Laboratory Assignment 1: Introduction to Linux system programming

Table of contents

1 Objectives

- Become familiar with the development of C applications in the GNU/Linux environment.
- Review the basics of C.
- Become familiar with the use of getopt for option handling.
- Become familiar with basic shell and its programming.

files_p1.tar.gz contains the files for the realization of some of the exercises in this laboratory.

2 Resources

In order to complete the laboratory, the student must read and understand the following documents:

- Slides: "Review: Programming in C".
- Slides: "Introduction to Bash", which presents a brief introduction to the Bash shell.
- Document/guide: "C application development on GNU/Linux".
- C programming examples available on the virtual campus.

3 Exercises

Exercise 1

In the exercise 1 directory of this laboratory (files_p1.tar.gz) there is a series of subdirectories with short C codes.

For each of these codes, the tasks to do and/or a series of questions to be answered are included next. Go through all the codes and follow the instructions, compiling and running the codes to test them. We refer to the environment manual to learn how to use the compiler. Vscode can be used as an editor:

1. Compilation.
    - Compile the code and run it.
    - Subsequently, get the output of the preprocessing stage (option -E or option –save-temps to get the output of all intermediate stages) and the hello2.i file.
    - What happened to the call of min() in hello2.i?
    - What effect has the #include <stdio.h> directive?
2. Make.
    - Examine the makefile, identify defined variables, objectives and rules.
    - Execute make and check the commands it executes to build the project.
    - Mark the aux.c file as modified by running touch aux.c. Then run make again. How is it different from the first time you ran it? Why?
    - Run the make clean command. What happened? Notice that the target clean is marked as phony in the .PHONY: clean command. Why? To check it you can comment out that line in the makefile, compile again by doing make, and then create a file in the same directory called clean, using the touch clean command. Now run make clean, what happens?
    - Comment out the line LIBS = -lm by putting a hash mark (#) in front of it. Rebuild the project by running make (do a clean before if necessary). What happens? Which step is the one that gives problems?
3. Sizes.
    - main1.c
        - Compile the code, run it, and answer the questions.
        - Why does the first printf() print different values for ‘a’?
        - How large is a char data type?
        - Why does the value of ‘a’ change so much by incrementing it by 6?
        - If a ‘long’ and a ‘double’ occupy the same, why are there 2 different data types?
    - main2.c
        - Compile the code and try to run it. Answer the questions
        - Do we have a compilation problem or an execution problem?
        - Why is the problem occurring? Fix it, compile and run again.
        - a,b,c, and x are declared consecutively. Are their addresses consecutive?
        - What does the modifier “%lu” in printf() mean?
        - What address does “pc” point to, does it match any previously declared variable? If so, do the sizes of the two match?
        - Does the value of the size of “array1” match the number of elements in the array? Why?
        - Do the addresses pointed to by string1 match the address of string2?
        - Why are the sizes (according to sizeof()) of string1 and string2 different?
4. Arrays.
    - array1.c
        - Compile and execute the code. Answer the questions
        - Why is it not necessary to write “&list” to get the address of the array list?
        - What is stored in the address of “list”?
        - Why is it necessary to pass as argument the size of the array in the init_array function?
        - Why the size (sizeof()) of the array in the “init_array” function does not match the size declared in main()?

    - Why is it NOT necessary to pass as argument the size of the array in the function init_array2?
    - Does the sizeof() of the array in the “init_array2” function match the size declared in main()? -
- array2.c
    - Compile and execute the code. Answer the questions:
    - Does the array copy perform correctly? Why?
    - If it is not correct, write a code that does perform the copy correctly.
    - Uncomment the call to the “tmo” function in the main() function. Compile again and run.
    - Is the problem that occurs a compilation problem or an execution problem? Why does it occur?
    - Find a MAXVALID value greater than 4 that does not cause the problem. Is it writing beyond the size of the array? If so, why does the code work?

5. Pointers.

    - pointers1.c
        - Compile and execute the code. Answer the following questions:
        - What operand do we use to declare a variable as a pointer to another type?
        - What operand do we use to get the address of a variable?
        - What operand is used to access the contents of the address “pointed to” by a pointer?
        - There is an error in the code. Does it occur in compilation or in execution? Why does it occur?
    - pointers2.c
        - Compile and execute the code. Answer the following questions:
        - How many bytes are reserved in memory with the call to malloc()?
        - What is the address of the first and last byte of this reserved area?
        - Why is the content of the address pointed by “ptr” 7 and not 5 in the first printf()?
        - Why is the content of ptr[1] modified after the *ptr2=15; statement?
        - What are two different ways of writing the value 13 in the address corresponding to ptr[100]?
        - There is an error in the code, does it manifest itself in compilation or in execution? Even if it does not manifest itself, the error is there. What is the error?
    - pointers3.c
        - Compile and execute the code. Answer the following questions:
        - Why does the value of ptr[13] change after the assignment ptr = &c;?
        - The code has (at least) one error. Does it manifest itself in compilation or in execution? Why?
        - What happens to the area reserved by malloc() after the assignment ptr = &c;? How can it be accessed? How can this area be freed?

6. Functions.

    - arg1.c
        - Compile the code and run it.
        - Why is the value of xc not modified after the call to sumC? Where is that information modified?
        - Comment out the two forward statements of sum() and sumC(). Compile again, what happens?
    - arg2.c
        - Compile the code and run it.
        - Why does the value of ‘y’ change after the call to sum()?
        - Why is sometimes the ‘.’ operator used and other times ‘->’ to access the fields of a structure?
        - Why does the value of zc become incorrect without using it in the code again?
        - Correct the code to avoid the error in zc.

7. Strings.

    - strings1.c
        - Compile and run the code. Answer the following questions:
        - What is the address of the letter ‘B’ in the Bonjour string, and the address of the letter ‘j’?

  - After the assignment p=msg2;, how can we retrieve the address of the string “Bonjour”?
  - Why is the length of the strings p and msg2 2 after line 30? 3 bytes are assigned to ‘p’ which modifies both, but then the length is only 2.
  - Why does strlen() return a different value than sizeof()?
- strings2.c
  - Compile and run the code. Answer the following questions:
  - The “copy” code does not work. Why?
  - Now use the function copy2() (uncomment the corresponding line). Does the copy work?
  - Propose a correct implementation of the copy.
  - What does the “mod” function do? Why does it work?
  - Uncomment the last call to the “mod” function. Compile and execute. Why does the error occur?

Exercise 2

The program *primes* whose source code is shown below, has been developed to calculate the sum of the *n* first prime numbers. Unfortunately, the programmer has made some mistakes. Using the C debugger `gdb` the student should find and correct the errors. Compile directly on the command line: `gcc -g -w -o primes primes.c`.

```c
/**
 *  This program calculates the sum of the first n prime
 *  numbers. Optionally, it allows the user to provide as argument the
 *  value of n, which is 10 by default.
 */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/**
 * This function takes an array of integers and returns the sum of its n elements.
 */
int sum(int *arr, int n);

/**
 * This function fills an array with the first n prime numbers.
 */
void compute_primes(int* result, int n);

/**
 * This function returns 1 if the integer provided is a prime, 0 otherwise.
 */
int is_prime(int x);

int main(int argc, char **argv) {

  int n = 10; // by default the first 10 primes
  if(argc = 2) {
    atoi(argv[2]);
  }
  int* primes = (int*)malloc(n*sizeof(int));
  compute_primes(primes, n);

  int s = sum(primes, n);
  printf("The sum of the first %d primes is %d\n", n, s);

  free(primes);
  return 0;
}
```

```
int sum(int *arr, int n) {
  int i;
  int total;
  for(i=0; i<n; i++) {
    total =+ arr[i];
  }
  return total;
}

void compute_primes(int* result, int n) {
  int i = 0;
  int x = 2;
  while(i < n) {
    if(is_prime(x)) {
      result[i] = x;
      i++;
      x += 2;
    }
  }
  return;
}

int is_prime(int x) {
  if(x % 2 == 0) {
    return 0;
  }
  for(int i=3; i<x; i+=2) {
    if(x % i == 0) {
      return 0;
    }
  }
  return 1;
}
```

Exercise 3

In this exercise, we will work on the use of `getopt()` an essential tool for command line option processing. The objective of the exercise is to complete the code of the `getopt.c` file so that it is able to process the `-e` and `-l` options as indicated by the program usage, which can be queried with the `-h` option:

```
$ make
$ ./getopt -h
Usage: ./getopt [ options ] title

    options:
        -h: display this help message
        -e: print even numbers instead of odd  (default)
        -l length: length of the sequence to be printed
        title: name of the sequence to be printed
```

Once completed, the program should print a sequence of `length` odd numbers (10 by default; we can change it with the `-l` option) or even numbers if the `-e` option is included. The `-l length` and `-e` arguments are optional, but the `title` argument must always be present on the command line.

Examples of outputs for different input combinations:

```
$ ./getopt hola
Title: hola
1 3 5 7 9 11 13 15 17 19
```

```
./getopt -l 3 hola1
Title: hola1
1 3 5

./getopt -l 4 -e hola2
Title: hola2
2 4 6 8
```

It is necessary to become familiar with the `getopt()` function by consulting the `getopt()` manual page: `man 3 getopt`.

- `int getopt(int argc, char *const argv[], const char *optstring);`

The function is usually invoked from `main()`, and its first two parameters match the `argc` and `argv` arguments passed to `main()`. The `optstring` parameter serves to tell `getopt()` in a compact form which options the program accepts –each identified by a letter–and whether they accept mandatory or optional parameters.

The following considerations should be taken into account:

1. The `getopt()` function is used in combination with a loop, which invokes the function as many times as the user has passed options on the command line. Each time the function is invoked and finds an option, `getopt()` returns the character corresponding to that option. Therefore, inside the loop, the *switch-case* construct is usually used to carry out the processing of the different options. It is advisable not to carry out the processing of our program inside the loop, but only to process the options and to give value to variables/flags that will be used in the rest of our code to decide the behavior that it must have.
2. A particular aspect of the `getopt()` function is that it sets the value of different global variables upon invocation. The most relevant of which are the following:
   - `char* optarg`: stores the argument passed to the current recognized option, if it accepts arguments. If the option does not include an argument, then `optarg` is set to `NULL`.
   - `int optind`: represents the index of the next element in the `argv` (elements left unprocessed). It is often used to process additional program arguments that are not associated with any option. We will see an example of this in practice.

To complete the code, include the `-l` and `-e` options in the `getpopt()` call and complete the `switch-case` structure to modify the default values of the `options` variable. To read the numeric value associated with the `-l` option, you must use the `optarg` global variable, keeping in mind that this variable is a string (type `char *`) and yet we want to store the option as an integer (type `int`). See the use of the `strtol()` function in the manual (`man 3 strtol`) for how to perform this conversion.

Also, since the `title` argument will not be processed by `getopt()` (since it is not preceded by an `-l` style option mark), we must continue processing the input string after the `for` loop. To do this, the `optind` variable will be used along with `argv` to store the string value that will be the title of our sequence.

Complete the code and answer the following questions:

1. What string should you use as the third argument of `getopt()`?
2. What line of code do you use to read the `title` argument?
3. Test the program using the following command lines:

   ```
   $ ./getopt hola -l 3
   $ ./getopt -l 3 hola -e
   ```

   The behavior is inappropriate. Why?

Exercise 4

Study the code and operation of the program `show-passwd.c`, which reads the contents of the system file `/etc/passwd` and prints on the screen (or in another given file) the various entries of `/etc/passwd` –one per line–, as well as the various fields of each entry. The `/etc/passwd` file stores in plain text format essential information about system users, such as their numerical user or group identifier as well as the default shell program for each user. For more information about this file, please refer to its manual page: `man 5 passwd`.

The usage mode of the program can be found by invoking it with the -h option:

```
$ ./show-passwd -h
Usage: ./show-passwd [ -h | -v | -p | -o <output_file> ]
```

The `-v` and `-p` options allow you to configure the format in which the program prints the `/etc/passwd` information. The `-v` and `-p` options enable the `verbose` (default) or `pipe` mode respectively. The `-o` option, which accepts a mandatory argument, allows you to select a file for program output as an alternative to the standard output.

One of the main objectives of this exercise is to familiarize the student with three very useful functions used by the `show-passwd.c` program, whose manual page should be consulted:

- `int sscanf(const char *s, const char *format, ...);`

  Variant of `scanf()` that allows formatted reading from a character buffer passed as the first parameter (`s`). The function stores in program variables, passed as an argument after the format string, the result of converting the various `s` tokens from ASCII to binary.

- `char *strsep(char **stringp, const char *delim);`

  It allows to split a character string into *tokens*, providing as a second parameter the delimiting string of those tokens. As can be seen in the `show-passwd.c` program, this function is used to extract the various fields stored on each line of the `/etc/passwd` file, which are separated by `":"`. The `strsep()` function is typically used in a loop, which stops as soon as the returned token is NULL. The first argument of the function is a pointer by reference. Before starting the loop, `*stringp` must point to the beginning of the string we wish to process. When `strsep()` returns, `*stringp` points to the rest of the string that remains to be processed.

Answer the following questions:

1. The `passwd_entry_t` data type (structure defined in `defs.h`) is used to represent each entry in the `/etc/passwd` file. Note that many of the fields store character strings defined as character arrays of prefixed maximum length, or by the `char*` data type. The `parse_passwd()` function, defined in `show-passwd.c` is responsible for initializing the various fields of the structure. What is the purpose of the `clone_string()` function used to initialize some of the aforementioned string-like fields? Why is it not possible in some cases to simply copy the string via `strcpy()` or by performing a `field=existing_string;` assignment? Justify your answer.
2. The `strsep()` function, used in `parse_passwd()`, modifies the string to be split into tokens. What kind of modifications does the string (variable `line`) undergo after successive invocations of `strsep()`? **Hint**: Look up the value and addresses of the program variables using a C debugger such as `gdb`.

Make the following modifications to the `show-passwd.c` program:

- Add the `-i <inputfile>` option to specify an alternate path for the `passwd` file. Make a copy of `/etc/passwd` in another location to verify the correct operation of this new option.
- Implement a new `-c` option in the program, which allows displaying the fields in each `passwd` entry as comma-separated values (CSV) instead of `":"`.

Exercise 5

In this exercise we are going to practice programming in bash making use of the internal *read* command (see help for the read function) to process files line by line:

```
read [-ers] [-a array] [-d delim] [-i text] [-n nchars] [-N nchars] [-p prompt...]
```

This command reads a line of standard input, breaks it into words, and assigns the first word to the first variable in the list of names, the second to the second variable, and so on.

If we want to add a special delimiter to separate words we can do this by assigning a value to the IFS variable before using the read operation. For example, to read words separated by ':' we would use the form:

```
while IFS=':' read var1 var2 ... ;
do
    # anything with $var1, $var2
done
```

And if we don't want to read from the standard input, we can redirect the input from the the whole loop to a file:

```
while IFS=':' read var1 var2 ... ;
do
    # anything with $var1, $var2
done < fichero
```

Use read to create a small script that does the same as the previous program 'show-passwd' (with its default options), i.e.:

- read the file /etc/passwd
- parse its entries made up of lines with words separated by ':'
- show each entry by the standard output with the same format as the 'show-passwd' program

For bash-formatted output see the -e option of echo (man echo). Alternatively the printf utility (man 1 printf) can be used.

Once this is done, modify the script to show only those entries in the /etc/passwd file where the user's home is a subdirectory of /home. To do this, it is useful to use the command *dirname* (man dirname) and the use of the if control flow structure in conjunction with *[*.

Finally, try to get a bash command, combining the cut and grep commands, to obtain from the /etc/passwd file all the homes starting with /home. We refer to the manual pages for cut and grep and review the use of pipes ('|') to combine shell commands.