Problem 1 -- using strace

The `strace` command under Linux is used to run a program with system call tracing enabled. This allows you to see the system calls that are being made along with their return values, and other events such as signal delivery and handling. strace can also be used to attach tracing to an already-running process. Please read the man page for strace. Then, write a very simple C program to output a fixed message to standard output. Run this program with strace and observe the system calls made. (no need to attach output for part 1)

Problem 2 -- pure assembly

Write a pure assembly language program to write a message to standard output using the `write` system call directly from assembly, with no help from the standard C library or the C compiler. Therefore you will write a .S file, assemble it to a .o file using as, and transform it into an executable a.out file using ld. **Repeat: do not use cc!**

The lecture notes explain the API for both 32-bit (using INT $0x80) and 64-bit (using SYSCALL). Be mindful of which API you are running under. For 32-bit, use the flag --32 to as and -m elf_i386 to ld. For 64-bit, use --64 for as and elf_x86_64 for ld. An a.out which has been flagged as 32-bit architecture by ld will be run by the kernel in 32-bit mode, even if your system is natively a 64-bit system. Since the APIs are incompatible, if you have written to the 64-bit API but assembled/linked as 32-bit, your program will be garbage and will not run. Conversely, a 64-bit program can not be run at all if you are natively running in 32-bit mode. The system header files /usr/include/asm/unistd_32.h and /usr/include/asm/unistd_64.h contain the system call numbers for each API. Or, you can "google" this information.

Attach a screenshot showing your assemble/link build process. Attach the strace output from running this program showing that it successfully made the write system call, and the output from the program showing that the message was written to the standard output.

Problem 3 -- exit code

If your program contains just a write system call and nothing else, what happens after the write? Why do you think this is (explain in your write-up)? Now, add an exit system call so that the program exits with a specific non-zero return code. Show that this worked via strace, and also by looking at the shell variable $? after execution.

Problem 4 -- system call validation

Introduce deliberate errors in your system call, such as passing an invalid address for the write string, or passing an invalid system call number. Show what happens via strace.

Problem 5 -- system call cost

This is an optional extra-credit problem for undergrad students, but a required problem for those taking the course at grad level:

Write a series of simple test programs in C: A) measure the cost of an empty loop using enough iterations to get a meaningful number. B) now add to that empty loop a call to an empty function. C) replace the empty function with a call to a very simple system call such as getuid. [Be careful...use strace to make sure you are actually making a system call. Some things like `getpid()` which are documented as system calls may be cached by user-level libraries]

Report on the cost in nanoseconds (look at the `clock_gettime` library function) of one loop iteration, one user-mode function call (not counting the loop iteration) and one system call (not counting the user-mode overhead). Approximately how much more expensive is a system call compared to a function call? Discuss your reasoning for why this is the case.