COMP 3000A: Operating Systems

Carleton University
Fall 2022 Final Exam Solutions

December 21, 2022

There are 28 questions on **4 pages** worth a total of 72 points. Answer all questions in the supplied text file template (available on Brightspace, titled `comp3000-final-template.txt`). Please do not corrupt the template as we will use scripts to divide up questions amongst graders. Your uploaded file should be titled `comp3000-final-username.txt`, replacing username with your MyCarletonOne username. Make sure you submit a UNIX text file and not another format (such as MS Word or PDF). Use a code editor, not a word processor! (You won't be graded for spelling or grammar, just try to make it clear.)

This test is open book. You may use your notes, course materials, the course textbook, and other online resources. If you use any outside sources during the exam, you **must cite** the sources. Your citation may be informal but should be unambiguous and specific (i.e., if you refered to the textbook, indicate what chapter and page you looked at rather than just citing the textbook). You **may not** collaborate with any others on this exam. This exam should represent your own work.

Do not share this exam or discuss it with others who have not taken it. Some students will be taking it at other times due to accommodations. Solutions will be released once everyone has finished th exam.

All explanations should be concise and to the point (generally no more than a few sentences, sometimes much less). If you find a question is ambiguous, explain your interpretation and answer the question accordingly.

You have 150 minutes. Good luck!

1. [2] In what circumstances would you expect the signal handler in 3000shell2.c to be called? Would you expect this to happen often?
   A: The signal handler will be called when 3000shell2 receives SIGHUP or SIGCHLD signals, because those are the signals signal_handler() has been registered for in main() (through calls to sigaction). We'd expect to get SIGHUP very rarely (say, when an ssh connection was terminated unexpectedly or a user sent the signal deliberately). SIGCHLD could be received every time a child process terminates, so it could happen for every external command run. As most commands will be run in the foreground, the call to wait in run_program() will take care of getting the return value of child processes, meaning the kernel doesn't need to send a SIGCHLD to the process; however, the signal handler will definitely have to handle the SIGCHLD generated by every backgound process. (1 point for identifying receiving SIGCHLD and SIGHUP, 1 point for reasonable explanations of when a process would get these signals. -0.5 if no mention of SIGHUP.)

2. [2] If you strace processes on the class VM, do you generally expect to see fork system calls? Why or why not?
   A: You don't expect to see fork system calls because the C library on the class VM makes clone system calls for fork library calls. (1 for not seeing them, 1 for explanation)

3. [2] What determines the available internal commands for a shell? What about the available external commands?

A: The availability of internal commands depends on the code of the shell itself, as the shell implements internal commands. External command availability depends on the programns installed in the directories in the current PATH, as any available executable file can be an external command. (1 for internal, 1 for external)

4. [5] For each of the following questions, answer `unshare`, `chroot`, both, or neither: **(For each, 0.5 for correct unshare classification, 0.5 for correct chroot classification.)**

   (a) [1] Can change how file paths are interpreted
   A: both

   (b) [1] Can cange the PID's associated with processes
   A: unshare

   (c) [1] Creates persistent files
   A: neither

   (d) [1] Can change how UID's are interpreted
   A: unshare

   (e) [1] execve's a new executable
   A: both

5. [1] When doing a call to `fork()`, how does the parent get the PID of the child process?
   A: The return value of fork, if it is greater than zero, is the PID of the child process.

6. [2] Is there a potential risk in running `sudo busybox --install` on the class VM? Why or why not? Assume you are running it in the normal student account just after login.
   A: This command creates hard links to the busybox binary in standard executable directories using the name of many common programs. If you didn't want to replace the standard versions with busybox's versions you could have a significant change/loss in system functionality. However, busybox checks for the existence of those files, so it only create links for programs that don't currently exist on the system. So this could be dangerous but in practice it isn't. (1 point for explaining what this command does, 1 point for either explaining how this could be bad or for noting it is pretty safe in practice. No points off if student didn't know busybox is smart about adding links.)

7. [8] Assume you run the following commands, and that the system's root filesystem is on the device /dev/sda2 (which is where the current directory is stored):

```
dd if=/dev/zero of=myimage bs=8192 count=60000
mkfs.ext4 myimage
mkdir mp
mount myimage mp
dd if=/dev/urandom of=file1 bs=8192 count=2
cd mp
dd if=/dev/urandom of=file2 bs=8192 count=2
```

   (a) [1] Could any of the above commands cause loss of data? Assume that nothing exists in the current directory with the name "myimage" and "mp". (Yes or No)
   A: No

(b) [1] How many `write` system calls were needed to create file1?
A: 2

(c) [2] Which of the above commands require root privileges? Why?
A: mount requires root privileges, because it changes the filesystems that are accessible and the system's directory structure.

(d) [2] What filesystem is file1 stored in? What about file2? Why?
A: file1 is stored in the root filesystem stored in /dev/sda2, because it was made in the top-level directory. file2 is in the myimage filesystem because it is mounted in the mp directory.

(e) [2] Did the creation of file1 increase the amount of data stored in /dev/sda2? What about file2? Explain briefly.
A: The creation of both file1 and file2 increased the amount of data stored in /dev/sda2 because file1 is stored in the root filesystem stored in /dev/sda2, and file2 increases the space used in the myimage filesystem and that filesystem's image is stored in the root filesystem which is stored on /dev/sda2.

8. [2] Answer the following questions about x86-64 assembly language:

(a) [1] What instruction is used to call a function?
A: call

(b) [1] What instruction is used to make a system call?
A: syscall

9. [1] It is a common convention to follow a call to `execve()` with a message output to standard error. What is the purpose of such a message?
A: If execve succeeds, the current program is replaced with that in the file given to execve. So if the statement after the execve runs at all, it means the execve has failed—so this is a natural place to output an error because an error indeed has happened (the execve failed).

10. [4] Assume a process opens a file X for writing, does an lseek that moves forward 1 MiB (2^{20}) bytes, does a write of 1024 null bytes, and then closes X. Answer the following questions. (Note: you may answer with expressions, you don't have to calculate the values.)

(a) [1] How many bytes can be read from X?
A: $2^{20} + 1024$**, or 1049600 bytes**

(b) [1] How many data blocks does the file use on disk? Assume each data block can hold 4096 bytes.
A: one data block

(c) [1] For this file, which is larger, its logical or physical size?
A: logical

(d) [1] Does X have a "hole"? Yes or no.
A: yes!

While you should have been able to figure out the answers based on your knowledge of lseek and file holes, you could have written a test program such as this:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>

int main()
{
        int fd;
        int shift;
        char buf[1024];

        fd = open("testfile", O_WRONLY|O_CREAT, 0444);
        shift = 1 << 20;

        printf("Seeking %d\n", shift);
        lseek(fd, shift, SEEK_CUR);

        memset(buf, 0, 1024);

        write(fd, buf, 1024);
        close(fd);

        return 0;
}
```

11. [2] In the confined environment, `nano` needs files from /lib/terminfo to function properly. `bash`, however, can function properly without these files. Why does `nano` need them and `bash` does not?
A: nano needs them because it operates in full-screen mode, i.e., it needs to position text at arbitrary positions in the terminal. In order to do this, it needs to know how to control the terminal, particularly where text is drawn on the screen. This is precisely what files in the terminfo directory document. bash just outputs text without regard to screen position, thus it can work on any terminal without requiring special information or having any special control. (1 for recognizing nano works in full screen and bash doesn't, and 1 for understanding that terminfo gives info on how to control terminals)

12. [4] Assume that your account is in the shadow group but has no extra privileges.

    (a) [2] Can your account validate passwords without having extra privileges? Why?
    A: Yes, because you can read /etc/shadow which has the password hashes for each account. /etc/shadow can be read by processes in the shadow group

    (b) [2] Can your account delete another user account? Why?
    A: No, because members of the shadow group cannot modify /etc/shadow or /etc/passwd, and they certainly can't delete a directory in /home, because those all require root privileges.

13. [2] Is it safe to give full access to system devices in a confined environment? Why or why not?

A: It is not safe, because then they can get access to critical devices such as the one holding the root filesystem (and thus will be able to mount it and modify any file on the system!

14. [4] Which of the following C functions/macros can be used from within a kernel module? Answer yes or no for each.

    (a) `printf`
    A: no

    (b) `snprintf`
    A: yes

    (c) `put_user`
    A: yes

    (d) `getpid`
    A: no

15. [2] After the c3000procreport module is loaded, does its code run continuously until it is unloaded? Why or why not?
    A: No, its code only runs in response to registered events because modules add code to the kernel but they don't create a new execution context (normally). Its functions only get called when the /dev/procreport is accessed; otherwise, its code isn't running.

16. [2] In the kernel source, `current` is a pointer to the task that made the system call that the kernel is currently handling. Does this pointer contain a virtual or a physical address? Explain briefly.
    A: Virtual address, because except for low-level memory management all pointers in the kernel are virtual because otherwise kernel code would have to deal with its code being spread out in page-sized chunks throughout the physical address space, or it would greatly make memory management more complicated because kernel memory would all have to be allocated contiguously.

17. [3] Can a `bpftrace` script be used to do which of the following? Answer yes or no for each.

    (a) Monitor calls to `puts()` made by 3000shell2.c
    A: yes

    (b) Monitor calls to `procreport_read()` in c3000procreport.c
    A: yes

    (c) Monitor write system calls made by 3000shell2.c
    A: yes

18. [2] When /dev/procreport is closed, what function in c3000procreport.c is called? How can you confirm this?
    A: procreport_release() is called when /dev/procreport is closed, we can verify this by either adding a printk statement to put a message in the kernel log or we can observe the output of watch_procreport.bt while a program opens, pauses, and then closes /dev/procreport.

19. [2] What special steps must a kernel module take before writing data to a userspace pointer? What happens in the class VM if you don't take these steps?
    A: The kernel must make sure to use the special macros/functions to access userspace such as put_user(). If it doesn't do this the memory operation will be detected as being not allowed and will generate a kernel oops log message.

20. [2] What is one function we can use to dynamically allocate memory in a kernel module? Why can't we just use `malloc()`?
A: get_free_page() is one that we've seen, there are many others for allocating memory in the kernel. We can't just use malloc() because it depends on system calls such as mmap and sbrk, and the kernel is what implement system calls so it cannot make them in order to implement its own functionality.

21. [2] What process's parent is always itself? Is this process special in any other way?
A: PID 1, the init process, is its own parent. It is special because it is the first process that runs when the system boots and when it terminates the system shuts down. Also, it becomes the parent of any process that loses its original parent.

22. [2] Does the hardware running the class VM have three, four, or five level page tables? How do you know?
A: It has four level page tables because while the Linux kernel assumes a five level page table, when we look at the lookups in procreport_get_physical() pgd and p4d are identical. We can also know this because you only need 5 level page tables to handle more than 256 TiB of RAM, and the SCS servers definitely don't have more than that!

23. [2] What is a C statement or declaration that could have generated the following assembly language code? Explain how each line is accounted for in your C code.

```
.LC0:
                .string  "alpha"
.LC1:
                .string  "beta"
.LC2:
                .string  "gamma"
animals:
                .quad    .LC0
                .quad    .LC1
                .quad    0
                .quad    .LC2
```

A: `char *animals[] = {"alpha", "beta", NULL, "gamma"}` **The first six lines are declaring the constant strings. "animals" labels the start of an array of four 64-bit values. All of these are pointers to the constant strings previously declared except for the third which is just a zero or NULL value.**

24. [4] Consider the following x86-64 assembly code and C code.
Assembly code:

```
movl the_number(%rip), %eax
cmpl %edi, %eax
jg .L6
jge .L4
leaq .LC1(%rip), %rdi
call puts@PLT
movl $-1, %eax
```

C code:

```
#include <stdio.h>

int the_number = 42;

int check_guess(int g)
{
        if (g < the_number) {
                puts("Higher!\n");
                return 1;
        } else if (g > the_number) {
                puts("Lower!\n");
                return -1;
        } else {
                puts("Got it!\n");
                return 0;
        }
}
```

(a) [2] What part of the C code does this assembly code implement? Be specific.
A: This code implements the core of the if statement, comparing g with the_number. The cmpl compares the two values, and the jg jumps to the code that implements the higher portion, the jge jumps to the got it branch code, and the rest of the code implements the lower portion, outputting the Lower! string and returning -1 (well, putting that value in the eax register to be returned).

(b) [2] Could the compiler have replaced the reference to `the_number` with the number 42? Why or why not?
A: No, because other parts of the program could modify the_number.

25. [2] How could you execve `/bin/ls`, giving it the command line argument of "-l /home"? Assume that you can use `environ` for the environment. Be sure to specify the exact arguments you would give to execve, defining any necessary data structures using C code.
A: For argv, you just need to define a suitable array of pointers to character arrays, with the last entry being NULL. You then give it as the second argument to execve. Code follows:

```
char *myargv[] = {"ls", "-l", "/home", NULL};
execve("/bin/ls", myargv, environ);
```

26. [2] How could you make a program "setuid student"? What privileges would such a program have that it otherwise wouldn't?
A: You make a program setuid student the same as you make it setuid root: you change its ownership to student with chown and then set the setuid bit on it (chmod u+s). Such a program would have access to student's files and could send signals to student's processes even if it was run by another non-privileged user. (So if you made a game that other users could run, if you made it setuid your user, you could have a high score file that other users couldn't update but your game could.)

27. [2] If 3000shell2 is setuid root, does it ever give up its root privileges? If so, when? If not, what are the security implications of this?
A: 3000shell2 gives up its root privileges just before running any external command unless the become command is used, specifically lines 188 and 189 in run_program() which sets the child process's effective gid and uid to the real ones, thus dropping root privileges.

28. [2] In bash, if I type "`ls > logfile`", what program **closes** `logfile`, bash or ls? Why?
A: ls closes the logfile because it is the one actually writing to the file. The file is initially opened by bash but then is handed off as ls's standard output; it can thus do whatever it wishes to with it and when it is done or it terminates, the file is closed.