Assignment

Re-submit Assignment

Due Dec 2 by 11:59am **Points** 20 **Submitting** a file upload
Available Nov 1 at 12am - Dec 2 at 11:59am about 1 month

task1-en-23-24-1-datamover

2023/24 autumn semester, Concurrent programming assignment

Conditions

- See the general conditions of the assignment on the Syllabus page in the theory Canvas
- The names of packages/classes/methods/etc. in the assignment must be used precisely as given
- The solution must be of the highest quality possible
  - Follow the standard coding conventions of Java
  - Name your variables well and use good code structure
- Submission
  - Upload the complete solution to Canvas archived in **zip** format
  - The `zip` file must only contain the source files of the solution in the correct directory structure
  - Do not include any other files (e.g. `.class`) or directories with your solution
    - Do not include the test classes included with the assignment...
    - ...but do include those which were created as parts of the task
  - The solution can be submitted multiple times before the deadline
    - The latest submission will be evaluated
    - *If you don't submit at all until the deadline, you're out!*

Base exercise (10 points)

In this task we will use threads to move elements of an array to new positions. This array has public static visibility and contains integers. Also, this array must be named `data`. In the base version of this task we will create and manage threads manually.

The top level class wrapping the entire program must be named `DataMover`. The program should also accept command line arguments. The first value is the simulated time of a move. The rest of the arguments represent the wait time for each thread. This way the number of arguments defines how many threads the program will use. The size of the array also matches the number of threads. The threads are numbered, with indexing starting at 0. The `Thread` objects are stored (similarly to `data`) in a public static list called `movers`.

In case no command line arguments are given, the program should run with default parameter settings. The defaults are the same as running the program with the following arguments: `123 111 256 404`

Initially, fill the array with the numbers `0`, `1000`, `2000` etc.

Each thread runs the following code `10` times:

1. Initially, the thread does nothing. The sleep time is defined by the corresponding command line argument.
2. Critical section with mutual exclusion:
    1. It subtracts its own index from the array element with an index corresponding to the thread's own index.
        - E.g. Thread#3 will subtract 3 from `data[3]`.
    2. After doing so, the thread logs the action by writing to standard output in this format: `#2: data 2 == 1996`
    3. Does nothing. Duration is defined by the move time, which is the first command line argument.
    4. It adds its own index to the array element following the one it just read from.
        - In case of the last thread, it adds to the first element of the array.
    5. After doing so, the thread logs the action by writing to standard output in this format: `#2: data 0 -> 14`

Let the program wait for all threads to finish. The `Thread` objects in the `movers` list are not to be removed, even if the corresponding threads are no longer running.

At the end, the program prints the final values of the array. For 3 threads these are the following: `[20, 990, 1990]`

To consider the exercise solved, the code has to be of good quality and has to compile free of errors/warnings, follows the structure outlined above, terminates in a timely fashion (with and without getting command line arguments), and produces the correct output.

Extended exercise (10 points)

Create a new program with a class named `DataMover2` in the default (anonymous) package. The program should accept a variable number of command line arguments, each representing a thread.

The following fields are public and static:

- `arrivalCount`, `totalSent`, `totalArrived`: three `AtomicInteger`, initially set to zero
- `pool`: a threadpool via `ExecutorService`
- `queues`: a list of `BlockingQueue` objects containing integers
- `moverResults`: A list of `Future` objects storing `DataMover2Result` objects.
    - `DataMover2Result` is a class with three publicly accessible integers: `count`, `data`, `forwarded`
- `discards`: list of integers

In case no command line arguments are given, use `123 111 256 404` as default (same as before).

At the beginning the main thread fills the `queues` list with elements and creates a `pool` with a maximum size of `100`.

The `pool` is given a task for each of its thread. Every task is solved by producing a `DataMover2Result` object. For this reason a `Callable` with such return type is to be used. Let's call the return value `result`.

1. The queue number `i` in the `queues` list connects the task number `i` with task number `(i+1)`, acting as the output queue for the former and as the input queue for the latter.
2. The theads are running the tasks as long as the produced values are less than `5n`, where `n` is the number of threads. Until then, the following runs in a loop:
    1. `sends <x>`: Produce integer between `0` and `10_000` named `x`, and put it into the output `queue` (using blocking insert).

- Add `x` to `totalSent`.

2. Try to get an integer from the input `queue`. Use a (pseudo-)randomly choosen timeout between `300` and `1000` ms.
   1. `got nothing...`: If the retrieval is unsuccessful, then start a new iteration immediately.
   2. `got <x>`: If the modulo `n` value of the retrieved number matches the index of the task, then the value has arrived at its destination.
      - Increment `arrivalCount` and `result.count`, add `x` to `result.data`.
   3. `forwards <x>`: Otherwise insert `x-1` to the output queue (also with blocking insert).
      - Increment `result.forwarded`.
   3. Pause for a duration determined by the corresponding command line argument.
3. We will soon make use of the `result`s. These will be available in `moverResults`.

The threads log their activity by printing to standard output. Format your output based on the following example:

```
...
total   9/20 | #0 got 7628
total   9/20 | #1 forwards 2426 [2]
total  10/20 | #1 sends 9281
total  10/20 | #3 got nothing...
...
```

The first two numbers represent the values received so far and the total amount expected (`5n`). `#m` is the task index, while the `[v]` value shows which task would accept the value.

After all tasks are submitted, the main thread should signal the `pool`, so it will no longer accept new tasks, then wait for a maximum of `30` seconds for the pool's termination.

The results of each task (`Callable`) is stored in the `moverResults` list. Add the values of `result.data` and `result.forwarded` together and store the value in the `totalArrived` container. It is possible that some `queues` elements still has numbers inside. Flush and sum these value into the `discards` list.

After summing the values, let's compare the results: the value of `totalSent` should match the sum of `totalArrived` and `discards`. The end of the output of an execution may look like this:

```
total  18/20 | #1 got 9341
total  19/20 | #1 sends 2386
total  19/20 | #0 sends 9761
total  19/20 | #1 got 9761
total  19/20 | #0 forwards 2771 [3]
discarded [0, 2770, 193509, 58277] = 254556
sent 370278 === got 370278 = 115722 + discarded 254556
```

In case the numbers do not match, use the following formatting. This must not happen in a solution worth full points.

```
WRONG sent 370278 !== got 152 = 24 + discarded 128
```