If you haven’t recouped the 5 points of question 5 on the midterm, NOW is the time to do it because if you don’t understand interleaving and how to analyze races by analyzing alternative interleavings, you will find the current assignment meaningless and impossible!

Note that in the detailed implementation of Peterson’s algorithm on page 181 of AOSC, the role played by the shared variable turn is reversed compared with my lecture notes. In AOSC’s version, the value of turn equals the number of the thread whose turn it is to run rather than to wait.

1. Explain why Peterson’s solution (Algorithm 3 in AOSC Chapter 7) requires the call to Thread.yield() whenever the scheduling is non-preemptive. Remember that Java runtime implementors are permitted to choose between preemptive (i.e., time sliced) Java thread scheduling or cooperative scheduling. Include an explanation of one or more specific failures that can occur when the scheduling is non-preemptive and the call to Thread.yield() is replaced by the statement “;” that does nothing. Which of the failures depend upon races? (Be sure to review the definition of race.)

(Remark: The solutions using yield() in AOSC also require the relevant Java Threads to have equal Java priorities. See AOSC page 205 for details. For the purposes of this problem you should assume the Threads have equal Java priorities and then you can ignore the issue.)

2. (a) Trace Peterson’s algorithm for all significantly different cases. You can apply symmetry considerations to reduce a lot of work. For example, you should assume that in all cases, the operation flag[t]=true executed by process (thread) 0 is done first. Demonstrate that the algorithm produces

   i. mutual exclusion,
   ii. no deadlocks, and
   iii. access to the critical region for each thread that requests it.

   in ALL cases.

(b) Figure out and demonstrate (with a specific interleaving) what goes wrong if Peterson’s solution were modified so the turn = other operation is replaced by turn = t. (In other words, each thread sets turn to command the other thread to wait if the other thread is requesting entry into, or is already executing in the critical section.)

(c) Figure out and demonstrate (with a specific interleaving) what goes wrong if Peterson’s solution were modified so the operations flag[t]=true and turn=other were programmed in reverse order in the programs of both processes.

(d) Figure out and demonstrate (with a specific interleaving) what goes wrong if Peterson’s solution were modified so the operations flag[t]=true and turn=other were programmed in reverse order in the programs of the first thread (thread 0) but not in the second.
3. What goes wrong if the operations semaphore “down()” operations `empty.P();` and `mutex.P();` were programmed in reverse order in the producer’s function in Figure 7.14? Demonstrate with one specific interleaving that causes failure. What is the failure?

4. Design an implementation of **counting semaphores** that uses binary semaphores and together with ordinary operations, and express the details in Java.

5. AOSC Chapter 7 problem 7.5 (Solve the readers-writers problem so that the waiting writers are protected against starvation due to the arrival of subsequent new readers.) Express the solution in Java; you might obtain it by modifying the code given with the textbook, but design the solution first on paper!