COMP 3000A: Operating Systems

Carleton University
Fall 2021 Final Exam Solutions

1. [2] How does /dev/null behave like a regular file? How is it different?
   A: You can open, read from, and write to it, just like a regular file. Unlike a regular file, data written cannot be read back, as reads always return nothing (a read of zero bytes).

2. [2] What is a C statement or declaration that could have generated the following assembly language code? How do you know? Explain briefly.

```
.LC0:
        .string "Hello world!\n"
        ...
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
```

   A: A puts("Hello world\n"); or printf("Hello world\n"); could have generated this because the compiler can replace simple printf() calls with puts() ones. .LC0 labels the constant hello world string, and the leaq instruction loads this address into the rdi register, which is used for the first integer/address parameter passed to a function. The call instruction then calls puts(). (Minus half point for saying puts() is a system call.)

3. [2] On x86-64 systems, which is larger, a process's virtual address space or a host computer's physical address space? How does this affect which part of a process's address space can be accessed without generating an error?
   A: A process's virtual address space is much larger than the physical address space, because regular computers aren't close to having 2^{64} bytes of RAM. This means that only a small fraction of a process's (virtual) address space can be accessed without generating an error, because only a small portion can possibly map to allocated physical memory.

4. [2] Describe how environment variable data is arranged in memory. What C data types are used? How are the key-value pairs stored?
   A: Environment variables are arranged as an array of pointers to arrays of characters, with the array of pointers and each array of characters being null terminated (terminated by a 0 value byte) and the array of string pointers is terminated by NULL (a pointer value of 0). Each array of characters (each string) contains the name of an environment variable (the key), an equal sign, and the variable's value, with that value being ended by a null byte.

5. [2] You're writing a program on a new version of Linux that has a new system call, `fastread`. Libraries have not been updated to support `fastread`. Can you make a pure, standards-compliant C program that calls the `fastread` system call? Why or why not?
   A: You can't make a standards-compliant C program because calling system calls requires using platform-specfic code (i.e., special assembly language instructions). You have to use inline assembly or a function that has inline assembly (i.e., syscall()).

6. [2] Which is bigger *on disk*, a statically-linked binary or a dynamically-linked binary? Why?
   A: Statically-linked binaries are bigger on disk because they must include all library code in them. Dynamically-linked binaries can load library code at runtime, thus reducing the size of the binary.

   Note that libraries are collections of object files and thus include functions and variables that can be linked against, with the code either being added to the program at compile time (static) or at runtime

(dynamic). Libraries do not contain full programs, as a program is a self-contained executable that can be run on its own. If we want to combine programs we typically use a shell, e.g., we can connect them using pipes. So saying that linking includes programs into our program is wrong. Many people found a web page that said linking includes programs. This web page is wrong and we deducted 0.5 for this.

7. [2] In the Microsoft Win32 API, the `CreateFile()` call is used to open a new or existing file. It returns an object handle (a pointer to a pointer that then refers to the object). What is the Linux system call equivalent to this? What is the type of its return value?
   A: The Linux equivalent is open, and it returns a file handle, which is an integer.

8. [2] When process A writes to a an existing file X, it freezes/locks up. Why could this happen? How could you get A to continue running? Assume that writes to most other files (new and old) work as expected.
   A: The file could be a named pipe, and the way to fix it would be for another process to read from the pipe.

9. [2] What is the purpose of `queue_nonfull` in Tutorial 8's 3000pc-rendezvous-timeout.c? Explain how it is being used in every place it is accessed in the program, outside of its initialization.
   A: Its purpose is to allow the consumer to wake up the producer after the producer has gone to sleep. The producer will sleep when the queue is full, meaning there is no room for new production. On line 163 in pthread_cond_timedwait() is where the producer leeps, and line 275 is where the consumer uses this variable to wake up the producer. (1 for understanding condition variables, 1 for usage specifics)

10. [2] What part of `3000makefs.sh` (from Assignment 3) is necessary to allow ps to work correctly? Why?
    A: Line 58, the mounting of /proc, is necessary for ps to work because ps looks in /proc to get information on running processes. (0.5 for saying busybox, 1 for identifying lines for installing busybox such as 56)

11. [2] In the chrooted environment created by `3000makefs.sh`, does `ls` depend on any dynamically linked libraries? Explain.
    A: No, because it is part of busybox, and busybox is statically linked.

12. [2] Can a mount command increase the space available for storing files? Explain. (Be sure to consider uses beyond those in `3000makefs.sh`.)
    A: Yes, because you can mount a filesystem on an additional device, such as a USB stick. This storage thus becomes available to files created under the mountpoint. (Note that storage is filesystem specific, so it won't make any space available in directories outside of the mountpoint.)

13. [2] On the class VM, what files have to be changed in order to add a new user? What about to add a group?
    A: You change /etc/passwd and /etc/shadow to add a new user, and /etc/group and /etc/gshadow to add a group. (1 point if you identify all four files but don't distinguish which is for users and which is for groups. No credit for home directory because you could use any directory.)

14. [2] `3000shell` can make many stat system calls for every command entered. Which function makes these stat calls? What are these stat calls for?
    A: find_binary() makes these calls (on line 124). They check whether the constructed absolute filename actually exists or not, thus telling us whether we've found the executable we are looking for.

15. [2] If you do a printf() that does not end in a newline, it will not be immediately output to a terminal; instead, it will only be output later once a newline is output. How can you force terminal output without a newline? Why does this work?
    A: You can force output with fflush(), because C library functions such as printf() do buffered output to terminals and so only issue write system calls when their buffer is filled, a newline is output, or the buffer is flushed.

16. [2] What is the relationship between sigaction() and kill()?
A: sigaction() is used to register signal handlers which are run when a process receives a signal. kill() is used to send signals.

17. [2] Tools like `gdb` and `strace`, that use the ptrace system call, have several significant limitations compared to `bpftrace` and other tools based on eBPF.

    What is one thing you can do with eBPF that you can't do with ptrace? And, what key restriction is placed on eBPF programs that isn't there for ptrace programs?
A: With eBPF you can observe all the processes on the system and change how the kernel works, potentially modifying how security or even scheduling decisions are made; ptrace can only allow one process to be observed at a time. eBPF programs must be loaded and run as root however. (Multiple acceptable answers.)

18. [2] What is one signal that can be sent directly from one process to another (via the kernel)? What is another signal that is sent by the kernel itself and received by a process? Briefly explain the purpose of each signal.
A: Many possible answers. Common ones are SIGTERM, SIGKILL, and SIGSTOP for sending from one process to another. SIGSEGV, SIGBUS, SIGCHLD and such are sent by the kernel to a process. (Should also briefly explain each signal.) (-.5 if two signals that are valid and properly explained but unclear on which is sent by a process vs kernel)

19. [2] Could you make a special file that, when read, returns a random sentence? Why or why not? Be sure to explain how it could be done or why it would be impossible. (No code is required in your answer.)
A: Yes you could, it would just be a kernel module like newgetpid, but it would choose a random sentence from a built-in database (or perhaps load it previously) and then return it. (1 for saying yes, 1 for the explanation) Note that character devices are not read or written one character at a time, that would be very wasteful! It is a character device because input and output can be arbitrarily sized; with block devices, we can only read or write entire blocks (i.e., only 4K at a time).

20. [2] After loading the `ones` module from Tutorial 7, running "cat /dev/ones" will produce an unbounded sequence of 1's. How is this possible, given that ones_read() only outputs a limited number of 1's? Explain briefly.
A: On every read, it will fill the given buffer completely and return the size of the buffer as the number of characters read. Thus there's never any indication of end of file (and indeed the offset is never changed), so subsequent reads will be indicated and will return the same, thus producing unbounded output.

21. [2] If the kernel accesses a process's data using standard C methods, such as dereferencing a pointer, it can result in errors. Why? How can it access process data safely?
A: User data is in a process with its own address space separate from the kernel (on most architectures), meaning that userspace pointers simply aren't valid in the context of the kernel's own address space. Further, it is possible a userspace pointer is pointing to memory that hasn't been loaded or isn't allocated memory; the kernel much check and deal with such conditions. To access userspace pointers safely, the kernel uses special functions such as get_user() and put_user() that do the necessary translations. 1 for put/get_user, 1 for different address spaces or something about the difference between memory in userspace and kernelspace.