Homework 2 - Anthony Keba, Teddy Chu, Tyler Edmiston

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 we reversed the order, It would be added to the wait queue, but never removed from the run queue.

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.

   Without getting the state as a snapshot, it is possible that declaring seatsRemaining could be referring to a different state than declaring cashOnHand.

3. **IN JAVA**: Write a test program in Java for the `BoundedBuffer` class of Figure 4.17 on page 119 of the textbook.
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.
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: No. Two-phase locking ensures serializability but at a price in concurrency, and hence throughput.

Short read locks: Yes, this is the same as a dirty read. A shared lock is acquired before each read and released right after. This method, though, allows for the possibility to read from the same location twice and get two different values.

Snapshot Isolation: Yes, snapshot isolation completely ignores and writes done since the most recently committed value, so if the snapshot is called while writing it is possible.

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 address of the first 4-byte word in page 6 is 24576, and the last word is 28668, which is 4 bytes before the next page at 28672. These physical addresses are 12288 and 16380, 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.

The page directory contains a total of 1024 page tables, which each contain 1024 pages. The last page table will contain pages of the values 1024 * 1023 as its first page, which is 1047552. Page 1047553 is the next page.

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.

Program ran on Online C Compiler in Google Chrome: At size 100,000,000 takes .5 seconds to find the final element (99999744). Constant lookup time of about 0.000016 seconds per element found.

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.

anthonykeba: ~/development/school/spring2019/lmu-cmsi-387 (master): ps | grep a.out
10552 ttys000 0:00.00 ./a.out
10553 ttys000 0:00.00 ./a.out
10554 ttys000 0:00.00 ./a.out
10555 ttys000 0:00.00 ./a.out
10561 ttys000 0:00.00 ./a.out
10562 ttys000 0:00.00 ./a.out
10563 ttys000 0:00.00 ./a.out
10564 ttys000 0:00.00 ./a.out
10557 ttys001 0:00.00 grep a.out

Our original hypothesis was that there would be three instances of ./a.out running, and we were surprised to see that there were 8 instances (plus the grep, which can be ignored, and is running on a different ttys instance). After careful consideration, we came to the realization that each fork would pickup the program where it began, and would continue execution from there, without resetting or ending the *for* loop. This meant that:
- Upon the first iteration of the loop, a second process was spawned, and the counter was decremented by one (Counter: 2, Instances: 2)
- Upon the second iteration of the loop, each of the two processes (with matching *for* loop counters) spawned a new subprocess. Meaning that there were now 2*2 processes running (Counter: 1, Instances: 4)
- On the final iteration of the loop, each of the remaining processes spawn their forks, resulting in 8 total processes, and an empty counter, triggering a break from the for loop. (Counter: 0, Instances: 8)

This explains the existence of 8 instances of ./a.out

Here is a simple graph meant to demonstrate the number of processes at each stage of the counter