Operating Systems [2025-2026]

Assignment 01 – C Programming in Linux

Introduction

The *C* Programming Language holds a small and simple grammar. *C* is well known and widely used for operating systems programming. It provides a straightforward access to system level functions (system calls), allows the manipulation of data at bit level (low level programming), and the implicit usage of pointers to the system's memory. *C* also provides an unmanaged access to the system resources, thus being able to produce highly efficient programs. But, on the other hand, this means that *C* is not a strongly typed language offering weak error detection and handling mechanisms. It is common to see programmers introducing fatal execution errors into their programs simply by not performing resource allocation correctly or/and by implementing inconsistent resource sharing and access rules.

Objectives

Students concluding this work successfully should be able to:

- Use all the necessary tools and commands to write, compile, debug and execute C programs in Linux systems
- Use memory pointers
- Read and write data to files
- Define, create, and use lists and data structures
- Use `make` and `makefiles`

Support Material

- Tutorial – "GNU debugger" – in the course materials
- Support documentation – "C and makefiles" – in the course materials
- K. A. Robbins, S. Robbins, "Unix Systems Programming: Communication, Concurrency, and Threads", Prentice Hall:
    - Chapter 4 – Unix I/O
    - Annex A – Unix Fundamentals

- Brian W. Kernighan, and Dennis M. Ritchie, “The C Programming Language, 2nd Edition (Ansi C)”, Prentice Hall:
    - Chapter 5 – Pointers and arrays
    - Chapter 7 – I/O
    - Chapter 8 – The Unix System Interface
- “Programming in C and Unix”, available at:
  http://gd.tuwien.ac.at/languages/c/programming-dmarshall/
    - C/C++ Program Compilation
    - Pointers
    - Dynamic Memory Allocation and Dynamic Structures
    - Advanced Pointer Topics
    - Input and Output *(I/O)*: *stdio.h*
- Splint
  http://www.splint.org
- “GDB: The GNU Project Debugger” available at:
  https://www.gnu.org/software/gdb/documentation/
- Guide to Faster, Less Frustrating Debugging” available at:
  http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html

Basic Shell commands

- **man** *[section]* keyword # Unix online manual
- **ls** *[-a] [wildcard]* # list directory contents
- **pwd** # return working directory name
- **cd** *directory* # change directory
- **mkdir** *directory* # make directory
- **rmdir** *directory* # remove directory
- **cp** *file1 file2* # copy file(s)
- **mv** *file1 file2* # move file(s)
- **rm** *file* # remove file(s)
- **cat** *file* # concatenate and print files to stdout
- **grep** *"string" file* # print lines from a file that match a pattern
- **more** *file* # print files to stdout page by page
- **ps** *[-a]* # process status
- **kill** *{-SIGNAL} pid* # terminate a process (by using SIGNAL = 9 -> SIGKILL)
- **diff** *file1 file2* # compare two files
- **sort** *file* # sort a file
- **who** or **w** # display who is on the system
- **pico** *file* # file editor
- **vi** *file* # file editor
- **emacs** *file* # file editor

Program compilation

GNU C Compiler will be used for C compilation.

Usage:

```
gcc –Wall file1.c file2.c (...) –o executable
```

Debugging C in Linux using gdb (GNU debugger)

Compiling with debug information:

E.g. `gcc –Wall –g prog1.c –o prog1`

Starting gdb:

E.g. `gdb ./prog1`

Exercises

Note: Only some of the exercises provided in this assignment will be done during the practical classes. The extra exercises should be done by the student as homework and any questions about them should be clarified with the teacher.

1. Debugging a program

In this exercise a program with errors will be debugged. To help you, the errors in the code are already indicated as comments in the source file provided (*odd_numbers.c*). Following the steps below, use the debugger to discover why they happen.

1) Open and analyze the file provided. The program gets a set of numbers from the command line and detects which are odd.

   Example of the expected result:

```
user@ubuntu:~/SO$ ./test 4 3 5 8 2
Number of args# = 6
4
3
5
8
2
Odd numbers:
3 5
```

2) Write a *makefile* to compile the program and create an executable called `test`.
3) The execution of the program has two major flaws, which become visible as the number of arguments increase. Find the errors, understand why they occur with the help of *gdb* and solve them.

Notes:

a) Make sure to previously compile the program with the *–g* flag.
b) Run *gdb* as `gdb ./test`
c) Error #1 - Segmentation fault
   i) Compile the program and test it with 2 arguments (e.g. `./test 2 1`).
   ii) To understand why the error occurs, run the program in *gdb* with the same 2 arguments (e.g. `run 2 1`).
   iii) Use `bt` command to display a back trace and discover where the error occurred. Each line with a # represents a different call.
   iv) Analyze the value of the variable *i* (`print i`). If this variable does not exist in the context, change the context using the command `frame` ; the different contexts correspond to the different # previously presented by `bt`.
   v) With the data obtained from *gdb* explain the reason of the segmentation fault
   vi) Fix Error #1 in the code and compile the new version using the makefile.

d) Error #2 – this error may result in different behavior depending on the total number and specific values of the arguments used.
   i) Create a breakpoint in the beginning of *main* function;
   ii) Run the program with 3 parameters and analyze step by step (using `step` or `next`) the changes in *i*, *numbers* and *is_odd*. If you want to repeat the display of the same variables at each stop, use command `display`. You may analyze memory using `x` command (e.g. `x /16x 0xbffff1c4`). Use `&` to get the address of a variable (e.g. `print &i`).
   iii) Fix the problem and compile the new version using the makefile.

2. Quick Sort

Some of the most common problems in computer science are related with the sorting of data. There are dozens of algorithms developed to support this task.

Quicksort is one example of these algorithms, which when correctly implemented, is one of the fastest sorting algorithms available and a fantastic example of recursion. Quicksort splits the array and sorts the two resulting smaller arrays to reduce the complexity of the task. An example of a quicksort function (with some errors) is provided in *quicksort.c* .

Start by analyzing the provided code, compile it, debug it if needed, solve the existing problems and run it. Test the provided algorithm with arrays of different sizes and compare the results of this quicksort function to the system *qsort()* (compare results and performance using command *time*).

3. Using a makefile

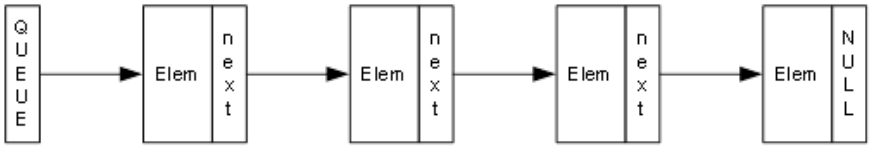
A palindrome is a sequence of characters (for example, a word) that reads the same backwards and forwards. Examples of common English words that are also a palindrome can be "civic", "radar" or "madam".

In this exercise, you will find a software program that receives a word or number from the console and prints whether it is a palindrome or not. This software program is composed by two different C source files (*palindrome.c* and *aux_file.c*) and a C header file (*aux_file.h*).

1) To facilitate the compilation of this application, create a makefile to automate this step. The makefile must generate an executable named "palindrome". (Note: Do not forget to use the `-Wall` flag when compiling).
2) Adapt your makefile to have a macro that you can use to easily change the compilation flags. Using this macro, add the "-g" flag (debug flag) to the list of flags that will be used during compilation.
3) Add a target "clean" that removes the executable and the intermediate object files that are generated during compilation.

4. Telephone List

Develop an application to manage a telephone list (FIFO queue) that interfaces with the user through a CLI (Command Line Interface) menu.



The following functionalities need to be implemented:

- Create a new queue
- Add one element to the queue
- Remove one element from the queue
- Print the queue
- Write the queue to a file
- Read a queue from a file

5. Searching lists using indexes

The objective of this assignment is to implement the functionality necessary to execute the code inside the `main()` function of the provided file *index_searches.c*. In this code it is possible to observe a list of function prototypes that must be implemented to get the program working.

Read a list of products from a text file and create a double linked list sorted by product name. Next, create an index that will allow us to speed up searches on the main product list. Each element on this index holds a pointer to the first element appearing on the product list with a different initial letter. Using the data on this index, only the elements on the main list with the same initial letter will be compared with the search string. When a product is located, the user is given the option to delete it from the list.

Notes:

- The string comparisons on the program are not case sensitive
- Each list must have a head element that serves only as a list header and does not hold any valid product info
- Your code should be as efficient as possible, have a small memory footprint and avoid unnecessary computations
- Implement all the necessary error detection and correction code