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 (the proverbial "hello world"). Run this program with strace and observe the system calls made. You will notice a lot of system calls made that bear no correlation to the code that you wrote. This is the shared library system getting initialized. Find the `write` system call associated with your output. Also find the `_exit` system call **(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!** Your `a.out` file will be only a few hundred bytes long, most of which is the a.out header. The `objdump` or `readelf` commands will allow you to explore this and provide hours of fun, for example `objdump -d a.out` will disassemble the binary a.out file and show you the opcodes inside. (no need to attach output for these commands).

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.

Note that the first opcode of your `.text` section will be the default start-of-execution address (unless you use additional flags to `ld`) so make sure it is what you want as the first opcode! Also note you will need some *pseudo-opcodes* such as for embedding a string in your program.

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.

Note: The program for part 2 should contain ONLY the write system call.

Problem 3 -- exit code

Now, observe the strace output after the write system call, and comment on what you see. How does your program exit? Why do you think this is? **Explain in your write-up**.

Write another version of the program that has the `_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

Write a version of your part 3 program that either passes an invalid system call number, or an invalid parameter to `write`. Show via the strace output what happened and discuss.

Problem 5EC -- scheduling

(+2 points)

Write a test program which creates a number of CPU-bound processes. One of those processes will have a non-default `nice` value. Both the number of processes and the nice value will be command-line flags. After spawning the processes, the parent process will sleep for the specified number of seconds, then kill all the children and collect their respective `rusage` structures. Report the sum total CPU time consumed by the children, and the CPU time by the child which had the non-default nice value. **Note:** include both system and user CPU time.

```
$ ./cputimes 16 0 5
Spawning 16 processes and waiting 5 seconds, first child process will have nice 0
Total CPU time was 39853000 us
Task 0 CPU time was 2493000 us
Task 0 received 6.25549% of total CPU

$ ./cputimes 16 10 5
Spawning 16 processes and waiting 5 seconds, first child process will have nice 10
Total CPU time was 39851000 us
Task 0 CPU time was 276000 us
Task 0 received 0.69258% of total CPU
```

Your output does not have to match the example above. In fact, to make charting easier, you might want to output in a different format that can export more easily into a spreadsheet.

Note that in the output above, this is running on an 8-CPU machine. Over a 5-second period, the maximum theoretical CPU time would be 40 seconds. We see that our 16 test processes got 39.9 seconds which tells us that the system was otherwise idle. When you pick a sample time, 5 seconds is probably a minimum otherwise brief fluctuations in system load can drastically affect your answers.

Run various combinations of #processes and nice value. In particular, I want you to experiment with a small number of processes (smaller than the total number of available cores on your system) as well as a large number. **Comment on** why the nice value appears to not affect things for small numbers of processes relative to CPU cores (run this on a system that is generally quiet, i.e. not a system which is running your machine learning projects!)

Present your results in graphical or tabular format, with analysis and commentary.