number N.")
        reverseSort    = flag.Bool("r", false, "Sort input lines in reverse order.")
        dirName        = flag.String("d", "", "dir-name that specifies a directory where it
must read input files from.")
    )
    flag.Parse()
    fmt.Println("===Started===")
    done := make(chan struct{})
    ListenSignal(done)
    var (
        inputFileIsPresent      = inputFileName != nil && *inputFileName != ""
        outputFileNameIsPresent = outputFileName != nil && *outputFileName != ""
        dirNameIsPresent        = dirName != nil && *dirName != ""
    )

    if inputFileIsPresent && dirNameIsPresent {
        log.Fatal("The application must print an error message, if two options -d and -i
are set at the same time.")
    }

    if !dirNameIsPresent {
        var records [][]string
        if !inputFileIsPresent {
            records = functions.WriteRecords()
        } else {
            records = functions.ReadCsvFile(*inputFileName)
        }
        functions.SortCsvData(records, *ignoreHeader, *reverseSort, *sortingField)
        if !outputFileNameIsPresent {
            fmt.Println(records)
        } else {
            functions.WriteCsvFile(*outputFileName, records)
        }
    } else {
        // new logic
        fnChan := ReadDir(*dirName, done)
        contChan := FileReadingStage(fnChan, 1, done)
        result := SortContent(contChan, *ignoreHeader, *reverseSort, *sortingField,
done)
        records := make([][]string, 0, 1000)
        for i := range result {
            records = append(records, i)
        }
        if !outputFileNameIsPresent {
            fmt.Println(records)
        } else {
            functions.WriteCsvFile(*outputFileName, records)
        }
    }
    fmt.Println("===Finished===")
}

func ReadDir(dir string, done chan struct{}) (fnames chan string) {
    fnames = make(chan string)
    go func() {
        defer close(fnames)
        fileList := ScanDir(dir)
        for _, f := range fileList {
            select {
            case fnames <- f:
                {
                    continue
                }
            case <-done:
                {
                    break
                }
            }
        }
    }()
    return fnames
}

func FileReadingStage(fnames chan string, n int, done chan struct{}) (allLines chan []
[]string) {
    lines := make([]chan [][]string, n)
    allLines = make(chan [][]string)
    for i := 0; i < n; i++ {
        lines[i] = make(chan [][]string)
        ReadFiles(fnames, lines[i], done)
    }
    go func() {
        defer close(allLines)
        wg := &sync.WaitGroup{}
        for i := range lines {
            wg.Add(1)
            go func(ch chan [][]string) {
                defer wg.Done()
                for line := range ch {
                    select {
                    case allLines <- line:
                        {
                            continue
                        }
                    case <-done:
                        {
                            break
                    }
                }
            }
        }(lines[i])
    }
    wg.Wait()
    }()
    return allLines
}

func ReadFiles(fnames chan string, lines chan [][]string, done chan struct{}) {
    go func() {
        defer close(lines)
        for fname := range fnames {
            select {
            case lines <- functions.ReadCsvFile(fname):
                {
                    continue
                }
            case <-done:
                {
                    break
                }
            }
        }
    }()
}

func SortContent(cont chan [][]string, ignoreHeader, reverse bool, field int, done chan
struct{}) (result chan []string) {
    result = make(chan []string)
    go func() {
        defer close(result)
        var buffer = make([][]string, 0, 1000)
        for line := range cont {
            buffer = append(buffer, line...)
        }
        functions.SortCsvData(buffer, ignoreHeader, reverse, field)
        for _, line := range buffer {
            select {
            case result <- line:
                {
                    continue
                }
            case <-done:
                {
                    break
                }
            }
        }
    }()
    return result
```

```go
}

func ScanDir(path string) (files []string) {
    filepath.Walk(path, func(path string, info os.FileInfo, err error) error {
        if err != nil {
            log.Fatalf(err.Error())
        }
        if !info.IsDir() {
            files = append(files, path)
        }
        return nil
    })

    return files
}

func ListenSignal(done chan struct{}) {
    sigs := make(chan os.Signal, 1)
    signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)
    go func() {
        sig := <-sigs
        fmt.Println()
        fmt.Println(sig)
        close(done)
    }()
}
```



```
1,2,3
4,5,6
2,3,4
```

```
→  project go run Task2.go -d test
===Started===
[[1 2 3] [2 3 4] [4 5 6]]
===Finished===
→  project go run Task2.go -d test
^Csignal: interrupt
→  project
```

Picture 1 — Result

Link to github - https://github.com/GladkiySS/Golang.git