COMP 3000: Operating Systems

Carleton University
Winter 2019 Final Exam Solutions

1. [2] Does `execve` close files that were previously open? How do you know?
   A: execve does not close files that were previously open. We know this because file descriptors 0, 1, and 2 (standard in, out, and error) are by convention always open when a program starts running. Thus they must be open before the execve and execve cannot close them.

2. [2] When does a call to `wait` return immediately? What does `wait` return?
   A: It returns immediately when a child process has already terminated. Wait returns the process ID of the process that has exited and its return value (the value returned by main).

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, and assume that you are running this as the user student.

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

   A: This program could either output "student" (if whoami exists, is executable by the student user, and operates as expected) or "Done!" (if the execve of whoami fails).

4. [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 kernel's information on each process is stored in each numeric directory in /proc. The comm file in each is the short name of the executable for that process.

5. [2] What is the PATH environment variable used for? Why is this variable used (rather than a process getting this information from another source)?
   A: The PATH environment variable stores the list of directories that should be searched for executables. This needs to be specified as otherwise a shell would have to search the entire filesystem for executables and might end up searching untrusted or otherwise problematic directories.

6. [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 stack-based variables can be allocated with just one instruction (assuming the stack doesn't need to grow). Heap allocation requires system calls.

7. [2] You're working on a Linux system where the setuid permission bit is ignored. What kinds of programs will break, and is this breakage significant?
   A: Programs that are run by the user but need root privileges, such as passwd and fusermount. These programs are not run routinely but will cause a non-privileged user problems when they wish to do certain tasks such as changing their password or using a userspace filesystem such as sshfs. (Full credit for mentioning at least one setuid program and what problems breaking it would cause.)

8. [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.

9. [2] Can signal handlers produce race conditions? How? Assume that the process is single-threaded (i.e., is a standard Linux process).
   A: Signal handlers can definitely produce race conditions by modifying data structures that are also being modified by the main code. For example, this could be a problem when a signal handler uses a standard library function that makes use of private static storage. If such a function is interrupted and then a signal handler calls it again, the internal storage will be in an undefined state. Lookup non-reentrant functions to learn more about this.

10. [2] Do you think a symbolic link can refer to a file on another filesystem? What about hard links? Explain briefly.
    A: 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.

11. [2] Why does `fsck` (on a non-journaled filesystem) have to walk through the entire filesystem hierarchy and inode table? Give an example of one specific error it could find.
    A: It has to walk through both because it has to look for inconsistencies, particularly with reference counts on inodes being inconsistent with the filesystem. If a file has a reference count of two and only one file refers to it, the reference count should be set to one. And, if an inode's reference count is 1 and it isn't referenced anywhere in the filesystem, a new filename should be made for it (and by convention it is placed in lost+found).

12. [2] Why is it important for file copying programs on Linux to treat sparse files differently? What problem can arise if sparse files are treated like regular files? And does this same problem apply to backup programs that compress their output?
    A: If sparse files are treated like regular files when copying, then it is possible for the copies to take up much more space than the original files (because the sparse regions in the original will have their blocks allocated in the copy). The same problem does not apply to backup programs when they are backing up, as the sparse regions will compress very well (as they are just a sequence of zeros). The same problem can arise, however, when restoring such a backup, in that the restored backup could take up much more space than the original files.

13. [2] When a process exits, does the kernel automatically reclaim the memory resources it was using? What about when a kernel module exits, is memory reclaimed? Explain briefly.
    A: When a process exits, the kernel automatically reclaims its memory because the kernel maintains a record of the memory associated with a process, its page table, and thus this record can be consulted to deallocate memory properly on process exit. When a kernel module exits memory is not automatically reclaimed because modules allocate memory as part of the kernel's own page table. No record is kept of what memory specifically belonged to the module (and indeed such ownership isn't well defined given the way modules can interact with the rest of the kernel). Thus the kernel doesn't and cannot deallocate all memory allocated by a module automatically when it exits.

14. [2] When a system call encounters an error (such as an invalid argument), how does it return an error to userspace? And, how does userspace receive this error? Explain briefly.
    A: To return an error, the kernel returns the error code (such as EINVAL) negated as the system call's return value. In userspace, this code gets put into the errno variable that can be checked

automatically using standard functions/macros. (Full credit for mentioning returning the code as the system call's return value and it being put into errno in userspace.)

15. [2] The functions `remember_read()` and `remember_write()` have an offset argument that is a pointer to `loff_t`. The read and write system calls, however, do not have offsets in their arguments. From the perspective of userspace, where is the offset stored? What about from kernel space?
A: From userspace offsets are an implicit part of the open file (referred to using a file descriptor). In kernel space the offset is stored as part of the FILE struct.

16. [2] In the remember module, `saved_data_order` specified the size of the static buffer. This variable does not specify the number of bytes to allocate (as you would with a call to malloc). What does it specify instead? Why is this different?
A: It specifies the number of pages to allocate as a power of two ($pages = 2^{order}$). Memory is allocated this way to minimize external fragmentation, as the powers of two mean you can always split up a non order zero allocation and get two of the previous (an order 3 allocation can be split into two order 2 allocations). This optimization is important in the kernel as the kernel often needs to allocate contiguous physical pages for tasks such as device I/O. Userspace allocations are all in an exclusive virtual address space so fragmentation is much less of an issue (although most userspace memory allocators do try to minimize fragmentation).

17. [2] The remember module code does not enforce mutual exclusion on the `saved_data` buffer. How could you enforce mutual exclusion? What would be the potential benefit of doing so?
A: You could enforce mutual exclusion by adding a semaphore (probably a spinlock) that could be used to enforce exclusive access to the saved_data buffer. Before remember_read or remember_write read or write to saved_data they would grab the semaphore (thus waiting if it was in use) and release it once they were done. The benefit of this is that is this prevents concurrent access to /dev/remember by multiple processes leading to undefined behavior, say by two of them writing at the same time, just as we did in 3000pc. Note the original remember module can crash when accessed concurrently.

18. [2] How can programs such as `bashreadline` observe the detailed behavior of large numbers of processes? Specifically, what is the key difference between bashreadline and other monitoring tools (such as `ps`), and why is this difference significant?
A: While ps just reads files in /proc to analyze kernel state, bashreadline uploads eBPF bytecode (produced by compiling a C program inlined in the python code) into the kernel. This code is verified and translated into machine code, giving it access to almost any code or data inside the kernel. eBPF thus allows for deeper and much more flexible monitoring of system behavior at the process and kernel level.

19. [2] When does the producer sleep in `3000pc`? Is this sleep essential for 3000pc to work, or is it simply there to improve performance? Explain briefly.
A: The producer sleeps when the queue is full (i.e., the producer as filled every slot and none have been emptied by the consumer yet). This sleep is essential because at this point the producer has nothing to do, it has to wait until the consumer makes progress. The only other alternative would be to do busywork, e.g. to loop until there was work to do. But this would consume CPU cycles with no benefit so it would be counterproductive.

20. [2] What's the difference between a regular mmap and an anonymous mmap? Why would you use one rather than the other?
A: A regular mmap associates a range of a process's address space with a range of bytes in a

file (thus the bytes in the file become "the same" as the corresponding the bytes in the mapped memory). An anonymous mmap has no associated file, thus it only allocates memory for an address range in a process's address space. In other words, an anonymous mmap does the memory allocation portion of a regular mmap without connecting it with a file.

21. [2] The Linux kernel's random number generator combines data gathered from multiple drivers in the kernel with a cryptographically secure pseudorandom number generator. Why must both of these components be used? (Note that the standard C `rand()` uses neither of these.)
A: The data gathered from multiple drivers produces the base entropy (unpredictability) that is necessary for a true random number generator. This gathered entropy however will generally be biased and the generator should output unbiased bits. The cryptographically secure pseudorandom number generator transforms these bits so they are unbiased and allows the kernel to then produce arbitrarily long that is unpredictable (as predicting its behavior would require reverse engineering the pseudorandom number generator, and a "cryptographically secure" one is resistant to this attack). (Full credit for saying the drivers provide uncertainty and the generator makes it unbiased.)

22. [2] Why do processes have a uid and an euid? Why not just use uid?
A: The euid is the user ID that is used for permission checks while the uid indicates the user controls the process. These two need to be different in UNIX/Linux because a user can run a program that has different privileges from them if the setuid bit is set.

23. [2] Why can it be hard to find race condition vulnerabilities? Explain with an example.
A: Race condition vulnerabilities can be hard to find because they only manifest under certain conditions, often very rare ones involving the relative scheduling of multiple processes or threads. In 3000log-write the TOCTTOU vulnerability (a race condition vulnerability) would most of the time not happen. Only when you ran it many many times (under specially crafted conditions) would the vulnerability show up.

24. [2] Eve wants to rewrite the semaphore implementation in the thread library she is using because she finds the code ugly and hard to follow. You are Eve's boss. Eve is a very talented but junior developer. What do you tell Eve?
A: Eve, please don't do this. Semaphore code is ugly and hard to follow because it must do very strange low-level things (such as use special machine language instructions) in order to ensure correctness; further, they are also generally highly optimized for performance. A clean implementation will likely have race condition errors and will be slower. If you really wish to improve the code, first study this code and other semaphore implementations. (Full credit for saying no and explaining how semaphores must use special instructions.)

25. [2] Alice wants to develop a security-related Linux kernel module. This module needs to read security policies from files. Bob suggests that she just use standard C library functions such as `fopen()` and `fgets()`, since the Linux kernel is written in C. Carol says Alice should instead have a regular process read the files and then write their contents to a character device. Whose advice should Alice follow? Why? (Be sure to analyze the merits of both Bob's and Carol's proposal.)
A: Alice should follow Carol's advice. Bob's advice is bad because you can't use file-related C standard library functions as the file descriptor abstraction isn't available in the kernel (as the kernel is what implements it). Further, reading files directly from kernel space can lead to many complications as the kernel code for doing this assumes it is running in the context of a system call (and a calling process). Carol's solution requires the kernel module to simply process data

via a write system call for the device, something we already have the code for, and the userspace code can be written in any language using whatever standard tools Alice wishes to use.

26. [2] The Linux kernel supports the signing of kernel modules. When this feature is enabled, only modules that have been properly signed can be loaded. Why would this feature be useful? And, if this was enabled, what information would the kernel need for it to accept a module that you compiled on your own?
A: This feature would be useful to prevent any user (including root) from loading unauthorized modules, as modules can violate any security guarantees the kernel would otherwise provide. The kernel would need the public key(s) that signed the modules added to it at compile time (so you'd have to add your public key to the kernel when you compile it if you wanted to sign modules).