CMT219 Algorithms, Data Structures and Programming

Lab Exercises Week 5

1. Write a Java program using multi-threading to simulate horse racing with 10 horses. Each thread simulates a horse (with an ID of 0, 1, until 9), and at every time step (10ms), it moves the horse forward by a random distance (for simplicity, an integer between 1 and 6 inclusive). A horse completes the game by running a distance of 1000. When a horse finishes, print the horse ID and the number of steps (time) involved. After all the horses finish, print the ID of the winner, i.e. the horse taking the minimum amount of time.

Hint: You may pass on the horse ID to the horse thread object as a constructor parameter.

2. Buffer is often used to store data temporarily, which is useful to synchronise between the device that produces the data, and the device that consumes the data. Write a Java program to simulate a buffer used with a producer (which produces the data and puts in the buffer) and a consumer (which uses the data from the buffer).

   You should implement a class **Buffer** that represents a buffer. The buffer can be implemented using an `int` array with a fixed number (called BUFFER_SIZE) of spaces. BUFFER_SIZE can be set to 10 for this exercise. You also define two variables, **head** which indicates the index of the first element currently in the buffer, and **count** which indicates how many elements are currently available in the buffer.

   In the following illustration, head = 1 (index starting from 0), and count = 5 (assuming only valid numbers currently in the buffer are shown).

	11	22	33	44	55				

↑head

   The buffer is treated in a circular fashion, which means, when the index reaches the end of the list, it wraps back from the beginning.

   Implement the following methods for **Buffer**:

   - **get**() which retrieves the first element in the buffer or throws an exception if the buffer is empty.

- **put(int value)** which adds the value to the end of the buffer or throws an exception if the buffer is full.

More methods can be implemented, e.g. to check if the buffer is empty or full. Remember to implement the **Buffer** class in a **thread-safe** manner.

In addition, implement the following two classes:

- **Producer** which is a thread that keeps putting numbers (1, 2, 3, … 100) to the buffer. Print the message to indicate this has happened and a suitable message in case the buffer is full. After putting each element in the buffer, waits for a random amount of time (50-150 milliseconds).
- **Consumer** which is a thread that keeps retrieving elements from the buffer (for 100 times). Print a message to indicate the number retrieved and a suitable message in case the buffer is empty. After putting every element in the buffer, waits for a random amount of time (50-150 milliseconds).

The main program simply starts both threads.

Test the program and see if the output is sensible. You may also change the rate of the producer or the consumer (e.g. to 50-100 milliseconds) and see whether the behaviour is as expected.

Hint: Use `index%BUFFER_SIZE` to map the index back to the range of 0 – BUFFER_SIZE-1.

3. Write a Java program to randomly generate N=100000 integer numbers and sort them. Implement this in three different ways:
   - Using BubbleSort to sort the whole array
   - Using BubbleSort to sort the two halves (elements 0-49999 and elements 50000-100000), one after another.
   - Using BubbleSort to sort the two halves (elements 0-49999 and elements 50000-100000). Use two separate threads so that the two sorting operations can be done at the same time (assuming the computer has multiple cores).

Although sorting two halves independently is not equivalent to sorting the whole array, an efficient algorithm can be developed to merge the two sorted arrays into one. You may try implementing this but this step is optional for this exercise.

[Hint: set two variables **p** and **q** indicating the next element from each sorted half array. Compare these elements with these indexes. The smaller element will be

chosen to put in the whole sorted array, and the corresponding index moved on to the next position.]

Use **System.nanoTime**() (or similar) to obtain the system time before and after running the program. Work out the time spent for each approach.

Hint: To make a fair comparison, you should make a copy of the random data and then supply one copy for each approach.

Try to structure your program such that 1) it splits the tasks into suitable classes 2) it avoids duplicating code and/or data as much as possible.