CS 346 – Answers to review for Test 1

1. Give a brief definition of each of the following OS terms.
   a. kernel
      the part of the OS that is always running, doing the most essential tasks

   b. shell
      the part of the OS that performs interactive I/O with the user, taking user commands and displaying standard output and error messages

   c. system call
      a function call in a program that requests an OS service

   d. context switch
      performing the steps necessary to swap out one job from the CPU and replace it with another from the ready queue

   e. POSIX
      a standard API for C programs to make system calls in a Linux or UNIX environment

   f. multiprogramming
      allowing the system to run several jobs at the same time by having them take turns in the CPU

   g. redirection
      having the standard output go somewhere besides the terminal screen (such as to a file), or having the standard input come from somewhere besides the keyboard

   h. device driver
      software that is necessary to use a piece of hardware that is installed on the system

   i. memory management
      deciding what should be in RAM versus secondary storage

   j. process
      a living program, one that has started by not yet finished

k. thread-join operation
a function call that allows a main thread to wait until other thread(s) have completed, analogous to wait( ) with processes

2. Why is it not possible to implement an entire operating system in a high-level language?

An operating system needs to be able to have direct access to specific registers, and certain assembly-level operations that manipulate the hardware. This functionality does not exist in a high-level language.

3. An assembler is a program that translates assembly language instructions into machine code that can be directly interpreted by the hardware. Would an assembler be considered part of the OS kernel? Justify your answer.

No. The assembler is only needed when one makes updates to source code. The assembling of instructions can be done at any time before a program needs to run, so it is not part of the operating system.

4. Describe the different states that a process can be in during its lifetime.

New – a process is created and acquires resources such as memory
Ready – eligible to execute, but there is already some other process controlling the CPU
Run – currently executing instructions on the CPU
Wait – cannot run because it needs to wait for I/O or other signal to wake it up
Terminated – temporarily hold exit status long enough for the parent to take note of it

5. What does this Linux command accomplish?

```
ls | grep 5 | wc
```

Determine how many files in the current directory have the digit 5 in their names.

6. Describe an example of a program where creating multiple threads would make more sense than creating multiple processes.

For example, we can split the data: I want to know how many 9 letter words there are in the dictionary. Split the dictionary up into parts, and give each part to a thread. The counting can be done within the threads, and when they have all completed, we can add their results.

7. Explain what an exec() system call does. When is the appropriate situation to invoke this call?

   This system call specifies to a child process what program it is to run, in place of the code of the parent process. We call exec( ) immediately after calling fork( ).

8. What information is shared among all the threads of the same process? What information can differ among each of the threads?

   They share the code, data segment, and open files.
   On the other hand, the contents of variables stored in registers, stack and heap will differ, as will the program counter (where in the program it is currently running).

9. Consider the following abbreviated process table.

PID	PPID	NAME
7205	7199	Bash
7325	7205	a.out
7326	7325	a.out
7327	7325	a.out
7329	7327	a.out
7330	7327	a.out

   a. Describe the hierarchical relationship among the 5 processes called a.out.

      The parent is 7325. It has two children 7326 and 7327. The second child, 7327, itself has two children 7329 and 7330 (making them grandchildren of the original a.out process).

   b. Give a pseudo-code design of a C program using fork() calls in order to create child processes suggested by the process table. Assume that fork() returns 0 in the child, and a positive integer in the parent.

```
pid1 = fork( )          // create first child
if (pid1 > 0)
   pid2 = fork( )       // create second child
   if (pid2 == 0)
      pid3 = fork( )    // create first grandchild
      if (pid3 > 0)
         pid4 = fork( )  // create second grandchild
```

10. Processes P1 and P2 need to access a critical section of code. Consider the following synchronization construct used by the processes. Assume that the critical sections of code have very short execution times.

```
/* Process P1 */
while (true) {
  wants1 = true;
  while (wants2 == true)
    ;
  /* Critical section */
  wants1 = false;
  /* Non-critical code */
}
```

```
/* Process P2 */
while (true) {
  wants2 = true;
  while (wants1 == true)
    ;
  /* Critical section */
  wants2 = false;
  /* Non-critical code */
}
```

Here, wants1 and wants2 are shared variables, which are initialized to false. Answer the following questions about the above implementation of P1 and P2. Justify your answers.

a. Does it ensure mutual exclusion?

Yes, each one will wait if the other one "wants." They can't both be in their critical section at the same time because of the while loop spinning before each critical section.

b. Is deadlock possible?

No, but something close, livelock, is possible. If P1 sets wants1 to true, and then control is immediately then passed to P2, and P2 sets wants2 to true, we now have both processes spinning. Each is waiting for the other, but they are not asleep.

c. Is starvation (unbounded waiting) possible?

Yes, see answer for (b).

d. Must the two processes access their respective critical sections in strict alternation? (Strict alternation means that a process having just finished its critical section cannot re-enter it until the other process has entered and exited its critical section.)

No. The way the code is written, P1 could go back to the top of the outer while loop and re-enter its critical section. This could happen if the scheduler happens to give P1 a lot of time to do this.