Homework #6: Building a Framework for Concurrent Task Execution

Due Thursday, May 4th

In this assignment, you will face concurrency in two distinct problems. First, you will make a small simulation thread-safe by adding and documenting synchronization. Second, you will build a framework for remotely executing concurrent tasks that you will subsequently use to build a simple parallel build system and a simple parallel analysis of GitHub data. The first problem focuses on fine-grained synchronization, whereas the second problem focuses on practical mechanisms for parallel programming in Java and will require you to make many design decisions about your overall design and concurrency strategy.

Your goals in this assignment are to:

- Practice writing frameworks or libraries for large-scale reuse.
- Make design decisions about whether and how to synchronize mutable state across multiple threads.
- Effectively use Java APIs for concurrent programming, including executor services and concurrent collections.
- Perform adequate (fine/coarse-grained) synchronization to make concurrent programs thread-safe.
- Perform remote method executions to distribute computations with Java.

Feel free to discuss aspects of your design (e.g. what kind of abstractions you are using for your tasks) with other students or with the course staff, but you may not share your code with other students and must submit your own solution for the assignment.

This assignment has no intermediate deadline, but we strongly encourage that you complete the much smaller first task within the first week (by Thursday, April 27th the latest) before tackling the second task.

Problem 1: Warmup with Synchronizing a Simulation

We provide a simple simulation of a small economy in which different simulated actors buy and sell products concurrently (package *edu.cmu.cs.cs214.hw6.bank*). In the provided form, the simulation is not threadsafe and may loose updates (visible by a change in the total amount of money in the economy). A simple solution would be to always lock on the entire economy for every individual change, but that would be terribly slow. Your task is to add and document synchronization in the simulation to make it safe, but do so in a fine-grained way that reduces lock contention as much as reasonably possible.

Use the `@GuardedBy`, `@Immutable`, `@ThreadSafe`, and `@NotThreadSafe` annotations from `http://jcip.net/` to document your lock strategy, possibly adding additional comments as needed. Do not significantly change the existing implementation; minor rewrites which preserve the general structure of the original simulation are okay.

We encourage you to track performance while you perform changes to see how fine-grained locking improves performance over coarse-grained locking. We encourage you to perform or automate tests or other forms of analysis to ensure safety, but we have no specific requirements.

Problem 2: Design a Remote Task Execution Service

Your goal is to design a service that can be used to schedule computationally intensive tasks on the same or on remote machines. Each task may consists of multiple subtasks; if possible the service may execute multiple subtasks in parallel to speed up computations.

To support a wide range of possible tasks, build the service as a framework or library that can be easily reused for different kinds of tasks. Demonstrate the flexibility of your design by using the service for three kinds of tasks:

- A build system or continuous integration task (like Travis-CI), in which sources and dependencies are downloaded and compiled, tests are executed, and a report is written. The build process may be faster when certain subtasks are executed in parallel. In package *edu.cmu.cs.cs214.hw6.sequential.buildsystem*, we provide a sequential implementation of the steps required to download, compile, and test the code of FindBugs as an example of the kind of steps that a build system may perform.
- An text analysis task that downloads data from some online source and performs some analysis on it, producing an aggregated result. Again, parallelizing subtasks may speed up the entire process. In package *edu.cmu.cs.cs214.hw6.sequential.sentiment*, we provide a sequential implementation of a sample analysis that gathers the text of commit messages from GitHub, performs sentiment analysis to judge whether they have a positive or negative tone, and compares the results across developers, programming languages, and organizations.
- A third task of your choice. Be creative but keep it simple. You are welcome to reuse code from your previous homework solutions this semester.

You have considerable freedom in how you *design* your service, as long as you follow the following constraints:

- A coordinator application schedules tasks, delegates them to workers, and keeps a log of all results. Worker applications receive tasks from a coordinator, perform the task, and return the result to the coordinator. Coordinators and workers should run in separate processes that may be executed on different machines.
- A task is always executed within a single process, but may be split into multiple concurrent threads within that process. Subtasks can therefore access the same shared data structures.
- Each worker performs all work in a working directory (if the task involves files). The entire working directory can be deleted once the task is completed.
- There may be dependencies among subtasks, such that some subtasks may only be executed after other subtasks have finished. Subtasks may dynamically create additional subtasks.
- The overall service should exploit parallelism where reasonable for performance.
- It is possible to read tasks descriptions from files, but for our purposes it is sufficient and likely much easier if the tasks are described in Java code, similar to the sequential implementations we provide.
- Document your synchronization strategy (if any) with `@GuardedBy` and similar annotations as in Problem 1.
- Attempt to gracefully terminate all workers when the coordinator is terminated and attempt to gracefully terminate all threads when a worker is terminated.
- It is encouraged but not required to robustly handle fault scenarios of distributed computations, such as rescheduling a computation if a worker crashes.
- It is strongly encouraged but not required to test your implementation with JUnit tests.

Provide a `design.pdf/design.md` document that describes the design of your solution. This should include a sketch of the overall structure (e.g., UML class or interaction diagrams) and a discussion of your key design decisions with regard to concurrency (granularity, concurrency design patterns used, synchronization strategy, etc) and why you made them.

Provide a `readme.md` file that explains how to start your coordinator and workers and how to schedule your sample tasks and see their results.

This task is intentionally *underspecified* to give you plenty of flexibility in the implementation. Where in doubt make and document reasonable assumptions or contact the course staff.

Handing in your work

Implement your framework/library code in package *edu.cmu.cs.cs214.hw6.taskservice* (and reasonable subpackages if appropriate). Place your sample implementation of the three tasks (build system, GitHub analysis, and your own task) in clearly recognizable packages.

Configure gradle, such that coordinators and workers can be started from the command line and document how to start them in your readme file (e.g., '`gradle run -worker <ip> <port>`'). You may, but are not required to, redesign the hw6 code structure into multiple projects if you prefer.

Place the readme and design files in the root of the hw6 directory.

Hints

In designing your service and the tasks, consider an appropriate strategy to ensure safety, liveness, and decent performance. Consider reusing appropriate APIs and design strategies rather than implementing everything from scratch using low-level concurrency primitives.

Think about the appropriate granularity of your subtasks and the amount of parallelism you can expect to exploit given the overhead costs of creating threads or synchronization. Not all computations may be efficiently parallelizable. It might be worth experimenting with and measuring actual speedups that can be achieved using parallelism.

To demonstrate the solutions, it is sufficient to reimplement the sequential tasks to be executed on your service, but you may, of course, also use the build system for other programs or perform additional analyses with GitHub data.

To communicate between processes, we recommend using sockets (either coordinator or worker would need the IP address and port of the other to establish a connection) and sending serialized Java objects. That is, the coordinator could create a task object and send it to the worker for execution. While it is generally fairly dangerous to send arbitrary code over the network to be executed, this is sufficient for our assignment.

Regarding GitHub analysis: GitHub has a rate limit on their APIs and will reject requests when too many are submitted. If you create an personal access token (`https://github.com/settings/tokens`; deselect all checkboxes except from "public_repo"), you will have a significantly higher rate limit. It is generally a good idea to not commit such tokens to the repository but to read them from a local file.

It is likely possible and advisable that the workers are completely independent of the kind of tasks the coordinator provides to them.

Evaluation

Overall this homework is worth 180 points, 50 for Part 1 and 130 for Part 2.

For Problem 1, we will primarily evaluate the correctness and efficiency of the synchronization strategy, which requires a well designed documentation to understand your synchronization strategy.

For Problem 2, we expect

- a well designed framework or library for tasks and their dependencies with clear API descriptions (Javadoc and additional documentation as appropriate),
- an appropriate design for exploiting concurrency among subtasks,
- the implementation of three nontrivial sample tasks safely exploiting concurrency to speed up the computations, including building FindBugs and performing the GitHub sentiment analysis,
- well written and well documented code as usual,
- and a discussion justifying the key design decisions made toward concurrency.