Lab Session 0: Linux Shell, C Tutorial
Sistemes Operatius

1 Introduction

The Operating Systems course is done in C Language and Linux. As most probably we are going to have sessions in-person we also provide an alternative if you don't have easy access to a Linux system (available in UPF) there is an IDE: `https://repl.it/languages/c`. Without creating a user you can run and edit the main.c program and have access to a bash shell console of an Ubuntu Linux distribution. If you create a user in the website you can in addition: edit other files, create folders, and save ; so better create one.

The objective of this lab session is to: (1) Learn the Linux Shell commands: ls, which, cat, echo, sleep, wc, ps, chmod, (2) Test a minimal shell script, (3) Review basic C programming, (4) Do the exercises focused on the understanding of memory addressing and handling in C. This lab session will be evaluated with an online survey to be done individually before class next week.

2 Bash Shell

Bash shell commands are programs that interface the SO by making system calls. Open the shell console `https://repl.it/languages/c`. Try the following commands that you will need to know understand and know for the OS course:

```
ls -la                  // list all files including permissions
which ls                // folder information
cat main.c              // print file content
echo $PATH              // print var or string content
sleep 5                 // wait in seconds
wc -l main.c            // counts lines in arg or standard input
chmod u+w main.c        // modify file permissions
```

There are many other commands with many other options (use –help for argument options). We will restrict to this ones that you will need to know. The shell prompt only executes commands that are in certain folders indicated in the so-called PATH system variable. Try for example the command "ls" which lists the files in a folder. "ls" is a program and you can know where it is by using the command "which ls". The usual location is in the main folder "bin" present

in every Unix like system. ("bin" standing for "binary", that is, executable"). The Linux file system starts at "/". The folder "bin" is in the root of the file system in "/bin".

```
$which ls
/bin/ls
```

2.1 Standard Input/Output and Redirection

Programs when executed in Unix-like systems have a standard input channel, a standard output channel. Usually all have their input assigned to the keyboard and their output to the screen (as an example "cat main.c" sends the content of the text file to the standard output, by default, the screen) but you can change that. In the command line there are two ways of changing the input output channels:(1) redirection to a file "Program > file_name" and (2) using pipes "Program1 | Program2". In the first case we redirect the standard output to a file, for example writing the files of a folder in a file "ls > files.txt". In the second case we redirect the standard output of Program1 to the standard input of Program2. Some examples:

```
ls > files.txt
ls | wc -l
echo "This is some text" | cat     // same as cat "This...
```

2.2 Creation of a Script Shell

We can create a text file with commands and execute it as a script shell.

```
>echo "ls" > myscript     // create txt file using redirection
>./myscript               // try to execute it
bash: ./myscript: Permission denied
```

In Linux files have permissions to indicate read (r), write (w) and exec (x) privileges of each file and folder for all types of users. Use "ls -la" to see the detailed info of every file. The first character that is shown for every file indicates if it is a folder (directory, "d", or not "-"). The three following characters indicate the permissions for the user and owner of the file (r,w,x).

```
>chmod u+x myscript    // also chmod 700 script
>./myscript
```

2.3 Execution in the background or concurrent execution

Experiment what happens when you run the command "sleep 5" with the & character at the end. The & character tells the shell to execute it concurrently and return immediately to shell console waiting for keyboard input.

```
>sleep 5 &
```

3 C Tutorial

The IDE `https://repl.it/languages/c` has the "Hello World" program:

```
#include <stdio.h>
int main(void) {
  printf("Hello World\n");
  return 0;
}
```

Use the gnu compiler gcc to compile it and execute it afterwards:

```
gcc main.c -o hello
./hello
```

3.1 Variable Declarations

Examine and understand the following C declarations:

```
char c = 'a';                          // a character
int num = -123;                        // an int variable initialized to -123
long l = -1000;                        // a big integer
unsigned int unsigned_num = 123;       // positive integer
unsigned long unsigned_l = 1000;       // positive long

float f = 3.45;                 // a floating point variable
double d = 3.45;                // a double variable
char* pointer_str = "hello";    // a char pointer declaration and inicialization
char array_str[] = "hello";     // an array declaration and inicialization

int* pointerToInt = malloc(5*sizeof(int));    // allocate five ints
void* pointerToVoid = (void*) pointerToInt;   // conversion: int to void pointer
int* pointerToInt = (int*) pointerToVoid;     // conversion: void to int pointer
```

Void type is like a joker type of maximum size. It is used to define pointers of any type. And for conversion from a pointer type to void pointer and viceversa you need to apply casting (conversion).

3.2 Input/output and conversion from text to int

Standard C lib file stdio.h defines printf, sprintf, scanf, sscanf functions:

```
int int_var = 10;
char*  txt = "20";
chr buff[80];
printf("Integer var: %d and txt var: %s", int_var, txt);
sprintf(buff, "Integer var: %d and txt var: %s", int_var, txt);
scanf("%d", &int_var);
sscanf(txt, "%d", &int_var);
```

Use **printf** to write to the standard output (the screen). Use **sprintf** to write to the specified char array buffer. Use **scanf** to read from standard input (the keyboard) and store the result into the corresponding variable. If variables are basic types like int, we need to give as parameter the direction of the variable, in this case &int_var (passing the address the variables can be updated. Use

sscanf to convert from string to integer: given an input character sequence and a format, extract the values into variables.

```
%d    for integer variables
%f    for printing float and double values
%p    for printing memory adress values (pointers)
%u    for printing unsigned integers
%lu   for printing long unsigned integers
%s    for char arrays / strings
```

Practice: Now add in the main.c program all declarations of previous section 3.1 and print each of them in the screen using printf. For the character pointer and pointer array print both the content and the pointers.

3.3 Variable Size

Use the function sizeof that returns the size in bytes of a variable (it returns an unsigned long as previously seen as %lu) to print in the screen the size of all precedent variable declarations.

Sizeof also accepts data types as input. Find out (printing on the screen) the size of all basic types in C: char, int, float, double, unsigned int, long, char*, int*, double*, long*, int**. Use sizeof like in one of the precedent programs.

3.4 Difference between pointers and arrays

These following declarations that we tested are not equivalent in C:

```
char array_str[] = "hello";
char* pointer_str = "hello";
```

The first one corresponds to an array of chars initialized to the char sequence "Hello", the second one corresponds to a char pointer from which we allocate 6 contiguous char positions filled with "Hello" plus the end character '\0'. Also be aware that this second initialization is an unmodifiable constant char sequence. To allocate modifiable space you should use malloc.

Discuss and understand the different results that you got for printing:

```
sizeof(pointer_str)
sizeof(array_str)
```

3.5 Memory Access

The * symbol can be used in two different ways in C language. When included in a declaration it indicates that is a pointer to a variable type. It can also be used to access the memory content indicated of a pointer (the content the memory address indicated by the pointer).

```
int* pi;  // we declare a pointer to an int (memory address)
          // *pi is now (without initialization) an invalid mem access
```

The **ampersand & operator** returns the memory address of its only operand. The **star * operator** returns the memory content of its only operand.

```
int anInt;
int* pint = &anInt;

anInt = 5;      // these two are equivalent operations, in both cases
*pint = 5;      // 4 bytes in memory are overwritten with a 5
```

The type of the pointer matters, because it indicates how many bytes of memory to overwrite.

3.6 Memory Allocation

The malloc function tells to the Operating System to allocate memory and returns a pointer to the reserved contiguous allocated memory portion:

```
int* pint = malloc(5*sizeof(int)); // allocate five ints

pint[0] = 10;           // these two are equivalent
*pint = 10;

pint[1] = 11;           // these two are equivalent
*(pint+1) = 11;         // +1 sums 4 bytes to the pointer  (size of an int)

int anInt = pint[1];        // these two are equivalent
int anInt = *(pint+1);

pint = &(*pint)         // this returns the same pointer unchanged

free(pint);             // from now pint[0] is an incorrect mem access
```

In the precedent code &anInt is a different memory address than &pint[1], it is the memory address of the integer variable (stored in another memory position). Remember:

- Understand the two usages of the * star/asterisk operator. A variable declared as a pointer to some type (int* pint), is a variable container which can contain a value that is going to be interpreted as an hexadecimal address to memory. The star when preceding the variable (*pint), refers to the content of the memory position referenced by the variable value.
- Always use the same type of pointer as the variables it examines: floats for floats, ints for ints, and so on (size matters!).
- Initialize a pointer before you use it! Set the pointer equal to the address of some variable in memory or use malloc (or new in c++).

3.7 Arrays to pointers

To obtain a char pointer equivalent variable of an array (pointing to the first element of the array) we have to do the following: access position 0 of the array and ask for its memory address using &:

```
char* pa = &a[0];
```

3.8 Double pointers

C Language allows also the declaration of a pointer to a pointer. This declaration is most useful to define two dimensional arrays (matrices).

```
int** mat;
```

The first pointer indicates the array of pointers of every row (or column) of the matrix. To initialize an n, m matrix, we need first to allocate space for the pointers to every row, and then we need to allocate each row.

```
int** mat = malloc( n * sizeof(int*) );
for(i=0;i<n;i++) {
 mat[i] = malloc( m * sizeof(int) );
}
```

And we can use the double bracket notation to access each element: mat[0][0]. Discuss how costly is to interchange two rows of your matrix.

3.9 Parameters to our program

The main function accepts the parameters argc and argv:

```
int main(int argc, char* argv[]) {
```

Argc is the number of parameters that has been passed to our program execution command line. Argv contains the string content of those parameters. Argv is like a two dimensional array: its declaration could be also written like a double pointer (considering the mentioned subtle differences between pointers an arrays). Argv uses the notation of two closed empty brackets to indicate that is an unspecified array length. Thus argv is an array of pointers to characters (strings in C language).

4 Exercises of the C tutorial

4.1 Initialize a dynamic memory array

Consider the following declarations:

```
int* pa = malloc(10*sizeof(int));
char* pch = malloc(10);
```

1. Write the program to initialize all the content of the allocated memory positions (pointed by "pa") with the first 10 odd integers starting from 1.
2. Then print the content of the array to the screen indicating: position of the array, memory address (remember is hexadecimal), and memory content.
3. Now use the explained two alternative methods to fill every memory position. You probably used the brackets access to a memory position, now use only the start operator (*).
4. Now initialize the content of the allocated memory positions (pointed by "pch") with the first 10 character letters.
5. Then print the content of the portion of memory pointed by pch as we previously did indicating: position of the array, memory address, and memory content.

4.2 A function to print memory content

Define a function to print memory that we will call "print_mem" with the signature above and use it to print the memory content of both arrays defined in previous section.

```
void print_mem(void* pointer, char* type, int number);
```

We use as first parameter to pass the pointer to the memory position to start printing. We use the second parameter to pass a string (char array in C) indicating the type of the memory use only ("int" and "char"). We use the last parameter to indicate how many elements to print.

If you want to correctly compare strings of each type you can use the function strcmp defined in standard C lib file string.h (shown below). Careful: returns 0 if both are equal. Also remember you will need to apply casting (conversion) when converting to void pointers and viceversa.

```
int strcmp (const char* str1, const char* str2);
```

4.3 Allocating arrays and matrices

Compile and execute the programs "memLeakInvalidAccess.c" and "memLeakMatrix.c". In `https://repl.it/languages/c` if you want to upload files you will need to create a user (but you can also copy the file content to main.c). Now use the debugger profiler tool "valgrind" and fix the errors of the two files:

```
>valgrind ./main        // valgrind argument is a program executable
```