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.**

If you unlock mutex.spinlock before removing the current thread from the runnable threads, the current thread is still runnable even though it is in mutex.waiters. When the code reaches mutex.spinlock it will search for the next thread which is still the current thread. This means the current thread will remain in the waiting position and that the code will never remove the current thread for the runnable threads, the next thread will never be reached.

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.
A non blocking version of the TicketVendor requires a more powerful atomic instruction that can package the actual updating of the TicketVendor with the obtaining of permission. The compare-and-set instruction meets this need. The sellTicket method attempts to make progress using the following method invocation: state.compareAndSet(snapshot, next).

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

In the actual code a snapshot is created, seats remaining and cash on hand is obtained from that snapshot alone. Whereas in the given modified

code(assignment) the snapshot is created twice and then the values are obtained which is a bug as both the snapshot are not from the same state.

3. IN JAVA: Write a test program in Java for the BoundedBuffer class of Figure 4.17 on page 119 of the textbook. ONLY WRITE THE TEST PROGRAM ~ DON'T MODIFY THE CODE FOR THIS ONE.

https://github.com/sgowdaks/Operating-Systems-CMSI-3510/tree/main/HomeWork/HomeWork2/Boundedbuffer

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.**

https://github.com/sgowdaks/Operating-Systems-CMSI-3510/tree/main/HomeWork/HomeWork2/Boundedbuffer_modified

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. For two-phase locking
Yes, because two-phase locking ensures that the old state of x will exist and be preserved in the serial history guaranteeing consistency from state to state and can be read as well as the new value of y.

b. For read committed isolation with short read locks

Yes, however a dirty read can take place where T2 can access the same memory twice and get two different values. If T2 tries to read the old value of x and new value of y many times, it may get different values.

c. For snapshot isolation
No, because any reads will only read the most recently committed value. So T2 will only be able to see the new value of both x and y.

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 6th page is mapped to 3rd page frames.
Hence the virtual address is
1st -> 4096 * 3 = 12,288
Last 4th byte -> (4096 * 4) - 4 = 16380
Physical address is
1st -> 4096 * 6 = 24,576
Last 4th byte -> (4096 * 7) - 4 = 28,668

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.**

The page directory can point to 1024 chunks of the page table that each point to 1024 page frames. Page frames are accessible in the range of [1024i, 1024(i + 1) - 1] where, i, is the page directory index. If we look at the last chunk, which is index 1023, we get the range 1047552 to 1048575. So, the first accessible page frame is page 1047552, and one index over we can access page 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.

System used -> macbook pro 13
IDE -> atom
Inputs given in Iterminal
The array was initiated with random values below 100, and the while loop continues until the given input value is reached. To get the time

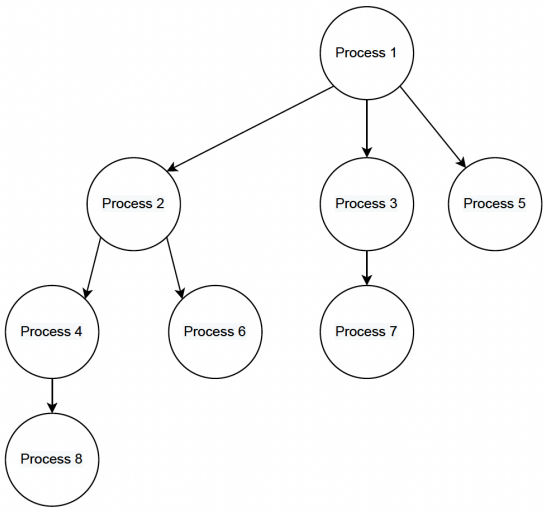
time ./a.out <arraySize>

was given. Which gave real, user and sys time, and for the report real time was considered.

ArraySize	Average Time1	Average Time 2	Average Time 3
10,000	0.007s	0.007s	0.007s
1,00,000	0.007s	0.007s	0.007s
10,00,000	0.017s	0.018s	0.017s
1,00,00,000	0.081s	0.079s	0.083s

As the array size increases the real time increases but for the same array size the real time is consistent.

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.**
There are 8 processes running from the program and 2 from bash for a total of 10 processes.


Process 1
Process 2
Process 3
Process 5
Process 4
Process 6
Process 7
Process 8