COMP 3000: Operating Systems

Carleton University
Fall 2019 Midterm Exam Solutions

October 16, 2019

1. [2] How are environment variables like normal C global variables? How are they different?
   A: Environment variables are global variables, just like other C globals: they can be accessed from anywhere and changed. They are different because they are defined by the arguments to execve rather than being defined in the data segment of the program binary.

2. [2] When does a call to `wait` pause a process? When does it not?
   A: wait pauses the process when it has at least one child process running and none have terminated since the last call to wait. wait returns immediately when a process's child has terminated since the last call to wait.

3. [2] To redirect standard input, what must a process do, and how can the process do it?
   A: The process just has to change what file is open on file descriptor 1. It can do this by opening a file and then using dup2 to copy the returned descriptor to 1.

4. [2] For each of the following, specify what system call these library functions generate on Ubuntu Linux (as reported by strace): open(), opendir(), read(), close()
   A: open(): openat, opendir(): openat, read(): read, close(): close

5. [2] When it is created by a shell running in a terminal, where does a process get its input from (assuming no I/O redirection)? Where does its output go? Explain from the perspective of the process. Be specific.
   A: Standard input (file descriptor 0) and standard output (file descriptor 1) refer to a TTY of some kind, typically a pseudo TTY such as /dev/pts/0. So input is read from the TTY device and is written out to the TTY device.

6. [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: My friend is incorrect, signal handlers can be called at any time. If a process is blocked on a system call, the signal can interrupt the system call; however, signals can also be delivered when a process is doing something computationally expensive (e.g., calculating the digit of π). It just so happens that 3000shell spends most of its time waiting for input, thus signals tend to interrupt it when it is in the middle of a read.

7. [2] Can the "cd" command be implemented as a separate binary? Explain.
   A: The cd command cannot be implemented as a separate binary as it needs to change the current working directory for the process (using chdir). A process has to do this for itself, you can't run chdir for another process (as would be the case with an external command trying to change the directory for bash.)

8. [2] Standard UNIX shells support patterns for specifying file patterns such as `*.c` for every C source file. Does 3000shell support such patterns? Why or why not?
   A: 3000shell does not support such patterns because it does not implement file pattern (globbing) support and file matching is not provided by the operating system automatically.

9. [2] Do pipes on Linux, such as `ls | wc`, involve the creation of temporary files? Explain.
   A: Pipes don't create temporary files, both the source and destination process run at the same time and the standard out of the source is connected to the standard in of the destination.

10. [2] Are there situations when you cannot make a hard link to a file but you can make a symbolic link? Explain.
A: Hard links have to be on the same filesystem, while symbolic links can be anywhere. Hard links have this restriction because they are mappings of names to inodes, and inode numbers are defined per filesystem.

11. [2] Compare, at a high level, the contents of an object file (a `.o` file produced by `gcc -c`) and the that of an executable binary for the same program. How are they similar and different? Assume a program created out of one C source file.
A: A .o file contains the machine code and data allocations corresponding to the C source file. An executable contains that same machine code, but it also has the code necessary for setting up the initial runtime environment and either library code or the code needed to load libraries at runtime.

12. [2] Another student says "storing file access time was a design mistake in UNIX." What is an argument for this? What is one against it?
A: An argument for this is access times means that the system has to write to disk every time a file is read, thus mixing read and write traffic. An argument against this is access time can be useful to find files that have not been recently read (and so potentially aren't needed).

13. [2] On Linux x86-64, how are function arguments passed and how does this compare to where local variables are stored?
A: On x86-64, function arguments are normally passed in registers while local variables are allocated on the stack, so they aren't even stored in the same place. (Also acceptable: function arguments are close to local variables because function arguments are pushed onto the stack, then the return address, and then local variables are allocated on the stack. This second version is acceptable because when there aren't enough registers then arguments will be passed on the stack.)

14. [2] On Linux x86-64, are environment variables stored close or far away from program code? How do you know?
A: If you look at the addresses of environment variables and compare them to addresses of functions, you'll find that they are very far away: the environment variables are at the top of the process's address space while the code is much lower. You can confirm this by having a program report the addresess of elements of *envp[] and and of main().

15. [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: fork: to create a child process, execve: to load the program binary in the child, exit: for the child to terminate and send the return value to the parent, wait: for the parent to receive the return value from the child.

16. [4] Outline the basic algorithm for copying a file using read and write system calls. 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_WRITE—O_CREAT—O_TRUNC)**
- **as long as read(a) returns data, read(a, buf) then write(b, buf)**
- **close(a), close(b)**

17. [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)**