COMP 3000: Operating Systems

Carleton University
Winter 2019 Midterm Exam Solutions

1. [2] Do `fork` and `execve` change where `printf()` sends its output? Why?
   A: They do not, as neither affects currently open file descriptors. Thus where printf() sends its output (to standard out, file descriptor 1) will be the same before and after both a fork and an execve.

2. [2] When does a call to `wait` return immediately, and when does such a call block the process (making it really "wait")?
   A: If a child process has exited and its return value has not been given to the parent (the one making the wait call), then wait returns immediately. If no child process has exited, the wait call in the parent will block until a child terminates (or it is interrupted.

3. [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.

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

   A: If /bin/ls exists and is executable by the current user, the program will output whatever /bin/ls outputs (the listing of files in the current directory or something similar based on what arguments it is passed). If /bin/ls does not exist or is not executable by the process's user, then "Done!" will be printed. Note that the output of ls and "Done!" will never be printed at the same time, as a successful execve prevents any further code in the process from being run.

4. [2] To receive a signal, what must a process do? Why? Assume the process is using the standard C library, and that you don't want to do anything specific in response to the signal—you just want to receive it.
   A: A process doesn't have to do anything to receive a signal—it just happens. The standard C library configures standard signal handlers that are run in response to signals.

5. [2] What is "stored" in the numeric directories in `/proc`? And, what is "stored" in the `comm` file in each of these numeric directories?
   A: The numeric directories in /proc contain information on the processes running on the system, with each process having a directory corresponding to its process ID. The comm file in each is the name of the executable running in the process (generally, the contents of argv[0]).

6. [2] Describe one standard environment variable. Why is this variable used (rather than a process getting this information from another source)?
   A: The PATH environment variable is used to tell the process where standard executables are located. This variable is needed because otherwise program would have to search the filesystem to find standard binaries such as ls. (Other environment variables are possible as an answer of course.)

7. [2] Is stack allocation of variables more or less efficient than heap allocation? Specifically, which requires more instructions and system calls? Explain briefly.
   A: Stack allocation is much more efficient as once memory for the stack has been allocated (generally when the program first starts up), allocations just require a decrement of the stack pointer register which can be done with one machine language instruction. Heap allocations, however, require function calls to malloc or equivalent, and they must either choose a portion of an already allocated memory region or they must make a system call (mmap or sbrk) to allocate more memory. Thus heap allocations can require tens, hundreds, or even thousands of instructions.

8. [2] Would it be possible to simulate execve in userspace using other system calls (plus other code)? Explain by outlining a possible simulation or by describing what aspect of execve couldn't be emulated.
   A: It would be possible to (mostly) simulate execve in userspace. You'd just need to mmap portions of the executable into properly allocated memory regions (for code and data), de-allocate existing code and data, and do the necessary relocation in order for the code to run in the memory regions it was loaded into. One thing a userspace execve couldn't do, however, is run programs with higher/different privileges. (It is fine to say that it could be completely done in userspace.)

9. [2] Are the same system calls used to read directories as to read files? Explain briefly.
A: Different system calls are used for files and directories. Files are opened with "open", read with "read" while directories are opened with "opendir" and read with "readdir". (Technically, these are the standard POSIX library calls. On Linux they translate to open/openat with a O_DIRECTORY flag and getdents to get the entries.)

10. [2] What are the two situations in which a user can modify a file that they do not own? Assume normal UNIX semantics and no elevated privileges.
A: If the file is writable by a group that the user is in, or if the file is writable by anyone (other), then a user can modify it without being its owner or any extra privileges.

11. [2] What happens when you mount a filesystem on a non-empty directory? Specifically, is any data lost or deleted when this happens? Explain briefly.
A: When you mount a filesystem on a non-empty directory, no data is lost or deleted but the files that were in the directory (before the mount) are hidden in favor of the files and directories in the newly mounted filesystem. When the filesystem in unmounted, the old files in the mountpoint again become accessible.

12. [2] Are signal handlers called "by the kernel"? Explain briefly.
A: Yes, they are called by the kernel, as the kernel calls the signal handlers to be invoked when delivering a signal. The kernel knows what code to call based on previously registered signal handling functions.

13. [2] How can a process change what file is being read on standard input? Explain briefly.
A: The process just has to close file descriptor 0 and then open a new file on this file descriptor (either using openat or open plus dup2).

14. [2] Do you think a symbolic link can refer to a file on another filesystem? What about hard links? Explain briefly.
A: Symbolic links can refer to any file on the system, because they refer to the destination file by name. Hard links are limited to files in the same filesystem, because they refer to an inode, and inode numbers are filesystem-specific.

15. [2] You have a hard drive that is failing—your system is getting read errors when accessing certain blocks. It is possible the filesystem is corrupted. Do you run fsck before or after copying data off of the disk? Why?
A: Run fsck after copying all the data that you can off the disk. fsck can modify the contents of a filesystem and so it could further damage the disk in the process of repairing it. (Ideally you'd make the copy at the block level first, copy all you can (from the block level copy), fsck the copy, then try copying some more.)

16. [2] "Sparse files are a poor man's compressed filesystem." What could this statement mean? (A compressed filesystem is one that compresses all files automatically using an algorithm similiar to zip files.) Explain.
A: Sparse files provide explicit compression for runs of all zeros in a file. Compression subsititues shorter representations of a variety of patterns (e.g, repeating 1's, 2's, alternating a's and b's). Thus sparse files are a very limited special case of generally compressed files.

17. [2] The messaging app Signal has a feature called "verify safety number". This feature asks you to compare a number with a person you message. If you both have the same number you should mark that person as being verified. What kinds of attacks do you think can be prevented by verifying a safety number? Explain briefly.
A: Signal safety numbers are like verifying a host key with ssh, except it verifies the keys of both ends of the connection (it is host key plus authorized_keys file contents). Like ssh's key verification, it is intended to stop intruder-in-the-middle attacks from hijacking the connection even on first use.