CMSI 3510
HW#2
Professor Johnson

Q#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 unlock the mutex.spinlock before removing the current threads from the queue of runnable threads. The current thread does not have the spinlock but is runnable and can grab the spinlock again, therefore this might cause other threads in the waiting queue to be starved out.

Q#2: Suppose the first three lines of the audit method in Figure 4.27 on page 144 were replaced by the following two lines:
Explain why this would be a bug

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

The above method chaining will not work because .getSeatsRemaining() method needs an object reference state.get() is returning an int value instead of an object reference.

Q#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.

Code:

Test Program:

```
//Test#1

package test.boundedBuffer;

import boundedBuffer.BoundedBuffer;



public class BoundedBufferTest {
```

```
                BoundedBuffer Test = new BoundedBuffer();
                Test.insert(1);
                Test.insert(2);
                Test.insert(3);
                Test.insert(4);
                Test.insert(5);
                Test.insert(6);
                Test.insert(7);
                Test.insert(8);
                Test.insert(9);
                Test.insert(10);
                Test.insert(11);
                Test.insert(12);
                Test.insert(13);
                Test.insert(14);
                Test.insert(15);
                Test.insert(16);
                Test.insert(17);
                Test.insert(18);
                Test.insert(19);
                Test.insert(20);
                //max capacity of buffer reached
                Test.insert(21);
        }

//Test#2
package test.boundedBuffer;
import boundedBuffer.BoundedBuffer;

public class BoundedBufferTest {
        //instantiating new Buffer
        BoundedBuffer Test = new BoundedBuffer();
         Test.insert(1);
         Test.insert(2);
         //output = 2
         Test.retrieve();
}
```

Q#4: **IN JAVA**: Modify the **`BoundedBuffer`** class of Figure 4.17 [page 119] creating **`BoundedBuffer2.java`**. Make the new program 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.

```
package boundedBuffer;

public class BoundedBuffer2 {
        private Object[] buffer = new Object[20]; // arbitrary size
          private int numOccupied = 0;
          private int firstOccupied = 0;
          /* invariant: 0 <= numOccupied <= buffer.length
```

```
    0 <= firstOccupied < buffer.length
    buffer[(firstOccupied + i) % buffer.length]
    contains the (i+1)th oldest entry,
    for all i such that 0 <= i < numOccupied  */
  public synchronized void insert(Object o)
    throws InterruptedException
  {
    while(numOccupied == buffer.length) {
      // wait for space
    wait();
    //buffer is empty, then call notifyAll()
    If (numOccupied == 0) {
            notifyAll();
    }
     buffer[(firstOccupied + numOccupied) % buffer.length] = o;
     numOccupied++;
  }

  public synchronized Object retrieve()
    throws InterruptedException
  {
    while(numOccupied == 0)
      // wait for data
      wait();
    //retrieving from a full buffer, then call notifyAll()
    If (numOccupied == buffer.length) {
            notifyAll();
    }

    Object retrieved = buffer[firstOccupied];
    buffer[firstOccupied] = null; // may help garbage collector
    firstOccupied = (firstOccupied + 1) % buffer.length;
    numOccupied--;

    return retrieved;
  }
}
```

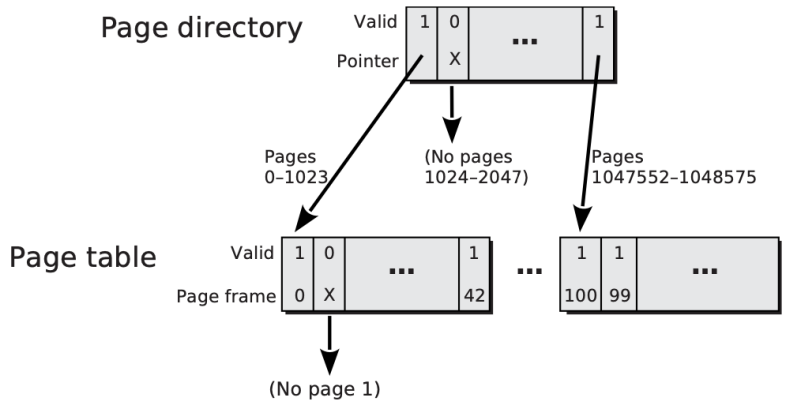
The new program does not work when tested with the test program from the previous question.

Q#6: 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 and 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

Q#7: 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 offset	Virtual address	Physical address
111111111100	28668	16380
111111111101	28669	16381
111111111110	28670	16382
111111111111	28671	16383

Q#8: At the lower right of Figure 6.13 on page 236 are page numbers 1047552 and 1047553. Explain how these page numbers were calculated.



In a paging system, the size of a virtual-page and a physical-frame is always the same.

Here, lowest 12 bits of virtual/physical address denote the page/frame offset, while the remaining higher bits represent the page/frame number.

Given virtual address is

= decimal 4290777130

= binary 1111 1111 1100 0000 0001 0000 0010 1010

Therefore,

page offset = bolded bits

= binary 0000 0010 1010

page number

= remaining bits

= binary 1111 1111 1100 0000 0001

= decimal 1047553

Now, according to the page-table, the frame number corresponding to page number 1047553 is

= decimal 99

= binary 110 0011

Thus, physical address is obtained by concatenating the frame number and the offset bits

= binary 110 0011 0000 0010 1010

= decimal 405546

Q#9: 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.

Report:

Given 1000 elements:

```
Output: Individual accesses per second: 16666667
```

Given 10,000 elements:

```
Output: Individual accesses per second: 53191491
```

Given 100,000 elements:

```
Output: Individual accesses per second: 66844915
```

Given 1,000,000 elements:

```
Output: Individual accesses per second: 68175620
```

Conclusion: Individual accesses per second seem to increase, approaching an upper limit, as array elements increase as if the system settles into a groove of accessing data from the array.

Hardware and Software specifications:

Processor: Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz

RAM: 16 GB

Code:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define NUM_LOOPS 1000

int main(int argc, char * argv[]) {
    long arrSize = atoi(argv[1]);
    char array[arrSize];

    srand(8723);
    for (long i = 0; i < arrSize; i++) {
```

```
        array[i] = rand() % 256;
    }
    printf("\n   Array filled\n\n");

    time_t startTime;
    time_t endTime;
    startTime = clock();

    for (int loopCount = 0; loopCount < NUM_LOOPS; loopCount++) {
        for (long i = 0; i < arrSize; i += 4096) {
            printf("     Array[%ld] = %d\n", i, array[i]);
        }
    }

    endTime = clock();
    float totalTime = (float) (endTime - startTime) / CLOCKS_PER_SEC;
    float timePerAccess = totalTime / NUM_LOOPS;
    int arrayAccessPerSecond = arrSize / timePerAccess;
    printf("   Start time: %ld    End time: %ld\n", startTime, endTime);
    printf("   Average time per whole array access: %f seconds\n",
      timePerAccess);
    printf("   Individual accesses per second: %d\n\n",
      arrayAccessPerSecond);

}
```

8. 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.

Console:

```
(base) claudiagusti@claudias-MacBook-Pro C Files % ps
  PID TTY           TIME CMD
 8409 ttys001    0:00.02 /bin/zsh -l
 8462 ttys001    0:00.00 ./multiforker
 8463 ttys001    0:00.00 ./multiforker
 8464 ttys001    0:00.00 ./multiforker
 8465 ttys001    0:00.00 ./multiforker
 8466 ttys001    0:00.00 ./multiforker
 8467 ttys001    0:00.00 ./multiforker
 8468 ttys001    0:00.00 ./multiforker
 8469 ttys001    0:00.00 ./multiforker
 8474 ttys002    0:00.02 /bin/zsh -l
(base) claudiagusti@claudias-MacBook-Pro C Files %
```

8 processes are shown running the program.