Lab Session 1: C language, read and read_split
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 saw in Seminar 1. We are going to practice time benchmarking of our program (using functions also provided in myutils). You will need to deliver a program bin2txt.c through 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);
int strlen(buf); // returns the length of string buf
```

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 (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 the standard input is closed (use control+C to close it) or a file finishes (you can pass a file to the standard input by redirecting the input like this: ./main < file.txt (inputs the content of file.txt to standard input 0).

```
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:

We are going to learn the differences of representing a number in text (a vector of its char digits) and in its integer format (its value) and how to convert from one another. We are going to learn what is a binary file (in which data is stored as its value) and a text file (in which data is stored as a sequence of character, a vector of char).

Open and understand the supplied program create_bin_file.c. Compile it (you will need to include the file myutils.c in the compilation) and use it correctly to create a binary file of 10 integers.

You can check using the command "cat" to show the content of the file; that the content is not shown correctly as it is a binary file.

1. create_bin_file.c is a program that first checks if arguments have been passed. Then converts the first argument (argv[1]) to an integer (using sscanf). As you can see argv[0] is the name of the executable that you created.
2. Then an integer array is created having as size the entered number (with dynamic memory using malloc) and the array is filled with random numbers (from 0 to 99) using the random generator function including header file stdlib.h: rand() % 100
3. A file is created with the flags indicated in the precedent section.
4. The file is filled with binary data with a single call to the sys-call "write", to write in one step all numbers.

4 Delivery exercise:

1. Do a program that you will call bin2txt.c that takes as first argument (argv[1]) the file that is going to be converted from binary to text file. Check that argv[1] exists.
2. We are going to check that the binary file of the precedent example was created correctly. Open the file using sys-call open.
3. You can now read the content of the file in two ways : (a) loop of calls to the sys call read or (b) using a single call to sys-call read (by declaring and allocating dynamic memory for an integer array). For the latter you need to know the size of the file, you can use the helper function in myutils.h get_file_size
4. Then write in a txt file called nums.txt all the numbers separated by spaces. You can use sprintf (to format the string adding a space) and then write using strlen to know the length of the string to be written.
5. You can check using the command "cat" to show the content of the file; the content is shown correctly as it is a text file.
6. Now open the generated file and use the command read_split supplied in myutils.h header and code files to read all numbers one by one.
7. Now benchmark the time taken and experiment using different numbers as parameters.