Problem 1 -- Short Answer

1A) Refer to the X86-32 `entry.S` code in the lecture notes. We see two instructions:

```
cmpl    $nr_syscalls,%eax
jae     syscall_badsys
```

These are essential for security. Imagine a world where these two instructions are missing. Explain how an attacker could exploit this as a security hole.

1B) For each of these scenarios (X86-32), explain if a new value for %esp is fetched from the TSS or not, or if the scenario is impossible. Justify your answer!

1B1) We are running a user-mode program when the PIT interrupt arrives.

1B2) During handling of a system call, a disk I/O completion interrupt arrives.

1B3) While handling an interprocessor interrupt (IPI), a page fault occurs.

1C) The root filesystem (volume) must be mounted before the kernel begins user-mode process #1 (init or systemd) since the executable for that process is in the filesystem. What would happen if the root filesystem can't be mounted by the kernel, e.g. because of a disk I/O error, or misconfiguration? This question requires some research. The answer does not appear directly in the lecture notes.

1D) Why does the `TIF_NEED_RESCHED` flag exist?

Problem 2 -- Assemble a cat

For those who don't enjoy coding in C, this problem will be a reprieve. Instead, we're going to write a program in pure assembly language! We've covered X86 assembly language only in class. For those who are running on Macs with ARM processors, you are free to do this on your own. The approach is similar, but the exact details of the system call API and of course the assembly syntax will be very different. You can always just do it on the class VM. On X86, you have a choice of 32 or 64 bit API.

First, a quick primer on how to assemble, link, and run an assembly language program on Linux: You won't be using the C compiler (e.g. gcc) at all. Write your program and give it a `.S` suffix, e.g. `catasm.S`. Assemble it using the system assembler:

```
as --32 -o catasm.o catasm.S #32 bit
as --64 -o catasm.o catasm.S #64 bit
```

Choose the correct option depending on whether you are writing in 32 or 64 bit mode. The assembler produces a relocatable object file `catasm.o`. The same thing happens when you compile a C program. Now we need to transform that into an absolute executable:

```
ld -m elf_i386 catasm.o      #32 bit
ls -m elf_x86_64 catasm.o #64bit
```

The linker (ld) produces `a.out` (use the -o option to give it another name). You can use the command `objdump -d a.out` to disassemble the executable (or the object file) if you want to see the actual binary opcodes.

You may see a warning message from ld such as

```
ld: warning: cannot find entry symbol _start; defaulting to 0000000008048054
```

This is OK. It just means ld is making the entrypoint of your program equal to the very first (executable) line of assembly. Assuming the assembler and linker are successful, you can just run the a.out file.

Since we aren't linking against the standard C library, you can't call any library functions. You have to make system calls by hand, using the API directly. That's the point of the assignment!

The program itself is very simple and is equivalent to the following in C:

```
while ((n=read(0,buf,4096)>0)
    write(1,buf,n);
_exit(n)
```

In other words, this is a primitive cat with no error checking or reporting. The buffer `buf` could be declared as a global BSS variable using the following assembly:

```
 .comm   buf,4096,4
```

This declares a global variable buf, in the BSS section, with a size of 4096 bytes. The last argument (4) is the alignment, don't worry about it!

If you do this declaration, then `$buf` is the assembly language expression for the address of the buffer. Alternatively, you could use the stack for the buffer, and very simply too, since we have no subroutines (functions) in this program. Reserve 4K with `subl $4096,%esp` and use `%esp` as the buffer address.

You will need to know the system call numbers for read, write, and exit. Look these up in `/usr/include/asm/unistd_32.h` or `/usr/include/asm/unistd_64.h`

Note that all examples in the lecture notes are based on the "UNIX" or "AT&T" assembly syntax for X86, as that is how the Linux kernel is coded. However, many internet sources will use the "Intel" or "NASM" syntax, which is quite different.

Obviously this program involves a loop with conditional branching. You will need to use the `cmpl` (or `cmpq` for 64-bit mode) instruction followed by an appropriate conditional jump such as `jle`. The lecture notes contain an appendix with a lot of information on X86 assembly.

I want you to use the `strace` utility. Perhaps you have already done this earlier in the course. Therefore your submission will comprise:

1) The assembly language source code

2) A screenshot of your build process

3) A screenshot of you running the cat program, and echo $? after to show the return value

4) A screenshot of running the cat program under strace, with input being a file that's bigger than 4K so that we can verify repeated read and write system calls. For this screenshot, suppress the standard output by redirecting it to `/dev/null`