CS533 Midterm Exam

Answer as many questions as you can. Explain your answers clearly and concisely using your own words. Do not simply reproduce terms and phrases from the papers you have read. Hand in your answers or email them to walpole@pdx.edu with the subject heading CS533 MIDTERM.

1. Explain how the value of a global integer variable `counter` might end up not reflecting the number of times it has been incremented when the following statement is executed concurrently by more than one thread. Illustrate the underlying cause of this problem by using low-level pseudo code to outline an execution sequence that produces the problem.

   ```
   counter = counter + 1;
   ```

2. In reasoning about the behavior of sequential programs we can depend on the property that values in memory will not change unless the thread of execution changes them. However, in concurrent programs this property sometimes holds and sometimes does not hold. Explain where it holds, and when it ceases to hold, in lock-based concurrent programs that follow the locking rules. Similarly, for concurrent programs with monitors and condition variables, explain where it holds and when it ceases to hold. Be sure to explain the reasoning behind your answers, and explain the implications this has for the programmer.

3. Explain why blocking I/O is problematic in an event handler? Outline a scheme for handling I/O in event-based programming. Illustrate your answer by outlining the steps involved in completing the execution of an event handler that contains an I/O call.

4. Why are page faults a problem for programs based on user-level threads? Why is a scheduler activations-like mechanism potentially helpful in dealing with them?

5. Explain how a remote procedure call, and corresponding return, can be performed, without kernel involvement, between threads in two different address spaces on a multiprocessor.

6. Explain why the thread_start and thread_switch functions of a user-level threads library have to be implemented at the assembly level.

7. In a program using the user-level threads system developed in Assignment 1, if `thread_wrap` is allowed to return, the program (including all other threads) will crash. Why? How can this be prevented?

8. As the implementer of a user-level threading system with a join operation, when is the earliest time you can free (deallocate or reuse) a thread’s (i) activation stack and (ii) thread control block?

9. Outline, using pseudo code, the implementation of a yield operation for a round-robin scheduler. Assume that the current thread’s thread control block can be referenced globally, and that the possible thread states are RUNNING, READY, and DONE.

10. When a user level thread calls `yield()` and enters the thread scheduler, certain scheduler-related variables, such as “current thread”, must be changed to refer to the next thread. Do these changes mean that the calling thread is no longer running? If not, is there a definitive point at which the calling thread ceases to run and the next thread starts to run, and if so, what is it? Illustrate your answer by outlining the sequence of low-level operations that take place during such a switch.