Concurrent Programming

Week 1 – Lab 1

Counting words

In this laboratory students will implement a series of programs that process files containing words. These files will have one word per line.

There are more optimized ways to implement the final program, but the way the resulting programs will get organized will help the study and development of additional new features in followup laboratories.

1 Exercise 1

Implement a program that opens the file containing words and stores every word into an array of strings:

```
char **word_array
```

Use the provided **lusiadas-words.txt** as input file of your program, that already contains one word per line.

After reading all the words into memory, the program should print in sequence the words, starting with each letter of the alphabet: first all words started with ‘a’, them all words started with ‘b’, … , and finally all words started with ‘z’.

Guarantee that words starting with **a** or **A** are considered as starting with the same letter.

2 Exercise 2

Develop a program that counts how many words start with each letter by following the next steps.

Develop a function (**`count_words`**) that receives an array of words, the total number of words in the array and a letter, and returns the number of words starting with such letter:

```
int count_words(char **word_array, int n_total_words, char letter)
```

After reading the whole words into the array, the program should print in the screen how many words start with each letter.

Guarantee that words starting with **a** or **A** are considered as starting with the same letter.

3 Exercise 3

Modify the previous program to print the number of occurrences of every unique word.

Guarantee that words' comparison is case insensitive:

Amor and **amor** should be considered the same word.

To implement this program, students will need to store in a linked list all unique words along with the number of times they appear on the array. To help implement this functionality define the **`word_info`** structure:

```
typedef struct{
      char *word;
      int count;
} word_info;
```

Implement a function (find_unique_words()) with the following arguments

- array of words
- length of array
- starting letter

The function should return a list of **`word_info`** structures containing all unique words starting with the given letter.

```
word_info *find_unique_words(char **word_array,
                             int n_total_words, char letter)
```

After printing the total number of words for each letter, print the corresponding number of unique words using the following function:

```
void print_unique_words_count(word_info *word_list, char c)
```

4 Exercise 4

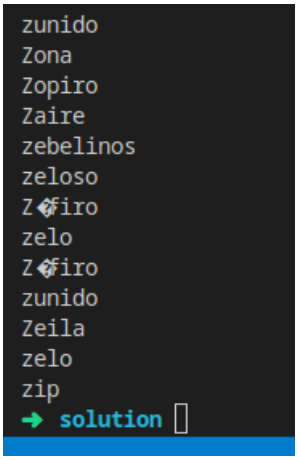
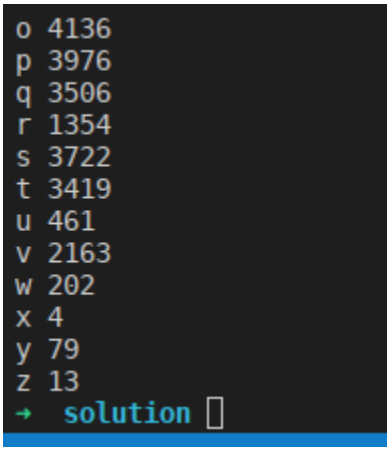
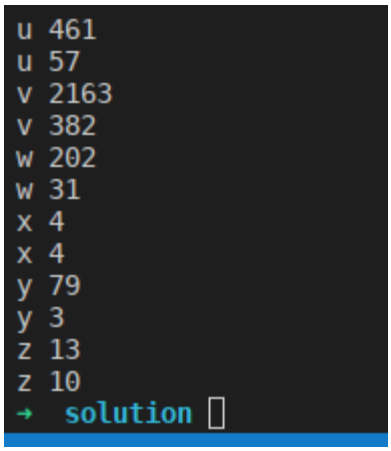
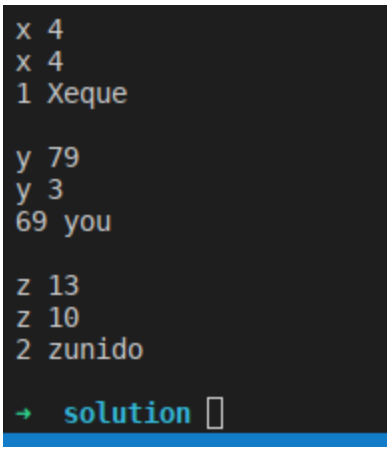
Modify exercise 3 to also print, for each letter, the word that is more frequent.

To do so Implement the following function:

```
word_info *more_freq_word(word_info *word_list)
```

If two or more words appear the same number of times, just print one of them.

5 Expected output

Exercise 1	Exercise 2
	
Exercise 3	**Exercise 4**
	

6 Application profiling

https://waterprogramming.wordpress.com/2017/06/08/

https://valgrind.org/docs/manual/cl-manual.html

https://kcachegrind.github.io/html/Home.html

Valgrind, besides verifying memory usage, can also be used to determine for how long each function in a program runs.

To determine what are those functions in the previous exercise, follow the next steps.

Execute the application in valgrid using the **callgrind** tool:

```
valgrind --tool=callgrind ./exercise-4
```

The output of valgrind does not give any hint on the functions execution times, but some of the generated auxiliary file will:

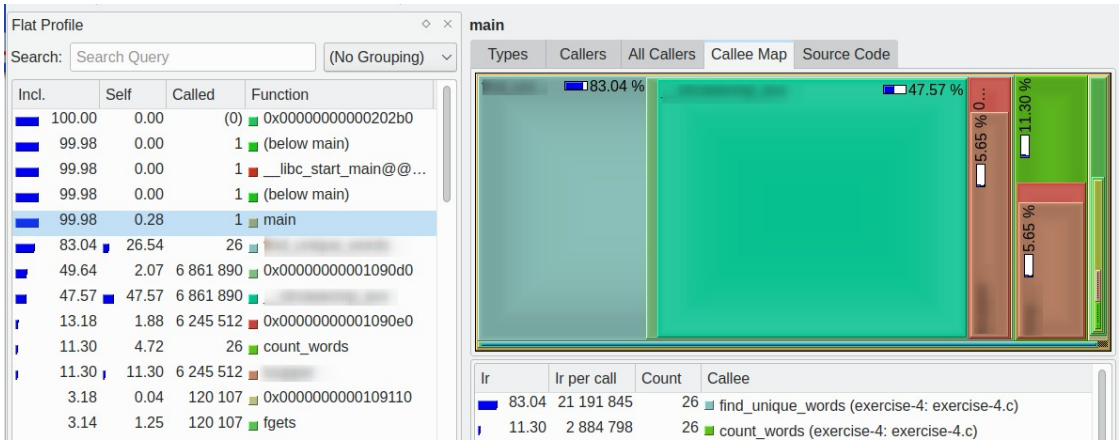
```
==1230261== I   refs:      663,530,440
➜ solution ls
callgrind.out.1230261  exercise-3.c
```

```
callgrind.out.xxxyyyzzz
```

This file should be loaded into the kcachegrind:

```
kcachegrind callgrind.out.1230261
```

Browse the various tabs in the kcachegrind to find the functions that takes more time during the execution



7

Exercise 5

Modify the solution of Exercise 4 to eliminate the possible functions that are called too many times.

The overall program organization and programmed functions should remain the same (as described in the assignments)