CSDS 293
Software Craftsmanship
2024 Fall Semester

Programming Assignment 3

Due at the beginning of your discussion session on

Sep 16 – 20, 2024

Reading

In addition to the following topics, the quiz syllabus includes any material covered in the lectures:

- Chapter 16 (controlling loops) in Code Complete.
- Item 48 in Effective Java.
- Section 19.5 (error-handling) in Code Complete.
- Section 17.3 (excluding "Rewrite with a status variable") in Code Complete.
- Java's BigInteger documentation (advanced operations).
- Section 19.1 ("Guidelines for Using Each Control Structure") in Code Complete.

Programming

In this assignment, you will enhance the Distributed Task Scheduling System by implementing advanced scheduling capabilities and basic concurrency control. You'll build upon the foundation established in the first assignment, adding new features and refining existing components to create a more sophisticated and efficient system.

System Architecture and Class Interactions

Building on the architecture established in Project 2, we'll now introduce task prioritization and enhance the `TaskScheduler` to support more complex scheduling scenarios. The existing classes (`Task`, `Server`, `TaskScheduler`, and `Duration`) will be extended or modified to accommodate these new features.

We'll introduce a new `PriorityTask` class that extends the basic `Task` interface, adding the concept of priority to our tasks. The `TaskScheduler` will be updated to consider these priorities when distributing tasks among servers. Additionally, we'll implement basic load balancing to ensure efficient utilization of server resources.

Concurrency control will be introduced to the `TaskScheduler`, making it thread-safe and capable of handling multiple simultaneous scheduling requests. This will involve careful synchronization of shared resources and the use of thread-safe data structures.

Task Prioritization

Extend the `Task` interface to include a priority concept:

- Add a `getPriority()` method returning an enum `TaskPriority` (HIGH, MEDIUM, LOW).
- Implement a `PriorityTask` class that extends `SimpleTask` and includes priority.

Ensure that `PriorityTask` maintains immutability, following the principles in Effective Java. The priority of a task will influence how the `TaskScheduler` assigns it to servers, with higher priority tasks generally being scheduled for execution before lower priority ones.

Enhanced TaskScheduler

Modify the `TaskScheduler` class to support priority-based scheduling. Implement a priority-based task queue that orders tasks based on their priority level. For example, assume the task scheduler receives three tasks with different priorities: 1. A low-priority task that takes 1 second to complete; 2. A high-priority task that takes 5 seconds to complete; and 3. A medium-priority task that takes 3 seconds to complete. These tasks are added to the scheduler in the order: low, high, medium. Despite the order in which the tasks were added, the scheduler should execute them in order based on their priority, which is the high-priority task, the medium-priority task, and the low-priority task.

Use streams and lambdas to sort and assign tasks based on priority, as discussed in Item 48 of Effective Java. This enhancement will allow the system to process more critical

tasks ahead of less important ones, improving overall system responsiveness to high-priority work.

Load Balancing

Implement a simple load balancing algorithm in the `TaskScheduler`. This algorithm should track the number of assigned tasks for each server and use this information when deciding where to schedule new tasks. For example, the task scheduler is configured with three servers: Server1, Server2, and Server3. Ten identical tasks, each taking 2 seconds to complete, are added to the scheduler. The scheduler should distribute the tasks relatively evenly across all three servers to balance the load. So each server should execute about 3 or 4 tasks. The exact distribution may vary slightly, but no server should have significantly more tasks than the others.

A more complicated example may need to combine priority with load balancing. For example, the task scheduler is configured with two servers: Server1 and Server2. Four tasks are added to the scheduler:

- A low-priority task that takes 1 second to complete
- A high-priority task that takes 5 seconds to complete
- Another high-priority task that takes 5 seconds to complete
- A medium-priority task that takes 3 seconds to complete

The scheduler should prioritize the high-priority tasks while also balancing the load between the two servers. So an expected output might be following:

- Server 1: High-priority task completed; Medium-priority task completed
- Server 2: High-priority task completed; Low-priority task completed

Your system can balance competing concerns: it executes high-priority tasks first, but also distributes work across available servers to maintain efficiency. Your system should also has the ability to adapt to changes in available resources and redistribute work accordingly, such as adding new servers.

Use streams to efficiently find the least loaded server for task assignment. The goal is to distribute tasks evenly across all available servers, preventing any single server from becoming a bottleneck in the system. Also, think about how to structure your code to handle these complex scenarios while maintaining clarity and adhering to the principles of good software design we've studied.

TaskScheduler Class

Develop a `TaskScheduler` class that manages task distribution across multiple servers. This class should include methods for adding servers to the scheduler, scheduling individual tasks, and executing all tasks across all servers:

- `void addServer(Server server)`: Adds a server to the scheduler.
- `void scheduleTask(Task task)`: Schedules a task to an appropriate server.
- `Map<Server, List<Task>> executeAll() throws SchedulerException`: Executes tasks on all servers.

Implement defensive copying in the `addServer` and `scheduleTask` methods, as recommended in Item 50 of Effective Java.

Concurrency Control

Make the `TaskScheduler` thread-safe to handle concurrent task scheduling requests. Use synchronized methods or blocks to protect critical sections of code where shared data is accessed or modified. Implement a thread-safe task queue using classes from the java.util.concurrent package, such as `ConcurrentLinkedQueue` or `BlockingQueue`.
Apply the principles of writing correct concurrent programs as discussed in Effective Java, paying particular attention to avoiding race conditions and ensuring that all shared state is properly synchronized.

Defensive Copying

Enhance defensive copying in the `TaskScheduler` and `Server` classes. Implement defensive copying in getters that return mutable objects to prevent external code from inadvertently modifying the internal state of these classes. Use unmodifiable views when returning collections, such as lists of tasks or servers. Follow the guidelines in Item 50 of Effective Java to ensure that your classes remain robust and resistant to errors caused by improper external modifications.

Error Handling

Expand the error handling capabilities of your system. Create a new `SchedulerFullException` to be thrown when no servers can accept new tasks, indicating that the system has reached its capacity. Implement proper exception chaining in all exception

handlers, as discussed in Section 19.4 of Code Complete. This will allow for more informative error reporting and easier debugging of issues in the distributed system.

General Considerations

As you implement these enhancements, continue to adhere to the principles of good software design discussed in our readings. Your classes may contain additional auxiliary private and package-private methods as needed to support their primary functionalities. Write JUnit tests for all new and modified methods to ensure their correct operation under various scenarios, including edge cases.

Continue to include appropriate comments, focusing on explaining complex algorithms or non-obvious design decisions. Remember that with the introduction of concurrency, clear documentation of synchronization strategies becomes particularly important.

You should write JUnit tests to make sure that your primary methods work as intended. However, we will revisit testing later on in the course, so extensive testing is not yet recommended. Similarly, your code should have a reasonable number of comments, but documentation is going to be the topic of a future assignment. As a general guideline at this stage of the course, comments and tests should be similar to those accepted in CSDS 132.

Discussion Guidelines

The class discussion will focus on:

- Implementation of priority-based scheduling
- Use of streams and lambdas for task management
- Concurrency control and thread safety (your choice of synchronization mechanisms and how you've ensured freedom from race conditions and deadlocks.)
- Application of defensive copying techniques

Be prepared to explain your design decisions, particularly regarding the trade-offs involved in your concurrency and load balancing implementations.

Your grade will be affected by the topics covered in this and in previous weeks. Your discussion leader may introduce material that has not been covered yet in the lecture. Although future topics will not contribute to your current grade, you are

expected to follow your leader's advice in revising the assignment.

Submission

Bring a copy to the discussion to display on a projector. Additionally, submit an electronic copy of your program to Canvas. In addition to your code, include a README file explaining how to compile and run the code. The code should be handed in a zip, tar.bz2, or tar.gz archive. Archives in 7z cannot be accepted.

Note on Academic Integrity

It is crucial to maintain academic integrity throughout this course. Copying another group's work or providing your code for others to copy, including posting it on public platforms or social media, is a violation of academic integrity. Ensure that the work you submit is your own, reflecting your understanding and application of the course materials.