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 use of standard library.
- To learn about the structure of a directory, and how to traverse it.

(This lab is to be done SOLO)

As usual, you should read and understand the reading material and complete task 0 before attending the lab. To be eligible for a perfect grade, you must complete at least tasks 1, 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 (except for task 0b), do not use the standard library!
Also, do not include stdio.h, stdlib.h, or any other "standard" external header files, although you may include your own header files, in case you have written 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.

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 lab4_start.s, lab4_main.c, lab4_util.c, and lab4_util.h.
2. Compile and link them without using stdlib (the C standard library) as follows:
   - Assemble the glue code:

     ```
     nasm -f elf64 lab4_start.s -o lab4_start.o
     ```

   - Compile the main.c and util.c files into object code files:

     ```
     gcc -m64 -Wall -ansi -c -nostdlib -fno-stack-protector lab4_util.c -o lab4_util.o
     gcc -m64 -Wall -ansi -c -nostdlib -fno-stack-protector lab4_main.c -o lab4_main.o
     ```

   - Link everything together:

     ```
     ld -m elf_x86_64 lab4_start.o lab4_main.o lab4_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 it assembly language.

Note that you can link files written in different languages: the object file format is a common format used by many languages and compilers. This means that object files can be linked together (to form a library or executable) even if they come from source code written in different languages and compiled by different compilers.

Task 0b: Discovering the file descriptors associated with stdin, stdout, and stderr

In this task, you need to find the number of the file descriptors associated with the standard input (stdin), the standard output (stdout), and the standard error stream (stderr). You will use these numbers in the rest of this lab; so recall them. To achieve this, do the following steps:

1. Look at /usr/include/stdio.h and search for the type of these variables (stdin, stdout, and stderr).
2. Look at /usr/include/libio.h and search for the definition of this type. Identify which filed in this struct holds the file descriptor.
3. Include stdio.h in your code.
4. Since C has no protection for struct members, you can then write a brief C program, which prints the values of the fields in the struct. Write a C program that prints the file descriptor of stdin, stdout, and stderr. As follows:
    a. Write the macro PRINT_FD(stdio) which prints the file descriptor of a given stream (stdio).
    b. In the main function, use the macro PRINT_FD to print the file descriptors of stdin, stdout, and stderr.

Task 1: 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.
Consider the following scenario:

Shira is very enthusiastic about her boy friend Dan's upcoming birthday. 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 like 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 the password to Dan’s email account. 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 that receives a string, and replaces Shira's name with the given string.

In this task you 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** 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 2: search program

In this section, you will practice how to search and display the files in your system.

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

SYNOPSIS
search OPTION

DESCRIPTION
List all the relative paths of the files in the working directory.

OPTIONS

-n <name>
Instead of listing all the files, list only the files named <*name*>

-e <name> <command>
Execute <*command*> on each file with name <*name*> .

Task 2a: restricted search

In this task you are required to list the relative path from the current directory of all the files in the working directory. That is, list all the files from the working directory and its sub-direcotireis and so on with their relative path.

EXAMPLES

```
#>./search
.
./DIR2
./DIR2/D2_DIR3
./DIR2/D2_DIR3/Dolphine
./DIR2/D2_DIR3/Ostritch
./DIR2/D2_DIR3/Pigeon
./DIR2/D2_DIR3/Cat
```

```
./DIR2/Dolphine
./DIR2/Pigeon
./DIR2/Cat
./DIR2/D2_DIR1
./DIR2/D2_DIR1/Dolphine
./DIR2/D2_DIR1/Tiger
./DIR2/D2_DIR1/Lizard
./DIR2/Dog
./DIR2/D2_DIR2
./DIR2/D2_DIR2/Monkey
./DIR2/D2_DIR2/D2_D2_DIR1
./DIR2/D2_DIR2/D2_D2_DIR1/D2_D2_D1_DIR1
./DIR2/D2_DIR2/D2_D2_DIR1/D2_D2_D1_DIR1/Tiger
.....................................
```

Some Guidelines

1. Your program should use the **sys_getdents** 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. Initially, it should be for the file `"."` that represents the current directory.
4. Search all the files in the directory (and its sub-directories) with this specific name, append to each file its relative path.
5. In case of an error, the program should terminate with exit code 0x55.
6. Don't forget not to use any standard library functions!

Task 2b: search specific files

Extend search, which is implemented in Task 2a, in the following way: When the flag `-n <name>` is supplied, it will only print the names of the files (with their relative paths) with this specific name. Similar to the Linux command command:

```
find . -name "filename"
```

EXAMPLES

```
#>./search -n Lion
./DIR2/D2_DIR2/D2_D2_DIR1/D2_D2_D1_DIR2/D2_D2_D1_D2_DIR1/Lion
./DIR4/D4_DIR1/Lion
./DIR4/D4_DIR2/Lion
./DIR4/D4_DIR3/Lion
./DIR3/Lion
./DIR1/D1_DIR2/D1_D2_DIR1/Lion
```

Task 2c: Extending Search: Execute commands on files

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

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.

In this task, you will extend your program by adding the ability to execute commands on files with a specific name. Do the following steps:

1. Add the option `-e <name> <command>` .
2. Append to the command the relative path of each file, and execute it using the function *simple_system* from lab4_util.c. *simple_system* is a simple implementation of the *system* stdlib

function (read about it using "man 3 system"). Fin). Example of the command *cat* when applied to a file: `cat file.txt`
The file can exist or not.

EXAMPLES

In directory : user@ubuntu:~/Desktop/WorkingDirectory/DIR2/D2_DIR2/ There is txt file "TxtFile" (user@ubuntu:~/Desktop/WorkingDirectory/DIR2/D2_DIR2/TxtFile) with content:
Hello everyone!
It's me.
I would like to see you.

```
user@ubuntu:~/Desktop/WorkingDirectory$ ./search -e TxtFile cat
Hello everyone!
It's me.
I would like to see you.
user@ubuntu:~/Desktop/WorkingDirectory$ ./search -e NotExistingFile cat
The file 'NotExistingFile' Does not exist.
```

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 1 task 2b and task 2c and also a makefile that compiles them. The source files must be named **task1.c task2b.c**, **task2c.c** and **start.s** and the makefiles named **makefile1**, **makefile2b** and **makefile2c**.

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.