Question 1

Answer saved

Marked out of 1.00

Flag question

Pilih kaedah-kaedah yang boleh terus menyebabkan sesuatu bebenang berhenti pelaksanaan. (Markah akan ditolak bagi jawapan yang salah.)

Select methods that can directly cause a thread to stop executing. (Marks will be deducted for wrong answers.)

[1 markah/mark]

Select one or more:

- [ ] Memanggil kaedah notify() pada suatu objek.
  Calling notify() method on an object.
- [ ] Tiada di atas.
  None of the above.
- [x] Memanggil kaedah sleep() pada suatu objek *Thread*.
  Calling sleep() method on a Thread object.
- [x] Memanggil kaedah wait() pada suatu objek.
  Calling the wait() method on an object.
- [ ] Memanggil kaedah read() pada suatu objek *InputStream*.
  Calling read() method on an InputStream object.

Next page

Question 3

Answer saved

Marked out of 2.00

Flag question

Berikan komen anda terhadap antaramuka *something* di bawah.

Give your comment on the something interface below.

```
interface something {
    public synchronized void somethingwrong();
}
```

[2 markah/*marks*]

The interface 'something' is implementing synchronization in its method called somethingwrong. This means that it is trying to implement a locking mechanism in the method whereby only 1 thread can access the designated critical section.

However, synchronization is not appropriate to be utilised in an interface. This is because it is not thread safe and in this way, we cannot force an implementation of the interface to be synchronized. Synchronized is an implementation detail (method implementation), but in interface all methods are abstract with no implementation. Therefore, the keyword does not belong in an interface.

In other words, synchronization is and should be part of the implementation and not part of the interface.

Question 4

Answer saved

Marked out of 5.00

Flag question

Pertimbangkan kelas Sometask di bawah:

Consider the Sometask class below:

```
public class Sometask {
    private int value = 0;
    synchronized public void increment() {
     value += 1;
  }

  synchronized public int getValue() {
    return value;
  }
}
```

Terangkan mengapa kaedah-kaedah *increment()* dan *getValue()* menggunakan *synchronized*. Cadangkan satu penyelesaian alternatif bagi *synchronized* untuk kaedah getValue().

Explain why the increment() and getValue() methods are synchronized. Suggest an alternative solution to synchronized the getValue() method.

[5 markah/*marks*]

The getValue() method needs to be synchronized because of the caching of local data. If the getValue() method not synchronized, it is possible that a thread that calls getValue() would see an old, cached value of count rather than the most current value. Synchronization ensures that the most current value of count will be seen.

For increment() method however, it will still need to be synchronized to prevent the race condition. A race condition occurs if it is possible for another thread to increment the value of count between the time when the first thread reads the old value and the time when it stores the new value.

An alternative solution to synchronize the getValue() method is to use volatile keyword for value variable. If count were declared to be a volatile variable, then getValue() would not have to be synchronized. By using volatile, it is guaranteed that any thread that reads volatile variable will see the most recently written value.

QUESTION 2

```java
import java.util.concurrent.Callable;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
import java.util.ArrayList;
import java.util.Scanner;

public class Q2 {

    /**
     * Write a multithreaded program to count the number of prime numbers between 1
     * and 10,000,000.
     */
    private static final int MAX_INT = 10_000_000;
    private static int TOTAL_PRIMES;

    /**
     * A class CountPrimesTask that implements Callable interface, receives a range
     * (first, last), and calls the countPrime() method to count prime numbers
     * between (first, last).
     */
    private static class CountPrimesTask implements Callable<Integer> {
        int first, last;

        public CountPrimesTask(int first, int last) {
            this.first = first;
            this.last = last;
        }

        public Integer call() {
            int count = countPrimes(first, last);
            return count;
        }
    }

    /**
     * A method countPrimesConcurrently() that starts a number of threads to count
     * prime numbers concurrently. The number of threads is specified as a
parameter
     * passed to countPrimesConcurrently(). This method returns the count of prime
     * numbers.
     */
    private static void countPrimesConcurrently(int noOfThreads) {

        System.out.println(
                "\nCounting primes between 1 and " + MAX_INT + " using " +
noOfThreads + " number of threads:\n");
        long startTime = System.currentTimeMillis();
```

```
        ExecutorService executor = Executors.newFixedThreadPool(noOfThreads);
        ArrayList<Future<Integer>> results = new ArrayList<>();

        /**
         * Each sub-task processes at most 1000 integers. Use Math.ceil to round up
the
         * number of sub-tasks to cater for MAX_INT which is not a multiple of
1000. -1
         * because exceed by 1.
         */
        int numberOfSubTasks = (int) Math.ceil(MAX_INT / 1000.0);

        for (int i = 0; i < numberOfSubTasks; i++) {
            int start = i * 1000 + 1;
            int end = (i + 1) * 1000;

            /*
             * The last task in that case will consist of the last (MAX_INT%1000))
ints.
             */
            if (end > MAX_INT)
                end = MAX_INT;

            CountPrimesTask subTask = new CountPrimesTask(start, end);
            Future<Integer> subResult = executor.submit(subTask);

            results.add(subResult);
        }

        executor.shutdown();

        /**
         * A method addToCount() that allows running threads to add their count of
prime
         * numbers to a global variable total. Total indicates total prime numbers
         * counted so far.
         */
        addToCount(results);

        long elapsedTime = System.currentTimeMillis() - startTime;
        /**
         * Subtract 1 from TOTAL_PRIMES due to excess in calculation
         */
        System.out.println("The number of primes is " + (TOTAL_PRIMES - 1) + ".");
        System.out.println("Total elapsed time:  " + (elapsedTime / 1000.0) + "
seconds.\n");
    }

    /**
     * A method addToCount() that allows running threads to add their count of
prime
```

```
     * numbers to a global variable total. Total indicates total prime numbers
     * counted so far.
     */
    private static void addToCount(ArrayList<Future<Integer>> results) {
        for (Future<Integer> res : results) {
            try {
                TOTAL_PRIMES += res.get();
            } catch (Exception e) {
            }
        }
    }

    /**
     * A method countPrimes() that returns the count of prime numbers within a
given
     * range (first, last).
     */
    private static int countPrimes(int first, int last) {
        int count = 0;
        for (int i = first; i <= last; i++)
            if (isPrime(i))
                count++;
        return count;
    }

    /**
     * A method isPrime() that returns True if a given number is a prime number, or
     * False otherwise.
     */
    private static boolean isPrime(int x) {

        /**
         * Use mathematical algorithm to quickly check if number is prime
         */
        int limit = (int) Math.sqrt(x);

        for (int i = 2; i <= limit; i++)
            if (x % i == 0)
                return false;

        return true;
    }

    /**
     * A main method that declares and initialises all the constants and variables,
     * gets and validates the number (1-8) of threads the user would like to use,
     * and calls countPrimesConcurrently() to start the process of counting prime
     * numbers
     */
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
```

```
        int numberOfThreads = 0;
        System.out.print("Enter number of threads (1-8): ");

        while (numberOfThreads < 1 || numberOfThreads > 8) {
            numberOfThreads = scanner.nextInt();
            if (numberOfThreads < 1 || numberOfThreads > 8)
                System.out.println("Please enter a number in the range 1 to 8!");
        }

        countPrimesConcurrently(numberOfThreads);
        scanner.close();
    }

}
```