Worksheet #1

Java Concurrency Mechanisms for Distributed Systems

Introduction

In the development of distributed systems, processes form the software backbone, and their internal organization often requires the use of multiple threads of control to achieve high performance and handle asynchronous events. While basic Java threads provide the primitive tools for concurrency, the `java.util.concurrent` package offers a high-level framework for building robust, scalable, and thread-safe applications without the complexity of low-level monitor locks.

This package provides a standardized set of utilities that address common concurrency patterns, such as thread management, thread-safe collections, and advanced synchronization aids. By utilizing these components, developers can avoid common pitfalls such as deadlocks and race conditions, which are particularly critical to manage in distributed environments where consistency and dependability are primary design goals.

Development Environment Setup

Java 21 Installation

To ensure compatibility with the modern concurrency features used in this worksheet, **you must use Java 21**.

1) **Standard Package Manager (APT)**: For students using Ubuntu, the simplest way to install the Java Development Kit (JDK) is through the official Ubuntu repositories. Run the following commands in your terminal:

```bash
sudo apt update
sudo apt install openjdk-21-jdk
```

   Verify the installation by running `java -version`.

2) **Version Manager (mise)**:

   As an alternative for users who need to manage multiple Java versions or want a more flexible environment, we recommend using `mise-en-place` (or simply `mise`). Once `mise` is installed, you can install Java 21 with

```bash
mise use --global java@21 # install java 21 and set it as global
```

Compiling and Running with Maven

This project uses Apache Maven for dependency management and build automation. To simplify the setup, a Maven Wrapper (`mvnw`) is included in the project root.

Note: On Unix-based systems (Linux/macOS), ensure the wrapper is executable.

To compile all exercises and verify that there are no syntax errors, execute the following command from the project root:

```bash
./mvnw clean compile
```

Running the Exercises

Each exercise contains a specific entry point (a class with a `main` method). You can run them using the `exec-maven-plugin` provided by the wrapper. Replace the `-Dexec.mainClass` value with the appropriate class name:

- **Exercise 1 (Consistent Counter)**:

```bash
./mvnw exec:java -q -Dexec.mainClass="deti.sd.ex1.CounterTest"
```

- **Exercise 2 (Task Dispatcher)**:

```bash
./mvnw exec:java -q -Dexec.mainClass="deti.sd.ex2.Gateway"
```

- **Exercise 3 (Bounded Buffer)**:

```bash
./mvnw exec:java -q -Dexec.mainClass="deti.sd.ex3.DistributedBufferSystem"
```

- **Exercise 4 (Distributed Barrier)**:

```bash
./mvnw exec:java -q -Dexec.mainClass="deti.sd.ex4.SystemController"
```

Exercise 1: The Consistent Counter (Mutexes vs. Atomic Variables)

In distributed and concurrent systems, multiple threads often need to update a shared state, such as a global request counter or a replica version number. Without proper synchronization, these updates can overlap, leading to the "lost update" problem where the final value is incorrect due to **race conditions**.

Goal: Compare different synchronization strategies in Java, Mutual Exclusion (Mutexes) and Atomic Variables, to ensure data consistency without using the `synchronized` keyword.

Base Code: `src/main/java/deti/sd/ex1`

Tasks

1) **The Unsafe Counter**:
   Analyze a provided base class that uses a primitive `int` and multiple threads to perform increments. Observe that the final count is non-deterministic and usually less than the expected total because the operation `count++` is not atomic at the machine level.

2) **Mutual Exclusion (Muxes)**:
   Implement a thread-safe version using `java.util.concurrent.locks.ReentrantLock`.
   - Ensure that the counter increment is wrapped within a `lock()` and `unlock()` sequence.
   - Use a `try...finally` block to ensure the lock is released even if an exception occurs, maintaining the dependability of the process.

3) **Atomic Variables (Lock-free)**:
   Refactor the counter using `java.util.concurrent.atomic.AtomicInteger`.
   - Utilize the `incrementAndGet()` method to perform the update.
   - Explain why this approach might be more efficient than using a Mutex in high-contention scenarios where threads frequently attempt to update the same variable.

4) **Advanced Logic**:
   Implement a method that updates the counter only if its current value is an even number. For the Mutex version, use standard conditional logic within the lock. For the Atomic version,

use the `compareAndSet(int expect, int update)` method within a loop to achieve a lock-free conditional update.

Exercise 2: The Task Dispatcher (ExecutorService)

Distributed servers often need to handle a burst of requests. Creating a new thread for every request is inefficient due to the high cost of context switching and resource consumption. The `ExecutorService` allows for the separation of task submission from task execution through the use of thread pools.

Goal: Implement a resource-managed task dispatcher that limits the number of concurrent active threads.

Base Code: `src/main/java/deti/sd/ex2`

Tasks

- Replace manual `Thread` creation with an `ExecutorService` using a fixed thread pool.
- Implement a simulation where a "Gateway" process receives 50 computational tasks but is restricted to using only 5 worker threads.
- Utilize the `shutdown()` and `awaitTermination()` methods to ensure the dispatcher waits for all background tasks to complete before the main process exits, ensuring reliable execution.

Questions for Reflection

- If 50 tasks are submitted to a pool of 5 threads, what happens to the 45 tasks that cannot run immediately?
- Why is it important to call shutdown() and awaitTermination() in a distributed system component rather than just letting the main method end?
- How does using a thread pool contribute to the "dependability" of a process compared to creating a new thread per request?

Exercise 3: The Bounded Buffer (Blocking Queues)

Communication between components in a distributed system often requires decoupling the producer of data from the consumer. A `BlockingQueue` acts as a synchronized channel that automatically handles the "wait-notify" logic when the buffer is full or empty.

Goal: Use a thread-safe queue to manage flow control between a network receiver and a data processor.

Base Code: `src/main/java/deti/sd/ex3`

Tasks

- Initialize a `LinkedBlockingQueue` with a fixed capacity to prevent memory exhaustion (back-pressure).
- Develop a "Producer" thread that simulates receiving network packets and placing them into the queue.
- Develop a "Consumer" thread that retrieves and processes these packets.
- Observe how the `put()` and `take()` methods automatically block threads to maintain synchronization without the need for manual locking.

Questions for Reflection

- In this exercise, the producer is faster than the consumer. What happens when the buffer reaches its capacity of 5? How does this protect the "dependability" of the consumer process?

- Why is a `LinkedBlockingQueue` preferred over a standard LinkedList for this multi-threaded interaction?

Exercise 4: The Distributed Barrier (CountDownLatch)

Coordination is a fundamental task in distributed systems, such as ensuring all nodes in a cluster have initialized their local state before a parallel computation starts. The `CountDownLatch` is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

Goal: Coordinate the startup phase of a multi-node system.

Base Code: `src/main/java/deti/sd/ex4`

Task:

- Implement a controller that waits for a group of "Worker" threads to finish an initialization sequence.
- Each worker must signal its readiness to the `CountDownLatch`.
- The controller must block using `await()` until the latch count reaches zero.
- Ensure that the final "System Start" signal is only printed after all workers are confirmed to be ready, simulating a synchronized start in a distributed cluster.

Questions for Reflection

- How does the `CountDownLatch`ensure that the "STARTING DISTRIBUTED SYSTEM" message is always the last thing printed?
- What would happen if one of the `DistributedWorker` threads encountered and exception and failed to call the signal method on the latch? How would this affect the dependability of the whole system?
- Unlike a `CyclicBarrier`, a `CountDownLatch` cannot be reset. In what distributed systems scenario would a reusable barrier (like CyclicBarrier) be more appropriate than a latch?