ESPLab B 2022

Lab 3

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

Lab 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.

You should use `valgrind` in the following manner: `valgrind --leak-check=full [your-program] [your-program-options]`

Task 0: Memory Leaks, Segmentation Faults, and Printing data from files in hexadecimal format

(To be done before attending the lab session)

Task 0a: Using `valgrind` to detect memory leaks

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, mergesort.c, is provided. The program should sort numbers specified in the command line and print the sorted numbers, like this:

```
$ mergesort 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). Another illegal memory access causes the program to print the result wrong. Moreover, the program has a few memory leaks.

First, solve the segfault using `gdb` (or just by reading the code). You should also read the code to find the source of the wrong printing. Then use `valgrind` to find the memory leaks and fix them.

Task 0b: Printing data from files in hexadecimal format

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 hexadecimal 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 (click to download):

```
#>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 lab, so make sure it is well-written and debugged.

Lab Instructions

Lab 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

DESCRIPTION

virusDetector compares the content of a file whose path is given in `stdin` byte-by-byte with a pre-defined set of viruses described in the signatures file.

Task 1: Virus detector using Linked Lists

(To be done during the lab session)

In the current task, 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 you will not yet need to read the suspected file's path. At a later stage (task 1c) you will compare the virus signatures from the list to byte sequences from a suspected file, whose name you will read from `stdin`.

Task 1a - Reading a binary file into memory buffers

The signatures file contains details of different viruses in a specific format. It consists of blocks ($\langle N, signature, name \rangle$) where each block represents a single virus description.

The name of the virus is a null-terminated string that is stored in 16 bytes. If the length of the actual name is smaller than 16, then the rest of the bytes are padded with null characters.

Notice the format is little-endian - the **numbers** (i.e. the length of the virus) are represented in little-endian order.

The layout of each block is as follows:

Block Layout

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

For example, the following hexadecimal signature:

```
05 00 31 32 33 34 35 56 49 52 55 53 00 00 00 00 00 00 00 00 00 00 00 00
```

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;
    unsigned char* sig;
    char virusName[16];
} virus;
```

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

- `void readVirus(virus* vir, FILE* input)`: this function receives a pointer to a virus struct and a file pointer and overwrites the given virus struct so that it represents the next virus in the file.
  To read from a file, use `fread()`. See man `fread(3)` for assistance.
- `void printVirus(virus* vir, 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 implement the auxiliary functions, implement the following two steps:

- Open the signatures file, use `readVirus` to read the viruses one-by-one, and use `printVirus` 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 lab3_out file.

Task 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* output);
     /* 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, link* to_add);
     /* Add the given link to the list
        (either at the end or the beginning, depending on what your TA tells you),
        and return a pointer to the list (i.e., the first link in the list).
        If the list is null - return the given 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 2.

```
1) Load signatures
2) Print signatures
3) 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 was loaded nothing is printed. You should read the user's input using `fgets` and `sscanf`.

Test yourself by:

- Read the signatures of the viruses into buffers in memory.
- Creates a linked list that contains all the viruses where each node represents a single virus.
- Prints the content. Here's an example output: lab3_out.

Task 1c - Detecting the virus

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

- Extend the prompt presented to the user as follows:
  `1) Load signatures`
  `2) Print signatures`
  `3) Detect viruses`
  `4) Quit`
  `Detect viruses` operates after the user runs it by entering "3".
- Open the file whose path is given in `stdin`, 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:

`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 signatures file.

Task 2: Anti-virus Simulation

(To be done during the lab session; can be done in a completion lab)

In this task, you will test your virus detector and use it to help remove viruses from a file. You are required to apply your virus detector to an infected file, which is infected by a 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.

Task 2a: Using `hexedit`.

After making sure that your virus detector program from task 1 can correctly detect the virus information, you are required to:

1. Download the infected file (click to download).
2. Set the file permissions (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 virus's location and cancel its effect by replacing all virus code with NOP instructions.

Task 2b: Killing the virus automatically.

Implement the functionality described above, as follows:

- Extend the prompt presented to the user as follows:
  ```
  1) Load signatures
  2) Print signatures
  3) Detect viruses
  4) Fix file
  5) Quit
  ```
- `Fix file` will request the user to enter the starting byte location in the suspected file (again the one given in `stdin`) and the size of the virus signature.
- The fix will be done with the following function:
  `void kill_virus(char *fileName, int signatureOffset, int signatureSize)`
- Hint: use `fseek()`, `fwrite()`

Deliverables

As for all labs, you should complete task 0 before attending the lab session. Tasks 1a, 1b, and 1c need to be done during the lab. Task 2 may be done in a completion lab.

The deliverables must be submitted by the end of the lab session.

You must submit a zip file that contains two folders. The first folder must be named t1 and contains the source file of task 1c that must be named **task1c.c**, and a makefile. The second folder must be named t2 and contains the source file of task 2b, which must be named **task2b.c**, and a makefile. That is, must be organized in the following tree structure (where '+' represents a folder and '-'

represents a file):

```
+ t1
   - makefile
   - task1c.c
+ t2
   - makefile
   - task2b.c
```

Submission instructions

- Create a zip file with the relevant files (student_id.ZIP).
- Upload zip file to the submission moodle page.
- Download the zip file from the same moodle page and extract its content to an empty folder.
- Compile and test the code to make sure that it still works.

Last modified: Wednesday, 11 May 2022, 11:22 PM

Administration

› Course administration

◄ Files

Jump to...

Lab 3: Reading Material ►