Donovan Moini, Ian Lizarda, Serena Zafiris
Operating Systems Homework02

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. (CH4 Exercises, 4.2)
    a. If you unlock *mutex.spinlock* prior to removing the current thread from runnable threads, the current thread would still be a runnable thread even though it is also in *mutex.waiters*. Once the code unlocks *mutex.spinlock*, it will look for the next runnable thread, which is still the current thread. This will result in the code infinitely placing the current thread on the waiting queue and never removing the current thread from the runnable threads, thus never 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. (CH4 Exercises, 4.11)
    a. When the code originally used *State snapshot = state.get()*, this ensured that when *getSeatsRemaining()* and *getCashOnHand()* were called they were referencing the same state. By removing *snapshot* and calling *state.get()* for both *getSeatsRemaining()* and *getCashOnHand()*, *seatsRemaining* and *cashOnHand* are referring to different states. This would be a bug because in the moments between calling *state.get().getSeatsRemaining()* and *state.get().getCashOnHand()* the state can be updated. This would result in the possible inconsistency of *getSeatsRemaining()* and *getCashOnHand()*.
3. IN JAVA: Write a test program in Java for the BoundedBuffer class of Figure 4.17 on page 119 of the textbook. (CH4, Programming Projects 4.5)
    a. Reference 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. (CH4, Programming Projects 4.6)
    a. Reference BoundedBuffer.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. (CH5 Exercises, 5.17)

   a. **For two-phase locking**
      i. Yes, it is possible because two-phase locking ensures that the system history obeying its rules is serializable, guaranteeing consistency from state to state. Thus the old state of x would exist and be preserved in the serial history and can be read, as can the new value of y.
   b. **For read committed isolation with short read locks**
      i. It is possible, but the possibility still exists that T2 could access the same memory twice and get two different values, also known as a dirty read. If T2 tries to read the old value of x and new value of y multiple times it may get different values.
   c. **For snapshot isolation**
      i. No because any reads will *only* read the most recently committed value. Thus, T2 would be able to see the new value of y but *only* the new 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? (CH6 Exercises, 6.5)
   a. The virtual addresses of the first and last 4-byte words are 12288 and 16380 respectively. These translate to the physical addresses 24576 and 28668 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. (CH6 Exercises, 6.9)
   a. The page directory (IA-32) can point to 1024 chunks of the page table, each of which can point to 1024 page frames. When looking at each chunk, we can access pages frames in the range [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, 1048575]. The first page frame we can access in this chunk 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. (CH6 Programming Projects, 6.1)
   a. I began with an array size of 1 and then increasing that size by a factor of ten until it maxed out at the size 1,410,065,408. The code begins with setting up the array's size and checking the existence and validity of the argument. Following such, it fills the array with random numbers (reusing the *vmArrayTimer.c* code). The next section accesses every 4096th element and uses a clock to account for the time it takes to access every 4096th element, with the execution time stored in another array. Once this concludes, the average execution time is calculated. Below are the results

in the format *arraySize* ⇒ *averageElementAccessTime*, where *averageElementAccessTime* is measured in seconds.
    i. 1 ⇒ 0.000002
    ii. 10 ⇒ 0.000002
    iii. 100 ⇒ 0.000002
    iv. 1,000 ⇒ 0.000002
    v. 10,000 ⇒ 0.000001
    vi. 100,000 ⇒ 0.000001
    vii. 1,000,000 ⇒ 0.000001
    viii. 10,000,000 ⇒ 0.000001
    ix. 100,000,000 ⇒ 0.000001
    x. 1,00,000,000 ⇒ 0.000001
    xi. 1,410,065,408 ⇒ 0.000001
b. Hardware
    i. MacBook Air (13-inch, 2017)
    ii. Processor: 1.8 GHz Intel Core i5
    iii. Memory: 8 GB 1600 MHz DDR3
    iv. Graphics: Intel HD Graphics 6000 1536 MB
c. Software
    i. macOS 10.14.2
    ii. Visual Studio Code version 1.31.1 (1.31.1)
    iii. Terminal version 2.9.1 (421.1)
        1. Oh My Zsh version 5.3
d. The following are the same tests but done on Windows OS.
    i. 1 ⇒ 0.000000
    ii. 10 ⇒ 0.000000
    iii. 100 ⇒ 0.000000
    iv. 1,000 ⇒ 0.000000
    v. 10,000 ⇒ 0.000000
    vi. 100,000 ⇒ 0.000000
    vii. 1,000,000 ⇒ 0.000000
    viii. 10,000,000 ⇒ 0.000000
    ix. 100,000,000 ⇒ 0.000000
    x. 1,00,000,000 ⇒ Could not allocate array of 1,000,000,000 size.
e. Hardware
    i. Surface Book 2
    ii. Processor: Inter(R) Core(™) i7-8650U CPU @ 1.9 GHz 2.11 GHz
    iii. Install RAM: 8.00 GB
    iv. System type: 64-bit operating system, x64-based processor
f. Software
    i. Microsoft Windows version 10.0.17763.316
    ii. Visual Studio Code version 1.31.1 (1.31.1)

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. (CH7 Exploration Projects, 7.1)

a. There are 8 total processes shown running the program.