COMP 3000B: Operating Systems

Carleton University
Winter 2020 Final Exam Solutions

April 13, 2020 (7-9 PM)

There are 29 questions on **3 pages** worth a total of 60 points. Answer all questions in the supplied text file template (available on cuLearn, titled `comp3000-final-template.txt`). Please do not corrupt the template as we will use scripts to divide up questions amongst graders. Your uploaded file should be titled `comp3000-final-username.txt`, replacing username with your MyCarletonOne username. Make sure you submit a text file and not another format (such as MS Word or PDF). Use a code editor, not a word processor! (You won't be graded for spelling or grammar, just try to make it clear.)

This test is open book. You may use your notes, course materials, the course textbook, and other online resources. If you use any outside sources during the exam, you **must cite** the sources. Your citation may be informal but should be unambiguous and specific (i.e., if you refered to the textbook, indicate what chapter and page you looked at rather than just citing the textbook). You **may not** collaborate with any others on this exam. This exam should represent your own work.

All explanations should be concise and to the point (generally no more than a few sentences, sometimes much less). If you find a question is ambiguous, explain your interpretation and answer the question accordingly.

You have 120 minutes. Good luck!

1. [2] When you interact with `bash` (via ssh or a graphical terminal) what file does `bash` read from in order to obtain user input? Is this file a regular file? Explain briefly.
   A: bash reads from a numbered file /dev/pts (such as /dev/pts/0) in order to obtain user input. This file is a character device, it is not a regular file.

2. [1] When you type the command `ls > ls.log` at a bash shell prompt, what process opens the file `ls.log` for writing, bash, ls, or another?
   A: bash

3. [2] When you type `ls -la` at a `bash` prompt, what system call does bash use to receive user input? What system call does bash use to pass the `-la` argument to ls?
   A: read, execve

4. [3] What system calls do the following C library functions make (on Ubuntu 18.04)? Note they may generate none, one, or multiple system calls. (a) fgets, (b) ioctl, (c) snprintf.
   A: a) read, b) ioctl, c) none

5. [2] Process A creates a child process B to open and write data to a file (and then terminate). What is the standard UNIX mechanism that allows B to inform A that the write to the file failed? Explain, indicating how the error message would be sent and received.
   A: B calls exit with a return code indicating failure. A calls wait to get B's exit code.

6. [2] If you decided you didn't want to run any pre-installed binaries and instead just wanted to run your own versions when you type in commands at a shell prompt, how could you do this? Would this change prevent other users of the system from using system binaries? Explain briefly.
   A: You would change your PATH environment variable (by changing your shell configuration

files, e.g., .bashrc, .bash_profile, .profile) to only list your directories, excluding /bin, /usr/bin, etc. This change wouldn't affect any other users on the system.

7. [2] If a sleeping process receives a signal, will the signal handler run immediately or will it run after the sleep finishes? Explain briefly, giving evidence for your answer.
   A: The signal handler will run immediately, and then once the signal finishes the sleep system call will return without having finished the sleep. If the restart flag is specified, the sleep system call will be automatically resumed where it left off. We could see this in 3000shell when we removed the SA_RESTART flag, 3000shell would terminate immediately when receiving a signal (e.g., SIGCHLD).

8. [2] SIGPIPE is sent to process to indicate a broken pipe, i.e., a write to a pipe that has no readers (but did previously). Alice, upon learning about SIGPIPE, says this is stupid, because the write would just return an error. Bob replies that SIGPIPE is useful just like SIGCHLD is. Is Alice right or is Bob? Explain how the signals are similar and a situation when SIGPIPE would be useful.
   A: The key benefit of SIGPIPE is informing a process that a pipe is broken even if it isn't in the middle of doing a write. In a producer/consumer situation using a pipe, the producer could be spending a considerable amount of time producing, assuming that the consumer is waiting; with SIGPIPE it will be immediately informed that something has gone wrong with the consumer and can take corrective action (or just terminate). This is similar to the situation with SIGCHLD, in that SIGCHLD tells the parent immediately that its child has terminated, while a call to wait would result in a significant delay in getting this information to the parent.

9. [2] If a process has a uid=1000, euid=1000, gid=1021, and egid=1021, what files can it read on the system? Why?
   A: The process can read files that are owned by uid 1000 with the owner read bit set, files with a group 1021 and with the group read bit set, and any files on the system not owned by 1021 and not with a group 1021 that have the other read bit set. Such regular files must also be accessible via a directory path where each directory is both readable and executable by uid 1000, gid 1021, or by others. This is because UNIX file permission checks involve checking whether the process owner has access, then the process group, and finally whether it has other access, with regular files requiring read access and directories requiring read access to list entries and executable access to get inodes associated with directory entries.

10. [2] With ssh, what is the purpose of the `id_rsa` file? What about `id_rsa.pub` file? What do they contain, and how are they used?
    A: The id_rsa file contain's a user's private key, and id_rsa.pub file contain's a user's public key. They are used to authenticate a user to a remote system by copying the contents of id_rsa.pub to the remote system's authorized_keys file.

11. [2] In a filesystem, can two files share the same inode? Explain briefly.
    A: Two filenames can both be hard links to the same inode. In this case, they are really two names for the same file rather than being two distinct files, as inodes represent the contents of files. Sharing an inode means they are essentially the same.

12. [2] Can you easily recover from erasing the primary and all backup superblocks of an ext4 filesystem? Explain why or why not.
    A: You cannot easily recover from erasing the primary and backup superblocks of a filesystem, as these are essential to mounting a filesystem. If these are erased, the filesystem is effectively

erased, and the only way to recover would be to reconstruct the superblock (say by guessing at the key parameters contained in it for the filesystem) or by doing a forensic recovery of file contents, both of which are non-trivial tasks.

13. [1] If you want the standard output of one program to be fed to the standard input of another program directly (without storing any data on disk), how could you do this **without** using the `|` operator? Explain with a short example.
    A: You would create a named pipe, and have one program use the pipe for standard out and the other for standard in. So instead of `ls | wc`**, you would do** `myfifo mypipe; (ls > mypipe &); wc < mypipe`

14. [2] Below is an implementaiton of `sem_wait()`. Does this version cause the process to sleep while waiting for the lock to be freed? Do you expect this implementation to work reliably in practice? Explain briefly.

```
void sem_wait(int *lock)
{
        while (*lock == 0) {
                /* wait */
        }
        *lock = 0;
}
```

    A: This version does not cause the process to sleep if the lock is currently taken; instead, it busy waits (loops, checking on each iteration whether the lock is available). This implementation will not work reliably in practice as the check and assignment are separate instructions, and thus another process could change the state of lock between them. A correct implementation would have to make use of special assembly language instructions (or special language mechanisms that would produce such instructions) that allow for the checking of a value and changing it both as one atomic instruction.

15. [2] From an API perspective, which is simpler to create, a thread or a process? Which do you think requires more work on the kernel's part? Explain briefly.
    A: A process is simpler to create from an API perspective than a thread, as a process can be created using fork, which takes no arguments, while threads are created using pthread_create, which takes four arguments. (At the level of system calls they are equivalent as they both result in a clone system call.) Process creation takes more work on the part of the kernel as the address space needs to be (logically) copied while with threads the address space is shared.

16. [2] How could you add support for lseek operations to a character device module? Specifically, what function(s) would you add, and how could you make sure those functions were called at the right time?
    A: You would add an llseek function to the module (say mymodule_llseek) with the same arguments as lseek, but with the fd being replaced with a struct FILE *. To make sure it was called, you would add an llseek entry, referring to mymodule_llseek, to the struct file_operations used in the module to define available file operations for the device.

17. [2] What is the difference between a process's uid and euid? Specifically, what does the kernel use each for?

A: The kernel uses the uid to determine process ownership (who can send STOP and KILL signals to control the process), while euid is used to determine what resources a process can access (mainly file access, comparing with a file's owner uid).

18. [2] Does the kernel know the names of the groups a user belongs to? How do you know?
A: The kernel doesn't know the groups that a user belongs to, not directly. It only knows the gid's of processes and files. To change the currently active group, a user has to run a program such as newgrp which is setuid root (thus allowing it to escape out of the kernel's normal security restrictions to allow the changing of a process's gid).

19. [2] When a process mmap's a file, can it (mostly) control where the file will be placed in virtual memory? What about physical memory?
A: A process can mostly control where a file is mmap'd into virtual memory, as it has control over its own virtual address space. A process has almost no control over where a process is mapped into physical memory, however, as that is managed by the kernel in a way that is mostly invisible to processes.

20. [2] Can two processes have data at the same virtual address? What about at the same physical address? Explain.
A: Two processes can have data at the same virtual address, but they will be completely distinct, i.e., data at address 2000 in process A is completely different from data at address 2000 for process B. Indeed, this difference is the whole purpose of separate virtual address spaces. The only exception to this is if two processes have an area of shared memory, but then that shared memory would not normally have the same virtual addresses in both processes.

    Two processes can share data at the same physical address by mmap'ing the same file or by creating a shared memory segment. The virtual addresses may be different but the underlying pages in RAM (and thus the underlying physical addresses) would be shared.

21. [3] For each of the following, state and explain whether they support lseek operations always, sometimes, or never: regular files, pipes, character devices
A: Regular files always support lseek operations, pipes never do, and character devices sometimes do, if they implement lseek support by defining the right entry in the corresponding module's file_operations struct. (In practice character devices almost never implement lseek.)

22. [2] At each level of the page table, we look up an entry which contains (the upper bits of) a pointer to another page, until the last one points to the desired data page. Do you think these pointers contain virtual or physical addresses? Why?
A: These pointers contain physical addresses, because otherwise we'd be using virtual addresses to resolve virtual addresses, and that could lead to a never-ending series of virtual to physical address lookups to figure out one virtual to physical address mapping. (Parts of the page table can be swapped to disk, but for the CPU to use them to map virtual to physical addresses, the necessary parts of the page table must be in RAM.)

23. [2] Is the size of a page consistent on all platforms that Linux runs on, or does it vary between platforms? How do you know?
A: The page size is not consistent because PAGE_SIZE varies between different architectures. We can see this in the Linux source code by seeing all the architecture-dependent definitions of PAGE_SHIFT and PAGE_SIZE (defined in terms of PAGE_SHIFT).

24. [3] Fill in the missing parts in the table below relating to file permissions in octal and symbolic form.

Octal	Symbolic
0666	(a) rw-rw-rw-
(b) 0674	**rw-rwxr–**
0775	(c) rwxrwxr-x
(d) 0544	**r-xr–r–**
0511	(e) r-x--x--x
(f) 0222	**-w–w–w-**

25. [2] Consider the following code:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
     printf("Hello!  The number is %d!\n", NUMBER);
}
```

(a) How can you compile this so, when run, it outputs "Hello! The number is 42!" (rather than reporting a compilation error)?
A: You just have to add a -DNUMBER=42 option to the compilation command line.

(b) What is one situation in which the technique you used in a) is useful (beyond those shown in class tutorials)?
A: It is useful for changing constants in code at compile time, say to define system-specific parameters, or more generally to define constants that are used for conditional compilation (e.g., #ifdef's).

26. [2] Python scripts importing from the bcc library can monitor all system calls on a system and arbitrary function calls in any process and in the kernel. Can regular python scripts or other userspace programs do this? Why or why not? Explain.
A: Regular python scripts, or indeed any non-BPF using userspace program, cannot do this kind of extensive monitoring. Processes are isolated from each other, and normally the only way to go beyond this is using ptrace, which allows one process to monitor exactly one other process. Running as root doesn't give you additional access without BPF. To monitor arbitrary kernelspace and userspace events, you need to use BPF. (Yes, there are traditional ways to get low-level access such as /dev/mem; however, even with these interfaces you would only get a static snapshot of the system. It wouldn't let you watch everything that is going on. The closest traditional mechanism would be system audit logs, but those don't give nearly the level of access as BPF.)

27. [2] Give an example of how an eBPF program can access a field in a task_struct. What task_struct is being accessed?
A: There are lots of examples in bpf code, as it is very common to get info from the currently running task's task_struct (i.e., current) in bpf programs. But in bpfprogram.c, the one example of this was bpf_get_current_pid_tgid() which gets the user-visible PID of the current task (i.e., the one that made the system call that is currently being processed).

28. [3] How often is the `filter()` function called in (the original version of) bpfprogram.c when tracing system calls? It it called for every system call made on the system, every system call made by the specified process, or every time a process running 3000shell makes a system call? How can you tell this from the code? What experiment(s) did you do to verify your interpretation of the code?
A: It runs for every system call made on the system. You can see this from the code because in bpfprogram.c, line 69, a tracepoint probe of every system call exit is defined. If you delete lines 72 and 73 (so the code doesn't call filter()) we get system calls from every process on the system. (Try it, it produces lots of output very quickly!)

29. [2] What is a way to load code into the kernel *without* it being verified for safety? What limits are placed on such code, relative to other kernel code?
A: You can load code using a kernel module (i.e., using insmod). No limits are placed on such code relative to other kernel code, kernel modules can do basically anything.