COMP 3000A: Operating Systems

Carleton University
Fall 2022 Midterm Exam Solutions

October 13, 2022

There are 14 questions on **3 pages** worth a total of 30 points. Answer all questions in the supplied text file template (available on Brightspace, titled `comp3000-midterm-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-midterm-username.txt`, replacing username with your MyCarletonOne username. Make sure you submit a 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 80 minutes. Good luck!

1. [2] Answer the following questions about x86-64 assembly language:

   (a) [1] What instruction is used to return from a function?
   A: ret

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

2. [2] On Linux, processes must make the `sbrk` or `mmap` system call to allocate memory from the kernel. Does a C program make one of these system calls every time it calls `malloc()`? How do you know?
   A: Every malloc does not result in a system call, we can see this by running a program that does a small malloc (for example, by getline() in A1), we don't see any additional memory allocation system calls. (This happens because the C library creates a default heap at program start and only grows it if that heap gets filled up.)

3. [2] Why are calls to `fork()` normally followed by an if statement that tests the number it returns? Explain briefly.
   A: After a fork we have two processes running the same code, the parent and the child. Without an if statement, both parent and child will continue to execute the same code, duplicating work for no benefit. However, by checking the return value of the fork with an if statement, each process can tell whether it is the parent or the child and thus run the code that it should.

4. [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  "fox"
.LC1:
                .string  "chicken"
.LC2:
                .string  "wolf
animals:
                .quad    .LC0
                .quad    .LC1
                .quad    .LC2
                .quad    0
```

A: `char *animals[] = {"fox", "chicken", "wolf", NULL}`

The left side accounts for the animals: line and the four .quad declarations, as this literally defines four pointers, with the fourth being a null pointer (a zero). The right hand initializer defines the values for the first three quad values and their associated character arrays defined at LC0, LC1, and LC2.

5. [4] Consider the following x86-64 assembly code and C code.
Assembly code:

```
cmpl %edi, the_number(%rip)
je .L7
ret
```

C code:

```
#include <stdio.h>

extern int the_number;

void check_guess(int g)
{
        if (g == the_number) {
                puts("Got it!\n");
        }
}
```

(a) [2] What part of the C code does this assembly code implement? Be specific.
A: This assembly code corresponds to the comparison with g and the if statement, along with the function return when the equality test fails. It does not implement the block executed if the comparison is true which has a call to puts().

(b) [2] Do you think `the_number` appears elsewhere in the assembly code for this C code? Why? Explain briefly.
A: If this is the entire file, then `the_number` is only referenced in the assembly code above because it is declared as extern in C, which means that the assembly code can assume that it is defined in another object file.

6. [2] How could you execve `/bin/ls`, giving it the command line argument of "-l"? 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", NULL};
execve("/bin/ls", myargv, environ);
```

7. [2] In 3000shell2.c, there are two calls to `wait()`. The one in `run_program()` can cause 3000shell2 to pause before continuing, while the one in `signal_handler()` will never cause 3000shell2 to pause. Why does a call to the same function potentially produce different behavior?
A: Whether wait blocks or not depends on whether or not we must wait for a child process to terminate. In run_program(), we pause because wait at this point of the program a child processhas just been launched and there is likely no other children. In signal_handler(), however, we are calling wait because the process received SIGCHLD, meaning that a child process has already terminated. Calling wait then returns immediately with that terminated child process's status.

8. [2] On the current version of Ubuntu Linux, what system call does the `fork()` library call make? Could it make a different system call and still work? Explain briefly.
A: The fork() library call makes the clone system call. This library call could make a fork system call and it would work fine (assuming the process is single threaded, clone allows for finer-grained control that is useful when forking a multithreaded process.).

9. [2] Are the internal commands of bash the same as 3000shell2? Are the external commands the same? Explain briefly.
A: The internal commands of bash are very different from 3000shell2. bash has many, many internal commands (just look at the size of its man page) while 3000shell2 has three: exit, pinfo, and become. (It also supports output redirection and backgrounding processes, but those aren't really internal commands.) If they have the same PATH environment variable, however, both programs support the same external commands, as those are just the programs available in the directories specified in PATH.

10. [2] What two things must you do make a program setuid root? Give each command along with a brief explanation of what it does.
A: You must do two things to make a program binary setuid root. First, you must make the file owned by root: "chown root foo". Then, assuming it is already executable, we must set the user s bit with chmod: "chmod u+s foo". Note that most must be run as root, so if the current user isn't root these commands should be run with sudo.

11. [2] When a process creates a file, the file's owner is set to the filesystem uid of the creating process. The filesystem uid is normally set to be equal to the process's effective uid. If you make a program setuid root, will that change the default ownership of the files it creates? Explain.

A: Yes it will, files created will be owned by root. This is because when a program is setuid root, its effective uid is set to root (euid=0), and thus its filesystem uid is also set to root (fsuid=0).

12. [2] If 3000shell2 is setuid root, will every program it runs have root privileges? Why or why not?
A: If 3000shell2 is setuid root, regular programs run will execute as the user that ran 3000shell2. It will only run programs as root with the become command (and specifying root as the user). This happens because of lines 188 and 189 where the effective user and group ID's are set to the process's regular user and group IDs, thus dropping the process's extra privileges gained from being setuid root.

13. [2] If you wanted to change a user's uid, what is one file that you would definitely have to change? Why?
A: You would definitely want to change /etc/passwd, as this is the file that defines the mapping between usernames and uid's. (/etc/group, /etc/shadow, /etc/gshadow don't have any information on uid's.)

14. [2] In bash, if I type "`ls > logfile`", what program opens `logfile`, bash or ls? Why?
A: bash opens logfile, because the shell (in this case, bash) is responsible for doing I/O redirection. ls is just outputting to its standard out, a file that is (almost) always open when a process starts.