Concurrent Programming 2023/2024

Java concurrency credit assignment

Data storage systems (*storage systems*) must simultaneously meet a number of requirements, in particular, the efficiency of data access operations, the capacity utilization of storage media and the fault tolerance of the devices that are the *storage* media. To this end, they often move chunks of data between different devices. Your task will be to implement in Java mechanisms that coordinate concurrent operations of such transfer according to the following requirements. Use the attached template to implement the solution.

Specification

In our system model, data is grouped into components and stored on devices in such units. Both each device and each data component are assigned an identifier that is immutable and unique within the system (class object `cp2023.base.DeviceId` and `cp2023.base.ComponentId`, respectively). In addition, each device has a certain capacity, that is, the maximum number of components it can store at any time. The assignment of components to devices is managed by the system (an object implementing the `cp2023.base.StorageSystem` interface, shown below):

```
public interface StorageSystem {

    void execute(ComponentTransfer transfer) throws TransferException;

}
```

More precisely, every component that exists in the system is on exactly one device, unless the system user has ordered the transfer of that component to another device (by calling the `execute` method of the aforementioned `StorageSystem` class and passing to it as a parameter an object implementing the `cp2023.base.ComponentTransfer` interface representing the ordered transfer).

```
public interface ComponentTransfer {

    public ComponentId getComponentId();

    public DeviceId getSourceDeviceId();

    public DeviceId getDestinationDeviceId();

    public void prepare();

    public void perform();

}
```

A component transfer is also ordered when the user wants to add a new component to the system (in which case the `getSourceDeviceId` method of the transfer object returns `null`) or remove an existing component from the system (in which case, symmetrically, the `getDestinationDeviceId` method of the transfer object returns `null`). In other words, a single transfer represents one of the three available operations on a component:

- *adding* a new component to the system device (`getSourceDeviceId` of the transfer object returns `null` and `getDestinationDeviceId` returns `non-null` indicating the identifier of the device on which the added component is to be placed),
- *transfer of* an existing component between the system's devices (`getSourceDeviceId` and `getDestinationDeviceId` both return non-nulls representing the identifiers of the current device on which the component is located and the destination device on which the component should be located after the transfer, respectively),
- *remove* the existing component from the device and thus - the system (`getSourceDeviceId` returns `non-null` indicating the ID of the device on which the component is located, and `getDestinationDeviceId` returns `null`).

The ordering of the above three types of operations by the user is beyond the control of the implemented solution. Instead, the task of your solution is to perform the outsourced transfers in a synchronous manner (i.e., if the outsourced transfer is valid, the `execute` method called on an object implementing a `StorageSystem` with a transfer represented as a parameter implementing the `ComponentTransfer` interface cannot complete its operation until that transfer completes). Since many different operations can be ordered simultaneously by the user, the system you implement must ensure that they are coordinated according to the following rules.

At most one transfer can be reported for a component at any time. As long as this transfer has not completed, the component will be called *transferred* and any subsequent transfer reported for this component should be treated as invalid.

The component transfer itself is a two-stage process and can take a long time (especially its second stage). The start of the transfer consists in its preparation (i.e. calling the `prepare` method on the `ComponentTransfer` interface object representing the transfer). Only after such preparation can the data representing the component be transferred (which is done by calling the `perform` method for the aforementioned object). Once the data is transferred (i.e., the `perform` method completes its operation), the transfer ends. Both of the above methods must be executed in the context of the thread ordering the transfer.

Security

A transfer can be either correct or incorrect. The following security requirements apply to correct transfers. Handling of invalid transfers, on the other hand, is described in the following section.

If the transfer represents an operation to remove a component, its initiation is *allowed* without any additional preconditions. Otherwise, the start of the transfer is allowed as long as there is space on the destination device for the component being transferred, in particular, this space is just being or will be released, so it can be reserved thanks to this. More specifically, the start of a transfer representing the transfer or addition of a *Cx* component is *permitted* if any of the following conditions apply:

- There is free space on the transfer target device for a component that has not been reserved by the system for another component that is/will be transferred/added to that device.
- On the target device there is a *Cy* component transferred from this device, the transfer of which has started or is allowed to start, and the space vacated by this component has not been reserved by the system for another component.
- Component *Cx* belongs to a certain set of transferred components such that the destination device of each component from the set is a device on which there is exactly one other component from the set, and the place of none of the components from the set has been reserved for a component outside the set.

If the transfer of a *Cx* component is permitted, but is to take place in a space still occupied by another *Cy* component being transferred (the last two cases above), the second stage of the *Cx* component transfer (i.e., the call to the `perform` function for that transfer) cannot begin before the first stage of the *Cy* component transfer (i.e., the call to the `prepare` function for that transfer) is completed.

Of course, if the transfer of a component is not allowed, it cannot be started (i.e. neither the function of the `prepare`, nor the `perform` function can be called on the object representing this transfer). Your solution

should absolutely provide all of the above security conditions.

Lifespan

As for liveness, on the other hand, the transfer (both its `prepare` and `perform` phases) should start as soon as it is allowed and the other security requirements are met. In case several transfers are competing for space on the device, among those that are allowed, your algorithm should locally prioritize transfers that wait longer on the device. Globally, this could potentially starve certain transfers (feel free to come up with a scenario for such a situation). The solution to this problem is of course implementable, but it complicates the code beyond what we would like to require of you. Therefore, it should not be implemented, especially since in practice, a system user seeing that a transfer to some device can't complete for a long time, could trash other components from that device.

Error handling

Finally, the proposed solution should check if the user-ordered transfer is invalid (which should result in the `StorageSystem` interface's `execute` method raising the corresponding exception inherited from the `cp2023.exceptions.TransferException` class). As explained earlier, a transfer is invalid if at least one of the following conditions exists:

- The transfer does not represent any of the three available component operations or does not point to any component (`IllegalTransferType` exception);
- the device indicated by the transfer as source or destination does not exist in the system (exception `DeviceDoesNotExist`);
- component with an identifier equal to the component being added as part of the transfer already exists in the system (`ComponentAlreadyExists` exception);
- A component with an ID equal to the component being removed or moved as part of the transfer does not exist in the system or is on a different device than the one indicated by the transfer (`ComponentDoesNotExist` exception);
- the component affected by the transfer is already on the device indicated by the transfer as the destination (`ComponentDoesNotNeedTransfer` exception);
- component affected by the transfer is still being transferred (exception `ComponentIsBeingOperatedOn`).

In the solution, you can take any sensible order of checking these conditions.

Requirements

Your task is to implement the system according to the above specification and the provided template using Java 17 concurrency mechanisms. Your source code should be written in accordance with good programming practices. Solutions based on active or semi-active (e.g. `sleep`, `yield` or other methods using time constraints) waiting will not receive any points.

To simplify solutions, we assume that threads reporting transfers are never interrupted (i.e., the `interrupt` method of the `Thread` class is never called for them). The response to the occurrence of a controlled exception resulting from such an interruption (e.g. `InterruptedException` or `BrokenBarrierException`) should be to raise an uncontrolled exception as follows: `throw new RuntimeException("panic: unexpected thread interruption");`.

Specific further formal requirements are as follows.

1. You may not in any way alter the contents of the packages `cp2023.base`, `cp2023.demo` and `cp2023.exceptions`.
2. You can only add classes implementing the solution in the `cp2023.solution` package, but you cannot create any subpackages in this package.
3. Your implementation must not create any threads.
4. Your implementation should not write anything to standard output (`System.out`) and standard diagnostic output (`System.err`).
5. In the `cp2023.solution.StorageSystemFactory` class, you need to add the content of the `newSystem` method, which will be used to instantiate the system you have implemented. Each call to this method should create a new system object. Multiple system objects should be able to run at the same time. However, you must not change the signature of this method or the name of the class or its location in any way. If the system configuration provided as arguments to this method is incorrect (e.g., some component is assigned to a device without the specified capacity or the number of components assigned to some device exceeds its capacity), the method should raise a `java.lang.IllegalArgumentException` exception with an appropriate text message.
6. You can create your own packages for testing, such as `cp2023.tests`, but these packages will be ignored for testing by us, so your system code in particular cannot depend on them.
7. You cannot use non-English characters (especially Polish characters) in Java source files.
8. Your solution should consist of a single file `ab123456.zip`, where `ab123456` should be replaced with your `students.mimuw.edu.pl` machine login (which is usually a concatenation of your initials and index number). This file must have the same structure as the template, that is, it must contain only the `cp2023` directory representing the package of the same name, which contains directories of the corresponding subpackages, at least `base`, `demo`, `exceptions` and `solution`, which in turn contain the corresponding source files (`*.java`).
9. Your solution must compile on the `students.mimuw.edu.pl` machine with the `javac` command `cp2023/base/*.java cp2023/exceptions/*.java cp2023/solution/*.java cp2023/demo/*.java`.
10. In your solution, the demo program, called by the `java cp2023.demo.TransferBurst` command, must work, that is, it must not report any exceptions.

Solutions that do not meet any of the above requirements will not be checked and will automatically get 0 points.

We ask for your understanding! We will have up to more than 160 solutions to test. These solutions will be tested automatically in the first phase. If we had to tweak each solution in any way in order to run the tests, we would lose a lot of time unnecessarily. Therefore, ensuring compliance with the above requirements is on your side.

To give you more information about the testing procedure itself, it will proceed roughly as follows for each solution.

1. The `ZIP` archive with the solution will be extracted to a dedicated root directory on the `students`' machine (or a compatible one as far as the Java version is concerned).
2. All files and subdirectories will be deleted from this directory except for the subdirectory `cp2023/solution` and its contents.
3. All files (and subdirectories) will be removed from the `cp2023/solution` subdirectory except for the files `*.java`.
4. The `cp2023/base`, `cp2023/demo` and `cp2023/exceptions` directories from the provided template will be copied to the root directory so that the interface files and the demo application are in the original version, and the directories with the code of our tests.
5. Everything will be compiled.
6. A demo application will run to see if it works.
7. If unpacking the archive, compiling or running the demo application `fails`, the solution automatically receives 0 points. Otherwise, further test applications and possible additional programs (e.g., anti-plagiarism verification) will be run as part of testing.

All questions and comments should be directed to Konrad Iwanicki via the Moodle forum dedicated to the task. Before posting a question, please check the forum to see if anyone has asked a similar one before.

Good luck!