Lab 6: Final Project (2013 Edition)

Choose one of the following final projects; you may also propose a different project, but don’t start work on it until you hear if it is approved. The project is due at the end of your exam period (earlier for seniors than everyone else); by one week before the end of classes (April 26), you must have chosen a project, decided who (if anyone) you will work with, and confirmed that all the necessary software infrastructure is working on the lab machines (or some other machine you can use). By the Monday of reading period (May 6) you must schedule a 30-minute time during exam week to demonstrate your project.

I. Alternate Paradigms for Concurrency.

Create an alternate implementation of your Lab 5 data structure, based on one of the other paradigms we have discussed but not studied in depth (refer to the CS356 readings page for references):

A) Initialize and Publish. If you choose this approach, you should be sure to address all the “caveats” discussed in JCiP, to make sure it is a correct Java program, i.e., don’t assume a sequentially consistent memory model, since Java does not promise this in the absence of synchronization.

One challenge with this approach is allowing concurrent updates, e.g. inserting two items simultaneously. Discuss whether or not you think this is possible; if it is, explain how with an example in your test suite; if it is not, clearly explain why not.

B) Transactional. If you choose this approach, you may either use Java along with the (minimally-documented) “multiverse” system, or one of the other languages described in “Programming Concurrency on the JVM”, e.g., Clojure, Scala, or Akka.

One challenge with this approach is deciding the “scale” of your transactions; with locks, holding one lock during an entire “insert” operation eliminated any concurrency. What happens if we try to make the entire “insert” a transaction? If this limits concurrency too much, how can we make smaller transactions that still work?

C) Actors. If you choose this paradigm, you can use Scala or Java with Akka-managed threads (described in “Programming Concurrency on the JVM”, in particular Chapter 8.7 has the example from lecture); you may also evaluate the Go language and explain whether or not it fits in this paradigm.

One challenge with this approach is to get used to the actor paradigm’s slight variation of the object-oriented “send a message to an object” terminology, in which the sending object does not wait for a response (so there is no need for a return at the end of the method — a method updates the object, sends other messages, and ends without a thread “going back to the object that sent the message”). You should demonstrate this difference in your class, if you choose this project.

I am also quite curious about the difference between having one object refer to another (as a Node in a Tree might refer to its parent node) vs. having one object contain another as part of its implementation (as a Fraction might have two Integer objects for the numerator and denominator). These concepts are handled in the same way (as data fields) in Java, and it seems to me that the actor-paradigm version of “send a message” might be necessary in the reference situation but not for containment. I would be curious to hear any insights you get about this difference.
You should include a test suite that shows both the strengths and weaknesses of the paradigm you have chosen. Your implementation should allow as much concurrency as possible (hopefully roughly the same amount as in your Lab 5), and your test suite should illustrate the available concurrency. Your tests should also demonstrate that delays between consecutive steps of an operation will not cause a problem.

Since most of the data structures from Lab 5 can be used to implement sets or dictionaries, you may want to envision a system in which a programming language simultaneously processes a bunch of independent variable definitions, e.g. while processing a function in which we've already seen a bunch of definitions \( x=1;\ y=17;\ z=42; \) followed by some code that uses \( x,\ y,\ z,\ \) and other variables. The language might want to add \( x,\ y,\ \) and \( z\) to its table of variables, then process the code that uses variables (e.g., \texttt{print a*x+b*y+z}), and then remove the local variables before going on to process the next function... you don’t have to actually write a test that does exactly this, but this is one motivating example for concurrency in dictionaries: can all the insertions happen at once? once they are done, can we process lookups for many variables at once? and after that, can we remove many variables simultaneously?

You may choose to undertake one such project by yourself, or work in a team of two if you do either (a) two paradigms for the same data structure, or (b) two significantly different structures with the same alternate paradigm.

II. Concurrency support for HERA.

Create a multi-threaded HERA program, based on any of the three approaches outlined in the practice questions for the co-design exam. You should include a demo of your system in which you can single-step through several instructions of one thread and then several instructions of another (either stepping through a thread-switch operation on a single-processor system, or watching the progress of each of several processors).

If you would like to create a multi-threaded system using a single processor, you’ll need a processor that implements the \texttt{CALL} and \texttt{RETURN} instructions. As this is not a central part of the project, I would be happy to advise anyone who wants to enhance one of our existing CS356 HERA processors in this way.

If you choose to create a multi-processor system, you have to create a single system in which the processors work together on the same data (rather than just using copy-paste to create several copies of a CS240 chip). Specifically, you must have all the processors share the same RAM (though they may have different instruction ROM's), and you must provide some mechanism by which one processor can achieve exclusive access to the RAM for several clock cycles (e.g., to update several variables atomically).

You may undertake this project by yourself, in which case you are allowed to limit your system to always running exactly two threads, or you may undertake it in a team of two if you allow additional threads to be created while the program is running.