Homework #2

1. **In the mutex-locking pseudocode of Figure 4.10 on page 111, there are two consecutive steps that remove the current thread from the runnable threads and then unlock the spinlock. Because spinlocks should be held as briefly as possible, we ought to consider whether these steps could be reversed, as shown in Figure 4.28 [on page 148]. Explain why reversing them would be a bad idea by giving an example sequence of events where the reversed version malfunctions.**
    a. In Figure 4.10 on page 111, when locking a mutex, where the mutex has been already locked, we place our thread into the wait queue of the mutex, remove the current thread from the runnable queue, and then we unlock the mutex's spinlock before we yield to a runnable thread. However, in Figure 4.28 on page 148, the mutex's spinlock is unlocked before the current thread is removed from the runnable queue. This is a bad idea giving the following example. Let's say thread A and thread B are executed concurrently. There exists an unlocked mutex where Thread A locks the spinlock, toggles the mutex's lock state, and then unlocks the spinlock. Therefore, Thread A is still in the Runnable Queue. Thread B comes along and attempts to lock the mutex. Since the mutex has already locked, after toggling the mutex's spinlock state to lock, Thread B is placed into the wait queue and unlocks the spinlock. Before Thread B can be removed from the runnable queue, Thread A is dispatched by the scheduler to the running state and attempts to unlock the mutex. Since the spinlock has been unlocked, after Thread A acquires the spinlock, Thread B is dispatched from the wait queue to the runnable queue. However, since Thread B has not finished executing after unlocking the spinlock, it continues by removing itself from the runnable queue which in turns allows itself to never be able to complete its task of locking the mutex for its intended purpose.
2. **Suppose the first three lines of the audit method in Figure 4.27 on page 144 were replaced by the following two lines: (Explain why this would be a bug.)**
    a. This would be a bug because let's say that after we declare and initialize the "seatsRemaining" variable, the AtomicReference is updated during a concurrent execution. Therefore, by the time we invoke the get method on the Atomic Reference, during the declaration and initialization of the "cashOnHand" variable the thread is dealing with a new snapshot that is inconsistent with the snapshot received for our "seatsRemaining" variable.
3. **IN JAVA: Write a test program in Java for the BoundedBuffer class of Figure 4.17 on page 119 of the textbook.**
    a. Source code can be found in our repository
4. **IN JAVA: Modify the BoundedBuffer class of Figure 4.17 [page 119] to call notifyAll() only when inserting into an empty buffer or retrieving from a full buffer. Test that the program still works using your test program from the previous exercise.**

a. Source code can be found in our repository

5. **Suppose T1 writes new values into x and y and T2 reads the values of both x and y. Is it possible for T2 to see the old value of x but the new value of y? Answer this question three times: once assuming the use of two-phase locking, once assuming the read committed isolation level is used and is implemented with short read locks, and once assuming snapshot isolation. In each case, justify your answer.**
    a. Two-phase locking
        i. Dealing with readers/writers locks during an atomic transaction, the case above would not be possible. In this serialized approach, the transaction awaiting to read the values would have to wait until the x and y values were both written. Also, if the transaction as a whole failed, then T2 would be reading the old values of x and y.
    b. Read committed isolation with short read locks
        i. In this case, transaction locks are exclusive for writes while all read operations use a shared lock for reads. An issue with read committed isolation is the idea of non-repeatable reads, or when the same thread accesses the same memory location twice and reads in two different values. With the write lock being exclusive only to writes the following can occur: T2 acquires the shared lock and reads in the old value of x while it's being written by T1. With the acquired write lock, T1 writes new values into x and y. Finally, T2 completes its operation by reading in the newly written value of y.
    c. Snapshot isolation
        i. Snapshot isolation is a result of multiversion concurrency control where every new write stores a new value in a different location than it's previous value. Any read operation only reads from the newly written/committed value and therefore eliminates the opportunity of T2 reading in the old value of x.
6. **Assume a page size of 4 KB and the page mapping shown in Figure 6.10 on page 225. What are the virtual addresses of the first and last 4-byte words in page 6? What physical addresses do these translate into?**
    a. Virtual address for first 4-byte word in page 6: 4096 * 6 = 24,576
        i. Physical address: 4096 * 3 = 12,288
    b. Virtual address for last 4-byte word in page 6: 24,576 + 4092 = 28,668
        i. Physical address: 12,288 + 4092 = 16,380
7. **At the lower right of Figure 6.13 on page 236 are page numbers 1047552 and 1047553. Explain how these page numbers were calculated.**
    a. The IA-32 Architecture contains a two-level page table that contains a page directory that points to chunks of the page table. Those chunks consist of page table entries and the page directory can point to 1024 of those chunks. The IA-32 architectures uses 4-KB pages containing 4-byte page table entries; therefore, each page-table chunk can store up to 1024 entries. Any one of those 1024

chunks can point to 1024 page frames. Using this knowledge we can compute the page numbers 1047552 and 1047553 by looking through each chuck pointer(1024), looking through each page frame pointer for a particular chuck(1024 * 1024) and then looking at the first, and second, page frame pointer in the last possible chunk ((1024 * 1024) - 1024) = 1047552 and ((1024 * 1024) - 1023) = 1047553.

8. **Write a program that loops many times, each time using an inner loop to access every 4096th element of a large array of bytes. Time how long your program takes per array access. Do this with varying array sizes. Are there any array sizes when the average time suddenly changes? Write a report in which you explain what you did, and the hardware and software system context in which you did it, carefully enough that someone could replicate your results.**
    a. Source code can be found in our repository
9. **Figure 7.20 [page 324] contains a simple C program that loops three times, each time calling the fork() system call. Afterward it sleeps for 30 seconds. Compile and run this program, and while it is in its 30-second sleep, use the ps command in a second terminal window to get a listing of processes. How many processes are shown running the program? Explain by drawing a family tree of the processes, with one box for each process and a line connecting each (except the first one) to its parent.**
    a. Source code can be found in our repository