COMP 3000A: Operating Systems
Carleton University
Fall 2021 Midterm Exam Solutions

1. [2] Can you make a regular file that consistently behaves the same as /dev/null? Why or why not? Explain.

   A: No you cannot, because you can write whatever you want to /dev/null and if you then read from it you get no data. If you write to a regular file, later reads will return the data written to the file.

2. [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.

   ```
   defaultName    .quad    .LC0
   .LC0           .string  "Jane"
   ```

   A: The following C code could generate the given assembly:

   ```
   char *defaultName = "Jane";
   ```

   The first line is represented by the character pointer declaration, while the second is represented in the default string value ”Jane”.

3. [2] On x86-64 systems (the type of systems we have been using in class), is it possible for the stack and heap to grow to the extent that they would both run into each other? Do you think this is likely to happen in practice? Explain.

   A: In principle they could run into each other, if we could consume all of the addresses in between them. However, with a 64-bit address space there is a huge amount of room between them, much more than the RAM in all but the largest modern computer. We can see this if we run 3000memview and subtract the lowest value for the stack from the highest value for the heap (sbrk), we get a number that is at least 2^{45}. Given that a tebibyte (TB) is 2^{40}, this is more than 32 TB.

4. [2] Describe the data structure that holds command line arguments. Is it a simple array? Be precise using the C type definition and explaining what it means.

   A: Command line arguments are defined with the declaration `char *argv[]`**. This is an array of pointers to character arrays. The actual data structure is an array of pointers to character arrays, wich each character array terminated with a null character, and with the array of pointers terminated with a null pointer.**

5. [2] A friend of yours says that environment variables are an inefficiently stored key-value store. Is your friend correct? Explain briefly.

   A: Your friend is mostly correct. It is a key-value store, with the keys being the environment variables. It is not that inefficient in terms of space, in that it requires one pointer worth of extra bytes per entry (so 8 bytes on our machines). It is very inefficient in terms of lookup time, in that lookup time is linear in the number of keys: you have to check each element of the environment variable array and compare its key to the desired key, there’s no way to go directly to the desired key as you would in, say, a hash table.

6. [2] Is it easy for a program to change a value in its argv array? Is it similarly easy to add elements to argv? Explain briefly.

   A: It is very easy for a program to change a value in its argv array, in that it can overwrite any of the string pointers and have them point to another string in memory. It is significantly harder to add elements to the argv array as there is no guaranteed safe space for it to grow in to. To grow the argv array it would have to be copied to a new location and grown there.

7. [2] As shown by 3000memview, memory layout is randomized every time a program is run. We can disable this by using setarch. If we do an strace of `setarch -R`, we notice it makes the following system call just before execve'ing a program:

   ```
   personality(PER_LINUX|ADDR_NO_RANDOMIZE) = 0 (PER_LINUX)
   ```

   What does this system call tell us about what part of the system randomizes process memory layout? Explain briefly.

   A: This system call tells us that the kernel is the one doing the memory randomization, because all we need is this one system call in order to stop it.

8. [2] You're writing a program on a new version of Linux that has a new system call, `fastread`. Libraries have not been updated to support `fastread`. Can you make a pure, standards-compliant C program that calls the `fastread` system call? Why or why not?

   A: You cannot make a pure C program that will call fastread because standard C cannot make system calls. We either need to use a platform-specific compiler directive or inline assembly code if we want to make a system call because we have to configure the stack properly to pass the system call arguments and we need to execute the syscall instruction.

9. [2] Which is bigger *in process memory*, a statically-linked binary or a dynamically-linked binary? Why? Ignore any sharing of memory between processes or other multi-process memory saving techniques.

   A: In process memory, a dynamically-linked binary will be bigger because the entire shared library is loaded into the process's address space. With static linking, only the needed parts of the library are included in the binary and so are loaded into memory.

10. [2] `3000shell` can make many stat system calls for every command entered. What is the maximum number of stat calls `3000shell` will make in a given situation (while executing a command)? Explain.

    A: 3000shell will make one stat call for every path in PATH until it finds one that contains the desired binary. So, if the binary doesn't exist, it will make exactly n stat calls, where n is the number of paths listed in the PATH environment variable.

11. [2] System calls are much more expensive than function calls (in terms of execution time and in other ways). At a high level, how does the C library minimize the number of write system calls it makes by default? And, what is one way you can force C library functions to make more write system calls?

    A: By default the C library buffers writes until a newline character is output. Thus, if there are 10 calls to printf and only the last one contains a newline, the output of all 10 inputs will be consolidated into one write call (assuming the total amount of data is smaller than the buffer

size). You can force the C library to output what is in its buffer with the fflush() call. (You can also change the behavior of streams to be unbuffered so anything written to them will immediately produce a write system call.)

12. [2] In the C standard library, there are multiple functions for outputting formatted string data from a program. Here is the declaration of four of them:

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int dprintf(int fd, const char *format, ...);
int sprintf(char *str, const char *format, ...);
```

What is the difference between these four functions? Are any of them inherently less portable than the others?

A: printf outputs to the STDOUT file stream. fprintf outputs to whatever FILE struct (stream) specified. dprintf outputs to the given file descriptor, while sprintf outputs to the given string. dprintf is the least portable as only UNIX-like systems use an int as a file descriptor. The whole purpose of C file streams is to abstract file I/O in a portable way.

13. [2] What standard library function is used to associate a signal with a signal handler? Does this library function actually call the signal handler? Explain briefly.

A: sigaction is used to associate a signal with a signal handler. This function doesn't call the signal handler; it just tells the kernel about the signal handler using a special struct containing a pointer to the handler. The kernel will then call the signal handler when the process receives the corresponding signal.

14. [2] Tools like `gdb` and `strace`, that use the ptrace system call, have several significant limitations compared to `bpftrace` and other tools based on eBPF.

What is one thing you can do with eBPF that you can't do with ptrace? And, what key restriction is placed on eBPF programs that isn't there for ptrace programs?

A: With ePBF we can watch the behavior of all the processes on the system rather than just one at a time. Thus, you can do things like count the total number of system calls being made. The price for this is that you must have root privileges to do things with ePBF while unprivileged users can run ptrace on their own processes.

15. [2] What system call is used to send a signal? Can signals be sent without using this system call? Explain briefly.

A: The kill system call is used to send signals. Signals are sent for reasons other than a kill system call being made, for example when a child process dies, when a process divides by zero, or when it accesses unallocated memory. In these situations, the signal is sent by the kernel to tell a process that something has happened.