Lab Session 1: C language and Online Survey
24296 - Sistemes Operatius

1 Introduction

The objective of this lab session is to continue practicing C language. We are also going to learn how to include header and source files in your compilation. The provided header and source file provided (myutils.h and myutils.c) contain the function "read_split" that we are going to use and understand. We are also going to practice time benchmarking of our program (using functions also provided in myutils files). We will end 45 minutes before the end of the class so that you have time to do an online survey in Aula Global.

2 System call signatures, program main structure and includes

Here you have the system calls signatures for open, read, write and close and standard C lib helper functions:

```
int open(char* name, O_CREAT | O_RDWR, 0644);
int read(int fd, void *buf, int nbyte);
int read_split(int fd, char* buff, int maxlen, char* ch_end);
int write(int fd, void *buf, int nbyte);
void close(int fd);
printf("Integer var: %d and txt var: %s", int_var, txt);
sprintf(buff, "Integer var: %d and txt var: %s", int_var, txt);
sscanf(buf, "%d", &int_var);
```

The function read_split defined in myutils.c is very similar to the sys-call read: it reads from file descriptor (or standard input 0, fd) until a space or new line are found (or a maximum length, max_len, is reached), puts the result in buffer (char pointer buff) and returns how many bytes were read. It indicates also if the last character read (ch_end) was space or new line to be able to distinguish words in the same line or in the next one.

Also, to facilitate writing, formatting and conversion you can also use the functions printf, sprintf, and sscanf (of the standard C lib, stdio.h file). In order to use them you need to include header files and add the main block of a C program. If we want to include custom made functions we need to include the header file, like myutils.h using the local include rule (with "").

```
#include <unistd.h>       // Unix-like sys-calls read and write
#include <fcntl.h>        // Unix-like sys-calls open and close
#include <stdlib.h>       // Standard c lib : malloc, free
#include <stdio.h>        // Stabdard c lib : printf, sprintf, sscanf
#include "myutils.h"      // Custom local include header

int main(int argc, char* argv[]) {
  return 0;
}
```

Write that main empty program in the web interface repl.it (used in the precedent lab session) in a file called main.c. Then compile and generate an executable:

```
gcc main.c -o main
```

See how we use the benchmarking functions startTimer, endTimer suplied in myutils files (their parameter is an identifier of the timer, it can handle 100 timers):

```
  startTimer(0);
  sleep(1);
  printf("Time: %ld\n",endTimer(0));
```

Add the supplied myutils files into your working folder. Compile it including the code (the .c file) of the additional header file:

```
gcc main.c myutils.c -o main
```

See also a usage of read_split in which words are read from standard input until nothing is read:

```
char buff[80];
char ch_end;
while(read_split(0, buff, 80, &ch_end) > 0) {
        printf("read: %s\n", buff);
}
```

3 Program the following exercise steps:

1. Do a program that takes as first argument (argv[1]) a number that we need to convert to integer (you can use sscanf). Print that number as an integer using printf to the screen.

```
./main 50       // We must print the integer value (given as first param) in the screen
```

2. Then create an integer array of that size (with dynamic memory using malloc) and fill it with random numbers (from 0 to 99). You can use the following code for the random generator function including header file stdlib.h:

```
rand() %100
```

3. Then write the numbers in two files: (a) one binary called nums.dat and (b) another in text called nums.txt with the following steps:
4. Create a file using the call to open with the following flags (create it if it does not exist). Use the flags indicated in the begging of the doc
5. For generating file (a) you can write all numbers using only one single call to the sys-call write.
6. For generating file (b) you can use sprintf and write.
7. For file (a), check that everything went correctly by reading the binary file and printing the numbers to screen.
8. For file (b), you can open the file and use the command read_split supplied in myutils header and code files to read all numbers one by one.
9. Now benchmark the time taken and experiment using different numbers as parameters.