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
2	16	The virus name represented
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 (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