THE UNIVERSITY OF WESTERN AUSTRALIA
SCHOOL OF COMPUTER SCIENCE & SOFTWARE ENGINEERING

CITS3242 Programming Paradigms

Project: Decentralized Experiment Scheduling

DUE DATE: 11:59am, Wednesday 1st June 2011

GROUP MEMBERS DUE: 5pm Wednesday May 4th

This is the programming project for CITS3242 Programming Paradigms. It contributes 40% of your final mark.

You should work on this project in groups of two. Both group members should submit the names, student IDs and logins of the group via the cssubmit system by the above date. Students working alone should also indicate this via cssubmit, but those choosing this option will be marked exactly the same as pairs.

Each individual should separately submit a report, and one member of each group should submit the groups completed project (the F# source code) by 11:59am Wednesday 1st June via https://secure.csse.uwa.edu.au/run/cssubmit.

Background and Overview

Following successful experiments with Large Hadron Collisions, physicists unexpectedly discovered 4 previously unknown particles of possible great importance.[1] This resulted in an international effort to construct a collection of automated labs for studying these particles by combining them in different ways. There are many scientists worldwide with access to use the labs, and each has a "client" computer that they will use to submit experiments via a network.

Your task is to build a prototype of the software that will be used to coordinate the experiments. This includes the following main features:

- Mutual exclusion: due to the potentially explosive nature of the particles in the labs, it is critical that each experiment complete without even very slight interruptions. Hence the labs must never contacted during an experiment, and all coordination of experiments needs to be done by the scientists' computers. In particular the client currently or most recently using a lab is responsible for maintaining a queue of clients waiting for that lab, and for passing this queue to the next client that uses the lab.
- Combining experiments together: in many cases different scientists will submit experiments that have common elements that allow just one of them to be performed, with the result sufficing for the other experiments. This allows a single experiment to be run, saving time and cost, with the result passed to all waiting clients requesting the same experiment, or an experiment that is "sufficed" for according to a set of rules. Each lab has its own set of rules, supplied when the system starts.
- Decentralization: to spread the role of coordination evenly, and avoid the possibility of attacks on central servers, the system will consist only of the scientists' computers (clients) and the labs computers, with a network connecting them. The coordination software will thus run on each of the client computers. Further, a negotiated arrangement strictly requires that client computers only be involved in computation and network communication when that client has submitted an experiment recently. Generally the client currently using a lab is responsible for maintaining a queue of waiting clients, and will hand over to the first client in the queue when it finishes its experiment.

[1] The "physics" in this document was invented just for this project. It is almost certain that it isn't correct.

Starting point provided

The page http://undergraduate.cs.uwa.edu.au/courses/CITS3242/project contains an F# project, as a zip file, that you should use as your starting point. Much of the structure of the program has already been written, allowing you to focus on the key aspects related to the various paradigms studied in this course. This starting point includes:

- Some basic data types, such as the representation of experiments (see below).
- The basic structure of the program, including starting implementations of types/classes for labs and clients. Most of your work will involve completing the coordination parts of the client class, as well as supporting functions for determining which experiments should be run together.
- Some code to help you with debugging - this includes code which allows multiple threads to print as events occur, with the interleaving between threads made obvious by using different colours an indentation based on what object the thread is running in. (This was surprisingly helpful in debugging a sample solution.) It also includes some test cases for the "suffices" part.

Experiments and unification

Experiments are represented by the following data type (**Var** is only for rules, ignore it to start with):

```
type exp = A | B | Mix of exp * exp | Var of string
```

Basic particles and mixing: The two new particles are represented as **A** and **B**

Mixing: An experiment involves introducing some particles, and then "mixing" them together, two at time to form compounds. Mixing is asymmetric – **Mix (A, B)** may have a different result from **Mix (B, A)**.

"Suffices" rules and variables: Due to the nature of the labs, each one has a set of rules that determine when performing one experiment suffices to determine the result of another experiment. As an example, a lab might have the following rule:

`Mix(A, B)` suffices instead of `Mix(B,A)`

To allow more general rules, they can include variables such as:

`Mix(x, y)` suffices instead of `x` (for all `x` and `y` - i.e., each can be any experiment).

Variables actually have the form `Var "x"` so this rule is actually written as:

`Mix(Var "x", Var "y")` suffices instead of `Var "x"`

Rules like this can be used for any values of the variables (x and y in this case). However, rules can also be conditional on other suffices relationships - we call these *subgoals*. Examples of such rules:

`Mix(x,y)` suffices for `xx` if `x` suffices for `xx`

`Mix(x,y)` suffices for `Mix(xx, yy)` if `x` suffices for `xx` and `y` suffices for `yy`.

Unification: To optimize the time and cost involved in each experiment, each time a lab becomes available, the client taking over coordination of that lab should check the queue of waiting clients for that lab for experiments that suffice for or equal its own experiment, and choose one that suffices for or equals the largest number of other experiments in the queue (choosing the smallest experiment in the case of ties).

Thus, your main task for the first part of the project is to write a function that determines whether one experiment suffices for another according to the set of rules for a lab. This involves matching the two experiments against the two experiments in a rule, which may include variables in which case the each variable should be given a value by the match. Then any subgoals should be solved, taking into account the values of the any variables. You need not worry about infinitely continuing to have further subgoals to solve – the rules will be chosen in a way that this will not happen.

Hint: use a type for *substitutions*, which are maps from variable names to experiments. For any two experiments, calculate a substitution that the replaces the variables the experiments in a way that an experiment is obtained that is a combination of both original experiments. (This is generally called *unification* and is an important concept in logic programming and type inference.) To also allow for failures, consider using an option type.

Decentralized Coordination

The coordination part of the project requires you to write code that will run on each client that allows the clients to coordinate with each other to ensure that only one client submits an experiment at a time, and to communicate their experiments to each other for combination according to the first part.

Your client class should meet the following requirements.

1) There are a certain number of clients, numbered 0, 1, ..., n-1, and a certain number of labs, numbered 0, 1, ..., r-1 (with $r < n$).

2) Each client will need to use one of the labs to perform an experiment from time to time, and only one client is allowed to be using each lab at any particular time (otherwise the labs may explode). Clients never submit new experiments until after they receive the result of their previous experiment.

3) When a lab is needed, the client will call the **`doExp`** method on its own instance of the client class, passing the requested experiment. The client class should firstly try to locate a lab that is not being used by another client, and submit the experiment. If all labs are busy, it should be placed in the queue for each lab and use the first lab that becomes available with the client at the front of the queue.

4) While waiting in a queue, if an experiment at the front of the queue is performed that suffices for the experiment requested by a client later in the queue, then the later client stops waiting and returns the result of the experiment at the front the queue. The same should happen if the clients experiment is identical to the one performed at the front of the queue.

5) It is the responsibility of the client that reaches the front of the queue to choose one of the experiments in the queue that suffices or equals the most other experiments in the queue, subject to the constraint that it's own experiments must be sufficed for or equal to the chosen experiment.

6) While using a lab, a Client should maintain a queue of other waiting clients.

7) When the client is done with the lab, it should pass the lab to the next client in the queue, and pass the result of the experiment on to all clients which have experiments that are sufficed for or equal to the experiment performed.

8) Each Client should interact directly with the other clients to locate labs, and wait for them to become available. This interaction should be via method calls only, and **all arguments and returned results must be immutable**. **No shared data structures are allowed.** Waiting should be done via the appropriate monitor mechanisms. **Busy waiting, or periodic polling are not considered appropriate.** Your prototype is unlikely to be useful if you don't follow this point, and you would only be solving a very easy concurrency problem, so expect an appropriately low mark. A partially working solution that follows this point will be given a higher mark than a fully working solution that violates it.

9) Each Client should keep track of which client holds which lab, but should only update this information when the Client is involved in the lab being allocated to a different client. When a lab is reallocated and a Client is not involved, its information on who has the lab will thus be out of date. (But still sufficient to locate the lab – see the following point.)

10) When a Client receives a request from its client, it should contact the Client of the holder of each lab, according to its information, to request the lab. If the contacted Client no longer holds the lab, it should forward the request on to the holder according to its information. When eventually the lab is located, the original client's Client should be informed when the lab becomes available, and the lab should be allocated to that client if no other lab has become available first.

11) When a Client obtains a lab, it should cancel its requests for the remaining labs.

12) When a Client obtains a lab, the Client releasing it should update its information on who holds the lab. Conversely, if a request is canceled, the original requester should update its information. In either case, any Client involved in forwarding the request should also update its information.

13) Initially client 0 holds lab 0, and 1 holds 1, and ... and client *r-1* holds *r-1*. The other clients hold no labs, and every Client knows this initial allocation.

14) A Client which holds no lab and does not request one should not be involved in any communication with other Clients, except for the limited period where it forwards requests if it just held a lab.

Submission and Assessment

You should submit the F# source code for your Client class each individual should submit and a short report. **Make certain that you submit both.**

Your code will marked on how well it meets the specification, as well as its elegance, clarity (how easy it is to read), and how appropriately it uses the features of F#. **Code that uses shared data structures, busy waiting or periodic polling is likely to receive a low mark, since this defeats much of the point of the project.**

25% of your mark will be based on a brief individual report (roughly 1-2 pages, 3 at most) that describes:

- How you shared the work involved in the project.
- The structure of your program and how it works, particularly focusing on the concurrency and functional aspects.
- Any particularly interesting features of your program.
- How well your program works, including any situations or tests for which it doesn't work.
- The steps you followed in building your program, focusing on the problems that you needed to solve in this process.
- A reflection on your experience with this project. Was it too easy or too hard? Too long or too short? Interesting or boring? A good learning experience or mostly pointless? What could have been better?

Getting Help and Forming Groups

Help will be available in the lab sessions. You are also encouraged to make use of https://secure.csse.uwa.edu.au/run/help3242 to discuss issues related to the project with other students and the teaching staff.

Please also use help3242 to find project partners – post if you need a partner, check to see who else is looking for a partner, and post to tell people when you've found a partner. (And pause to consider the concurrency issues involved in this project partner system.)

Rowan Davies <rowan@csse.uwa.edu.au>