Exercises 3

In the following exercises, **make sure to rely on thread management facilities discussed in the lecture**.

Task 3.1. Sort and Latch the Array

1. Finish the program that sorts an array ex03_sort2Proc.java. It can sort individual halves of an array already, so you only need to write a merge function (merging algorithm can be found, e.g., in Wikipedia).
2. Replace the semaphore in the code with a CountDownLatch object.

Task 3.2. Find the Maximum

Write a concurrent program that finds the largest number in the unsorted array using four processors. The main routine runs four concurrent threads, then each of them searches for the largest array element in the corresponding quarter. The main routine waits until all of them finish their work, and prints the largest of the four found numbers.

Note that when a thread finishes its work, the thread object is still accessible (the thread is not running, but the object is not destroyed). This means that you can safely store the results of your computation in the thread objects.

Task 3.3. Concurrent Game

Write a “concurrent game” for P processes. The main routine creates and runs P processes with identifiers $1 \cdots P$. Each process selects a random number between 0 and 9, and shuts down. The main routine selects the process with the largest result and gives it one point. If several processes provided the same largest number, each of them gets one point. Then the whole game is repeated until some process scores 3 points.

During the game, the main routine should print the results of each game round, and report the winner(s) when the game is over.

Example input data: `4`

Example output data:

```
round 1 winners
1 3

round 2 winners
1 4
```

```
round 3 winners
4

round 4 winners
1 2

winner(s): 1
```

Task 3.4. Money Monitor with a Barrier

Improve the logic of the program ex03_money.java as follows.

1. The bank account should be an object of a separate class `AccountType` that implements two functions: `GetValue()` (returns the current amount of money on the account) and `AddOneUnit()` (adds one money unit to the account). The `AddOneUnit()` function should be atomic to make sure that `AccountType` can be safely used in concurrent programs.
2. Use `synchronized` keyword instead of a semaphore to implement atomicity.
3. Use CyclicBarrier instead of a semaphore to wait for all threads to finish before printing final results.