Problem 1 -- Short Answer

1A) A user mode process attempts to make a system call with the system call number set to -1. Refer to the X86-32 entry.S code in the lecture notes and explain what happens.

1B) After a system call is completed and the kernel returns control to the user-level process which invoked the system call, how does the user/supervisor privilege level get reset to user mode?

1C) Process 999 has made a system call which encountered a blocking condition: waiting for disk I/O. Assume a single-CPU system for simplicity. While this is taking place, the CPU has moved on and is running process 123. Process 123 makes a system call. During the handling of that system call, the disk I/O interrupt arrives. The completion of disk I/O unblocks pid 999, which is now in the "READY" state. Describe what happens next in (i) a fully pre-emptive Linux kernel (ii) a non pre-emptive Linux kernel

Problem 2 -- some assembly required

In this problem, you will be writing a pure assembly language program that makes system calls, and examining its operation using the `strace` utility program, which allows you to run a program with system call tracing enabled. Thus you can 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.

Problem 2A -- strace a C program

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. As part of your submission, attach a PDF screenshot where these system calls are highlighted.

Problem 2B -- Pure assembly

Now, write a **pure assembly language** program which writes a brief message to standard output using the `write` system call directly from assembly, and then calls the `exit` system call with the exit code being the return value from write. Your program will **not** use the standard C library or the C compiler in any way!. 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 `-m 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 code for 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 to MacOS users:** I have not evaluated this assignment on the Mac platform. However, in the past, students were able to do a similar assignment. The API is

similar to Linux but some additional/different steps are required.

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 screenshots showing : 1)your assemble/link build process. 2) the strace output from running this program showing that it successfully made the write and exit system calls 3) the output from the program showing that the message was written to the standard output 4) The `$?` value showing the exit code of the program.