# Lab 4: System Calls

## Lab Goals

- To get acquainted with the low-level interface to system calls.
- To understand how programs can work without the standard library.
- Basics of directory listings, through a first attempt at executable file viruses.
- Debugging programs via printouts (the idea of debug mode).

As usual, you should read and understand the reading material and complete task 0 before attending the lab. To be eligible for a full grade, you must complete at least tasks 1a, 1b, 2a and 2b during the lab. Task 2c may be done in a make-up lab if you run out of time.

In this lab you will use the following system calls: open, close, read, write, lseek, exit, and getdents which you can read more about in the Reading Material.

For the entire lab, do not use the standard library!
Also, do not include stdio.h, stdlib.h, or any other "standard" external header files, although you can include your own header files, if any.
This also means that you cannot use any library functions like printf, fopen, fgetc, strcmp, etc.

## Task 0: Using nasm, ld and writing the patch program

**Task 0 is crucial for the successful completion of this lab! make sure you finish it and understand it before your lab session.**
**File descriptor numbers:**
stdin 0
stdout 1
stderr 2

### Task 0a: a trivial program using only system calls

We will build a program which prints its arguments to standard output without using the standard C library.

1. Download start.s, main.c, util.c, and util.h.
2. Compile and link them without using stdlib (the C standard library) as follows:
   - Assemble the glue code:

     ```
     nasm -f elf 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. Run the program several times, each with a different number of arguments, and observe the results.
4. Look at the source code of the files, and **make sure you understand it**. In particular, if you are taking the "architecture and assembly language" course you are expected to understand the role of every instruction in start.s, and if not you are still expected to understand what this code does (see basic explanation below and comments in start.s).
5. Write a makefile to perform the compilation steps automatically.
6. Write a new main.c that prints "hello world", or some other message of your choice, to standard output, again **not** using stdlib, using the scheme explained above, and test it.

#### 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 0b: Patching executable files

This is a preliminary exercise in the open, close, read, write and lseek system calls that you will need to use in the lab.

Download the executable file used in this task greeting and use the command `chmod +x greeting` to be able to execute the file.

Shira is very enthusiastic about Dan's upcoming birthday – her boyfriend. She wants to make him something special. Since she's studying to become a programmer, she wants to write a program to print out delightful things for her boyfriend when he runs it - sort of a birthday card. For this, she sat days and nights and made tens of sketches of how the card should look like, and what it should contain; then she wrote a program in the C programming language.

Mira, Dan's ex-girlfriend, knows Dan's email password. She logged into his email and saw the email Shira sent. Mira got very jealous of Shira and her great idea. She plotted a plan! She wanted to replace Shira's name, with her own name in the program. Unfortunately, she does not have the source code, but only the compiled program. So she comes to you in despair and asks you to write a program which receives a name, and replaces Shira's name with it.

In this task we will implement the **patch** program:

SYNOPSIS
patch *FILE_NAME X_NAME*
DESCRIPTION
Changes the file *FILE_NAME*, so that it would print *X_NAME* instead of Shira's name.

### Some Guidelines

- In case of any error, the program should terminate with exit code 0x55.
- You may want to use the **sys_lseek [19]** system call.
- Note that the file on which you are operating is known in advance, so you also know its size, but you can also use lseek to find this.
- When using the `open(const char *pathname, int flags, mode_t mode)` system call, you might want to set mode to **0**777 or **0**644. In this case, the value will be ignored because no new file is being created.
  The "mode" argument will make more sense after the lecture on file permissions in Unix (just before lab5).
- Use hexedit in order to find the address you should patch.

Once again, remember not to use any standard library functions – only the system_call function provided.

## Task 1: The encoder program

In this task we will write a simple version of the encoder program from Lab 1 without using standard library functions. Note that we are adding a **debug mode** option, in order to help you debug your code under developement. Debug modes typically are available in a program at all levels of development and sometimes even after delivery. This is done here so that you will get used to the idea and consider using it in **all** your code henceforth.

## Task 1a: A restricted encoder version

### A reminder: the encoder program

NAME
encoder - encodes the input text as lowercase letters
SYNOPSIS
encoder
DESCRIPTION
encoder receives text characters from standard input and prints the corresponding lowercase characters to the standard output. Non uppercase letters remain unchanged.
OPTIONS
**-d**
Activate debug mode (in which debugging messages are printed to stderr).
EXAMPLES

```
#> encoder
Hi, my name is Noah
hi, my name is noa
^D
#>
```

### Comments and tips

1. **Note that there's now a new option flag (-d)**. This flag enables the debug prints (see next section).
2. This program should use the system_call function from start.s and should not use any stdlib functions (like printf, fopen, etc.).
3. Make sure you compile and link in the same way as in task0. Feel free to adapt the makefile you wrote in task 0.
4. It is highly advisable **for this lab task** to read and write one character at a time (as opposed to using a larger buffer), this way the program is much simpler. However, note that this implementation is **inefficient** and should not be used in production code due to a large number of system calls that could have been minimized by using a buffer.
5. You may or may not want to download the work you submitted at Lab 1 from the submission system and adept it to the new requirements.

6. For more information about the requirements of the encoder program you can refer to Lab 1.

### Debug prints

When (and only when) the debug flag (-d) is given, your program should add the following prints to **stderr**:

- For each system call (except for "exit"), its ID (the value of the first argument) and its return code.
- Whatever else you think will help you.

## Task 1b: Working with files

NAME
encoder - encoders the input text as lowercase encrypted letters.
SYNOPSIS
encoder [OPTION]…
DESCRIPTION
encoder receives text characters from standard input or from a file and prints the corresponding lowercase encrypted characters to the standard output.

If option '-i' is given, encoder reads its input from a file, if it is not, encoder reads its input from standard input.
If option '-o' is given, encoder writes its output to a file, if it is not, encoder writes to standard output.

OPTIONS
-d
Activate debug prints.
-i FILE
Input file. Read input from a file, instead of the standard input.
-o FILE
Output file. Outputs to a file, instead of the standard output.
EXAMPLES

```
#> echo "Hi, my name is Noah" > in
#> ./encoder -i in -o out
#> cat out
hi, my name is noa
```

### Comments and tips

1. See comments and tips from Task 1a.

2. Make sure to check if there is an error when opening files, your program should print an appropriate error message and exit in the case of an error.
3. If you'll read the whole filename at once you might need to remove some characters from the end of the strings. Ask yourself, Why are they there? Knowing how strings are represented in C, how can you easily remove any suffix from them?

### Debug prints

In addition to all the debug prints listed in previous tasks, add these prints:

- The input and output file paths (print "stdin" or "stdout" if -i or -o were not used).
- The ID and return code for all added system calls.

## Task 2: First step towards Flame 2

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 code to files in the current directory.

In the following tasks you will implement the `flame2` program:

**SYNOPSIS**

**flame2** OPTION

**DESCRIPTION**

Print a comment line of your choice (such as "Flame 2 strikes!"), and then list all the file names in the current directory.

**OPTIONS**

**-d**

Activate debug prints

**-s <suffix>**

Instead of printing all the file names, print only the names of the files in the current directory, that end with the single character *<suffix>*

**-a <suffix>**

Attach the executable code of flame2 at the end of each file in the current directory, that end with the single character *<suffix>.*

## Task 2a: A restricted flame2 version: printing a list of all files

A restricted version of flame2 is implemented, as follows:

- Print the names of all files in the current directory.

### 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 (can be found in /usr/include/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. Dont forget not to use any standard library functions!

### Debug prints

- As before, all return codes and arguments sent to system calls.
- The length and name of each dirent record.

## Task 2b: Extending flame2: printing a list of all files with a given suffix

Extend `flame2`, which is implemented in Task 2a in the following way:
When the flag `-s <suffix>` is supplied, it will only print the names of the files in the current directory that end with `suffix`.
The suffix parameter will contain a single character.

Consult the **sys_getdents [141]** manual page for a way to check the type of the file.

## Task 2c: Extending flame2: Add the executable code of flame2 to each file obtained from Task2b

This task may be done in a completion lab **if you run out of time** during the regular lab.

The implementation requirements for this task depend on which course you're taking. Students who take the "Computer Architecture and System Programming Laboratory" course should do steps 1 through 4 in assembly language, while students who take the "System Programming Laboratory" course should implement the equivalent code in C.

**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. For example using the 'c' suffix will infect all the c files in the directory, so be careful **not** to destroy your own source code files!

The following contains code you need to write in **assembly language**. People doing only SPlab should implement this part in C.

Extend `flame2`, which is implemented in Task 2b in the following way:

1. Starting assembly language implementation: begin with a label "code_start".
2. Write a function `void infection(void)` that prints to the screen the message "Hello, Infected File". **Note**: this should be done using just **one**system call! If you have more than 15 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 assembly language part with a label "code_end".
5. The rest of the task below can be done in C (recommended). Print both the addresses of `code_start` and `code_end` to the screen, infer from this information how to reach to the executable code of the function `infector` and the other functions you need to copy to the output file.
6. Modify flame2 such that when the flag `-a <suffix>` is supplied, it will call the infection and infector functions in order to add the executable code of `infection` to the end of each file in the current directory (i.e. all the files obtained in a manner similar to Task2b). Also, this option will print out the names of these files like the "-s" option. For simplicity, you may assume that only one of the options "-a", "-s" will be set in any single run of the program.

   **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().

   Make sure you understand the part of the code that follows the "system_call:" label in file "start.s". You are encouraged to work with an appropriate registers in order to implement all the file handling issues.

After completing the task, execute one of the files that was "infected" by flame2 and see what happens: does the virus really work?

1. Does it attach its code and infect the executable file(s)?
2. Does it run when the infected is executed (if not, why not?)

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.

### Debug prints

- Again, all return codes and arguments sent to system calls.

## Deliverables:

Tasks until 2b must be completed during the regular lab. Task 2c 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 day.
You must submit source files for task 1b task 2b and task 2c and also a makefile that compiles them. The source files must be named **task1b.c**
**task2b.c**, **task2c.c** and **start.s** and the makefiles
named **makefile1b**, **makefile2b** and **makefile2c**.