

**Problem 1—this is due NEXT CLASS, 10/4**

Suppose the concurrent execution of two threads with a shared integer variable `COUNT` is modelled so that the *atomic operations* are `regA=COUNT` `regB=COUNT`, `COUNT=regA+1` and `COUNT=regB-1`. Assume `regA` and `regB` are separate “registers”.

Assume the initial value in `COUNT` is 0.

Assume the programs followed by the threads are

ThreadA:	ThreadB:
(A1) <code>regA=COUNT</code> ;	(B1) <code>regB=COUNT</code> ;
(A2) <code>COUNT=regA+1</code> ;	(B2) <code>COUNT=regB-1</code> ;

There are  $6 = \binom{4}{2}$  interleavings of these atomic operations; they represent all possible variations of concurrent execution of these threads under the given model. List (1) all 6 interleavings; and for each, (2) show what happens **each** time the value of `COUNT` changes, (3) show the final value of `COUNT`, and (4) indicate for which interleavings the result is wrong because of the bug caused by races.

**Problem 2—this and the rest are due 10/6**

Note that in the detailed implementation of Peterson’s algorithm on page 223 of OSCJ, the role played by the shared variable `turn` is reversed compared with my lecture notes. In OSCJ’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 OSCJ 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 OSCJ also require the relevant Java Threads to have equal Java priorities. 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.

### Problem 3

1. 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 7.12? Demonstrate with one specific interleaving that causes failure. What is the failure?
2. Construct at least one specific scenario in which the failure described in section 7.8.1.1 occurs. A specific scenario includes the detailed sequence in which concurrent Java operations are executed as they are interleaved from multiple threads. You will have to make certain assumptions about how the Java scheduler operated when threads have different priorities. Your answer to this question must state explicitly what assumptions you had to make in order to bring about your one or more scenarios.
3. Construct at least one specific scenario in which the failure due to race conditions mentioned at the very beginning of section 7.8.1.2 occurs. To do this, assume the `synchronized` keywords were removed from the code of Figure 7.29. Find one or more specific interleavings in which demonstrate how race conditions lead to data structure inconsistency and application failure.

For each scenario, explain specifically what would happen if the `synchronized` keywords were replaced. Your answers should demonstrate how the failures you found before are now prevented.

4. The rest of section 7.8.1 motivates the Java `wait()/notify()` facility. Construct one or more specific scenarios in which the failure described on page 208, top paragraph, does occur. For credit, your construction must show a specific sequence of Java operations constructed by interleaving operations from multiple threads.
5. I think the authors made a mistake when they used the word “blocked” in “(1) the producer is blocked waiting for the consumer to free space in the buffer” Exactly which code is the producer thread executing in Fig. 7.28 when it is waiting for the consumer to free the space? Do you agree with me about whether the producer is “blocked?” To answer this question you must cite a specific definition for the word “blocked”.