COMP 3000A: Operating Systems

Carleton University
Fall 2019 Final Exam Solutions

December 17, 2019

The exam had 60 points total in 17 questions. Note that portions of solutions in parentheses are explanatory comments and were not required for full marks.

1. [1] When setting up ssh key-based authentication, what part of the key pair goes in the remote system's `authorized_keys` file, the public or private key?

   A: public

2. [1] If you run `fsck.ext4` on a damaged ext4 filesystem and it complains that there isn't an ext4 filesystem to recover, what data is missing?

   A: primary superblock or all superblocks

3. [1] Is stack allocation of variables more or less efficient than heap allocation? Specifically, which requires more instructions and system calls?

   A: Stack allocation is more efficient. (Stack allocation just takes two instructions: you save the current stack pointer register to the base pionter register and decrement the stack pointer register. Heap alloction requires many more instructions in order to remove a block from the free memory pool and potentially to increase the size of the memory pool using system calls.)

4. [1] When a process exits, does the kernel automatically reclaim the memory resources it was using? How do you know?

   A: Yes. You can tell because even if a program doesn't free allocated memory, the process's memory returns to the free memory pool as shown by top.

5. [1] Can a process directly access kernel data structures?

   A: No. (Virtual memory means each process has its own private memory map, as does the kernel. So no direct access is possible.)

6. [1] Can a kernel module access all kernel data structures?

   A: Yes (Modules are loaded directly into the kernel and thus occupy part of the kernel's private virtual address space.)

7. [2] When are environment variables initially allocated? What system call causes them to be allocated?

   A: Environment variables are allocated when a program is loaded into a process, by execve (using its envp parameter).

8. [2] To redirect standard input, what must a process do, and how can the process do it?

   A: A process must change what file is open on file descriptor 0. One way to do this is to open the desired file and then use dup2 to copy the return file descriptor to fd 0.

9. [2] A friend tells you, "A signal handler will only be called while a process is in the middle of a system call. We know this because in 3000shell the only time the signal handler is called is while 3000shell is waiting for input, blocked on a read system call." Is your friend correct? Explain briefly.

A: The friend is incorrect, as signals can be delivered at any time. The read call in 3000shell is the one that always gets interrupted because 3000shell spends most of its time waiting on user input.

10. [2] Can the "cd" command be implemented as a separate binary? Explain.

    A: The cd command cannot be implented as a separate binary because cd changes the current working directory of a process, and this must be done by a process using the chdir system call. One process cannot call chdir on behalf of another process.

11. [2] Do pipes on Linux, such as `ls | wc`, involve the creation of temporary files? Explain.

    A: Pipes on Linux do not involve the creation of temporary files; instead, the standard out of the first process (ls) is connect to the standard in of the second process (wc). (It turns out they are connected using a pipe, see the pipe(2) man page.)

12. [2] Are there common situations when you **can** make a hard link to a file but you **cannot** make a symbolic link? Explain.

    A: There are no common situations where you can make a hard link to a file but cannot make a symbolic link to it. Hard links can normally only be made within the same filesystem while symbolic links can cross filesystem boundaries, referring to any file on the system.

13. [2] On Linux x86-64, are environment variables stored close or far away from program code? How do you know?

    A: Environment variables are stored for from program code. We can see this if we compare pointers to environment variables to pointers to functions (e.g., envp[0] vs &main). Normally environment variables are near the top of the address space while code in towards the bottom.

14. [2] Say you run the commands `cp -a A B; rm A; mv B A`. Can you tell that A has been replaced with a duplicate, or is the new A indistinguishable from the original A? (Note that the -a flag tells cp to preserve timestamps, permissions, and ownership if possible.) Explain.

    A: The new A will be almost indistinguishable from the original, except that the new one will have a different inode number. This happens because every new file created gets a new inode number, and renaming a file preserves its inode. So the new A will refer to the inode created by cp; the old inode will be freed by the rm command.

15. [2] Assume X and Y are two regular files which have exactly the same size as shown by `ls -l`. Do X and Y necessarily take up the same number of blocks on disk? Explain.

    A: X and Y do not necessarily take up the same number of blocks on disk because one or both could have a hole in it, thus reducing its physical size relative to its logical size. (It is also possible that sizes could vary based on how data from the files were put in blocks, because X and Y were on different filesystems with different block sizes or filesystem layouts, or because the filesystem employed on-the-fly encryption or deduplication.)

16. [2] What do the `bs` and `count` parameters to `dd` specify, in terms of their effect on the read and write system calls emitted by `dd`?

    A: bs is blocksize, specifies the size of the buffer for reads and writes. Count specifies the number of read and write calls. Example: count=10, bs=512, means 10 reads from input and 10 writes to output, each using a buffer of 512 bytes.

17. [2] If I type `rsync -a /a/myfiles/ /b/otherfiles/`, what have I done to the files in `/b/otherfiles/` that were there previously? Explain.

    A: If there are files with the same name/path in /b/otherfiles as were in /a/myfiles, those will be updated/overwritten so they are identical to the version in /a/myfiles. The files in otherfiles that aren't in myfiles, however, will be left unchanged. (To get rid of them, you have to add the –delete and –force options.)

18. [2] If you know a file's inode number (for a file on an ext4 filesystem), can you access its contents from user space? What about kernel space?

    A: With an inode number only, you can't access its contents from userspace but you can from kernelspace. There are no system calls that let you open a file using an inode; however, the kernel has to be able to access inodes directly because the kernel implements the entire file and filesystem abstraction, and in fact does so every time a file is opened.

19. [2] Why does the kernel use `printk` rather than `printf`? What's the difference between the two?

    A: The kernel uses printk rather than printf because 1) printf goes to standard out, and there is no standard out for the kernel and 2) the kernel doesn't have access to the standard C library, it has to be self-contained. Instead of going to standard out, printk stores messages in a log buffer that can be retrieved from userspace (by some sort of kernel logging daemon, klogd or systemd's journal) and/or a terminal device (e.g., the Linux text console or a serial line console).

20. [2] If you upgrade the kernel installed on your system, do you also have to upgrade the modules? Why or why not?

    A: You have to upgrade the modules as well. because there is no fixed API between modules and the main kernel. Module code is just kernel code, and thus as internals of the kernel change module behavior also must change. Thus, when you install a new kernel, modules must also be upgraded (recompiled and installed) so they match.

21. [2] What are two specific types of information that you get by reading files in `/proc` and/or `/sys`?

    A: You can get process information (PIDs, file descriptors, memory maps, owner, group, etc), system information (/proc/cpuinfo, modules and devices in /sys), mounted filesystems, filesystem types available...basically, almost anything about the currently running kernel and its state.

22. [2] If you wanted to find out what was the user ID associated with a process, what type of data structure would you check? What else does this data structure contain (at a high level)?

    A: (This question should have said "in the kernel", so I was flexible in the answers I accepted. If you described a

23. [2] Are system calls slower, faster, or the same speed as a function call? Why?

    A: System calls are much slower than function calls. While function calls can invoke system calls, a function call on its own is just a few instructions (instructions to put arguments in registers and

24. [2] Are the pages for a process necessarily contiguous in physical memory? What about virtual memory? Contiguous here means the memory addresses form a continuous range, i.e. there are no gaps.

    A: Pages for a process are contiguous in virtual memory and are normaly not contiguous (and in fact are quite fragmented) in physical memory. The whole purpose of virtual memory is to provide a contiguous, private address space to processes.

25. [2] Can the kernel "trust" data passed to it as arguments to system calls? Why or why not? (Explain what you mean by trust.)

    A: The kernel cannot trust data passed to it as arguments. The kernel must validate parameters to make sure errors or malicious data do not corrupt or compromromise the system. (It is true that root users are given more latitute and thus can do more harm to the system than regular users; even for root users that trust has limits, and the kernel verifies things as best it can.)

26. [2] When does a call to `wait` return immediately? What does `wait` then return?

    A: A call to wait returns immediately when a child process has already terminated or if there are no child processes. If the child process has terminated, it returns with the PID of a child that has exited and its return value is stored in passed-by-reference (pointer) wstatus. If there was no child process, it returns -1.

27. [2] If your program has the following code, what could it potentially output (depending upon the state of the system)? Assume that this code compiles with no warnings, and assume that you are running this as the user student.

```
execve("/usr/bin/whoami", argv, NULL);
printf("Done!\n");
```

    A: This code either prints "student" or "Done!" to standard out. Only one or the other, never both. This assumes that /usr/bin/whoami, on success, prints out the current user's username to standard out (as it normally does).

28. [4] What are two specific things you can do with the `trace` command (that is part of eBPF)? Can you do these two things with `strace`? Explain briefly. (You don't need to give the precise `trace` commands.)

    A: You can do many, many things with trace, and most of those things cannot be done with strace. strace allows you to report on the system calls of a specific process and its children. trace, however, can report on system calls by any process on the system, can report on function calls in processes and in the kernel, can do so selectively (by system call, process, or other criteria), and can report on the kernel and userspace backtrace when those functions are called. For this question, I expected answers such as describing opensnoop and bashreadline (tasks from Tutorial 6) or answers from question 10 on the third assignment.

29. [4] What are the four key system calls used when a shell runs an external command? Be sure to consider all processes. Explain the role of each.

    A: The four key system calls are fork/clone (make a new child process), execve (load the command binary into the child process and run it), exit (the child process terminates with a return value), and wait (get the return value from the child process). I gave most if not all the credit for just naming these four system calls.

30. [4] Outline the basic algorithm for copying a file using mmap. Be sure to specify any key arguments to the necessary system calls.

    A: For copying source to dest:

    - **a = open("source", O_READ), b = open("dest", O_READ—O_WRITE—O_CREAT,O_TRUNC)**
    - **stat(a) to get length of file (len)**
    - **lseek to len-1 on b, write one byte**

- **s = mmap(a, READ, SHARED, len), d = mmap(b, READ/WRITE, SHARED, len)**
- **copy len bytes from s to d**
- **close(a), close(b)**