COMP 3000: Operating Systems

Carleton University
Fall 2017 Mid-Term Exam Solutions

1. [1] What system call is used to load a program binary in Linux? Does this system call create a new process? (yes/no)
   A: execve. No. (fork is used to create a process. execl, execv, etc. are library wrappers aound the execve system call and so are not correct answers.)

2. [1] Dynamically-linked programs generally make many mmap calls at the start of program execution that a statically-linked version of the program doesn't do. What are those mmap system calls for?
   A: They map libraries into the process's address space.

3. [1] Symbolic links map filenames to filenames. What do hard links map filenames to?
   A: inodes

4. [1] In the producer/consumer problem, when the producer encounters a filled queue (i.e., there is no room for it to produce anymore), what should it do?
   A: The producer should go to sleep until the consumer signals it/wakes it up.

5. [2] Consider the following implementation of fill_rand_buffer():

```
void fill_rand_buffer(rand_state *r)
{
        ssize_t count;

        count = read(r->fd, (void *) &r->buffer,
                     BUFSIZE * sizeof(unsigned long));

        r->current = (count / sizeof(unsigned long)) - 1;
}
```

   Explain the calculation of `r->current`. Why isn't it just set to the value of `count`?
   A: 1 mark for explaining that we divide count by sizeof(unsigned long) to get the number of unsigned longs read into the buffer and subtract 1 since `r->current` acts like an index (and indexing starts at 0). Effectively this points `r->current` to the last unsigned long in buffer. 1 mark for explaining that we don't set it to count since count is the number of bytes read into buffer, and it would make no sense as an index. (Any variation/other way to write the answers above gets full marks or part marks depending on amount of detail.)

6. [2] What are two standard uses of signals in Linux? (Do not list uses that depend upon application-specific behavior, e.g. SIGUSR1.)
   A: 1 marks each for explanations of any two signals from the man page of signals. eg. SIGTERM, SIGKILL, SIGINT, SIGCHLD, etc. Exception is SIGUSR1, or any custom signals. The signal names don't need to be mentioned. (0.5 as part marks for saying signals are used for IPC, even though this was specifically excluded by saying no to SIGUSR1).

7. [2] Do pointers in C contain virtual or physical addresses? Why?
   A: 1 mark for "pointers contain virtual addresses". 1 mark for "because the kernel/OS/MMU abstracts the physical memory and/or users only have access to virtual memory"

8. [2] Does a process make a system call to allocate memory? Why?
A: Yes. A process must make a system call like mmap() in order to allocate memory. The kernel controls all system resources, including the virtual-to-physical mapping of memory address space. As such, a process has to talk to the kernel to get more memory. System calls are how processes talk to the kernel. (1 point for Yes, 1 point for something like "because we need to ask the kernel for memory")

9. [2] At the prompt of `shell`, a user types `something > output.txt`. Which program opens the file `output.txt`, `shell` or `something`? Why?
A: 'shell' opens 'output.txt'. The shell parses the prompt string, extracts the filename, opens the file, and gets a file descriptor for it. Then, after forking, the child process—at this point still a copy of 'shell'—replaces its own file descriptor for 'stdout' with the output file's fd with a function like dup2. Finally, we run execve on 'something'. The modified descriptor for 'stdout' persists through the call to execve.

execve preserves file descriptors so a running program can do I/O in a generic way that allows programs to be combined together into larger solutions. A program can have as part of its specification, "Writes output to file descriptor 4", and whatever calls it can make that file descriptor refer to anything - a file, a process, even a device. Standard input, output, and error (file descriptors 0, 1, and 2) are the universal standards in UNIX, but there is nothing special about these descriptors other than convention, and in fact some programs specify purposes for other file descriptors.

(1 point for "shell opens output", 1 point for a mix of how and/or why)

10. [2] 3000test used mmap to compare the contents of two files. Standard tools for comparing files, however, use read rather than mmap. How could you use read system calls to compare the contents of two files of arbitrary size? Does this approach have any advantages over using mmap?
A: We can use the read() call to copy chunks of data into a buffer where as mmap() may load the entire file into memory. If we compare the contents of distinct and arbitrarily large files using mmap, it is possible to have both files entirely kept in memory (mmap does optimizations behind the scenes and loads data lazily, but makes no guarantees). We have no use for the bytes we previously looked at, so we can reuse the read buffers for the next chunk of comparisons.

We can create two small buffers for each file, read() into the buffers, compare them, and carry on. If at some point the buffers are not equal, we can stop there. If close to the start of the file, this approach isn't much better than using mmap(). It is at the end of the comparision, where we see that the files are equal, that using buffers has the clear advantage: the program uses a fixed amount of memory (2*buffer_length), whereas mmap() could use up to (2*file_size) bytes of memory.

(1 point for "use small buffers", 1 point for "mmap [might] load both entire files into memory")

11. [2] The `shared` struct in 3000pc.c is allocated using an mmap call with the MAP_SHARED flag. If this flag is changed to MAP_PRIVATE, how will the behavior of 3000pc.c change? Why?
A: If the flag is changed to MAP_PRIVATE, the shared data structure will no longer be shared between the producer and consumer, because MAP_PRIVATE memory regions become copy on write across a fork (like all other process memory normally). Thus the consumer will try to consume from an empty buffer while the producer will produce to a buffer that, once it is filled, will never be emptied. In other words, 3000pc will be completely broken. (1 point for saying something about things being broken, 1 for the explanation.

12. [2] You are trying to port 3000pc to a system that does not provide any implementation of sem_wait() or sem_post(). You search online and you find the following implementation of sem_wait():

```
int sem_wait(int sem)
{
        while (sem < 0) {
            /* wait */
        }

        sem--;
        return sem;
}
```

Does this code implement a correct semaphore? Why or why not? Explain briefly.

A: This code DOES NOT implement a correct semaphore for multiple reasons. First, the semaphore is passed by value not reference, so nobody else can change its value. Thus if the while loop condition is ever true this function will block forever. Second, because we aren't using special instructions it is possible for another process/thread to change the value of sem between the time the loop finishes and the decrement; thus while it may work correctly some of the time this implementation will break sometimes. Doing a busy loop rather than sleeping is inefficient for long waits but is very efficient for short ones; thus the Linux kernel sometimes uses semaphores that busy wait. They are known as spinlocks. (1 point for saying no, 1 for a correct reason why)