Group 4 Project 1 Report

Design Details

The basic design of the "Work day" project broke down into 5 separate classes:

Main: Instantiates the different thread objects and sets them onto a synchronized start.

Office: Manages the clock thread, manager thread, and employee threads, as well as the shared use of the conference room.

Clock: A thread that counts through time in a day, adding events to a list to interrupt the clock for situations like lunch and the end of the day.

Employee: A thread that attends meetings throughout the day, and randomly chooses to ask questions which are either answered by another employee (the Team Lead) or the Manager.

Manager: Similar to an employee, but has a different schedule and answers questions rather than asking them.

We made a few assumptions when writing our code:

- We believed employees should ask around 5-20 questions in the average day of work.
- We assumed employees would eat lunch between 12 and 2
- We assumed that, if a team lead left (for lunch, a meeting, or the end of the day) that an employee wouldn't be able to ask any questions and the question they generated would be dropped. (You need to follow the chain of command, and a manager wants to make sure the team lead got the question first)
- Concerning the output, we assumed that it would be helpful to compile the statistics for employees at the end of a given day, such as time spent working, in meetings, waiting for a question, and more. So we chose to include that statistical output at the end of each program run.

We made a few design choices specific to concurrent problem solving:

- The question of how employees know about events like lunchtime or meetings was handled through notifications sent out by the Clock thread, which compiles them in a Vector of event times.
- All of the threads were launched together with a CountDownLatch, and joined at the end so the program could finish.
- Employees have a few volatile variables, such as Boolean values for if they have eaten lunch or arrived at work.

- The Manager has a volatile Blocking Queue to keep the employee questions in order, and volatile Booleans for if he has attended certain meetings or lunch yet.
- The Manager also has a question lock that allows him to only answer one question at a time.
- The Office was where the bulk of the concurrency handling was done.
- Three barrier setups were used in the Office. A CyclicBarrier for the standup meeting and end of day meeting, and an array of CyclicBarriers for each of the team meetings, which ensured that each meeting did not reserve the conference room until every employee was ready.
- Volatile ints were used in the Office to keep track of which team was in the conference room and how many people were present inside it.
- Three locks were used to limit access to different factors – specifically, whether an employee was working, whether the conference room was being reserved, and whether a team lead was answering a question.

Possible Changes Considered

Rather than running all employees on a single "Employees" thread, we opted to run each employee as its separate thread that checked in with the clock. This allowed our synchronizing to be much less cumbersome overall.

We thought about using a barrier combined with sleeps to represent the time elapsed when a manager tries to answer a question, but decided that a wait would serve the purpose better when synchronizing the time between questions.

Results of Testing

During our testing, we found a few problems that caused us to make changes to our code and our design. One thing we noticed was a race condition for the conference room as the teams sought to have their team meetings. We ran into trouble where a member of two different teams would each enter the conference room at the same minute to reserve it for a meeting, leaving one team in the lurch permanently. To fix that problem, we created a variable that allowed a team to call "dibs" on the conference room before anyone stepped inside.

At the start, we had employees with about a 1 in 20 chance of generating a question. The problem that emerged was that there were thousands of opportunities in each tick of the clock, and so we were receiving hundreds of questions from each employee in a day. To fix that, we put a randomizer into the clock that would only generate questions a small fraction of the time.

A lot of bugfixes resulted in minor synchronized blocks scattered throughout the code. Instances like scheduling an event in the clock or reserving the conference room were synchronized to prevent more deadlock on those scenarios.

The end result is a program that seems robust and capable of simulating a work day effectively to the given requirements.