Homework02

Ameya Mellacheruvu, Clarissa Bernardo, and Krysten Tachiyama

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.**

   By unlocking mutex.spinlock before removing the current thread from runnable threads, the current thread remains runnable. When the code unlocks mutex.spinlock, there is a possibility that the current thread never gets removed from the runnable threads and therefore prevents the code from getting to the next thread.

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.

   To ensure a safe way to dispense a ticket we must atomically compare and set a snapshot of the current situation in seats that we have remaining and available cash on hand. Storing the snapshot state and retrieving information from that state will yield a consistent output. The only way to change the output is by getting another snapshot to then store. In the code:

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

   state.get() changes the snapshot. The code above calls to change the snapshot twice leading to two different snapshots that getSeatsRemaining() and getCashOnHand() are getting information from. This is a bug because the 2 methods are reading from different snapshots.

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

See Test.java and 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.**

   See BoundedBuffer_Modify.java and Test_Modify.java

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.**

   Two-phase locking - not possible. This type of locking serializes between transactions.

   T1 writes into x:
   e1(x); r1(x; 5);w1(x; 8; w1(x;10)(recent value;ē1(x)(unlocks);
   T1 writes into y:
   e1(y);r1(y;5);w1(y;7) recent value.ē1(y)(unlocks);
   T2 reads into x:
   e2(x);r2(x;10); ē2(x)
   T2 reads into y:
   e2(y); r2(y;7);ē2(y)

   Read committed isolation with short read locks - not possible. The exclusive locks are write locks. Read committed acquires a shared lock before each read operation and release immediately after the read. The shared lock prevents reading data written by a transaction that is still in progress as that transaction will hold the lock in exclusive mode.

   Snapshot isolation - it is possible. Snapshot Isolations can read recent and older versions; They can read all entities from the version that was most recently committed when the transaction started. No read locks are needed at all. As a similar problem like in #2, snapshots could grab different states and therefore make it possible to retrieve an older y value.

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?

Page 6 is mapped to page frame 3. Each page frame starts as a multiple of 4096 (4KB):

PAGE FRAME	ADDRESS
0	0
1	4096
2	8192
3 (first 4 byte)	12288
3 (last 4 byte)	(4096*4) - 4 = 16380

To get the physical addresses of the bottom 2 addresses, multiply the page by 4096. Therefore, the virtual addresses of the first and last 4 byte word of page 6 is 12288 and 16380. And their physical address of the first and last 4 byte word of page 6 is 24576 and 28668 ((4096*7)-4).

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 table to the right points to pages 1047552 and 1047553. Since the page table is 1024 chunks of information, we multiply the last page, 1023 by 1024, and get the values 1047552 and 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.**

   We performed this on a Windows OS using VSCode IDE and used the command prompt to run the code. The size of the array is determined by a user input that is prompted at the beginning of the program. The array is initiated with random values and a

timer is set before entering the for loop. The array is iterated through until the program hits a position that is a multiple of 4066. It then stops the timer and adds it to a totalTimes list that is ultimately averaged out.

Below shows the sizes tested and resulting times (each size was run 3 times).

Array Size	Average Times
50,000	0.001583, 0.001250, 0.001667
100,000	0.004958, 0.004458, 0.003833
1,000,000	0.050111, 0.039684, 0.045066
100,000,000	3.668414, 3.386332, 4.623444

The average times seem to be consistent for each array size, with larger arrays giving a little more variance.

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's 8 processes running in the program