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.

The mutex spinlock is unlocked before the current thread stops being marked as runnable. This will allow the thread to be run again immediately, and, if the scheduler handles this too consistently, could result in an endless loop of a single thread constantly checking and seeing a locked mutex.

2. Suppose the first three lines of the audit method in Figure 4.27 on page 144 were replaced by the following two lines:

```
int seatsRemaining = state.get().getSeatsRemaining();
int cashOnHand = state.get().getCashOnHand();
```

Explain why this would be a bug.

If the state is used to get the values of remaining seats and cash instead of a snapshot, incorrect values of cash on hand and seats remaining may be obtained. This is because a snapshot ensures that values obtained are those before the start of any update. If the state is used, one value might have been updated by another thread while one value shows the old value and incorrect values will be used in the system to vend tickets.

3. **IN JAVA**: Write a test program in Java for the **BoundedBuffer** class of Figure 4.17 on page 119 of the textbook.

The code is under the homework02 folder in the repo. It is titled BoundedBuffer.java

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.

The code is under the homework02 folder in the repo. It is titled BoundedBufferForQuestion4.java

The tests for both questions 3 and 4 can be found in the BoundedBufferTest.java file.

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) Assuming Two Phase Locking: It is possible for T2 to see the old value of x but the new value of y. This is because in Two Phase locking, the transactions acquire locks before accessing data items and hold the locks until the transaction is complete. So if T1 writes new values into x and y, it will hold the shared lock on both x and y until the end of the transaction. For T2 to read values of x and y, it has to wait for the shared lock to release. If T2 acquires the shared lock on x and y after y is updated but before T1 does, it will see the new value of Y but the old value of X.

b) Assuming read committed isolation level with short read locks: It is possible for T2 to see the old value of x but the new value of y. This is because in this system, T2 acquires short shared locks on x and y during reads and releases them immediately after reading. T1 can write new values into x and y without locking them as locks are not required during writing. So if T2 reads the value of x before T1 updates it, T2 will see the old value of x while y might have been updated by T1.

c) Assuming Snapshot Isolation: It is not possible for T2 to see the old value of x but the new value of y. This is because in snapshot isolation, each transaction sees a snapshot of data as it existed at the start of the transaction. Hence, T2 will always see a snapshot of x and y as they existed before T1 began the transaction to update both values. T2 will only see a snapshot of both updated values of x and y after T1 has completed updating them.

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?**

    The virtual addresses of the first and last 4-byte word in page 6 are 24576 and 28668 respectively. Their physical addresses are 12,288 and 16,380 respectively.

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.

    There are 1024 chunks and each chunk references 1024 frames. The final page chunk, therefore, starts at page frame number 1023*1024=1047552.

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.

The program is titled vmArrayTimer.c in the homework02 folder. To run the program and get the time it took, I did it this way. I run the command:

```
date; ./vmArrayTimer size_of_array; date;
```

This printed out the start time of the program, run the program, and gave me the end time of the program at the end. To find how long the program took, I just subtracted start time from end time. This was done on a macOS computer with intel chip. The results are listed below.

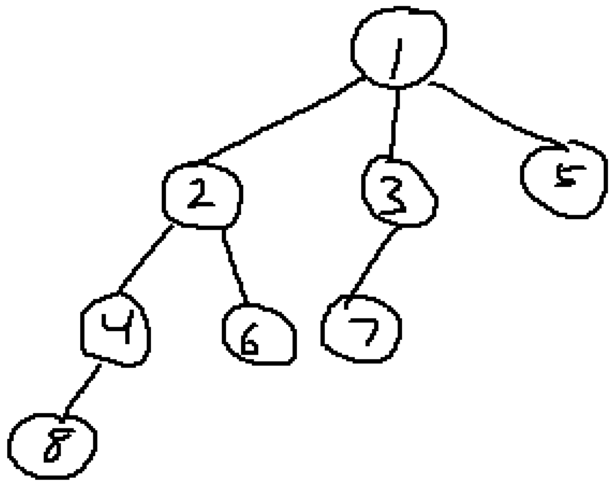
Array size	Start time	End time	Time taken
100 000	16:36:31 PST	16:36:31 PST	0s
1000 000	16:45:00 PST	16:45:00 PST	0s

100 000 000	16:50:32 PST	16:50:33 PST	1s
1 000 000 000	17:07:22 PST	17:10:36 PST	3minutes
10 000 000 000	17:01:57 PST	17:06:39 PST	5minutes

The time jumped significantly at size 1 000 000 000. The time taken increases as size increases.

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.

8 processes are running the program.



Assumes the threads run in order of creation, which is probably false.