Tigerlilly Zietz
Josh Seaman
Nick Morgan

HOMEWORK 02

---

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.

   Reversing them using the pseudocode from Figure 4.28 would be problematic because the current thread could stay as part of the runnable threads since unlock mutex.spinlock is called before removing the current thread. Thus this would leave the possibility of executing this thread over and over again, disrupting the sequence of executions that the thread must execute.

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.

   The bug could occur due to the fact that calling state.get()... on two separate lines leaves the possibility that .getSeatsRemaining() and .getCashOnHand() could be called on two different states, thus using snapshot to save a specific instance of the State guarantees that .getSeatsRemaining() and .getCashOnHand() are being called on the same object.

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

Code can be found in BoundedBuffer/BoundedBufferTest.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.

Code can be found in
BoundedBuffer/BoundedBufferModifiedQ4.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	It is **impossible** for T2 to see the old value of x and the new value of y because two-phase locking ensures serializability and it holds all the locks until the end of the transaction. Thus, the system doesn't predict the transaction's future read or write actions and is unable to see the new value of y since it ensures one transaction runs at a time, and commits or aborts before the next transaction is able to begin.
Short Read Locks	It is **impossible** for T2 to see the old value of x and the new value of y because with read committed isolation and short read locks, writes have an exclusive lock resulting in T1 having write access for

	the entirety of the transaction. With short read locks, doing the two transactions simultaneously could result in swapping x and y's old values, rather than making the two equal.
Snapshot Isolation	It is **possible** for T2 to see the old value of x and the new value of y because snapshot isolation does not truly offer serializability. One transaction could copy x to y while another copies y to x. In snapshot isolation, every write action stores the new value for an entity in a different location than the old value. Thus, a read action doesn't need to read the most recent version, resulting in varying values for the old value of y and 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 on page 6? What physical addresses do these translate into?

4-Byte Word:	**Virtual Addresses:**	**Physical Addresses:**
First:	12288	24576
Last:	16380	28668

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 values 1,047,552 and 1,047,553 are calculated by finding the index of the 1024th chunk, or the last page table. In the IA-32 page table's two-level hierarchy,

since it indexes starting at 0, we can multiply 1023 by 1024 page frames to get the value of 1,047,552. Then, the subsequent value of page 1,047,553 is the result of adding 1 to 1,047,552 to get the following 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.

Code can be found in question_8_loop.py

```
Size: 6473
avg time:       1.2159347534179687e-06

Size: 16278
avg time:       6.993611653645833e-07

Size: 31782
avg time:       5.006790161132813e-07

Size: 62123
avg time:       4.5299530029296873e-07

Size: 123123
avg time:       7.128715515136719e-07

Size: 243140
avg time:       5.067405054124735e-07

Size: 976563
avg time:       4.868547455603335e-07

Size: 7812500
avg time:       8.033593412338281e-07

Size: 15938728
avg time:       6.304826444784188e-07

Size: 692657839
avg time:       6.089583142822172e-07

Size: 125937563
avg time:       7.862191127984795e-07

Size: 269272864
avg time:       6.258581995347513e-07

Size: 1000000000
avg time:       7.173117972557009e-07
```

This code was written and ran in Python 3.7.3 and executed on a 2018 MacBook Pro via the terminal.

Our findings clearly show that it takes more time to access each element the shorter the array is. As seen in the screenshot, when the array is length is only 6473 the average time is 1.22 e-4 seconds compared to a larger size of 1000000000 where the average time is 7.17 e-7. From our findings, when the array size was double 4096 since it would then have to loop through and access multiple elements in the array.

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 a total of 10 processes running since there are 8 processes from the program and 2 processes from bash.

Family tree of the processes: