LAB\_A

**Introduction to C Programming in a Unix (Linux 32 bits) Environment**

**This lab assignment is to be done SOLO!**

**Home-Lab A -- Assignment goals:**

* C primer
* Parsing command-line arguments
* Understanding character encoding (ASCII)
* Implementing a debug mode for your program
* Introduction to standard streams (stdin, stdout, stderr)
* Simple stream IO library functions

**Preparation - Part 0: Maintaining a project using make**

You should perform this part **before** starting to implement the hand-in assignment. It is basic knowledge that is not checked, but needed to complete the assignment.

For this part, 3 files are provided: **add.s**, **main.c**, **numbers.c**. The first file is assembly language code, and the other 2 are C source code.

1. Log in to Linux.
2. Decide on an ASCII text editor of your choice (vi, emacs, kate, pico, nano, femto, or whatever). It is **your responsibility** to know how to operate the text editor well enough for all tasks in all labs.
3. Using the text editor that you decided to use, write a makefile for the given files (as explained in the introduction to GNU Make Manual, see the **Reading Material for Lab A**. The Makefile should provide targets for compiling the program and cleaning up the working directory.
4. Compile the project by executing make in the console.
5. Read all of lab A reading material, and make sure you **understand** it.
6. Read the puts(3) and printf(3) manuals. What is the difference between the functions? To read the manuals type man followed by the function name (e.g. man puts) in a "console".

**Important**

To protect your files from being viewed or copied by other people, thereby possibly earning you a disciplinary hearing, employ the Linux permission system by running: chmod 700 -R ~ In order to make sure you have sufficient space in your workspace, run the following command once you're logged in du -a | sort -n Then you can see a list of your files/directories and the amount of space each file/directory takes. If you need space and KNOW which files to remove, you can do that by: rm -f [filename]

**Unix: Control+D, Control+C and Control+Z**

* What does Control+D (^D) do? Control+D causes the Unix terminal driver to signal the EOF condition to the process running in this terminal foreground. You can read more about it [here](http://en.wikipedia.org/wiki/End-of-transmission_character).
* What does Control+C (^C) do? Pressing Control+C in the terminal, causes the Unix terminal to send the SIGINT signal to the process running in the terminal foreground. This will usually terminate the process.
* What does Control+Z (^Z) do? Pressing Control+Z in the terminal, causes the Unix terminal to send the SIGTSTP signal to the process running in the terminal foreground. This will suspend the process (meaning the process will still live in background).
* Do not use Control+Z for terminating processes!!!

**Writing a simple program**

Write a simple echo program named my\_echo:

NAME

my\_echo - echoes text.

SYNOPSIS

my\_echo

DESCRIPTION

my\_echo prints out the text given in the command line by the user.

EXAMPLES

#> my\_echo aa b c aa b c

**Mandatory requirements**

* Create a proper makefile as described in the reading material.
* Test your program to see that it actually works.

**The actual assignment**

In this simple assignment you will be writing a simple **encoder program**. The program has three functionalities:

1. Parsing the command-line arguments and printing debug messages.
2. The actual encoder.
3. Redirecting the input and output according to the command-line arguments.

Although you will be submitting a single program containing all the above, it is highly recommended that you implement each step in the above order and test it thoroughly before proceeding to the next one. There are several reasons for this. First, the step-by-step scheme is how physical labs will be run. But more in general, it is important to be able to partition the work and test functionalities separately, this leads to much more efficient and correct code development.

**Part 1: Command-Line Arguments and Debugging**

First, re-read and understand the arguments of main(argc, argv), which represent the command-line arguments in the line used to run any program using a "console". Recall that argv is the number of arguments, and that argv is an array of pointers to locations containing "null terminated strings" - the command line arguments, with argv[0] pointing to the program file name used in the command line to run the program.

Second, introduce a debug mode into your program. For this we will develop an easy debugging scheme which can be used with any program and allows for special debugging features for testing. The minimum implementation prints out important information to stderr when in debug mode. Printing out the command-line parameters allows for easy detection of errors in retrieving them. Henceforth, code you write in most labs and assignments will also require adding a debug mode, and it is a good idea to have this option in **all** programs you write, even if **not required** to do so!

For this scheme, you must simply loop over the command-line arguments, and if in debug mode, print each argument on a separate "line" to stderr. Debug mode is a variable that you can choose to initialize to "on" or "off" (default: off), but if there is a command line argument "+D" it turns debug mode on, and if there is a command-line argument "-D" it turns the debug mode off. Use fprintf( ) -- see manual -- for simple printing of "strings" on separate lines. Note, that the output should be to stderr, rather than stdout, to differentiate between regular program output (currently null) and debug/error messages from the program.

**Part 2: The Encoder**

In this part you will first use the command-line parsing to detect a possible encoding string, and use that to modify the output behavior. With no encoding string, every input character (from stdin) is simply sent to the output (stdout). That is, you read a character using fgetc( ), possibly encode it, and then print it after modification using fputc( ), until detecting an EOF condition in the input, at which point you should close the output stream and exit "normally". We recommend here that you use variables such as infile and outfile as arguments to fgetc() and fputc() respectively, initialized by default to stdin and stdout, respectively. This will allow you to do the last part with very little effort.

The encoding works as follows. The encryption key is of the following structure: +e{key}. The argument {key} stands for a sequence of digits whose value will be **added** to each input characters in sequence, in a **cyclic** manner.  
This means that each digit in sequence received by the encoder is added to the corresponding character in sequence in the key. When and if the end of the key is reached, re-start reading encoding digits from the beginning of the key. You should support both addition and subtraction, +e{key} is for addition and -e{key} is for subtraction.

Implementation is as follows. The key value, if any, is provided as a command-line argument. As stated above, this is indicated by a command line argument such as "+e1234" or "-e13061". The first is a sequence of numbers to be **added** to the input characters before they are emitted, while the second is a sequence of numbers to be **subtracted** from the input characters. Assumptions are: only at most one of "+e" or "-e" are present, and the rest of the command line argument is always (only) a non-empty sequence of decimal digits, terminated as usual by a null character.

Encoding is as follows: to the first character of the input, add the numerical value of the first encoding digit, to the second input character add the (numerical value of the) second digit, etc. If you reach the end of the encoding string (null character!) before you reach EOF in the input, reset to the beginning of the encoding string. Observe that this is ASCII encoding, so it should be **very simple** to compute the numeric value of each digit, which you should do directly using no special functions. Note that we advance in the encoding key once for each input character, but encoding, if indicated, should only be applied to alpha-numeric characters, that is 0-9, A-Z, and a-z, and should use "wrap around", that is assume z+1 is a, and A-1 is Z. etc. Examples are provided below to fully clarify this.

**Examples**

In the first example below see how the A,B,C,D,E are encoded adding 1,2,3,4,5 respectively and then for the next character the encoding key is reset to 1 for the next character, Z. But Z+1 is wrapped around and becomes A. Then there is a newline character, which is output with no change (still advancing the encoding key) so to the next character 3 is added, and to the one after that 4 is added. The last 2 characters again are output with no change. The 2nd example below is similar but now with a key to subtract, rather than add.

#> encoder +e12345

ABCDEZ

BDFHJA

12#<

46#<

^D

#> encoder -e4321

GDUQP523

CASPL202

^D

**Part 3: Input and/or Output to Specific Files**

The default operation of your program is to read characters from stdin (the "console" keyboard), encode them as needed, and output the results to stdout (the "console" display). After checking correctness of all the previous parts, now add the option for reading the input from a specified input file: if command-line argument "-ifname" is present, the input should be read from a file called "fname" instead of stdin (or in general the file name starts immediately after the "-i" and ends at the null character). Likewise, if command-line argument "-ofname" is present, the output should go to a file name "fname" (or in general, file name immediately after the "o").

Observe that if you did things right and heeded our advice above, this part is only a few lines of code: while scanning the command-line arguments simply check for "-i" and "-o" and open input and/or output files as needed using fopen( ), and use the file descriptor it returns for the value of "infile" and/or "outfile". The rest of the program does not need to change at all. Just make sure that if fopen( ) fails, print an error message to stderr and exit. Note that your program should support encoding keys, input file setting, output file setting, and debug flag setting, in any combination or order. You may assume that at most one of each will be given (e.g. no more than one encoding key, and no more than one output file setting).

**Helpful Information and Hints**

* stdin and stdout are FILE\* constants than can be used with fgetc and fputc.
* Make sure you know how to recognize end of file (*EOF*).
* Control-D causes the Unix terminal driver to signal the EOF condition to the process running in this terminal foreground, using this key combination (shown in the above example as ^D) will cause *fgetc()* to return an *EOF* constant and in response your program should terminate itself "normally".
* Refer to [ASCII](http://www.asciitable.com/) table for more information on how to convert characters to lower-case or upper-case.
* In every do-at-home lab, we designate some issues as "may obtain help". By this we mean there will be an **optional** task in a concurrent physical-attendance lab (in this case lab 1), where you may obtain help from a TA and make sure you are doing things correctly. This does **not** mean that you can expect the lab TA to do your work for you, only help to clarify misconceptions and difficulties. In this assignment the following issues are so designated:
  1. Correctly detecting the end-of-file condition in the input file
  2. Correct use of makefiles

**Mandatory requirements**

* You must read and process the input **character by character**, there is no need to store the characters you read at all, except for the character currently being processed.
* Important - you cannot make any assumption about the line lengths.
* Check whether a character is an lowercase (letter resp. uppercase, or number) by using a single "if" statement with two conditions. How?
* You are **not** allowed to use any library function for the purpose of recognizing whether a character is a letter, and its case.
* Read your program parameters in the same way as in Part 0's: main.c. First, set default values to the variables holding the program configuration and then scan through *argv* to update those values. Points will be reduced for failing to do so.
* Program arguments may arrive in an **arbitrary** order. Your program must support this feature.
* Output to stdout (or ofname) must contain **only** the (possibly encoded) characters from the input, this will undergo automated checking so any deviation will cause failing automated tests and a grade reduction. For the debug printouts (in stderr) we will not enforce a specific format.

**Submission**

In the following submission instructions and deliverables as well as point distribution.

**Submission instructions**

* Create a zip file with the relevant files (only).
* Upload zip file to the submission system.
* Download the zip file from the submission system and extract its content to an empty folder.
* Compile and test the code to make sure that it still works.
* In this case the makefile should be set so that "make encoder" generates an executable file of your program with the name "encoder" in its current directory, and with no additional directory structure.

**Deliverables**

All required subtask solutions are to be submitted within a single C file (encoder.c) inclusive makefile. I.e., a zip folder containing exactly the following files:

1. makefile
2. encoder.c

**Credit Points per Part**

|  |  |
| --- | --- |
| **Part** | **Points** |
| 1 | 20 |
| 2 | 40 |
| 3 | 40 |

LAB\_1

**Physical Participation Lab 1**

**Program Memory and Pointers, Debugging and Simulating Object Oriented Programming**

**Lab goals**:

* C primer
* Understanding storage addresses, introduction to pointers
* Pointers to basic data types, to structures, and to functions
* Simulating object-like behavior in C

**Remark : All the files mentioned in the lab text below are available on the course Moodle page.**

**(This lab is to be done SOLO)**

**Task 0: Using gdb(1) to debug segmentation fault**

*You should finish this task****before****attending the lab session.*

C is a low-level language. Execution of a buggy C program may cause its abnormal termination due to *segmentation fault* --- illegal access to a memory address. Debugging segmentation faults can be a laborious task. GNU Debugger, is a powerful tool for program debugging and inspection. When a program is compiled for debugging and run inside gdb, the exact location of segmentation fault can be determined. In addition, the state of the processor registers and values of the variables at the time of the fault can be examined.

The source code for a buggy program, count-words, is provided in file count-words.c. (You can find the file in the course Moodle) The program works correctly most of the time, but when called with a single word on the command line, terminates due to segmentation fault.

1. Write a Makefile for the program.
2. **Specify compiler flags appropriate for debugging using** gdb.
3. Find the location and the cause of the segmentation fault using gdb.
4. Fix the bug and make sure the program works correctly.

**The tasks below are to be done only during the lab session! Any code written before the lab will not be accepted.**

**Task 1: Understanding memory addresses and pointers**

Logical virtual memory layout of a process is fixed in Linux. One can guess from the numerical value of a memory address whether the address points to:

* a static or a global variable,
* a local variable or a function argument,
* a function.

**T1a - Addresses**

Read, compile and run the addresses.c (You can find the file in the course Moodle) program (**remember to use the *-m32* flag**).  
Can you tell the location (stack, code, etc.) of each memory address?  
What can you say about the numerical values? Do they obey a particular order?  
  
Check **long** data size on your machine using *sizeof* operator. Is *long integer* data type enough for **dist** (address difference) variables ?

**T1b - Arrays memory layout**

In this task we will examine the memory layout of arrays.  
Define four arrays of length 3 as shown below *in the function main* and print the memory address of each array cell.  
  
int iarray[3];  
float farray[3];  
double darray[3];  
char carray[3];  
  
Print the hexadecimal values of **iarray, iarray+1, farray, farray+1, darray, darray+1, carray and carray+1** (the values of these pointers, **not** the values pointed by the pointers). What can you say about the behavior of the '+' operator?  
  
Given the results, explain to the TA the memory layout of arrays.

**T1c - Distances**

Understand and explain to the TA the purpose of the distances printed in the point\_at function.  
Where is each memory address allocated and what does it have to do with the printed distance? Given the results, explain to the TA the memory layout of arrays.

**T1d - Pointers and arrays**

Array names are essentially pointer constants. Instead of using the arrays, use the pointers below to access array cells.  
int iarray2[] = {1,2,3};  
char carray2[] = {'a','b','c'};  
int\* iarray2Ptr;  
char\* carray2Ptr;  
  
Initialize the pointers iarrayPtr and carrayPtr to point to the first cell of the arrays iarray and carray respectively. Use the two pointers (iarrayPtr,carrayPtr) to print all the values of the two arrays.  
  
Add an uninitialized pointer local variable p, and print its value (not the value it points to). What did you observe?

**T1e - Address of command-line arguments**

Add a prinout of the address and content of the command line arguments (argv, argv[0], argv[1], etc.), and run the program with some command-line arguments. What can you say about the memory location of the command-line arguments visible in main( )?

**Task 2 - Structs and pointers to functions**

Let us recall the following definition:

* **Pointers to functions** - C allows declaring pointers to functions. The syntax is: function\_return\_type (\*pointer\_name)(arguments\_list); for simple types of return value and arguments. You can read more about pointers to functions [here](http://en.wikibooks.org/wiki/C_Programming/Pointers_and_arrays#Pointers_to_Functions).

The base.c (You can find the file in the course Moodle) file is the base file for task 2 - you should complete it as stated in the sub tasks.  
  
**During Task 2 we read individual characters (bytes) from stdin using fgetc( )**

* Please read the Deliverables section before continuing.

**T2a**

Implement the map function that receives a pointer to a char (a pointer to a char array), an integer, and a pointer to a function. Map returns a new array (after allocating space for it), such that each value in the new array is the result of applying the function f on the corresponding character in the input array.

1. char\* map(char \*array, int array\_length, char (\*f) (char))

Example:  
  
char arr1[] = {'H','E','Y','!'};  
char\* arr2 = map(arr1, 4, xprt);  
printf("%s\n", arr2);  
free(arr2);  
  
Results:

48

65

79

21

* Do not forget to free allocated memory.

**T2b**

Implement the following functions, and test them:  
  
char my\_get(char c);  
/\* Ignores c, reads and returns a character from stdin using fgetc. \*/  
  
char cprt(char c);  
/\* If c is a number between 0x20 and 0x7E, cprt prints the character of ASCII value c followed by a new line. Otherwise, cprt prints the dot ('.') character. After printing, cprt returns the value of c unchanged. \*/  
  
char encrypt(char c);  
/\* Gets a char c and returns its encrypted form by adding 1 to its value. If c is not between 0x20 and 0x7E it is returned unchanged \*/  
  
char decrypt(char c);  
/\* Gets a char c and returns its decrypted form by reducing 1 from its value. If c is not between 0x20 and 0x7E it is returned unchanged \*/  
  
char xprt(char c);  
/\* xprt prints the value of c in a hexadecimal representation followed by a new line, and returns c unchanged. \*/  
  
Note that array length is constant i.e. if the initial array is of length 5, then the new array that we receive with my\_get function is of the same length.  
  
Example:  
  
int base\_len = 5;  
char arr1[base\_len];  
char\* arr2 = map(arr1, base\_len, my\_get);  
char\* arr3 = map(arr2, base\_len, cprt);  
char\* arr4 = map(arr3, base\_len, xprt);  
char\* arr5 = map(arr4, base\_len, encrypt);  
char\* arr6 = map(arr5, base\_len, decrypt);  
free(arr2);  
free(arr3);  
free(arr4);  
free(arr5);  
free(arr6);  
Result:  
  
Hey! // this is the user input.  
H  
e  
y  
!  
.  
48  
65  
79  
21  
.

* Do not forget to free allocated memory.
* There is no need to encrypt/decrypt letters in a cyclic manner, simply add/reduce 1.

**Task 3 - Menu**

In this task we will be simulating objects in C. We will have a menu consisting of menu "objects", each of which has a name to be printed on the menu, and a "method" (which you will implement using a pointer to a function as a part of a "struct", as there are no "methods" or "objects" in C). **From here on we read complete lines from stdin (using fgets( ) )instead of individual chars.**

**struct** - A struct in the C programming language is a structured type that aggregates a fixed set of labeled items, possibly of different types, into a single entity similar to an "object".

The struct size equals the sum of the sizes of its objects plus alignment (if needed). You can get the size by using the **sizeof** operator as follows: sizeof(struct struct\_name).

**T3a - Preliminary: Main Menu Loop and Terminating upon EOF.**

Write and check a program called "menu" that prints to stdout a line stating "Select operation from the following menu:", reads an input line from stdin, and repeats forever unless it encounters an EOF condition for stdin. In the latter case, your program should exit normally.

Recall that typing 'Ctrl+D' in the console simulates an EOF condition for stdin.

**T3b - Implementing the menu**

In this task we will implement the menu. The menu must offer options for all your functions (Get String, Print String (crpt), Print Hex (xprt), Encrypt and Decrypt). See also the file Task 3 Example.

A function pointer can be a field in a structure, thus several functions can be accessed through a single data structure or container.  
  
An array of function descriptors, each represented by a structure holding the function name (or description) and a pointer to the function, can be used to implement a program menu. Using the following structure definition:  
  
struct fun\_desc {  
char \*name;  
char (\*fun)(char);  
};  
  
Alternatively, you can define this as a "typedef".  
  
Below is an example of declaration and initialization of a two-element array of "function descriptors":  
  
struct fun\_desc menu[] = { { "hello", hello }, { "bye", bye }, { NULL, NULL } };  
  
Extend your basic menu from T3a to offer the functions from task 2 in the following way:

* 1. Define a pointer 'carray' to a char array of length 5, initialized to the empty string (how?).
  2. Defines an array of fun\_desc and initializes it (in the declaration, not as program code within a function) to the names and the pointers of the functions that you implemented in Task 2. The last fun\_desc in the array should contain a null pointer name and a null pointer to function (**the length of the array should not be kept explicitly after constructing it**).
  3. Displays a menu (as a numbered list) of names (or descriptions) of the functions contained in the array. The menu should be printed by looping over the menu item names from the fun\_desc, **not** by printing a string (or strings) that contain a copy of the name.
  4. Displays a prompt asking the user to choose a function by its number in the menu, reads the number, and checks if it is within bounds. The bound should be pre-computed only **once**, and **before** the loop where the prompt is printed. If the number is within bounds, "Within bounds" is printed, otherwise "Not within bounds" is printed and the program exits gracefully.
  5. Evaluate the appropriate function over 'carray' (using map) according to the number entered by the user. Note that you should call the function by using the function pointer in the array of structures, and not by using "if" or "switch".
  6. After calling any menu function , let 'carray' point to the new array returned by map( ).
  7. A user should be able to terminate the program when asked for a choice, by signaling EOF using 'Ctrl+D' on an empty line.

Task3 examples can be found in the course Moodle  
  
Is it possible to call a function at an invalid address in your version of the program?

**Bonus item (0 points)** Add a menu item for "junk", where the pointer to function is initialized to point to something that is not known function code, such as your fun\_desc array. Compile and run the modified program, and select the junk menu item. What do you observe?

**Optional Tasks**

Some topics in this course can be challenging, particularly for beginners , The "Optional Tasks" section of the lab is meant to encourage students to seek assistance from the TAs and to ensure you comprehend the parts do-at-home labs (in this case Lab A and Lab B) with which we believe you may face difficulties. If you and the TA have free time during the lab, you may wish to do the following tasks from these labs during lab 1.  
  
**Makefile task(Optional 1, from Lab A)**  
  
**Correct detection of EOF (Optional 2, from Lab A)**  
  
**Using Valgrind (Optional 3, from Lab B)**

**Deliverables**

As for all labs, you should complete task 0 before the lab, and make sure you understand what you did.  
  
During the lab, you should complete all tasks, 1, 2, and 3. If you did not complete task 3 submit whatever you did complete in the lab. Your lab grade is determined and decided in most cases by your lab TA during the lab. Still, you are required to submit your code at the end of the lab, and the grade may be adjusted based on what you submit. For example, failure to quote code copied from elsewhere, even a permitted source, could cause a reduced grade, or worse.  
  
The deliverables must be submitted until the end of the day.  
  
You must submit source files for task3 (note that task3 includes the functions in task 2) in respective folders, and also a makefile that compiles them. The source files must be organized in the following tree structure ( where '+' represents a folder and '-' represents a file):  
+ task3  
   - makefile  
   - menu\_map.c

**Submission instructions**

* 1. Create a zip file with the relevant files (only).
  2. Upload zip file to the submission system.
  3. Download the zip file from the submission system and extract its content to an empty folder.
  4. Compile and test the code to make sure that it still works.

LAB\_B

**C Programming: debugging, dynamic data structures: linked lists, patching binary files.**

**This lab assignment may be done in pairs.**

**Home-Lab B -- Assignment goals:**

* Pointers and dynamically allocated structures and the "Valgrind" utility
* Understanding data structures: linked lists in C
* Basic access to "binary" files, with application: simplified virus detection in executable files

In this lab you are required to use Valgrind to make sure your program is "memory-leak" free. If you use the VM we supplied you, you should install the library libc6-dbg:i386 by running sudo apt-get install libc6-dbg:i386 You should use Valgrind in the following manner: valgrind --leak-check=full [your-program] [your-program-options]

**Preparation - Part 0: Memory Leaks, Segmentation Faults, and Printing data from files in hexadecimal format**

Programs inevitably contain bugs, at least when they are still being developed. Interactive debugging using valgrind(1) helps locate and eliminate bugs. Valgrind assists in discovering illegal memory access even when no segmentation fault occurs (e.g., when reading the n+1 place of an array of size n). Valgrind is extremely useful for discovering and fixing memory errors (leaks, double free, illegal access, etc.).

To run Valgrind write: valgrind --leak-check=full [program-name] [program parameters]. Using the command line argument --leak-check=full gives detailed information regarding each leak. Useful for finding the source of the leak and fixing it.

You might be able to get more information by running Valgrind in verbose mode like so:  
valgrind -v --leak-check=full [program-name] [program parameters]. You can even increase the level of verbosity by multiplying the "v" command line option (in some versions of Valgrind): valgrind -vvv --leak-check=full [program-name] [program parameters].

The source code of a buggy program, named **Bubblesort**, is provided. The program should sort numbers specified in the command line and print the sorted numbers, like this:

$ bubblesort 3 4 2 1  
Original array: 3 4 2 1  
Sorted array: 1 2 3 4

However, an illegal memory access causes a segmentation fault (segfault). In addition, the program has a few memory leaks.

**Part 0 assignments are:**

First, solve the segfault using gdb (or just by reading the code). Then use Valgrind to find the memory leaks and fix them.

Then, write a program that receives the name of a binary file as a command-line argument, and prints the hexadecimal value of each byte in the file in sequence to the standard output (using printf). Consult the printf(3) man page for hexadecimal format printing.

NAME  
    hexaPrint - prints the hexdecimal value of the input bytes from a given file  
SYNOPSIS  
    hexaPrint FILE  
DESCRIPTION  
    hexaPrint receives, as a command-line argument, the name of a "binary" file, and prints the hexadecimal value of each byte to the standard output, separated by spaces.

For example, your program will print the following output for this **exampleFile**(download using right click, save as):

#>hexaPrint exampleFile  
63 68 65 63 6B AA DD 4D 79 0C 48 65 78

You should implement this program using:

* fread(3) to read data from the file into memory.
* A helper function, PrintHex(buffer, length), that prints length bytes from memory location buffer, in hexadecimal format.

You will need the helper function during the rest of the assignment, so make sure it is well written and debugged.

Additionally, make sure your code for the menu at the end of lab 1 (physical presence lab 1) is working and you understand it, as you will need to implement something similar in this lab.

**The Actual Assignment (Instructions)**

**Assignment goals** - understanding the following issues: **implementing linked lists in C, basic manipulation of "binary" files.**  
In this lab you will be writing a **virusDetector** program, to detect computer viruses in a given suspected file.

NAME  
   virusDetector - detects a virus in a file from a given set of viruses  
SYNOPSIS  
   virusDetector FILE  
DESCRIPTION  
   virusDetector compares the content of the given FILE byte-by-byte with a pre-  defined set of viruses described in the file. The comparison is done according to a naive   algorithm described in task 2.  
   FILE - the suspected file

**Part 1: Virus detector using Linked Lists**

In the current part you are required to read the signatures of the viruses from the signatures file and to store these signatures in a dedicated linked list data structure. Note, that the command-line argument FILE is not used in subparts 1a and 1b below. At a later stage (part 1c) you will compare the virus signatures from the list to byte sequences from a suspected file, named in the command-line argument.

**Part 1a - Reading a binary file into memory buffers**

The signatures file begins with a **magic number** of 4 bytes, that is used to quickly check that this is the right type of file, followed immediately by the details of different viruses in a specific format. The magic number of the signature file is the character sequence "VIRL" for little-endian encoding, and "VIRB" for big-endian encoding. The rest of the file (after the magic number) consists of blocks (< N,name,signature>) where each block represents a single virus description.  
  
Notice the format is little endian - the numbers (i.e., the length of the virus) are represented in little endian order.  
  
The name of the virus is a null terminated string that is stored in 16 bytes. If the length of the actual name is less than 16, then the rest of the bytes are padded with null characters.

The layout of each block is as follows:

|  |  |  |
| --- | --- | --- |
| **offset** | **size (in bytes)** | **description** |
| 0 | 2 | The virus's signature length N, up to 2^16 little endian |
| 2 | 16 | The virus name represented as a null terminated string |
| 18 | N | The virus signature |

For example, the following **hexadecimal** signature.  
05 00 56 49 52 55 53 00 00 00 00 00 00 00 00 00 00 00 31 32 33 34 35  
represents a 5-byte length virus, whose signature (viewed as hexadecimal) is:  
  
31 32 33 34 35  
and its name is VIRUS

You are given the following struct that represents a virus description. You are required to use it in your implementation of all the tasks.  
  
typedef struct virus {  
unsigned short SigSize;  
char virusName[16];  
unsigned char\* sig;  
} virus;

First, you are required to implement the following two auxiliary functions and use them for implementing the main parts:

* virus\* readVirus(FILE\*): this function receives a file pointer and returns a virus\* that represents the next virus in the file. To read from a file, use fread(). See man fread(3) for assistance.
* void printVirus(virus\* virus, FILE\* output): this function receives a virus and a pointer to an output file. The function prints the virus to the given output. It prints the virus name (in ASCII), the virus signature length (in decimal), and the virus signature (in hexadecimal representation).

After you implemented the auxiliary functions, implement the following two steps:

* Open the signatures file, check the magic number, and exit print an error message if the magic number is incorrect (i.e. different from "VIRL") Then if magic number is OK, use readVirus in order to read the viruses one-by-one, and use printVirus in order to print the virus (to a file or to the standard output, up to your choice).
* Test your implementation by comparing your output with the file. Tip for Linux: use diff to compare files line by line. (type man diff for more info)

**Reading into structs**  
The structure of the virus description on file allows reading an entire description into a virus struct in 2 fread calls. You should read the first 18 bytes directly into the virus struct, then, according to the size, allocate memory for sig and read the signature directly to it.

**Part 1b - Linked List Implementation**

Each node in the linked list is represented by the following structure:  
typedef struct link link;  
  
  
struct link {  
link \*nextVirus;  
virus \*vir;  
};  
  
You are expected to implement the following functions:

* void list\_print(link \*virus\_list, FILE\*);  
  /\* Print the data of every link in list to the given stream. Each item followed by a newline character. \*/
* link\* list\_append(link\* virus\_list, virus\* data);  
  /\* Add a new link with the given data to the list (at the end **CAN ALSO AT BEGINNING**), and return a pointer to the list (i.e., the first link in the list). If the list is null - create a new entry and return a pointer to the entry. \*/
* void list\_free(link \*virus\_list);  
  /\* Free the memory allocated by the list. \*/

To test your list implementation you are requested to write a program with the following prompt in an infinite loop. You should use the same scheme for printing and selecting menu items as at the end of lab 1 (physical presence lab 1).  
  
1) Load signatures  
2) Print signatures  
3) Detect viruses  
4) Fix file  
5) Quit  
  
Load signatures requests a signature file name parameter from the user after the user runs it by entering "1".  
  
After the signatures are loaded, Print signatures can be used to print them to the screen. If no file is loaded, nothing is printed. You should read the user's input using fgets and sscanf. Quit should exit the program. Detect viruses and Fix file should be stub functions that currently just print "Not implemented\n" (note that these printouts are dropped in the final version of your program).  
Test yourself by:

* Read the viruses into buffers in memory.
* Creates a linked list that contains all of the viruses where each node represents a single virus.
* Prints the content. Here's an example output. File: **example output**

**Part 1c - Detecting the virus**

Now, that you have loaded the virus descriptions into memory, extend your virusDetector program as follows:

* Implement Detect viruses: operates after the user runs it by entering the appropriate number on the menu,
* Open the file indicated by the command-line argument FILE, and fread() the entire contents of the suspected file into a buffer of constant size 10K bytes in memory.
* Scan the content of the buffer to detect viruses.

For simplicity, we will assume that the file is smaller than the buffer, or that there are no parts of the virus that need to be scanned beyond that point, i.e., we will only fill the buffer once. The scan will be done by a function with the following signature:  
  
1. void detect\_virus(char \*buffer, unsigned int size, link \*virus\_list)  
  
The detect\_virus function compares the content of the buffer byte-by-byte with the virus signatures stored in the virus\_list linked list. size should be the minimum between the size of the buffer and the size of the suspected file in bytes. If a virus is detected, for each detected virus the detect\_virus function prints the following details to the standard output:

* The starting byte location in the suspected file
* The virus name
* The size of the virus signature

If no viruses were detected, the function does not print anything. Use the **memcmp(3)** function to compare the bytes of the respective virus signature with the bytes of the suspected file.  
You can test your program by applying it to the given file.

**Part 2: Anti-virus Simulation**

In this task you will test your virus detector, and use it to help neutralizing viruses from a file. The neutralization assumes that the virus is a function that does something and returns.

**Part 2a: Using hexedit**

In this part, you are required to apply your virus detector to a file, which is infected by a very simple virus that prints the sentence **'I am virus1!'** to the standard output. You are expected to cancel the effect of the virus by using the hexedit(1) tool after you find its location and size using your virus detector.  
After making sure that your virus detector program from part 1 can correctly detect the virus information, you are required to:  
1. Download the file (using right click, save as).  
2. Set the file permissions (in order to make it executable) using chmod u+x infected, and run it from the terminal to see what it does.  
3. Apply your virusDetector program to the infected file, to find the viruses.  
4. Using the hexedit(1) utility and the output of the previous step, find out the viruses location and nuetralize them by replacing the first byte of the virus code by a simple [RET](https://pdos.csail.mit.edu/6.828/2005/readings/i386/RET.htm) (near) instruction. This neutralizes the virus code by making the virus function return immediately without doing anything else.

Note that part 2a is not submitted, but you will be required to do it during the frontal check of the lab after submmission. It is also a good idea to do this before implementing part 2b, so you know that the code you write is getting the correct locations for actually neutralizing the viruses.

**Part 2b: Neutralizing the virus automatically**

Implement the functionality that is described above, do it as follows:

* Implement the "Fix file" option: scan for all viruses in the suspected FILE (the one given as the command-line argument), and neutralize them automatically by modifying first byte of the virus (equal to the first byte of the signature) to the RET instruction.
* The fix will be done by the following function:  
  void neutralize\_virus(char \*fileName, int signatureOffset)
* Hints: use fseek( ), fwrite( )

**Deliverables**

Note, that the assignments in part 0 are not checked and graded.  
For parts 1 and 2, we expect ONE programs, containing all subparts solutions and requirements (since part 2 builds on part 1), in a single C source-code file, plus an appropriate makefile.  
The deliverables must be submitted before the labs deadline.  
  
You must submit a zip file named:{YOUR\_ID}.zip, which contains only two files: makefile, and AntiVirus.c

**Submission instructions**

* Create a zip file with the relevant files (student\_id.ZIP).
* Upload zip file to the submission system.
* Download the zip file from the submission system and extract its content to an empty folder.
* Compile and test the code to make sure that it still works.

**Credit Points per Part**

|  |  |
| --- | --- |
| **Part** | **Points** |
| 1 | 60 |
| 2 | 40 |

LAB\_2

**lab 2**

In this lab, you should perform **task 0 before attending the lab session**. This lab may be done in pairs (as a rule, members of a pair must be in the same lab group, and should inform the TA(s)).

**Goals**

* Get acquainted with command interpreters ("shell") by implementing a simple command interpreter.
* Understand how Unix/Linux fork() and exec() work.
* Introduction to Linux signals.
* Redirection, and introduction to pipes.
* Learn how to read the manual (man).

**Note**  
You will be extending your code from lab 2 for use in lab C, so try make your code readable and modular.

**Motivation**

Perhaps the most important system program is the **command interpreter**, that is, the program that gets user commands and executes them. The command interpreter is thus the major interface between the user and the operating system services. There are two main types of command interpreters:

* Command-line interpreters, which receive user commands in text form and execute them (also called **shell** in UNIX-like systems).
* Menu-based interpreters, where the user selects commands from a menu. At the most basic level, menus are text driven. At the most extreme end, everything is wrapped in a nifty graphical display (e.g. Windows or KDE command interpreters).

**Lab Goals**

In this sequence of labs (2 and C), you will be implementing a simple shell (command-line interpreter). Like traditional UNIX shells, your shell program will **also** be a **user level** process (just like all your programs to-date), that will rely heavily on the operating system's services. Your shell should do the following:

* Receive commands from the user.
* Interpret the commands, and use the operating system to help starting up programs and processes requested by the user.
* Manage process execution (e.g. run processes in the background, suspend them, etc.), using the operating system's services.

The complicated tasks of actually starting up the processes, mapping their memory, files, etc. are strictly a responsibility of the operating system, and as such you will study these issues in the Operating Systems course. Your responsibility, therefore, is limited to telling the operating system which processes to run, how to run these processes (run in the background/foreground) etc.

Starting and maintaining a process involves many technicalities, and like any other command interpreter we will get assistance from system calls, such as execv, fork, waitpid (see **man** on how to use these system calls).

**lab 2 tasks**

First, download [LineParser.c](https://moodle.bgu.ac.il/moodle/pluginfile.php/3724005/mod_resource/content/1/LineParser.c) and [LineParser.h](https://moodle.bgu.ac.il/moodle/pluginfile.php/3724006/mod_resource/content/2/LineParser.h). These files contain some useful parsing and string management functions that will simplify your code substantially. Make sure you appropriately refer to LineParser.c in your makefile. You can find a detailed explanation [here](https://moodle.bgu.ac.il/moodle/mod/resource/view.php?id=2316251).

Throughout the lab pay close attention to the difference between **processes**(things that you run with execvp() after fork()) and **shell commands**. Think about when do you need a new process and when to use the process of the shell. Running things in a different process preserves inter-activeness with the shell. However, not all things can be run in a new process.

**Task 0a**

Here you are required to write a basic shell program **myshell**. Keep in mind that you are expected to extend this basic shell during the next tasks. In your code write an infinite loop and carry out the following:

1. Display a prompt - the current working directory (see man getcwd). The path name is not expected to exceed **PATH\_MAX** (it's defined in **linux/limits.h**, so you'll need to include it).
2. Read a line from the "user", i.e. from stdin (no more than 2048 bytes). It is advisable to use **fgets** (see man).
3. Parse the input using **parseCmdLines()** (LineParser.h). The result is a structure **cmdLine** that contains all necessary parsed data.
4. Write a function **execute(cmdLine \*pCmdLine)** that receives a parsed line and invokes the program specified in the cmdLine using the proper system call (see man **execv**).
5. Use **perror** (see man) to display an error if the execv fails, and then exit "abnormally".
6. Release the cmdLine resources when finished.
7. End the infinite loop of the shell if the command "quit" is entered in the shell, and exit the shell "normally".

Once you execute your program, you will notice a few things:

* Although you loop infinitely, the execution ends after execv. Why is that?
* You must place the full path of an executable file in-order to run properly. For instance: "ls" won't work, whereas "/bin/ls" runs properly. (Why?)

**Now replace execv with execvp (see man) and try again**.

* Wildcards, as in "ls \*", are not working. (Again, why?)

In addition to the reading material, please make sure you read up on and understand the system calls: fork(2), exec(2) and its variants, signal(2), and waitpid(2), before attending the "official" lab session.

**Task 0b**

Add the signal handler [looper.c](https://moodle.bgu.ac.il/moodle/pluginfile.php/3724008/mod_resource/content/1/looper.c) that prints the signal with a message saying it was received, and propagates the signal to the default signal handler. This is what really makes the process sleep/continue. The signals you need to address are: SIGTSTP, SIGINT, SIGCONT. The signals will be sent to the looper by the shell that you are going to write to test the functionality of the signal commands in task 3 and the process manager that you are going to implement in lab C.

* Use strsignal (see: man strsignal) to get the signal name.
* See signal(2) you will need it to set your handler to handle these signals.
* Use signal(signum, SIG\_DFL) to make the default handler handle the signal.
* Use raise() to send the signal again, so that the default signal handler can handle it.
* After handling SIGCONT, make sure you reinstate the custom handler for SIGTSTP
* After handling SIGTSTP, make sure you reinstate the custom handler for SIGCONT

**Task 1**

In this task, you will make your shell work like a real command interpreter (tasks 1a and 1b), and then add various features.  
When executed with the "-d" flag, your shell will also print the debug output to stderr (if "-d" is not given, you should not print anything to stderr).

**Task 1a**

Building up on your code from task 0, we would like our shell to remain active after invoking another program. The **fork**system call (see man) is the key: it 'duplicates' our process, creating an almost identical copy (**child**) of the issuing (**parent**) process. For the parent process, the call returns the process ID of the newly-born child, whereas for the child process - the value 0 is returned.

**You will need to print to stderr the following debug information in your task:**

* PID
* Executing command

Notes:

* Use fork to maintain the shell alive (recall mandatory lecture 2) by forking before **execvp**, while handling the return code appropriately (again as stated in the lecture). (Although if fork( ) fails you are in real trouble anyway (e.g. fork bomb!), so you might as well ignore this case).
* If execvp fails, use **\_exit()** (see man) to terminate the process. (Why?)

**Task 1b**

Until now we've executed commands without waiting for the process to terminate. You will now use the **waitpid** call (see man), in order to implement the wait. Pay attention to the **blocking** field in cmdLine. It is set to 0 if a "&" symbol is added at the end of the line, 1 otherwise.  
  
Invoke waitpid when you're required, and only when you're required. For example: "cat myshell.c &" will not wait for the cat process to end (cat in this case runs in the **background**), but "cat myshell.c" will (cat runs in the **foreground**).

**Task 1c**

Add a **shell command** "cd" that allows the user to change the current working directory. Essentially, you need to emulate a simplified version of the "cd" internal shell command: use **chdir** for that purpose (see man). No need to implement anything beyond transferring the argument of "cd" to "chdir". **Print appropriate error message to stderr if the cd operation fails.**

**Task 2: Redirection**

Add standard input/output redirection capabilities to your shell (e.g.,   
**"cat < in.txt > out.txt"**). Guidelines on I/O redirection can be found in the reading material.  
  
Notes:

* The **inputRedirect** and **outputRedirect** fields in cmdLine do the parsing work for you. They hold the redirection file names if exist, NULL otherwise.
* Remember to redirect input/output only in the child process. We do not want to redirect the I/O of the shell itself (parent process).

**Task 3 - Signals**

Every program you run using the shell runs as a process. You can get a list of the running processes using the ps program (see: man 1 ps and man 2 ps). In this task we are going to implement shell commands to help manage the processes using signals. Implement and test the following commands:

 suspend <process id> - Suspend a running process (SIGTSTP).

 wake <process id> - Wake up a sleeping process (SIGCONT).

 kill <process id> - Terminate a running/sleeping process.  
In both cases, use the kill( ) system call wrapper, see man 2 kill, to send the relevant signal to the given process id. Check if kill() succeeded and print an appropriate message.

Test your shell using your looper code from task0b in the following scenario:

#> ./looper&

#> ./looper&

#> ./looper&

#> ps

PID TTY TIME CMD

17998 pts/11 00:00:00 bash

24207 pts/11 00:00:00 task2

24246 pts/11 00:00:00 looper

24279 pts/11 00:00:00 looper

24326 pts/11 00:00:00 looper

24336 pts/11 00:00:00 ps

#> kill 24326

#> ps

PID TTY TIME CMD

17998 pts/11 00:00:00 bash

24207 pts/11 00:00:00 task2

24246 pts/11 00:00:00 looper

24279 pts/11 00:00:00 looper

24326 pts/11 00:00:00 looper <defunct>

24336 pts/11 00:00:00 ps

**Now that you finished tasks 1, 2, 3, save your code aside. You will need it for submission and for Lab C.**

**Task 4: Exercise in Pipe System Call**

Recall from the lecture that a pipe is a pair of input stream/output stream, such that one stream feeds the other stream directly. All data that is written to one side (the "write end") can be read from the other side (the "read end"). This sort of feed becomes pretty useful when one wishes to communicate between processes, such as when implementing a shell pipe as described in the lecture. This task is to help you execise the basic pipe mechanism, towards achieving a shell pipe implementation (part of what you will be doing in lab C).  
  
**Your task:** Implement a simple program called **mypipe** (a program **separate** from your shell), which creates a child process that sends a message such as "hello" to its parent process. The parent then prints the incoming message and terminates. Use the **pipe** system call (see man) to create the pipe.

**Deliverables: lab 2**

Tasks 1, 2, 3, 4 must be completed during the regular lab. The deliverables must be submitted until the end of the lab session.  
You must submit 2 source files:  
The first one is the shell including task 1,2, 3. Name it myshell.c  
The second one is mypipe.c (task 4).  
Also, submit a makefile that compile both of them, that is: running "make myshell" should generate the "myshell" executable, and "make mypipe" should generate the "mypipe" executable

**Submission instructions**

* Create a zip file with the relevant files (only) (named either [student-id-num].zip [student1-id-num\_student2-id-num.zip] in case of pair submission).
* Upload zip file to the submission system.
* Download the zip file from the submission system and extract its content to an empty folder.
* Compile and test the code to make sure that it still works.

LAB\_C

**Lab C**

Lab C builds on top of the code infrastructure of the "shell" from lab 2. Naturally, you are expected to use the code you wrote for lab 2 in this lab, and extend it. This do-at-home lab may be done in pairs.

**Motivation**

In this lab you will enrich the set of capabilities of your shell by adding job control, pipes, and history. Note that all parts below, except for part 1, are features that should be added to your shell as one program (in however many C functions you wish), that supports all these features.

**Part 0: preparation**

**History**

Check out the "history" mechanism in the Linux shell. For example, see [this link](https://opensource.com/article/18/6/history-command) and try it out in a Linux shell.

Reexamine your command interpreter (shell) code from lab 2, and if not done so before, reorganize it to be as modular and extensible as possible,

**Lab C Parts**

**Part 1: An Exercise in Pipes**

**Note**  
This part is independent of the shell, and a preparation for implementing a pipe command in the shell. You should not use the LineParser functions in this task, nor read any command lines. However, you need to declare an array of "strings" containing all of the arguments and ending with 0 to pass to execvp() just like the one returned by parseCmdLines().

Here we wish to explore the implementation of a pipeline. In order to achieve such a pipeline, one has to create a pipe and properly redirect the standard output and standard input of processes.  
Please refer to the 'Introduction to Pipelines' section in the reading material.  
  
**Your task:** Write a short program called **mypipeline** which creates a pipeline of 2 child processes. Essentially, you will implement the shell command line **"ls -l | tail -n 2"**.  
(A question: what does "ls -l" do, what does "tail -n 2" do, and what should their combination produce?)  
  
**Follow the given steps as closely as possible to avoid synchronization problems:**

1. Create a pipe.
2. Fork a first child process (child1).
3. In the child1 process:
   1. Close the standard output.
   2. Duplicate the write-end of the pipe using **dup** (see man).
   3. Close the file descriptor that was duplicated.
   4. Execute "ls -l".
4. **In the parent process: Close the write end of the pipe.**
5. Fork a second child process (child2).
6. In the child2 process:
   1. Close the standard input.
   2. Duplicate the read-end of the pipe using **dup**.
   3. Close the file descriptor that was duplicated.
   4. Execute "tail -n 2".
7. **In the parent process: Close the read end of the pipe.**
8. Now wait for the child processes to terminate, in the same order of their execution.

After implementing the above code, debug and test it as follows: Compile and run the code and make sure it does what it is supposed to do, by adding debug messages printed to stderr as follows:

* + In the parent process:
    - Before forking, "(parent\_process>forking…)"
    - After forking, "(parent\_process>created process with id: )"
    - Before closing the write end of the pipe, "(parent\_process>closing the write end of the pipe…)"
    - Before closing the read end of the pipe, "(parent\_process>closing the read end of the pipe…)"
    - Before waiting for child processes to terminate, "(parent\_process>waiting for child processes to terminate…)"
    - Before exiting, "(parent\_process>exiting…)"
  + In the 1st child process:
    - "(child1>redirecting stdout to the write end of the pipe…)"
    - "(child1>going to execute cmd: …)"
  + In the 2nd child process:
    - "(child2>redirecting stdin to the read end of the pipe…)"
    - "(child2>going to execute cmd: …)"

1. How does the following affect your program:
   1. Comment out step 4 in your code (i.e. on the parent process:**do not** Close the write end of the pipe). Compile and run your code. (Also: see "man 7 pipe")
   2. Undo the change from the last step. Comment out step 7 in your code. Compile and run your code.
   3. Undo the change from the last step. Comment out step 4 and step 8 in your code. Compile and run your code.

**Part 2: Implementing a Pipe in the Shell**

Having learned how to create a pipe between 2 processes/programs in Part 1, we now wish to implement a pipeline **inside** our own shell. In this part you will extend your shell's capabilities to support pipelines that consist of just one pipe and 2 child processes. That is, support a command line with one pipe between 2 processes resulting from running executable files mentioned in the command line. The scheme uses basically the same mechanism as in part 1, except that now the program to be executed in each child process is determined by the command line.

Your shell must be able now to run commands like: ls|wc -l which basically counts the number of files/directories under the current working dir. The most important thing to remember about pipes is that the write-end of the pipe needs to be closed in all processes, otherwise the read-end of the pipe will not receive EOF, unless the main process terminates.

Notes:

* The line parser automatically generates a list of cmdLine structures to accommodate pipelines. For instance, when parsing the command **"ls | grep .c"**, two chained cmdLine structures are created, representing **ls** and **grep** respectively.
* Your shell must still support all previous features, including input/output redirection from lab 2. Obviously, it makes no sense to redirect the output of the left--hand-side process (as then nothing goes into the pipe), and this should be considered an error, and likewise redirecting the input of the right-hand-side process is an error (as then the pipe output is hanging). In such cases, print an error message to stderr without generating any new processes. It is important to note that commands utilizing both I/O redirection and pipelines are indeed quite common (e.g. **"cat < in.txt | tail -n 2 > out.txt"**).
* As in previous tasks, you must keep your program free of memory leaks.

**Part 3: Process Manager**

Every program you run using the shell runs as a process. You can get a list of the running processes using the ps program (see: man 1 ps and man 2 ps). In this task we are going to add to your shell an internal "process manager" to manage the process we run in our shell (everything you fork). The process manager will provide 4 operations:

* procs - prints current processes including sleeping, running, and "freshly" terminated processes.
* wake <process id> - wakes up a sleeping process.
* suspend <process id> - suspends a running process.
* kill <process id> - terminates a running/sleeping process (was implemented in lab 2).

**Part 3a - Process List**

In this part we will create and print a list of all processes that have been forked by your shell.

**Representation**

Create a linked list to store information about running/suspended processes. Each node in the list is a struct process:

typedef struct process{

cmdLine\* cmd; /\* the parsed command line\*/

pid\_t pid; /\* the process id that is running the command\*/

int status; /\* status of the process: RUNNING/SUSPENDED/TERMINATED \*/

struct process \*next; /\* next process in chain \*/

} process;

The field *status* can have one of the following values:

#define TERMINATED -1

#define RUNNING 1

#define SUSPENDED 0

**Implementation**

Implement the following functions that create and print the process list:

* void addProcess(process\*\* process\_list, cmdLine\* cmd, pid\_t pid);: Receive a process list (process\_list), a command (cmd), and the process id (pid) of the process running the command. Note that process\_list is a pointer to a pointer so that we can insert at the beginning of the list if we wish.
* void printProcessList(process\*\* process\_list);: print the processes.
* Add support for the command procs to the shell which prints processes using printProcessList()in the following format:  
  <index in process list> <process id> <process status> <the command together with its arguments>

Example:

#> sleep 3 # foreground, takes 3 seconds until we get prompt back

#> procs

PID Command STATUS

14952 sleep Terminated

#>

#> sleep 5& # background, we get prompt back immediately

#> procs

PID Command STATUS

14962 sleep Running

#> # Wait for the process to finish

#>

#> procs

PID Command STATUS

14962 sleep Terminated

**Part 3b - Updating the Process List**

Implement the following to add some functionality to your process list:

* void freeProcessList(process\* process\_list);: free all memory allocated for the process list.
* void updateProcessList(process \*\*process\_list);: go over the process list, and for each process check if it is done, you can use waitpid with the option WNOHANG. WNOHANG does not block the calling process, the process returns from the call to waitpid immediately. If no process with the given process id exists, then waitpid returns -1.  
  **In order to learn if a process was stopped (SIGTSTP), resumed (SIGCONT) or terminated (SIGINT), It's highly essential you read and understand how to use waitpid(2) before implementing this function**
* void updateProcessStatus(process\* process\_list, int pid, int status): find the process with the given id in the process\_list and change its status to the received status.
* update void printProcessList(process\*\* process\_list);:
  + Run updateProcessList() at the beginning of the function.
  + If a process was "freshly" terminated, delete it after printing it (meaning print the list with the updated status, then delete the dead processes).

**Part 3c - Manipulating the Processes**

In this part you add to your shell process manipulation commands, one of the following (some you already implemented in lab 2):

* suspend <process id> - suspends a running process. Send SIGTSTP to the respective process. This is similar to typing CTRL-Z in the shell when running the process.
* kill <process id> - terminates a running/sleeping process. Send SIGINT to the respective process. This is similar to typing CTRL-C in the shell when running a process.
* wake <process id> - wakes up a sleeping process. Send SIGCONT to the respective process. This is similar to typing fg in a standard shell, right after typing CTRL-Z.

Use kill(), see man 2 kill, to send the relevant signal to the given process id. Check if kill() succeeded and print an appropriate message. Remember to update the status of the process in the process\_list.

Test your shell using your looper code from task0b in the following scenario:

#> ./looper&

#> ./looper&

#> ./looper&

#> procs

PID Command STATUS

18170 ./looper Running

18171 ./looper Running

18174 ./looper Running

#> kill 18170

#> Looper handling SIGINT # Message from the child process

#> suspend 18174

#> Looper handling SIGTSTP # Message from the child process

procs

PID Command STATUS

18170 ./looper Terminated

18171 ./looper Running

18174 ./looper Suspended

#> wake 18174

#> Looper handling SIGCONT # Message from the child process

#> wake 18171 # What will happen to the process? (it is already running)

#> Looper handling SIGCONT # Message from the child process

procs

PID Command STATUS

18171 ./looper Running

18174 ./looper Running

**Part 4: Adding the History Mechanism**

Here you will add a history mechanism to your shell. The history mechanism works as follows. Your shell should keep HISTLEN previous command lines in a queue, where HISTLEN is a constant with a value of 20 as a default. The history list is maintained in an array of size HISTLEN of pointers to (copies of) previous commands. Note that you need to allocate space for these copies. Also note that you should keep the UNPARSED command lines in the history list, and NOT the parsed version.

When a new command line is entered after the history list is full (already has HISTLEN entries), delete the oldest entry and insert the new one. You should implement the history as a circular queue, using "newest" and "oldest" indices.

The user can now perform the following functions as a shell command (not a process!):

* "history": print the history list (number of entry in the array and the appropriate command line), for all valid entries.
* "!!": retrieve the last command line (non-history, for clarification please refer to lab reading material) CL, enter CL again into the queue, and execute it (needs to be parsed again!).
* "!n": With n a number between 1 and HISTLEN, as in "!!" except with CL being the command line at history index n. If n is an invalid number (out of range, or no such entry yet) print an error message to stdout and ignore this command line.

Note that your shell should support history on top of all the other features: pipes, redirection, etc. This should not be hard if your code is well-designed.

**Submission**

Submit a zip file (named either [student-id-num].zip [student1-id-num\_student2-id-num.zip] in case of pair submission) with the following files: mypipeline.c that implements the stand-alone pipe from part 1, and myshell.c, the source code of a shell supporting all features from lab 2 and from this lab (parts 2,3,4) and a makefile to compile and link them into the executables "mypipeline" and "myshell", respectively.

LAB\_3

**Lab 3: Assembly Language and System Calls Primer**

**Lab Goals**

* Initial introduction to writing simple programs in assembly language.
* To get acquainted with the low-level interface to system calls.
* Calling assmebly code from C and vice-versa.
* Basics of directory listings.

**This lab may be done in pairs!**

As usual, you should read and understand the reading material and complete task 0 before attending the lab.  
**For the entire lab, do not use the standard library! This means you shouldn't include stdio.h, stdlib.h, and do not link to any other file from the C standard library. You can, however, include your own files or any files we provide you. This also means that you cannot use any library functions like printf, fopen, fgetc, strcmp, etc.**

**Task 0: Using nasm, ld and the Arguments Printing Program**

In this task we will build a program which prints its arguments to standard output without using the standard C library.

**Task 0.A: C Implementation**

1. Download the files main.c, start.s, util.c and util.h.
2. Compile and link them without using the C standard library as follows:
   * Assemble the glue code:  
     nasm -f elf32 start.s -o start.o
   * Compile the main.c and util.c files into object code files:  
     gcc -m32 -Wall -ansi -c -nostdlib -fno-stack-protector util.c -o util.o  
     gcc -m32 -Wall -ansi -c -nostdlib -fno-stack-protector main.c -o main.o
   * Link everything together:  
     ld -m elf\_i386 start.o main.o util.o -o task0
3. Write a makefile to perform the compilation steps automatically.
4. Write a new main.c that prints the elements of argv to the standard output, without using stdlib. This part is important, as here is where you make sure that you have the compiler set up correctly to work using the CDECL C calling convention, as described in class.

**Explanation**  
The file "start.s" has two purposes:

1. Each executable must have an entry point - the position in the code where execution starts. By default, the linker sets this entry point to be a library supplied code or function that begins at \_start. This code is responsible for initializing the program. After initialization, this code passes control to the main() function. Since we are not using any standard libraries, we must supply the linker with \_start of our own - which is defined in start.s.
2. The assembly-language source code in start.s also contains the system\_call function, which is used to get a direct system call without requiring you to write in assembly language. Note that you can link files written in different languages: an object file is an object file, no matter where it came from. All is machine code at some point!

**Task 0.B: Assembly Language Primer**

Implement a stand-alone program in assembly language that prints a constant string, such as "hello world" and a linefeed to stdout. For this sub-task you need to read about argument passing to a system call in assembly, and look at start.s as an example. Your program should not use anything other than Linux system services.

**Task 0.C: Recalling encoder from lab 1**

Make sure you have a **good** implementation in C of the encoder from lab 1, in the sense that it has handled the tasks with a good understanding of low-level features. That is because later on in the lab you will need to implement a simplified form in assembly language with direct system calls. A good low-level C implementation will make the task immediate, whereas an implementation without a care for details and understanding of low-level features (e.g. what is the end of a "string"?) will be extremely unhelpful. Consider that any place you have used "strlen", "strcpy", strcmp", and especially if you did something like "strlen{encoding)%i" you are **not** well prepared.

If you **do not** have a good implementation of the encoder from lab A, consider the following advice. You may also borrow lab 1 solutions in C from other people (**only**) as preparation for lab 3, but of course you should say your implementation is based on (appropriate citation) C code from another source. In this instance only this is permitted and will cause no grade reduction! If, on the other hand, you **do** have a good low level implementation of the encoder from lab A, this task 0.C can be considered a "nop", and ignore the rest of task 0.C. That is, you are ready in this aspect due to a previously well done job! (As the saying goes **"Mi She Tarakh Be Erev Shabat, Yokhal Be Shabat"**).

**Low-level encoder tips**

A good low-level implementation of the encode is easy to transfer to assembly language. Good is according to the tips below:

1. Note that you never need to "copy" a "string" in this task. Rather you can always maintain a pointer to its start, as do not need to modify such "strings". Therefore, you should not be copying any such, and certainly not use "strcpy". See next tip.
2. Recall that a "string" is simply an array of bytes, and that a pointer can be seen as a reference to the array, or any part thereof if it is advanced. So no need to use "strcmp" to detect command-line flags. For example, to do the output file case, can simply do in the loop on arguments:
3. char \* OutFileName;
4. FILE \* OutFile;
5. if(av[i][0] == '-' && av[i][1] == 'O') { /\* Can actually be done in 1 instruction, e.g.: CMP word [eax], '-'+(256\*'O') ; equivalently "-O" \*/
6. OutFileName = (av[i])+2;
7. OutFile = fopen(OutFileName, "w");
8. if(Outfile == NULL) { /\* error, print "cannot open file" and exit \*/}
9. }
10. To find the start of the encoding string, same (av[i])+2 as above works:
11. unsigned char \* EncoderString, \* CurrentEncodeP, EncodeByte;
12. if(av[i][0] == '+' && av[i][1] == 'e') {
13. CurrentEncodeP = EncoderString = (av[i])+2;
14. if(\*EncoderString == 0) /\* Error, null encoder string, exit \*/
15. }
16. /\* And then, later on, after getting each character c: \*/
17. EncodeByte = (\*CurrentEncodeP)-'0';
18. c += EncodeByte;
19. CurrentEncodeP++;
20. /\* And then below wrap around to start at null termination \*/
21. if(\*CurrentEncodeP == 0) { CurrentEncodeP = EncoderString; }

**Task 1: Simplified Encoder In Assembly Language**

Recall the encoder program from lab A (a long time ago!). We want to implement a simplified version of the encoder program in **assembly using system calls**. Please note again that your code (including the function main) should be exclusively in assembly language, and use no library functions. You may still call functions from "util.c" which we provide.  
Overall, the encryption program should support the following command-line arguments:

* -i{file} - get input from the given file.
* -o{file} - direct output to the given file.

The encoder will be a simplified version that reads a character (from stdin by default), encodes it by adding 1 to the character value if it is in the range 'A' to 'z' (no encoding otherwise), and outputs it (to stdout by default). We do this in the 3 subtasks below.

**Task 1.A: Debug printout in assembly language**

This part achieves access to arguments in cdecl C calling convention, and acts like the debug printout of lab A. Except now debug is always on, so all command line arguments are printed to stderr. Write a function called main in assembly language that prints all the command line arguments to stdout, each on a separate line, using only direct system calls (that is, use "write" system call, number 4). You are supposed to use no library functions, but may use our "strlen" provided in util.c. The program should then exit "normally" using the exit system call (system call number 1).

**Task 1.B: Basic Encoder Version**

In this section you are required to implement the encoder program from stdin to stdout (use the appropriate system calls for reading or writing). Extend the code from Task 1.A to do this. We suggest using another function called encode to do this. For simplicity we also suggest using global variables "Infile" and "Outfile" (appropriately initialized) rather than constants in the system calls, in order to speed up the next subtask.

**Task 1.C: Encoder Version With Input and Output Support**

Add to your program the option of the following command line arguments:

* -i{file} - get input from the given file.
* -o{file} - direct output to the given file.

Recall from Task 0.C how to access the file name, to be used in the system call. Also, note that the system call is equivalent to "open" (except for the return value) rather than "fopen".

**Task 2: Attach Virus Program**

Many computer viruses attach themselves to executable files that may be part of legitimate programs. If a user attempts to launch an infected program, the virus code is executed before the infected program code. The goal is to write a program that attaches its own code at the end of files in the current directory.

**In this task you are required to write both in C and assembly language. Your C program should call sys\_getdents in order to get all the entries in the current directory and print them, and when the -a{prefix} argument is provided should call assembly functions that will attach the executable code to the desired files.**

In this task you are required to implement the following program:

**DESCRIPTION**  
Print all the file names in the current directory.

**COMMAND-LINE ARGUMENT**  
**-a{prefix}**  
Attach the executable code (be discussed more below) at the end of each file in the current directory that begins with the given prefix. For every file changed you should print a message to the user, i.e., when printing the file name as a part of the file list add a print comment **"VIRUS ATTACHED"** next to the file name.

**Some guidelines**

1. Your program should use the **sys\_getdents [141]** system call.
2. The declarations of the dirent type constants can be found in the file dirent.h
3. Please note that the first argument for getdents is a file descriptor open for reading - it should be for the file "." that represents the current directory.
4. In case of an error, the program should terminate with exit code 0x55.
5. To make things easier, you may assume that the entire directory data (returned by the getdents call) is smaller than 8192 bytes.
6. Dov not forget not to use any standard library functions!. Instead, in "util.h" and "util.c", you can find few implementation for some helpful functions. You may use them.

**Task 2.A: directory listing**

Print all te filenames in the current directory to stdout. Note that this part is to be written in C.

**Task 2.B: attaching the virus**

Now add the "-a" option to attach the virus. **Warning:** You probably want to be very sure that the mechanism for selecting files works correctly at this point, e.g. you may not want the program to operate on your C source code files, etc. Be careful not to destroy your own source code files!  
  
**The following contains code you need to write in assembly language.**

1. Starting assembly language implementation: begin with a label "code\_start".
2. Write a function **void infection( )** that prints to the screen the message "Hello, Infected File". Note: this should be done using just one system call! If you have too many lines of code here then you are doing something wrong!
3. Write a function **void infector(char \*)** that opens the file named in its argument, and adds the executable code from "code\_start" to "code\_end" after the end of that file, and closes the file. Note: this should be done using just a few system calls: open (for append), write, close, each using less than 10 lines of assembly code. Again, if your code is longer then you are doing something wrong!
4. End infected and infector program part with a label "code\_end".
5. When the flag -a{prefix} is supplied, your program will call the infection and infector functions in order to add the executable code of infection to the end of each file with the given prefix. Also, this option will print out a comment such as "VIRUS ATTACHED" next to each file with the given prefix (this print will be performed in the C program).

**Note: it is recommended to open the file with the append option enabled (also need write of course). You may open for reading/writing rather than append, but then you will have to perform the lseek system call to the end of the file.**  
  
**Note for assembly language implementation:** The part of the code that is responsible for actual file handling (i.e. opening the file, adding the executable code of the infection , etc.) should be written in assembly language and done inside the file "start.s". You can add the code after the end of the code for system\_call. You can either call the system\_call code (note that it uses C calling conventions, as until now you used it through function-calls from C), or re-use part of it to do the system call yourself (shorter and simpler!). Also, it is a good idea to test your infection() function first, before proceeding to infector().  
  
Test your implementation on at least two files. You can use your previous lab solutions as input. Use the command **chmod u+wx {filename}** to give user write/execute permissions.

**Submission**

Task 1 and Task 2 (without -a option) are mandatory for this lab. You can complete the -a option in task 2 later on in the completion lab.  
You are required to submit a zip file named [your id].zip that contains the following:

+ task1

- start.s

- makefile

+ task2

- main.c

- start.s

- makefile

LAB\_D

**Lab D: Assembly Language "do at home" lab.**

**Lab Goals**

* Assembly language primer: improving proficiency in assembly language features.
* Interfacing C to assembly code-continued
* Using dynamically allocated memory
* Multi-precision addition.
* Pseudo-random number generation.

**This lab may be done in pairs!**

As usual, you should read and understand the reading material and complete part 0 before attempting to do the lab assignment.  
For this lab, unlike the previous lab (lab 3) you **are** supposed to use stdlib functions. Make sure you compile and link with the CDECL conventions, as otherwise the C to assembly interface you have used before will not work!

**Part 0: Basic Command-line Arguments Printing using stdlib**

**Part 0 is crucial for the successful completion of this lab! make sure you finish it and understand it before implementing your program to be submitted.**  
Read the Assembly lecture [Assembly Language Primer](https://moodle.bgu.ac.il/moodle/mod/resource/view.php?id=2299524). For this task you must understand the arguments of main( ), how to access the arguments of a function in assembly language (discussed in class), and how to pass arguments to a function in the C CDECL calling convention. Be careful not to mess up your stack!

In this preliminary you need to write function starting with the (global) label "main" in assembly language which performs the following:

* print **argc** in decimal format to stdout using printf
* print **argv[i]** to stdout using puts, for all i from 0 to argc-1

Now, write a makefile to compile the assembly code you wrote, and to link the resulting object file with te C standart library (gcc myfile.o). This makefile will be useful throughout the lab.

**The lab assignment: Multi-Precision Integer IO and Adder**

We have partitioned the lab work into parts, suggesting the order of implementation and testing. Nevertheless, you are supposed to submit a single program that ties it all together as statesd in part 4 below.

**Part 1: Structs and Multi-precision Integer Hexadecimal Printing and Reading**

Read about the difference between little endian and big endian [little vs. big endian](https://en.wikipedia.org/wiki/Endianness).

**Part 1.A: Printing a Multi-precision Integer**

Implement print\_multi(struct multi \*p)): gets a pointer to struct multi{unsigned char size; unsigned char num [ ]} where size is the number of bytes in the num array (always greater than 0), and the num array is a multi-precision unsigned integer in **little endian**. The function should print the value of the **entire** number in hexadecimal by calling printf("%02hhx") once for every byte in the array. If the number contains leading zeros, you may wish to remove them in the output, but this is not a requirement in the assignment.  
**Warning: please note that C library functions do not maintain the value of all your registers!**

Test this by initializing a global struct, as in the following lines, and call print\_multi from main with a pointer to the struct x\_struct:

x\_struct: db 5

x\_num: db 0xaa, 1,2,0x44,0x4f

The output in this case should be (with a linefeed at the end):

4f440201aa

**Part 1.B: Reading a Multi-precision Integer**

After you implement and test the printing, you should implement a function getmulti that reads a line from stdin using fgets, containing only a sequence of hexadecimal digits, and stores it in the above type of structure. You may assume that the input contains no leading zeros. Use your printing function to see that your input is correct. You may assume that the input line contains less than 600 characters. Note that your code will be simpler if you process hexadecimal characters in **pairs**.

Think: how do you **very simply** make sure you always need to process an even number of hex digits?

**Part 2: Addition of Multi-Precision integers**

**Overview**

In this task you need to implement the function **struct multi\* add\_multi(struct multi \*p, \*q)**;  
The function should perform an addition between two such numbers represented as structs, creating a third number represented the same way. This is done by byte-wise addition between the two arrays defined in the given structs while maintaining the carry between additions. The result should be placed in a newly allocated array in a new allocated struct of size 1+max(len1, len2).

Input:  
Two arrays **array1, array2** (defined as "variables" in the code), of size **len1, len2** respectively.  
For example:

x\_struct: db 5

x\_num: db 0xaa, 1,2,0x44,0x4f

y\_struct: db 6

y\_num: db 0xaa, 1,2,3,0x44,0x4f

Output:  
Without loss of generality, assume that len1 > len2. Therefore

* **max\_len = max(len1,len2)=len1**
* **min\_len = min(len1,len2)=len2**

The function will return an array, dynmically allocated using malloc, **result\_array**, of size **max\_len** such that:

* **result\_array[i]=array1[i]+array2[i]+cy** for 0 <= i < min\_len.
* **result\_array[i]=array1[i]+cy** for min\_len <= i < max\_len.

cy is the result of the carry from the previous addition.

**Part 2.A: Get MaxMin**

Implement this assembly language function **not** in the C calling convention. Given pointers to number structures in eax and ebx, return the pointer to the one with the higher length field in eax, and the other pointer in ebx.

**Part 2.B: add\_multi Implementation**

Use the MaxMin function and Print\_multi you wrote to implement and test the element-wise addition, and print each number to be added and the result in separate lines to stdout.

**Test your function by defining appropriate initialized number structs and printing the resulting array.**

**Part 3: Pseudo-Random Number Generator (PRNG)**

Implement a function name **rand\_num** that uses basic assembly instructions in order to generate a random number using a "linear-feedback shift register". See [LFSR in Wikipedia](https://en.wikipedia.org/wiki/Linear-feedback_shift_register) The function uses a global initialized (not to zero!) unsigned 16-bit (word) STATE variable, and a constant MASK variable. Use the mask for the Fibonacci LFSR for 16 bits. Each pseudo-random operation does:

* Use the MASK to get just the relevant bits of the STATE variable.
* Compute the parity of the above relevant bits. Note: we recommend, but not require, that you use the parity flag!
* Shift the bits of the (non-maked) STATE variable one position to the right, with the MSB determined by the parity you just computed.

First, test your function by printing some generated pseudo-random numbers in hexadecimal using printf. Once you have done that, write a function PRmulti: uses the PRNG to create a pseudo-random Multi-precision Integer as follows: the first 8 bits generated by the PRNG determine the length n in bytes of the number (generate a new random byte instead if this is zero!), and then 8\*n PRNG bits determine the actual value to be insetred into the appropriate struct.  
If done properly, you should be able to use your printing function from part 1.A to print the resulting numbers, do so and thoroughly test your code.

**Part 4: Putting it all together**

Your final program "multi" should combine all the above as follows. The program should print to stdout the numbers to be added (in hexadecimal), and then their sum. Then the program exists normally. The source of the numbers is to be determined as follows:

* By default (no command-line arguments), the program operates (i.e. prints and adds) on the numbers encoded by x\_struct and y\_struct.
* If argv[1] is "-I", the program operates on numbers obtained from stdin, one number per line (as in part 1.B)
* If argv[1] is "-R" the program operates on numbers obtained from the Pseudo-Random number generator, as in part 3.

Some example of program runs follow. First, the default with no command line arguments:

$ ./multi

4f440201aa

4f44030201aa

4f9347040354

In the examples with the "-I" flag below, the first two lines are input lines. You may have an extra leading zero in each number's output.

$ ./multi -I

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

1

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

1

100000000000000000000000000000000000000000000000000000000000000000000000

$ ./multi -I

2

1

2

1

3

When running with "-R" flag, the result may vary depending on the seed and how the 16-bit pseudo-random is used exactly, and the numbers may be very long, as in the following example:

$ ./multi -R

5588c47a8814c200cf54a8e1dc23036edd0b196e7d5a8510c1698fdeba56294dd970a69c59249962913d80e3bde487a991057c8a1d15d26e45a6ecb0303c480a19c61360f0eac2c039db49a247b74ac3a0be451ecc8d7f1378196e

7f933c689717b3b90785688ffe379eb82f8faecb679a4bebaa78018761582c968fb622e3bdf4414d5934d765abe8b3597cc218ae337dbaee915580733f71b66af6c0f19a83baa6842885a8f1922439bbeeb1d848e25d2768070d039ee84b539a83aa6060e0b4031e24d9588cd7d52c56e11cd58c8f9e908d27d027304cb1d85824b9ef59ace2b563897c6afe77b4632112508b8db7ca9717a37fe34d5924118163d918b6ca3f79990a194ef092147297b7e22579395b85304ca11ebcec8844b61a1fd454b0189ee074f591c59ac3189e7851e3bd6c645b759d29fdc6ebe2f55994963f319c

7f933c689717b3b90785688ffe379eb82f8faecb679a4bebaa78018761582c968fb622e3bdf4414d5934d765abe8b3597cc218ae337dbaee915580733f71b66af6c0f19a83baa6842885a8f1922439bbeeb1d848e25d2768070d039ee84b539a83aa6060e0b4031e24d9588cd7d52c56e11cd58c8f9e908d27d027304cb1d85824b9ef59ace2b563897cc0873c2eeb35d4515ae260ac733aa6eec05872928edbe8e9da205a1e33ef3367286138b0cbbc5144b6b6ba3f4314d44aafc2691261cbec8e19fb9cc8cf1cbcffab8bae2409893b121d98b606a32ce7ed9e853101c1e713a9b74b0a

**Submission**

You need to submit a single assembly language code file, "multi.s", and a makefile which compiles it and links it into an executable named "multi". (Compile: "nasm -f elf32 multi.s -o multi.o" Link: "gcc -m32 multi.o -o multi") The code is as completed in part 4, which as stated above contains all the other parts. You are required to submit a zip file in the format [your\_id].zip that contains "multi.s" and "makefile".

LAB\_4

**Lab 4: ELF - introduction**

This lab **may** be done in pairs.

You must complete task 0 before attending the lab, and we also recommend doing tasks 1a, 1b, and 1c before the lab.

**Lab Goals**

* Extracting useful information from files in ELF.
* Fixing files using this information: reverse engineering.

In the following labs, you will learn to handle object and executable files. We begin by learning just some of the basics of ELF, together with applications you can already use at this level - editing binary files and writing software patches. Then, we will continue our study of ELF files, by beginning to parse the structures of ELF files, and to use them for various purposes. In particular, we will access the data in the section header table and in the symbol table.

**Methodology**

* Learn how to use the *readelf* utility. By using *readelf* you can get, in a human readable format, all the ELF structural information.
* Experience basic ELF manipulation.

**Recommended Operating Procedure**

**This advice is relevant for all tasks.** Note that while at some point you will no longer be using *hexedit* to process the file and *readelf* to get the information, nevertheless in some cases you may still want to use these tools for debugging purposes. In order to take advantage of these tools and make your tasks easier, you should:

* Support debugging messages: in particular the offsets of the various items, as you discover them from the headers. Also, whenever the user is required to enter values, you should print the **parsed** values in their respective representation (e.g. string, decimal or hexadecimal).
* Use *hexedit* and *readelf* to compare the information you are looking for, especially if you run into unknown problems: *hexedit* is great if you know the exact location of the item you are looking for.
* Note that while the object files you will be processing will be linked using *ld*, and will, in most cases, use direct system calls in order to make the ELF file simpler, there is no reason why the programs you write need use this interface. You are allowed to use the standard library when building your own C programs.
* In order to preserve your sanity, even if the code you MANIPULATE may be without stdlib, we advise that for your OWN CODE you DO use the C standard library! (Yes, this is repeated twice, so that you notice it!)
* In order to keep sane in the following labs as well, **understand** what you are doing and **keep track** of that and of your code, as you will be using them in future labs.

All the executable files we will work with in this session are 32-Bit ELF binaries.  
Compile your code accordingly.

**Lab 4 Tasks**

**Task 0**

**Task 0a:**

Download the executable file *abc* (from the [Lab 4 auxilliary files](https://moodle.bgu.ac.il/moodle/mod/folder/view.php?id=2378992)), and answer the following questions (be prepared to explain your answers to the lab instructor):

1. Where is the entry point specified, and what is its value?
2. How many sections are there in "abc"?

**Task 0b**

Write a program called *hexeditplus*:

./hexeditplus

The hexeditplus program performs operations (read and write) on files and memory. File operations are done on a file *file\_name* as defined below. Each operation is done in units of *size* bytes, which indicates a unit size, i.e. the number of bytes we want to use as the basic unit in each operation of our program, such as "display memory contents". Size can be either 1, 2 or 4, with 1 as the default.

First, define a menu for the user with a number of predefined functions (as done in Lab 1), to which we will add functions as we go. The program prints the menu, obtains a choice from the user, acts on it, and repeats infinitely. Specifically, the command line:

./hexeditplus

Should print:

Choose action:

0-Toggle Debug Mode

1-Set File Name

2-Set Unit Size

3-Load Into Memory

4-Toggle Display Mode

5-Memory Display

6-Save Into File

7-Memory Modify

8-Quit

For this part, use an array with the above menu names and pointers to appropriate functions that implement each option, using the same menu scheme from lab 1.

At this point implement "Toggle Debug Mode", "Set File Name", "Set Unit Size", and "Quit". All other functions should at this point be stubs that print "Not implemented yet" and return without doing anything else.

Toggle debug mode means turn the debug flag on (if it is currently off, which it is in the initial state), and print "Debug flag now on". If the debug flag is on, this function prints "Debug flag now off", and turns the flag off. When the debug mode is on, you should print the value of the variables: unit\_size, file\_name, and mem\_count, every time just before the menu is printed.

All functions should be of the form:

void fun(state\* s); // Getting the state as a pointer allows the functions to change it.

where the state struct is defined as:

typedef struct {

char debug\_mode;

char file\_name[128];

int unit\_size;

unsigned char mem\_buf[10000];

size\_t mem\_count;

/\*

.

.

Any additional fields you deem necessary

\*/

} state;

**Set File Name** queries the user for a file name, and store it in *file\_name*. You may assume that the file name is no longer than 100 characters. If debug mode is on, the function should also print (to stderr, as are all debug messages): "Debug: file name set to 'file\_name' " (obviously, replacing 'file\_name' with the actual name).

**Set Unit Size** option sets the size variable. The steps are:

1. Prompt the user for a number.
2. If the value is valid (1, 2, or 4), set the size variable accordingly.
3. If debug mode is on, print "Debug: set size to x", with x the appropriate size.
4. If not valid, print an error message and leave size unchanged.

**Quit** is a function that prints "quitting" (in debug mode), and calls exit(0) to quit the program.

The rest of the functions will be written in the next tasks. The menu should be extensible, you will change and extend it in each sub-task of task 1. It should be printed using a loop iterating over the menu array, and be {NULL, NULL} terminated.

Be sure to implement this code and test it carefully before the lab (that is why you have the debug option), as you will need to extend it during the lab!

**Task 1: hexeditplus**

In this task we will write our own version of *hexedit* for working with binary files. You will extend your code from task 0b.  
Note: You should verify that there is no error when opening a file. In case of an error, you should print a message and abort the rest of the operation.  
For this task you will be working with the following ELF file: *abc* (see [Lab 4 auxilliary files](https://moodle.bgu.ac.il/moodle/mod/folder/view.php?id=2378992)).

**Note:** For any functions that handle files, the file needs to be opened and closed **within that function**

**Task 1a: Load Into Memory**

**For this task, you may assume that the size of the data being loaded is less than 10000 bytes.**

Write the function for the "Load Into Memory" option, which works as follows:

* Check if *file\_name* is empty (i.e. equals to ""), and if it is print an error message and return.
* Open *file\_name* for reading. If this fails, print an error message and return.
* Prompt the user for *location* in **hexadecimal**, and *length* (in decimal).
* If debug flag is on, print the file\_name, as well as *location*, and *length*.
* Copy *length* \* *unit\_size* **bytes** from *file\_name* starting at position *location* into *mem\_buf*.
* Close the file.

Assume that the user has already set the file name to "abc". If the user chooses 3 on the menu, he is prompted for *location* and *length*. It should look as follows:

3

Please enter <location> <length>

12F 10

The program should open the file abc and load the 10 bytes (assuming unit size is set to 1), from byte 303 (which is the decimal value of 0x12F) to byte 312 in the file into *mem\_buf*. The output should look like:

Loaded 10 units into memory

**Remember**

* To read *location* (hexadeciomal) and *length* (decimal) use fgets and then sscanf, rather than scanf directly.
* Note again that *location* is always entered in hexadecimal representation.

**Task 1b: Toggle Display Mode**

Write the function for the "Toggle Display Mode" option, which switches between display using a decimal representation, and display using a hexadecimal representation.

Toggle display mode means turn the display flag on and print using a hexadecimal representation, the initial state is off and print using a decimal representation. Print "Display flag now on, hexadecimal representation", and if the display flag is on, this function prints "Display flag now off, decimal representation", and turns the flag off. For exmple, (assume that the display flag is off) entererint option 4 in the menu:

4

Display flag now on, hexadecimal representation

4

Display flag now off, decimal representation

**Task 1c: Memory Display**

Write the function for the "Memory Display" option:  
This option displays *u* units of size *unit\_size* starting at address *addr* in memory. Unit\_size is already defined in *state*, but *u* and *addr* should be queried from the user by this function. *u* will be given in decimal and *addr* in hexadecimal. Entering a value of 0 for *addr* is a special case, in which the memory to be displayed starts at your mem\_buf.

The units should be displayed according to the display flag. If the display flag is on then print using a hexadecimal representation, and if the display flag is off print using a decimal representation.

If the user set the unit size to 2 and loaded a file into memory, then the output should look something like this (remember, we previously loaded 5 pairs of bytes from the file *abc*, located 303-312):

Choose action:

0-Toggle Debug Mode

1-Set File Name

2-Set Unit Size

3-Load Into Memory

4-Toggle Display Mode

5-Memory Display

6-Save Into File

7-Memory Modify

8-Quit

> 5

Enter address and length

> 0 5

Decimal

=======

256

0

12032

26988

12130

Choose action:

0-Toggle Debug Mode

1-Set File Name

2-Set Unit Size

3-Load Into Memory

4-Toggle Display Mode

5-Memory Display

6-Save Into File

7-Memory Modify

8-Quit

> 4

Display flag now on, hexadecimal representation

Choose action:

0-Toggle Debug Mode

1-Set File Name

2-Set Unit Size

3-Load Into Memory

4-Toggle Display Mode

5-Memory Display

6-Save Into File

7-Memory Modify

8-Quit

> 5

Enter address and length

> 0 5

Hexadecimal

===========

100

0

2F00

696C

2F62

Note that, depending on the chosen unit size, the printed hexadecimal values may differ in order when compared with the output of *hexedit*. Why is that?

Use your newly implemented functionality (load into memory and memory display) to answer: what is the entry point of your own *hexeditplus* program? Verify your answer using readelf -h

**Implementation note: working with units**  
You are required to write code that handles data in unit sizes (i.e. not necessarily single bytes). This might confuse you into writing much more code than needed. See below how to handle multiple unit sizes when reading, printing etc. without writing too much code. Relevant to this task is the function print\_units, and also see below.

Note that, you can use the following lines to use for printing

printf(dec\_formats[u-1], val);

printf(hex\_formats[u-1], val);

where *u* is the current unit size and *val* is the val that we want to print, and with the arrays defined as follows:

static char\* hex\_formats[] = {"%#hhx\n", "%#hx\n", "No such unit", "%#x\n"};

static char\* dec\_formats[] = {"%#hhd\n", "%#hd\n", "No such unit", "%#d\n"};

**Task 1d: Save Into File**

Write the function for the "Save Into File" option, which works as follows:  
This option replaces *length* units (each of size determined by the current unit size) at *target-location* of *file\_name* with bytes from the **hexeditplus** memory starting at virtual address *source-address*. Note that the filename is the last file name set by option 1 "Set File Name".  
  
When the user chooses option 6, the program should query the user for:

* *source-address* (source **virtual memory** address, in hexadecimal), *source-address* can be set to 0, in which case, the source address is start of *mem\_buf*, in any other case, use *source-address* as an address in (virtual) memory.
* *target-location* (target file offset, in hexadecimal),
* *length* (number of units, in decimal).

Implement the checks that the file can be opened (for writing and NOT truncating), and print appropriate debug messages in debug mode as in the previous task. Close the file after writing.

For example, after the file name was set to "abc" and unit size to 1 bytes, choosing option "6-Save Into File" using *source-address* 960c170, *target-location* 33 and *length* 4, the program should write *length* = 4 bytes from (virtual) memory, starting at address 0x960c170 to the file *abc*, starting from offset 0x33 (overwriting what was originally there). It should look as follows:

6

Please enter <source-address> <target-location> <length>

960c170 33 4

Note again that the target file is the one specified using option 1 in the menu.  
  
Also observe that after you execute this option, **only** *length* units of the file *file\_name* should be changed.

If <target-location> is greater than the size of <file\_name> you should print an error message and not copy anything.

You should use *hexedit*, to verify that your code for tasks 1c and 1d works correctly, by loading a portion of a file into memory and saving it to another file.  
Here is some of *hexedit*'s output for the file abc, verify that you understand why the output is as it is.

00000070 01 00 00 00 01 00 00 00 00 00 00 00 00 80 04 08 ................

00000080 00 80 04 08 EC 05 00 00 EC 05 00 00 05 00 00 00 ................

00000090 00 10 00 00 01 00 00 00 14 0F 00 00 14 9F 04 08 ................

000000A0 14 9F 04 08 0C 01 00 00 14 01 00 00 06 00 00 00 ................

000000B0 00 10 00 00 02 00 00 00 28 0F 00 00 28 9F 04 08 ........(...(...

000000C0 28 9F 04 08 C8 00 00 00 C8 00 00 00 06 00 00 00 (...............

000000D0 04 00 00 00 04 00 00 00 48 01 00 00 48 81 04 08 ........H...H...

000000E0 48 81 04 08 44 00 00 00 44 00 00 00 04 00 00 00 H...D...D.......

000000F0 04 00 00 00 51 E5 74 64 00 00 00 00 00 00 00 00 ....Q.td........

00000100 00 00 00 00 00 00 00 00 00 00 00 00 06 00 00 00 ................

00000110 04 00 00 00 52 E5 74 64 14 0F 00 00 14 9F 04 08 ....R.td........

00000120 14 9F 04 08 EC 00 00 00 EC 00 00 00 04 00 00 00 ................

00000130 01 00 00 00 2F 6C 69 62 2F 6C 64 2D 6C 69 6E 75 ..../lib/ld-linu

00000140 78 2E 73 6F 2E 32 00 00 04 00 00 00 10 00 00 00 x.so.2..........

00000150 01 00 00 00 47 4E 55 00 00 00 00 00 02 00 00 00 ....GNU.........

00000160 06 00 00 00 0F 00 00 00 04 00 00 00 14 00 00 00 ................

00000170 03 00 00 00 47 4E 55 00 C1 4E 4D 18 B9 A6 21 8F ....GNU..NM...!.

**Task 1e: Memory Modify**

Write the function for the "Memory Modify" option:  
This option replaces a unit at *location* **in the memory buffer** (not virtual memory address!) with *val*.  
The steps are:

1. Prompt the user for *location* and *val* (all in hexadecimal).
2. If debug mode is on, print the location and val given by the user.
3. Replace a unit at *location* in the memory with the value given by *val*.

When the user chooses option 7, the program should query the user for:

* *location* (memory buffer location, in hexadecimal)
* *val* (new value, in hexadecimal)

For example, if unit size was set to 4, choosing option "7-Memory Modify" using *location* 0x40, *val* 0x804808a, will overwrite the 4 bytes starting at location 0x40, with the new value 804808a. It should look as follows:

7

Please enter <location> <val>

40 804808a

As in the previous task, you should check that the location chosen to be modified, given the currentunit size, is valid, and act accordingly.

You can test the correctness of your code using hexedit (by saving the memory buffer to a file and then looking at the resulting change using hexedit).

**Task 2: Reading ELF**

Download the following file: deep\_thought from Lab 4 Files.  
  
deep\_thought is an executable ELF file. It does not run as expected. Your task is to understand the reason for that.  
  
Do the following:

1. Run the file.
2. Which function precedes main in execution ? (Hint: see assembly code in Lab 3).
3. What is the virtual address to which this function is loaded (Hint: use readelf -s)

**Fixing the buggy executable file**

Use your *hexeditplus* program from task 1 to display the entry point of a file.

What are the values of *location*/*length*? How do you know that?  
  
Use the edit functions from *hexeditplu*s program to fix the *deep\_thought* file, so that it behaves as expected.

**Task 3: Delving Deeper into the ELF Structure**

The goal of this task is to display the compiled code (in bytes) of the function main, in the *offensive* executable in the lab 4 auxiliary files.

In order to do that, you need to:

1. find the offset (file location) of the function main.
2. find the size of the function main.
3. use your *hexeditplus* program to display the content of that function on the screen.

Finding the needed information:

1. Find the entry for the function main in the symbol table of the ELF executable (readelf -s).
2. In that reference you will find both the size of the function and the function's virtual address and section number.
3. In the section table of the executable, find the entry for the function's section (readelf -S).
4. Find both the section's virtual address (Addr), and the section's file offset (Off).
5. Use the above information to find the file offset of the function.

**Hacking the executable file**

Hack this executable file so that it does nothing when it is run: replace the code of the main function by [NOP](https://en.wikipedia.org/wiki/NOP_(code)) instructions.  
Make sure you do NOT override the ret instruction ([Opcode](https://en.wikipedia.org/wiki/Opcode): c3) in main.  
Alternately, you can plant just one ret instruction (where?).

**Task 4: Hacking: installing a patch using hexeditplus**

The following file ntsc file was meant to be a digit counter. Download it from Lab 4 Files, and run it in the command-line.

./ntsc aabbaba123baacca

./ntsc 1112111

What is the problem with the file? (hint, try this string: 0123456789)  
  
Create a new program with a correct digit counter function (should get a char\* and return an int), compile and test it. Remember to compile with the -m32 flag in order to produce an ELF compatible with 32 bits, and also **must use the compile flags: `-fno-pie' and '-fno-stack-protector`**  
  
Use *hexeditplus* to replace (patch) the buggy digit\_cnt function in the *ntsc* file with the corrected version from the new program.  
You should do it using options 3 & 6 in *hexeditplus*.  
(think: are there any kinds of restrictions on the code you wrote for the digit\_cnt function?)  
Explain how you did it, and show that it works.

**Deliverables:**

Tasks 1,2, and 3 must be completed during the regular lab. Task 4 may be done in a completion lab, but only if you run out of time during the regular lab. The deliverables must be submitted until the end of the lab session.  
You must submit source files for task 1 and task 4 and a makefile that compiles them. The source files must be named task1.c, task4.c, and makefile.

**Submission instructions**

* Create a zip file with the relevant files (only).
* Upload zip file to the submission system.
* Download the zip file from the submission system and extract its content to an empty folder.
* Compile and test the code to make sure that it still works.

LAB\_E

**Lab E: Linking ELF Object Files**

This lab may be done either solo or in pairs.

In the previous lab, you learned to investigate and change ELF files using hexedit, and other command-line tools. In this lab, you will continue to manipulate ELF files, this time using your own code (written in C), and perform a limited pass I operation of the linkage editor ("linker").

We will parse the ELF file and extract useful information from it. In particular, we will access the data in the section header table, and in the symbol table. We will also learn to use the mmap system call.

**Important**

This lab is written for 32-bit machines. Some of the computers in the labs already run on a 64-bit OS (use*uname -a* to see if the linux OS is 64-bit or not). 32-bit and 64-bit machines have different instruction sets and different memory layout. Make sure to include the *-m32* flag when you compile files, and to use the Elf32 data structures (and not the Elf64 ones).

In order to know if an executable file is compiled for 64-bit or 32-bit platform, you can use readelf, or the *file* command-line tool (for example: file /bin/ls).

**Useful Tips**

You will no longer be using *hexedit* to process the file and strings to find the information; nevertheless, in some cases you may still want to use these tools for debugging purposes. In order to take advantage of these tools and make your tasks easier, you should:

* Print debugging messages: in particular the offsets of the various items, as you discover them from the headers.
* Use *hexedit* and *readelf* to compare the information you are looking for, especially if you run into unknown problems. *hexedit* is great if you know the exact location of the item you are looking for.
* Note that while the object files you will be processing will be linked using *ld*, and will, in most cases, use direct system calls in order to make the ELF file simpler, there is no reason why the programs you write need use this interface. You are allowed to use the standard library when building your own C programs.
* In order to preserve your sanity, even if the code you MANIPULATE may be without stdlib, we advise that for your OWN CODE you DO use the C standard library!
* In order to keep sane in the following lab as well, **understand** what you are doing and **keep track** of that and of your code, as you will be using them in a future lab.

**Lab E Assignment**

The goal of this lab is to implement a limited pass I (merging) of a linkage editor. You begin by allowing access to ELF object files, examining, and printing out their structures (section headers and symbol table). After this part is correctly implemented and debugged, you will be implementing the part that does the merging. This assignment will be limited to merging 2 ELF files, and with additional simplifying assumptions and restrictions specified later on.

You must use only the mmap system call to read data from your ELF files from this point onwards. However, you should use write( ) to generate the merged output file.

**Part 0**

This part is about learning to use the mmap system call. Read about the mmap system call (man mmap).

Write a program that uses the mmap to examine the header of a 32-bit ELF file (include and use the structures in elf.h). The program is first activated as:

myELF

The program then uses a menu similar to lab 4, with available operations, as follows:

Choose action:

0-Toggle Debug Mode

1-Examine ELF File

2-Print Section Names

3-Print Symbols

4-Check Files for Merge

5-Merge ELF Files

6-Quit

Note that the menu should use the same technique as in lab 1, i.e. an array of structures of available options. Toggle Debug Mode is as in Lab 4. Quit should unmap and close any mapped or open files, and "exit normally". Examine ELF Files queries the user for an ELF file name(s) to be used and examined henceforth. For now, options 2, 3, 4, 5, should call stub functions that print "not implemented yet". All file input should be read using the mmap system call. You are NOT ALLOWED to use read, or fread.

To make your life easier throughout the lab, for each ELF file, map the entire file with one mmap call.

In Examine ELF File, after getting a file name, open the file for reading, and then print the following:

1. Bytes 1,2,3 of the magic number (in ASCII)
2. Entry point (in hexadecimal)

Check using *readelf* that your data is correct.

Once you verified your output, extend *examine* to print the following information from the header:

1. Bytes 1,2,3 of the magic number (in ASCII). Henceforth, you should check that the number is consistent with an ELF file, and refuse to continue if it is not.
2. The data encoding scheme of the object file.
3. Entry point (hexadecimal address).
4. The file offset in which the section header table resides.
5. The number of section header entries.
6. The size of each section header entry.
7. The file offset in which the program header table resides.
8. The number of program header entries.
9. The size of each program header entry.

The above information should be printed in the above exact order (print it out as nicely as *readelf* does). If invoked on an ELF file, examine should initialize a global file descriptor variable for this file, and leave the file open. When invoked on a non-ELF file, or the file cannot be opened or mapped at all, you should print an error message, unmap the file (if already mapped) close the file (if already open), and set the respective file descriptor variable to -1 to indicate no valid file. You probably also should use a global variable to indicate the memory location of the mapped file.

Your Examine ELF File should be able to handle up to two ELF files, keep both open and mapped at the same time. So you should keep 2 separate file descriptors, map\_start variables, and other relevant information for each file. That is, each time this function is called it should get a new file name, open the file and map it, and print out the above stated information, while keeping any previous file information as well. Calling the function for the 3rd time or more you may print out an error message and do nothing, or (bonus) you may decide to support more than 2 ELF files.

You can test your code on the following file: [a.out](https://moodle.bgu.ac.il/moodle/mod/resource/view.php?id=2378996), and also any of the files in the "ELF object file examples" folder.

**Part 1 - Sections**

Extend your myELF program from Part 0 to allow printing of all the Section names in an 32-bit ELF file (like readelf -S). That is, modify the menu to add a "Print Section Names" option.

For each ELF file already opened by Examine ELF File, Print Section Names should visit all section headers in the section header table, and for each one print its index, name, address, offset, size in bytes, and type number. Note that this is done for all files currently mapped, so there is no file just print an error message and return.

The format for each ELF file should be:

File ELF-file-name

[index] section\_name section\_address section\_offset section\_size section\_type

[index] section\_name section\_address section\_offset section\_size section\_type

[index] section\_name section\_address section\_offset section\_size section\_type

....

Verify your output is correct by comparing it to the output of *readelf*. In debug mode you should also print the value of the important indices and offsets, such as shstrndx and the section name offsets.

You can test your code on the following file: [a.out](https://moodle.bgu.ac.il/moodle/mod/resource/view.php?id=2378996), and also any of the files in the "ELF object file examples" folder.

**Hints**  
Global information about the ELF file is in the ELF header, including location and size of important tables. The size and name of the sections appear in the section header table. Recall that the actual name **strings** are stored in an appropriate **section** (.shstrtab for section names), and not in the section header!

**Part 2 - Symbols**

Extend your myELF program from part 1 to support an option that displays information on all the symbol names in a 32-bit ELF file.

The new Print Symbols option, for each open ELF file, should visit all the symbols in that ELF file (if none, print an error message and return). For each symbol, print its index number, its name and the name of the section in which it is defined. (similar to readelf -s). Format should be:

File ELF-file0name

[index] value section\_index section\_name symbol\_name

[index] value section\_index section\_name symbol\_name

[index] value section\_index section\_name symbol\_name

...

Verify your output is correct by comparing it to the output of *readelf*. In debug mode you should first print the size of each symbol table, the number of symbols therein, and any other useful information.

**Hints:**

Symbols are listed in the designated sections. The section in which a symbol is defined (if it is defined) is the index of the symbol, which is an index into the section header table, referring to the section header of the appropriate section, and from there the section name can be retrieved as above. Symbol name is an attribute of the symbol structure, but recall again that the actual name string is stored in a string table, a separate section(.strtab).

**Part 3 - Linker pass 1**

In this part you shall write a limited version of a linker pass 1 (as mentioned in the last lecture). It will merge 2 ELF files (the ones opened by Examine ELF File) into one relocatable object file. In this part, you should be using as input the files from the the "ELF object file examples" folder, where "F1a.o", F1b.o", and "F1c.o" are intended for use as the first file, and "F2a.o", "F2b.o" are intended for use as the second file. Merging files should result in a single relocatable object file, mimicking the one generated by pass I of the linker. For example, invoking pass I of the linker: "ld -r -m elf\_i386 F1a.o F2a.o -o F12a.ro" generates the relocatable object file "F12a.ro". Your program should generate the same type of output ELF file given these same files. The linker can be used on (only) the resulting file to create a working executable file: "ld -m elf\_i386 F12a.ro -o F12a" generates the "F12a" executable file that can actually be run.

**Part 3.1: Check Files for Merge**

Implement Check Files for Merge: CheckMerge. The CheckMerge function first checks that 2 ELF files have been opened and mapped, (print an error message and return otherwise). Here we assume that there are exactly 2 such ELF files, and each file contains exactly one symbol table (otherwise, print "feature not supported" and return). CheckMerge then, for each ELF file, loops over all symbols (except for symbol number 0, which is a dummy symbol) in its symbol table SYMTAB1 (except for the first symbol, which is a dummy null symbol) and for each symbol sym:

* If sym is UNDEFINED, search for sym in SYMTAB2, the symbol table of the other ELF file. If not found in SYMTAB2, or found in SYMTAB2 but UNDEFINED there as well, print an error message: "Symbol sym undefined".
* If sym is defined, i.e. has a valid section number, again search for sym in SYMTAB2. In this case, if sym is found in SYMTAB2 amd is defined, print an error message: "Symbol sym multiply defined".

Continue scanning symbols even after errors were found. Note that using "F1a.o" and "F2a.o" should result in no errors, but using "F1b.o" and "F2b.o" should result in errors.

**Part 3.2: Merge ELF Files**

In a real linkage editor, this is done only if checking for merge passes with no errors. However, your implementation will allow this even if the previous step found errors. Here you should write a Merge function, which does the following:

* Create a new ELF file called "out.ro".
* Create an ELF header for this file and write it into the file "out.ro".
* Create a new section header table for this file.
* Create merged sections.
* Write the merged sections and the new section header table into the file "out.ro".
* Close the file "out.ro". (You may or may not need to update the ELF header before you close the file, depending on your implementation.)

To implement the above, you should follow the procedure stated for pass I in the lecture. However, below are some simplifying assumptions and restrictions that should make implementation easier. So you may assume all the following restrictions:

* There are exactly 2 ELF files being merged.
* The second ELF file does not contain any sections that do not exist in the first ELF file.
* The second ELF file does not contain any symbola that do not exist in the first ELF file.
* The second ELF file does mot have any relocations.

With all the above simplifying assumptions, the following method can be used for merging.

* Use a copy of the ELF header of the first file as the ELF header for the merged file. You need to modify only the "e\_shoff" field, as specified below.
* Use a copy of the section header table of the first ELF file as an initial version of the section header table for the merged file. You will need to modify the "sh\_off" and "sh\_size" fields in each section header, as specified below.
* Mergable sections: ".text" ".data", ".rodata" sre merged as follows. Concatenate the section contents from the first ELF file and the section contents from the second ELF file, to create the merged appropriate section. E.g. to create the merged ".text" section, copy the contents of ".text" from the first ELF file, and to that append the contents of ".text" from the second ELF file. The merged section obviously has a size that is the sum of the sizes (should accordingly change the appropriate section header).
* Section ".shstrtab" of the first ELF file can be used without changes as the ".shstrtab" of the merged file, and the same for ".symtab" and any relocation sections (because of the restricting assumptions above).
* The symbola table of the first ELF file can be used as that of the merged file, after copying over symbol values and definition of symbols from the second ELF file, for every symbol that is UNDEFINED in the first ELF file.

A practical implementation of the above then looks as follows:

* Create "out.ro" and copy an initial version of the ELF header as its header.
* Create (in memory) an initial version of the section header table for the merged file by copying that of the first ELF file.
* Loop over the entries of the new section header table, and process each section according to the above scheme (concatenate ".text", copy ".shstrtab" as-is, etc.), and immediately write (append) the appropriate merged section to "out.ro". Note that to concatenate e.g. ".text", simply write the contents of ".text" from the first ELF file, at the end of "out.ro", then find the contents of ".text" in the second ELF file (by finding ".text" in its section header table) and write it again at the end of "out.ro" (no need to merge them in memory!) So now you know the file offset of the merged section and its length, so update the appropriate section header table entry fields (offset and size).
* Write the merged (and modified) section header table entry, appended to the end of "out.ro".
* Fix the "e\_shoff" field in ELF header of "out.ro" to point to the location where you actually wrote the section header table, and close "out.ro".

The basic requirement here is that "readelf" on "out.ro" will show that all sections have been merged as stated above, and that the symbols have been resolved, when operating on pairs of ELF files that obey the restrictions stated above. For a bonus, running ld (the linker pass II) on "out.ro" should work correctly in these cases, and generate an executable file that runs correctly!

**Deliverables:**

You need to submit "myELF.c" and a makefile which compiles and links your code.

LAB\_5

**Lab 5 (C) - Loader**

**Introduction**

In this lab you will implement a static loader. Your loader will be able to load (run) static executable files. These are executable files that do not use dynamic library code. In particular, your loader will be able to load your code which uses the system\_call interface, and no standard libraries. **Note**: Remember to compile your code with the -m32 flag.

**Task 0**

Write a program, which gets a single command line argument. The argument will be the file name of a 32bit ELF formatted executable.

Your task is to write an iterator over program headers in the file. Implement a function with the following signature:  
int foreach\_phdr(void \*map\_start, void (\*func)(Elf32\_Phdr \*,int), int arg);

The function arguments are:

1. map\_start: The address in virtual memory the executable is mapped to.
2. func: the function which will be applied to each Phdr.
3. arg: an additional argument to be passed to func, for later use (not used in this task).

This function will apply func to each program header.

Verify that your iterator works by applying it to an 32bit ELF file, with a function that prints out a message: "Program header number i at address x" for each program header i it visits.

**Task 1**

**Task 1a:**

In this task you will use the iterator you created in Task 0, and implement the *readelf -l* functionality.

Using the functions from task 0 (the iterator), your task is to go over the program headers in a file and for each header, print all the information which resides in the corresponding Elf32\_Phdr structure.

The output should look similar to *readelf -l*:

Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align  
PHDR 0x000034 0x04048034 0x04048034 0x00100 0x00100 R E 0x4  
INTERP 0x000134 0x04048134 0x04048134 0x00013 0x00013 R 0x1  
LOAD 0x000000 0x04048000 0x04048000 0x008a4 0x008a4 R E 0x1000  
LOAD 0x0008a4 0x040498a4 0x040498a4 0x0011c 0x00120 RW 0x1000  
DYNAMIC 0x0008b0 0x040498b0 0x040498b0 0x000c8 0x000c8 RW 0x4  
NOTE 0x000148 0x04048148 0x04048148 0x00020 0x00020 R 0x4

**Task 1b:**

Note that tasks 0 and 1 should work on any Linux 32 bit ELF file whatsoever, as at this point we are merely looking at the program headers (empty if the ELF is non-executable, of course). But starting from task 2, we actually attempt to load and execute, so there you only need to support static linked files, i.e. without program headers of type DYNAMIC or INTERP.

In the next task we will map the needed segments to memory, so we need to prepare the appropriate data for mapping. The loader uses the Program Headers to find the information needed for loading the program to memory and uses mmap system call for actually mapping the needed segments to process's memory. Specifically, it is VERY important to pass mmap the appropriate protection and mapping flags.

For each program header that should be mapped to memory, in addition to the information in task 1a, print the appropriate protection flags and mapping flags that should be passed to the mmap function for this header. **Note**: The protection flags used by mmap, though similar to the flags in the program header, differ in bit position!

**Task 2**

Now you will write the actual loader, using the iterator from task 1. The loader is mainly a single function, which maps each relevant chunk of the executable into memory. In the end, you should produce an executable program, which receives one command-line argument (an ELF executable), loads it in to memory and passes control to the loaded code.

**Note**: In the auxiliary files you may find a file named [workingLoader](https://moodle.bgu.ac.il/moodle/pluginfile.php/3793911/mod_folder/content/0/workingLoader?forcedownload=1). That executable file is an official solution to this task. You may use it to test that your loader outputs the appropriate results.

**Be careful!** Do not take this executable and present it as your solution. We've put in a print to let the instructors know it is the official solution.

**Task 2a**

Recall that in order to actually load the files using mmap as required, you need to obey the instructions in the reading material so as to avoid memory space clashes. Download the linking\_script file, and compile your code as shown in the reading material. Note that we link the program without using any standard libraries, but the loaded program still needs to get a specific stack before it starts executing. Also, we still want the ability to use the system\_call interface (what the "glue" code file [start.s](https://moodle.bgu.ac.il/moodle/pluginfile.php/3793911/mod_folder/content/0/start.s?forcedownload=1) from lab 4 was about). So you have to link your code together with start.o (you do not need startup.o yet). Verify linking worked properly using *readelf -h loader*

You should be able to explain to the lab instructor why the linking script is needed and how you verified that it worked.

**Task 2b**

Implement the following function:  
void load\_phdr(Elf32\_Phdr \*phdr, int fd);

This function takes two arguments, a pointer to the Phdr struct and the file descriptor of the executable file. It should map each Phdr that has the PT\_LOAD flag set, into memory, starting from the specified offset, and place it at the virtual address stated in the Phdr. Each map should be according to the flags set in the Phdr struct. In addition, this function should print to the screen the information about each program header it maps (you can use the function from Task 1 to print the information).

Recommended operating procedure: make sure system calls succeed before proceeding, most especially mmap.

**Task 2c**

After successfully completing the previous function, you should now pass control to the loaded program. To achieve this, we provide the code in assembly language (startup.s), you should examine the code we provide. You may download its object file [startup.o](https://moodle.bgu.ac.il/moodle/pluginfile.php/3793911/mod_folder/content/0/startup.o?forcedownload=1). You need to execute the loaded program using our function startup(), with the following signature:  
  
int startup(int argc, char \*\*argv, void (\*start)());  
  
and start is the entry point of your executable.

Your loader should be able to load and run all code from previous labs which uses the system\_call interface, provided that they are compiled with the -m32 flag and according to the compilation instructions in the system calls lab.

However, first try it for a program that does not expect command-line arguments, such as this file: [loadme](https://moodle.bgu.ac.il/moodle/pluginfile.php/3793911/mod_folder/content/0/loadme?forcedownload=1). In case of a bug note the following:

* The "arg" of foreach\_phdr is used correctly - it should be the file descriptor of an open file.
* The file/s that the segments are mapped from must remain open and mapped to memory.
* The correct segments are mapped to memory and flags are set appropriately.
* startup is used correctly as following:  
  startup(argc-1, argv+1, (void \*)(elf\_head->e\_entry))

**Task 2d**

Now, we assemble the command line arguments and pass them to the loaded program. A command line looks like:  
  
my\_loader my\_test\_program arg1 arg2 ...

Where *my\_loader* is the executable program you implemented in this lab, *my\_test\_program* is the program you are trying to load and run, and arg1, arg2, etc. are the command-line arguments that *my\_test\_program* should see. These are to be passed to it using the startup function.

**Deliverables**

This lab is not too much work ASSUMING you read the reading material and did task 0 before the lab. So you should complete everything from task 1 to task 2d during lab hours. having to do completion labs next week.

**Submission:** Inside a zip file: task2.c , makefile.