CS2106: Lab 2 (Graded) - Understanding System Calls in Linux

Lab 2 is graded and must be run on Linux using Ubuntu 16.04.X (the lab PCs are using 16.04.1). This is because different versions of Linux could possibly give different answers.

In the command line examples for Labs, the '`$`' is used to indicate the shell prompt. Your actual prompt may differ (and can be changed). I will usually use `teletype` font for command lines, C code, system calls, functions and programs.

After this lab, you will appreciate better what actually happens when a process runs (with both static and dynamic linked executables). You will get to understand the connection between the C code and the system calls. You will also see how `strace` can be used to investigate a program and do some debugging.

Remark: You should treat every lab as having something to learn. In many labs, also something to "play with". By "playing", I mean that you can experiment with the programs or methodology in the lab. For example, in Ex1 you can modify the program and observe the result of `strace`, you are not restricted to just what is stated in the assignment. Obviously submission should be based on the lab itself.

1 Exercise 1: System Call Tracing (Graded)

Goals: Learn how to use `strace` to examine the system calls generated by a running process

On Linux, the `strace` utility can show you ALL the system calls made by a program. First, read the `strace` man page. Its long enough that you might not immediately understand the whole man page, however, it will make more sense once you have started using `strace`.

It will also be useful to read some other man pages. This is because given the interconnected nature of OS, some system calls may be from topics which we haven't covered yet. As such, the man pages will give you an idea what is going on. Take note that the actual system calls used can vary due to differences in versions of Linux and C libraries, for example `exit()` can vary between Unixes.

You will see also that since `strace` shows the low level system calls, the precise behavior may be somewhat different from what the higher level documentation says. You may find that `strace` output looks "*complex*" — this is because it is trying to show you what is happening at the system call interface level, i.e. *doesn't try to hide what is going on.* This of course also can make it more difficult to understand but a large part of the OS behavior of a process can be understood simply by looking at the trace. To simplify matters, we will ignore some portions of the `strace` output.

First, compile `lab2-hello.c` program to the executable `lab2-hello`. To simplify your initial system call tracing experiment, we will make a statically linked executable which does not need dynamic linking (eg. DLLs, in Windows these are files with the extension `dll`, while in Linux these have filename suffix `so`)). (Static linking means that the libraries are incorporated into the executable itself while dynamic linking will link at runtime the executable with the dynamic libraries used).

```
$ gcc -static -o lab2-hello lab2-hello.c
```

The use of `-static` tells `gcc` to use static rather than dynamic linking to create the executable. For this lab, static linking will reduce the amount of output since dynamic linking will introduce extra system calls (we may discuss dynamic linking when we deal with the memory section of the course).

You should first understand what `lab2-hello.c` is doing. It uses `scanf()` to read input into a string, `printf()` for formatted output, and `getpid()` to find the process identifier (pid).

We can now trace all the system calls made by `lab2-hello` as follows:

```
$ strace ./lab2-hello
```

The output from `strace` is quite verbose. It consists of the `strace` output (a trace of the system calls) and any output from the process being straced. Both kinds of output are mingled together. To make it easier to understand and analyze, it is useful to save the output from `strace` into some files. `strace` sends the output of the program into the stream which is called standard output (`stdout` for the C `stdio` library) and the output from trace goes to standard error (`stderr` also defined by `stdio`). Standard error is where error messages are normally sent to in Unix, and `strace` sends its own output to `stderr`. When you run in the shell (terminal) then the output from the program running and the strace output is mixed together, sometimes, this can seem a bit confusing.

Using the `bash` command shell, you can save the output from `strace` as follows:

```
$ strace ./lab2-hello 2>strace.txt
```

Check the file `strace.txt` to see what it contains, e.g. `less strace.txt`. The syntax with (`>`) sends the output of `strace` which goes to file descriptor 2 into the file `strace.txt`. This is called *I/O redirection*. Later, when we cover files in the last part of the course, we will examine how I/O redirection works.[1]

You can also save the process output into another file. An example with both `stdout` and `stderr` streams going into separate files is: (there are 2 I/O redirections below)

```
$ strace ./lab2-hello 2>strace-output.txt >lab2-hello-output.txt
```

You should also read the man page for `strace` before and after Exercise 1. While it is not necessary to understand all the options, it is useful to understand what `strace` is doing and we will need some of those options for later.

HINTs:
All system calls are documented in the Linux manual, e.g. `man 2 read`. The first `execve()` is from running the executable itself. If you get confused by which system call is coming from where, you can modify the C program just for your own testing but please submit your answer using the original code.

1.1 Question 1(a)

For lines labelled as LINE 4-LINE 8 in the C program, correlate it with the system calls from `strace` by explaining which system calls you think come from which line of the program. You don't need to explain what they do, which is Question 1(b). To simplify the lab, you can ignore the following system calls: brk, readlink, access, fstat, lseek.

[1] Of course, `strace` can itself be used to investigate I/O redirection.

1.2 Question 1(b)

Try to explain for LINE 4-8 why you think those system calls occurred in the trace. Just make your best attempt (reading the man pages may help). You do not need to know precisely what is happening but as this is a very simple program, you should be able to have a good guess what it's doing.

1.3 Question 1(c)

Instead of compiling your executable to be linked statically, now compile it with dynamic linking (which is the default with `gcc`).

```
$ gcc -o lab2-hello lab2-hello.c
```

Briefly describe the difference in the trace with part 1(a). Explain whether a dynamic library is being used and what is the pathname. You do not need to understand what is going on with the dynamically linked executable but just that it changes the behavior of the program.

In the remainder of Question 1, you can use the static linked version.

1.4 Question 1(d)

What happens when LINE 9 is replaced by LINE 10 (just comment LINE 9 and uncomment LINE 10). You can see that LINE 6, `exit()` rather than `_exit()`. Read the man page to see what the system call in the trace does. (Some remarks on threads: as we do not go into the details of *threads*, you can think of threads as something like a sub-process within a process. To be more precise, a thread is like another process which shares memory (rather than a copy as in `fork()`) and OS context with the parent).

Try the following three variations using `strace` to see what system call is used to terminate the process:

- the original: using "`exit(0)`" (LINE 9 with LINE 10 commented out)
- use "`_exit(0)`" (LINE 10 with LINE 9 commented out)
- comment out all the exit statements, i.e. fall through the end of `main()`

Explain what do you think is happening with the system call usage to "exit from the process" in each of the three variations. You should read the man pages for `exit(3)` and `_exit(2)` (i.e. `man 3 exit` and `man 2 _exit`).

1.5 Question 1(e)

Replace LINE 2 by LINE 3 by recommenting the code. What is the difference in the trace? Explain what you think causes the difference.

1.6 Remarks

This exercise shows you that what happens in the actual system can sometimes be a bit different/more complex than you might realize. The documentation may sometimes be hard to interpret (in some cases, documentation may be wrong or out of date) and may not be presenting precisely what is happening. Occasionally, debugging tools may also have problems. (`strace` cannot properly trace certain programs—I leave students to think why this might be a an important feature [hint: think "security"]).[2] You may also realize that `strace` is a user program running in user mode. Thus there isn't any `strace` code in the OS kernel

When in doubt, read the source — is perhaps one way of finding out, but this may be difficult for the Linux code. For this question, you are supposed to extrapolate from the experiment, so reading the source code of the C library is not needed! Finally, a thought for the curious student to ponder upon — how does `strace` itself work since `strace` is a normal program in user mode.[3]

2 Exercise 2: Using `strace` for debugging (Graded)

Goals: Learn how system call tracing can be used to understand or debug a program.

Extract `lab2-myst` from `lab2-myst.zip`, e.g. `unzip lab2-myst.zip`. The file `lab2-myst` is a x86 Linux executable. In order to be able to run `lab2-myst`, you may need set the file mode to be executable:

```
$ chmod u+x lab2-myst
```

Try running `lab2-myst` a few times (make sure you have set the file mode). You may notice a feature that `lab2-myst` is a bit unusual in that it behaves *non-deterministically*. A deterministic program behaves the same way every time it is run while a non-deterministic program is possibly different every time it is run. You may not be familiar with non-deterministic programs since most programs are deterministic.

To set the context of this question:

> Think of yourself as being an investigator. In real life, you could be doing something similar to debug a program or determine why the system is not working. For example, a program suddenly stops working. Reinstalling the program also happens not to work. Why? How to debug the problem? You may not have the source code or understand the code.

Your task is to use `strace` to understand an approximation of what the program does given that the source code is not given. Answer the following questions:

- Question 2(a): Explain why the output is non-deterministic.
- Question 2(b): Describe as best as you can what you think `lab2-myst` is doing? It might be the case that you are unable to explain all aspects of `lab2-myst` but try to explain as much as possible.
- Question 2(c): You will see "42" in the output, explain how it arises.

[2] The curious student might want to think of whether the act of being observed has a difference on the observee, i.e. it makes a difference in Quantum Mechanics.

[3] Its too complex to explain here but students who want to know more can see me. Can you strace strace?

HINT:
In Linux, the `fork` system call may be actually implemented using the `clone` system call. Reading the `strace` man page will help.

3 Submission

Your submission for a lab must be **individual** work. A joint or collaborative submission is not allowed for labs.

Submit using the file naming convention in Lab 1, a single PDF (or TXT) file with your answers to Exercise 1 and 2. Your student information should also be at the start of your file. Submit your file to the appropriate workbin. Note no code needs to be submitted for Lab 2.