Eight Simple Rules for Designing Concurrent Systems

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 5828 — Lecture 10 — 02/11/2010

© University of Colorado, 2010
Lecture Goals

- Review of material in Chapter 4 of Breshears
  - Eight Simple Rules…
In the chapter, Breshears presents eight guidelines for designing concurrent applications.

- We make use of guidelines as designing multithreaded applications is still more of an art than a science.
- That is not to say that we don’t have methodologies or techniques to draw upon.
- We just covered the main approaches in the prior lectures.
- But, for any particular program, there are multiple ways to make it concurrent and it may not be clear which way to go.
Rule 1

- Identify Truly Independent Computations
  - If you can’t identify (in a single threaded application) computations that can be done in parallel, you’re out of luck
  - And, in last lecture, we looked at situations that indeed can’t be made parallel
- But opportunities will be there if you’re willing to look hard enough: from the real world, DVD rental fulfillment
  - pulling discs, packing them, shipping them: all independent
- Consider: File Browsers: what might be independent?
Rule 2

- Implement Concurrency at the Highest Level Possible
  - When discussing “What’s Not Parallel” a common refrain was “you can’t make this parallel, so see if its part of a larger computation that CAN be made parallel”
  - This is such good advice, it was promoted to being a guideline!
  - Two approaches: bottom up, top down
Rule 2: Bottom Up

- Our methodology says to create a concurrent program
  - start with a tuned, single-threaded program
  - and use a profiler to find out where it spends most of its time
- In the bottom-up approach, you start at those “hot spots” and work up; typically, a hotspot will be a loop of some sort
  - See if you can thread the loop
    - If not, move up the call chain, looking for the next loop and see if it can be made parallel...
    - If so, still look up the call chain for other opportunities, first.
      - Why? Granularity! You want coarse-grained tasks for your threads
Rule 2: Top Down

- With knowledge of the location of the hot spot
  - start by looking at the whole application and see if there are parallelization opportunities on the large-scale structure that contains the hot spot
    - if so, you’ve probably found a nice coarse-grained task to assign to your threads
    - If not, move lower in the code towards the hot spot, looking for the first opportunity to make the code concurrent
Rule 3

- Plan Early for Scalability
  - The number of cores will keep increasing
  - You should design your system to take advantage of more cores as they become available
    - Make the number of cores an input variable and design from there
  - In particular, designing systems via data decomposition techniques will provide more scalable systems
    - Humans are always finding more data to process!
  - More data, more tasks; if more cores arrive, you’re ready
Rule 4

- Make use of Thread-Safe Libraries Wherever Possible
  - First, software reuse!
    - Don’t fall prey to Not Invented Here Syndrome
    - if code already exists to do what you need, use it!
  - Second, more libraries are becoming multithread aware
    - That is, they are being built to perform operations concurrently
  - Third, if you make use of libraries, ensure they are thread-safe; if not, you’ll need to synchronize calls to the library
    - Global variables hiding in the library may prevent even this, if the code is not reentrant; if so, you may need to abandon it
Rule 5

- Use the Right Threading Model
  - Avoid the use of explicit threads if you can get away with it
    - They are hard to get right, as we’ve seen
  - Look at libraries that abstract away the need for explicit threads
    - We’ll be looking at OpenMP and Intel Threading Building Blocks in Chapter 5
    - And, I’ll be discussing Scala’s agent model, Go’s goroutines and Clojure’s concurrency primitives
      - all of these models hide explicit threads from the programmer
Rule 6

Never Assume a Particular Order of Execution

- With multiple threads, as we’ve seen, the scheduling of atomic statements is nondeterministic.
- If you care about the ordering of one thread’s execution with respect to another, you have to impose synchronization.

But, to get the best performance, you want to avoid synchronization as much as possible.

- In particular, you want high granularity tasks that don’t require synchronization; this allows your cores to run as fast as possible on each task they’re given.
Use Thread-Local Storage Whenever Possible or Associate Locks with specific data

- Related to Rule 6; the more your threads can use thread-local storage, the less you will need synchronization
- Otherwise, associate a single lock with a single data item
  - in which a data item might be a huge data structure
- This makes it easier for the developer to understand the system; “if I need to update data item A, then I need to acquire lock A first”
Dare to Change the Algorithm for a Better Chance of Concurrency

Sometimes a tuned, single-threaded program makes use of an algorithm which is not amenable to parallelization

They might have picked that algorithm for performance reasons

Strassen’s Algorithm $O(n^{2.81})$ vs. the triple-nested loop algorithm to perform matrix multiplication $O(n^3)$

Change the algorithm used by the single-threaded program to see if you can then make that new algorithm concurrent

BUT: when measuring speedup, compare to the original!!
Coming Up Next

- Lecture 11: Good Enough Design
  - Chapter 5 of Pilone & Miles
- Lecture 12: Model-Based Approach to Concurrency
  - Material will come from optional textbook