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: it prepares the stack so as to have argc and argv in the format expected by main(). 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 can be used to get a direct system call without requiring you to write in assembly language. That is, it receives arguments on the stack in CDECL format, places them in the appropriate registers and performs the system call. You can either use it if you wish, but it is better to look at this as an example how "arguments" to system calls are placed in registers before the INT 0x80 instruction actually executing the system call. 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 A

Make sure you have a **good** implementation in C of the encoder from lab A, 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 A 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 (file descriptor or error), which is in the EAX register, 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 a given file. Note that here you will be writing partly in C and partly in assembly language, so your program code will consist of 2 files: main.c, and (an extended version) of start.s, beginning with the start.s we provide.

DESCRIPTION
Your program receives a command line argument, which includes the name of a file:

COMMAND-LINE ARGUMENT
-a{file} - attach the executable code to the given file.
Attach the executable code (be discussed more below) at the end of given file name. After attaching the code, the program should print the message **"VIRUS ATTACHED"** after the file name.

Some guidelines

1. In case of an error, the program should terminate with exit code 0x55.
2. Do 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: Parse and print the file name

Write the function main( ) in C: retrieve and print the file name given in the command line to the screen, and then call the functions "infection( )" and "infector(filename)", with filename as given in the command-line argument.

Both these functions are to be written in assembly language, at this point make them "stubs": both just consist of the "ret" instruction. Now, compile, link, and test your code.

Task 2.B: attaching the virus

Now implement the "-a" option to attach the virus. **Warning:** You probably want to be very sure that the mechanism for determining the file name 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 (all in assembly language).

1. Starting assembly language implementation: begin with a label "code_start".
2. Write the code for 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 does the following, in this order:
    1. Prints the file name given as its argument
    2. Opens the file named in its argument
    3. Adds the executable code from "code_start" to "code_end" after the end of that file
    4. 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".

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 executable files from your previous lab solutions as input. Use the command **chmod u+wx {filename}** to give user write/execute permissions if needed. Use hexedit to check that your infector code actually works (how?).

Submission

Task 1 and Task 2 are mandatory for this 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