Andrey Varakin
Jeremy Goldberg
Miliano Mikol

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.**

```
to lock mutex:
  lock mutex.spinlock (in cache conscious fashion)

  if mutex.state = 1 then
    let mutex.state = 0
    unlock mutex.spinlock
  else
    add current thread to mutex.waiters
    unlock mutex.spinlock
    remove current thread from runnable threads
    yield to a runnable thread
```

Bug: if the spinlock is released first before removing the thread from runnable threads, then the code would run in an infinite loop of locking/unlocking a spinlock for the current thread.

Original
T1 runs
T1 is added to mutex.waiters
T1 is removed from runnable threads
mutex.spinlock is unlocked
mutex.spinlock is yielded to the next runnable thread

Reversed
T1 runs
T1 is added to mutex.waiters
mutex.spinlock is unlocked
T1 immediately starts running again, producing an infinite loop

Andrey Varakin
Jeremy Goldberg
Miliano Mikol

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.

Instead of looking at a single snapshot of the state, this code would look at two different states and therefore get different seatRemaining and cashOnHand variables for different states. This would lead to computational errors.

Andrey Varakin
Jeremy Goldberg
Miliano Mikol

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

```java
public static void main(String args[]){
 BoundedBuffer buff = new BoundedBuffer();
 Thread childThreadInsert = new Thread(new Runnable() {
   public void run() {
     for (int i = 0; i < 100; i++) {
       try {
         buff.insert((Integer) i);
       } catch (InterruptedExceptioninterruptedException) {
         System.out.println("cannot insert");
       }
       System.out.println("inserted: " + i);
       sleep(500);
     }
   }
 });
 Thread childThreadRetrieve = new Thread(new Runnable() {
   public void run() {
     for (int i = 0; i < 100; i++) {
       try {
         System.out.println(
             "retrieved: " + buff.retrieve().toString()
         );
       } catch (InterruptedException interruptedException) {
         System.out.println("cannot retrieve");
       }
       sleep(1000);
     }
```

Andrey Varakin
Jeremy Goldberg
Miliano Mikol

```
   }
 });
 childThreadInsert.start();
 childThreadRetrieve.start();
 System.out.println("Done");
}
private static void sleep(int milliseconds){
 try{
   Thread.sleep(milliseconds);
 } catch(InterruptedException e){
   // ignore this exception; it won’t happen anyhow
 }
}
```

Andrey Varakin
Jeremy Goldberg
Miliano Mikol

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.**

```java
public class BoundedBuffer {

  private Object[] buffer = new Object[20];

  private int numOccupied = 0;

  private int firstOccupied = 0;

  public synchronized void insert(Object o)  throws
InterruptedException {

    while(numOccupied == buffer.length) wait();

    buffer[(firstOccupied + numOccupied) % buffer.length] = o;

    numOccupied++;

    if(numOccupied == 1) notifyAll();

}

  public synchronized Object retrieve() throws InterruptedException {

    while(numOccupied == 0) wait();

    Object retrieved = buffer[firstOccupied];

    buffer[firstOccupied] = null;

    firstOccupied = (firstOccupied + 1) % buffer.length;

    numOccupied--;

    if(numOccupied == buffer.length) notifyAll();

    return retrieved;

}
```

Andrey Varakin
Jeremy Goldberg
Miliano Mikol

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.**

1) Two-Phase Locking

No this situation is not possible. T1 would acquire all locks first (perform its operations, then return the resources in reverse order. T2 would then take over and read both of the new values of x and y.

2) Read Committed Isolation Level (Short Read Locks)

Yes this is possible as T1 would take a write lock for x and y but could release y after it is done writing and hold on to x as it writes there. T2 could grab the read locks for x and y and would then be able to see the new value of y and the old value of x.

3) Snapshot Isolation

This situation could also occur with snapshot isolation. In snapshot isolation, there are no read locks. T1 could grab the write locks for x and y, write to y and release it, then T2 could read y and x. Since T1 has not yet released the write for x, T2 would see the old value of x and the new value of y.

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?**

Virtual address of first word: 24,576	Virtual address of last word: 28,668
Physical address of first word: 12,288	Physical address of last word: 16,380

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 points to chunks of the page table, where each chunk points to 1024 pages. The lower right half of the tree in Figure 6.13 is the 1023rd chunk of the page table, which points to page 1047552, the first page of that chunk. Since each next page number is just one plus the previous one, the next page will then be numbered 1047553.

Andrey Varakin
Jeremy Goldberg
Miliano Mikol

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.**

Hardware
Device: MacBook Pro (Retina, 13-inch, Early 2015)
Processor: 2.7 GHz Dual-Core Intel Core i5
Memory: 8 GB 1867 MHz DDR3
Graphics: Intel Iris Graphics 6100 1536 MB
Software: MacOS Catalina 10.15.3

Test 1 (Base Case)
Number of elements in array: 10000; Number of tests: 10
Average time to access an array element: 0.000045 seconds

Test 2
Number of elements in array: 100000; Number of tests: 10
Average time to access an array element: 0.000286 seconds
Conclusion: the average access time increases by over a factor of 10 when increasing the number of elements by the same factor.

Test 3
Number of elements in array: 1000000; Number of tests: 10
Average time to access an array element: 0.002832 seconds
Conclusion**:** same as previous test.

Test 4
Number of elements in array: 10000000; Number of tests: 10
Average time to access an array element: 0.026756 seconds
Conclusion**:** same as previous test.

Test 5
Number of elements in array: 1000000000; Number of tests: 2
Average time to access an array element: 3.898516 seconds
Conclusion**:** when increasing the number elements by a factor of 100, there is a HUGE difference in access time. This test also used well over .5 GB of memory to execute with only two loops for the outer loop.

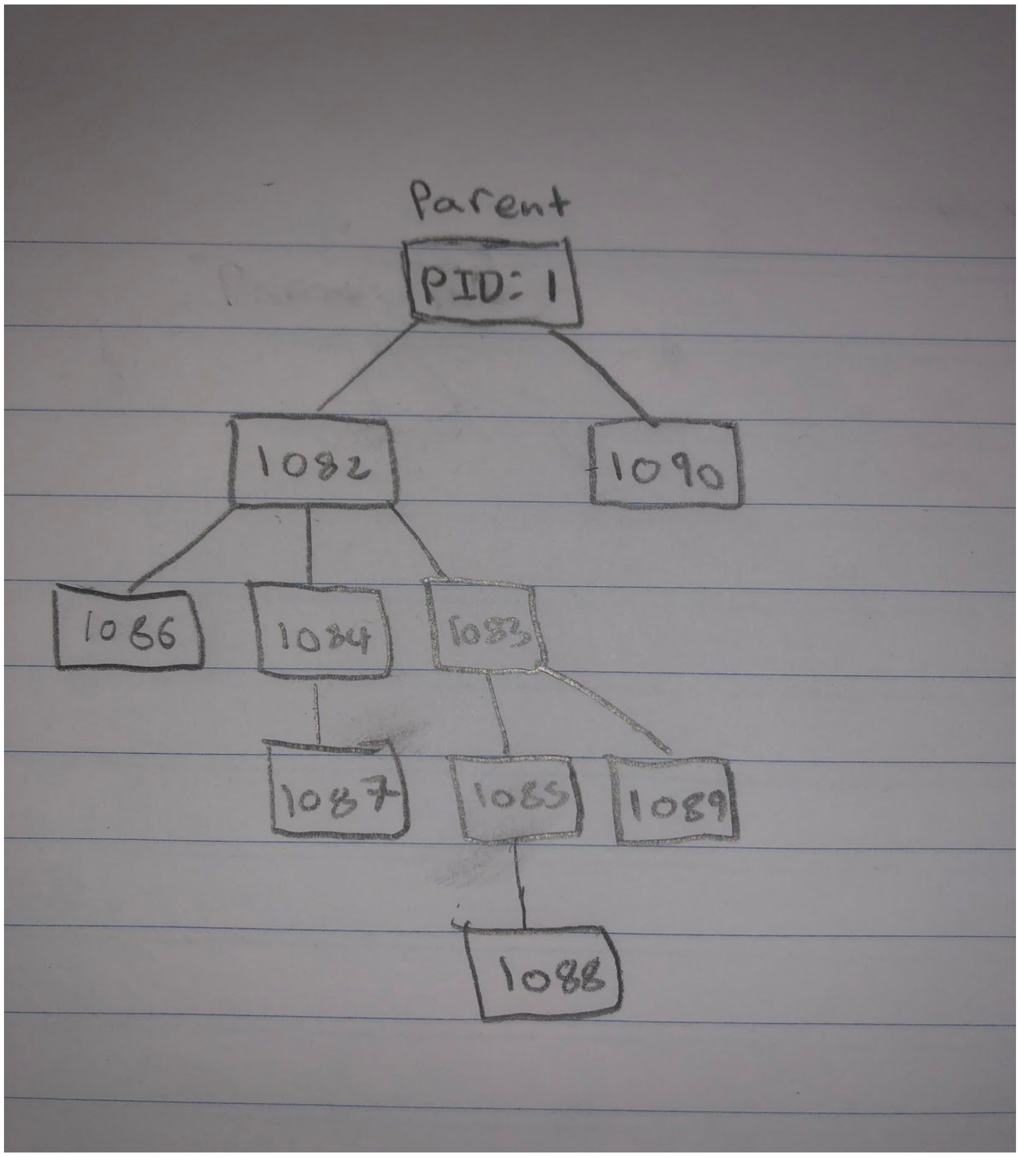
Andrey Varakin
Jeremy Goldberg
Miliano Mikol

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

int main(int argc, char * argv[]) {
  int arr_size = atoi(argv[1]);
  int * arr = malloc(arr_size * sizeof(int));

  for (int i = 0; i < arr_size; i++) {
      arr[i] = (rand() % 100) + 1;
  }

  int num_tests = 100;
  float avg_access_time, total_time = 0.0;
  clock_t start, end;

  for (int i = 0; i < num_tests; i++) {
    int start = clock();
    for (int j = 0; j < arr_size; j++) {
      if (j % 4096 == 0) {
        int end = clock();
        float t = ((double) end - start)/CLOCKS_PER_SEC;
        printf("\nArray element accessed: %d . . .\n", arr[j]);
        printf("Time to access: %f seconds\n", t);
        total_time += t;
      }
    }
  }

  avg_access_time = (total_time / (arr_size / 4096)) / num_tests;

  printf("Total average time: %f\n", avg_access_time);

  free(arr);

  return 0;
}
```

Andrey Varakin
Jeremy Goldberg
Miliano Mikol

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 9 processes shown when running the program while simultaneously calling ps.


Parent
PID: 1
1082
1090
1086
1084
1083
1087
1085
1089
1088