COMP 3000: Operating Systems

Carleton University
Winter 2020 Midterm Exam Solutions

16 questions, 34 points total.

1. [1] When you interact with `bash` (via ssh or a graphical terminal) what file does `bash` read from in order to obtain user input?

   A: It normally reads from a tty device such as /dev/pts/0 or similar.

2. [1] When a user specifies the > operator in `bash`, what system call(s) at minimum must `bash` make?

   A: bash must do an open call to open the file and then dup2 to copy the returned file descriptor to file descriptor 1.

3. [1] When you type `ls -la` at a `bash` prompt, how does `ls` get the `-la` argument?

   A: ls gets the argument via the execve system call that loads the ls binary (specifically, the argv argument).

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) fork, (b) execve, (c) snprintf.

   A: fork: clone, execve: execve, snprintf: none

5. [2] How can a process send a return value to its parent process? How can the parent process receive this value?

   A: A process returns a value via the exit system call. The parent process receives this value via wait (through the pointer passed to wstatus).

6. [2] Both `fork` (clone) and `execve` are essential system calls for a shell to run external commands. What would be the effect of just using `fork` on its own? What about `execve`?

   A: With just fork, you get a copy of the process, so two processes after the fork are running the same program code with the same data (but are copies). If you just use execve, the code and data in the current process is discarded and a new program binary is loaded in.

7. [2] What is the PATH environment variable used for? What is the immediate source of PATH, i.e., from where does a running program get the value of PATH?

   A: PATH is used to find program binaries that can be run from the command line. A running program gets PATH where it gets all of its environment variables, from the arguments to the execve call that loaded the program (specificially, envp).

8. [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, interrupting the sleep. We can see this in 3000run (from Assignment 2) where the 3 second sleeps almost never took three seconds, they generally took less because they were interrupted by a signal.

9. [2] SIGSEGV is sent to process when it makes an invalid memory access. Could you make a signal handler to intercept SIGSEGV? What should such a signal handler do? (Should it ignore the signal, log the message, or do something else?) Explain briefly.

   A: Yes, you could make a signal handler to intercept SIGSEGV. The handler should probably cause the progarm to terminate or restart as an invalid memory reference probably means the program has encountered a serious problem such as a corrupted or invalid pointer.

10. [2] If a process has a uid=1000, euid=0, gid=1021, and egid=1021, what files can it read on the system? Why?

    A: The process can read essentially any file on the system because it is running with root privileges (euid=0, and 0 is the root user).

11. [2] On Ubuntu Linux 18.04, what program runs when you first log in via ssh or on a text console? How can you change this program?

    A: When you first log in /bin/bash runs, because this is the default shell on Ubuntu 18.04. You can change this program using chsh (or by editing /etc/passwd directly).

12. [2] With ssh, what is the purpose of the `authorized_keys` file? What does it contain?

    A: This file determines what remote users are authorized to access an account using public key cryptography (instead of the normal method of passwords). This file contains authorized public keys belonging to authorized remote users, one per line.

13. [2] Where is a file's filename stored? Is this the same place as where its last modified timestamp is stored? Explain briefly.

    A: A file's filename is stored in its directory (e.g., the name "ls" is stored in the /bin directory), meaning it is stored in that directory's inode or associated data blocks (for large directories). The last modified timestamp is contained in the file's inode, a completely different structure containing all of a file's metadata except for its filename.

14. [2] Can you recover from an erased superblock? How? Afterwards, is there any lasting damage? Explain.

    A: You can recover from an erased superblock with no lasting damage as filesystems have backup superblocks.

15. [2] Can you recover from an erased inode? How? Afterwards, is there any lasting damage? Explain. Be sure to consider different kinds of inodes.

    A: If an inode is erased, it and any data blocks associated with it are lost. Thus if you erase an inode for a regular file, you lose the file entirely. If you erase a directory inode, you lose all the filenames stored in it, but the inodes it referred to can be recovered and placed in lost+found (but their filenames are lost). Special files (block devices, character devices, pipes) are lost when their inodes are erased, but they can be re-created with no data loss as they don't actually store any data (other than the device numbers, ownership, and permissions).

16. [6] Consider the following listing of files, produced using the command `ls -lais`:

```
total 164
23609523   4 drwxr-xr-x 2 soma soma     4096 Feb 27 10:07 ./
23603346   4 drwxr-xr-x 6 soma soma     4096 Feb 27 10:10 ../
23604018   4 -rw-r--r-- 1 soma soma   100012 Feb 27 10:10 data
23604016 48 -rwxr-xr-x 2 root root    47480 Sep  5 06:38 id1*
23604017 48 -rwsr-xr-x 1 root root    47480 Sep  5 06:38 id2*
23604016 48 -rwxr-xr-x 2 root root    47480 Sep  5 06:38 id3*
23604015   4 -rw-r--r-- 1 root root     3236 Nov  8 21:22 passwd
23603788   4 -rw-r----- 1 root shadow   2207 Nov  8 21:22 shadow
```

(a) What is the logical size of `passwd`? What is its physical size? Recall that `ls` by default uses a 1K block size.

A: logical size 3236 bytes, physical size 4096 bytes

(b) What (if anything) can you say about the data stored in `data`?

A: data is mostly zero bytes (null bytes) as the physical size is much smaller than the logical size, meaning that it is full of holes. At most 4096 of its bytes are non-zero as that is its physical size.

(c) What is the inode number of `id1`?

A: 23604016

(d) Are any of these files only readable by some users on the system? Explain briefly.

A: All of the files are readable by everyone except for shadow, which is only readable by the root user and those in the group shadow.

(e) What can you tell about how similar `id1`, `id2`, and `id3` are based only on the information above?

A: id1 and id3 are identical because they are both hard links to the same inode. id2 appears to be similar to id1 and id3 based on the ownership, timestamps, physical, and logical sizes; however, id2 could contain completely different data from id1 and id3. At best id2 is a copy of id1/id3.

(f) The permissions of `id2` are different from that that of `id1` and `id3`. If we assume these all contain the code for the standard `id` command (which reports on the current uid, euid, gid, egid, and available groups), how will the output of these commands compare? Will they all produce the same output, or will there be differences? Explain briefly.

A: id1 and id3 will produce the same output because they are actually the same program binary. id2 will produce the same output when run by the root user; when run by other users, however, it will report that the euid=0 because this binary is setuid root.