Alternative Approaches to Concurrency

Kenneth M. Anderson
University of Colorado, Boulder

CSCI 5828 — Lecture 26 — 04/16/2009

Goals

- Review alternative approaches to concurrency
  - MapReduce
  - Agent Model of Concurrency
    - Examples from Erlang and Scala

Problems with Concurrency

- As mentioned at the beginning of this semester
  - Designing and Implementing Concurrent Systems is hard
    - Data structures shared between threads need to be protected
      - requiring locks (monitors and condition sync) to avoid interference
    - Deadlock and Race conditions are always a concern
      - and sometimes not easy to reproduce
- We've used model-based techniques to attempt to address these concerns ; but this simply shifts the problems to the design phase

Alternative Approaches

- As a result of these concerns, computer scientists have searched for other ways to exploit concurrency
  - in particular using techniques from functional programming
- Functional programming is an approach to programming language design in which functions are
  - first class values (with the same status as int or string)
    - you can pass functions as arguments, return them from functions and store them in variables
  - and have no side effects
    - they take input and produce output
    - this typically means that they operate on immutable values

Example (I)

- In python, strings are immutable
  - a = “Ken @@@”
  - b = a.replace(“@”, “!”)
  - b
    - 'Ken !!!'
  - a
    - 'Ken @@@'
- Replace is a function that takes an immutable value and produces a new immutable value with the desired transformation ; it has no side effects

Example (II)

- Functions as values (in python)
  - def Foo(x, y):
    - return x + y
  - add = Foo
  - add(2, 2)
    - 4
- Here, we defined a function, stored it in a variable, and then used the “call syntax” with that variable to invoke the function that it pointed at

Example (III)

- continuing from previous example
  - def DoIt(fun, x, y): return fun(x,y)
  - DoIt(add, 2, 2)
    - 4
- Here, we defined a function that accepts three values, some other function and two arguments
  - We then invoked that function by passing our add function along with two arguments ;
  - DoIt() is an example of higher-order functions: functions that take functions as parameters
  - Higher-order functions is a common idiom in func. prog.

Relationship to Concurrency?

- How does this relate to concurrency?
  - It offers a new model for designing concurrent systems
    - Each thread operates on immutable data structures using functions with no side effects
    - A thread's data structures are not shared with other threads
    - Work is performed by passing messages between threads
      - If one thread requires data from another that data is copied and then sent
- Such an approach allows each thread to act like a single-threaded program; no danger of interference

Map, Filter, Reduce

- Three common higher order functions are map, filter, reduce
- map(fun, list) -> list
  - Applies fun() to each element of list; returns results in new list
- filter(fun, list) -> list
  - Applies boolean fun() to each element of list; returns new list containing those members of list for which fun() returns True
- reduce(fun, list) -> value
  - Returns a value by applying fun() to successive members of list (total = fun(list[0], list[1]); total = fun(total, list[2]); …)

Examples

- list = [10, 20, 30, 40, 50]

- def double(x): return 2 * x
- def limit(x): return x > 30
- def add(x,y): return x + y

- map(double, list) returns [20, 40, 60, 80, 100]
- filter(limit, list) returns [40, 50]
- reduce(add, list) returns 150

Implications

- map is very powerful
  - especially when you consider that you can pass a list of functions to it and then pass a higher-order function as the function to be applied
    - for example
      - def DoIt(x): return x()
      - map(DoIt, [f(), g(), h(), i(), j(), k()])
- But the real power, with respect to concurrency is that map is simply an abstraction that can, in turn, be implemented in a number of ways

Single Threaded Map

- We could for instance implement map() like this:
  - def map(fun, list):
    - results = []
    - for item in list:
      - results.append(fun(item))
- This would implement map in a single threaded fashion

Multi-threaded Map

- We could also implement map like this (pseudocode):
  - def Mapper(Thread):
    - def __init__(… fun, list): …
    - def run():
      - self.results = map(fun, list)
  - def xmap(fun, list):
    - split list into N parts where N = number of cores
    - create N instances of Mapper(fn, list_i)
    - wait for each thread to end (in order) and grab results
    - append thread results to xmap results
    - return xmap results

Note: threads can complete in any order since each computation is independent

Super Powerful Map

- We could also implement map like this:
  - def supermap(fun, list):
    - divide list into N parts where N equals # of machines
    - send list_i to machine i which then invokes xmap
    - wait for results from each machine
    - combine into single list and return
- Given this implementation, you can apply a very complicated function to a very large list and have (potentially) thousands of machines leap into action to compute the answer

Google

- Indeed, this is what Google does when you submit a search query:
  - def aboveThreshold(x): return x > 0.5 <-- just making this up
  - def probabilityDocumentRelatedToSearchTerm(doc): …
  - searchResults =
    - filter(aboveThreshold,
      - map(probabilityDocumentRelatedToSearchTerm,
        - [<entire contents of the Internet]))

Difference between map and xmap?

- The team behind Erlang published results concerning the difference between map and xmap
  - They make a distinction between
    - CPU-bound computations with little message passing vs.
    - lightweight computations with lots of message passing
- With the former, xmap provides linear speed-up (10 CPUs provides a 10x speed-up, then declining) over map
  - the latter less so (10 CPUs provided 4x speed-up)
  - Indeed, xmap's performance in the latter case tends to max out at 4x no matter how many CPUs were added

Linear speed-up: Hard to achieve!

- On my machine a program to double each member of a large list actually runs faster in single threaded mode!!
  - When using map, you are building just one results list and do not incur any overhead with respect to threading
  - When using xmap, three lists are being created (one per thread, one to collect the results) and
    - you incur overhead to
      - create each thread
      - wait for each one to start running
      - wait for each one to join the main thread

Demo

Agent Model

- The functional language Erlang is credited with creating an approach to concurrency known as the agent model
  - A concurrent program consists of a set of agents
    - Each agent has its own set of data structures that are not shared with other agents
  - Agents can perform computations and send messages
  - Messages sit in an actor's mailbox until it is ready to process them; they are always processed one at a time
    - An actor does not block when sending a message
    - An actor is not interrupted when a message arrives

Examples

- Examples will be presented in Scala
  - Scala is a language which nicely combines both the imperative and functional programming styles
  - It is implemented on top of Java and thus is cross platform
    - I won't spend much time explaining Scala; I'll just focus on the agent model

Example 1


```
import scala.actors._
object SillyActor extends Actor {
  def act() {
    for (i <- 1 to 5) {
      println("I'm acting!")
      Thread.sleep(1000)
      }
    }
  }
}
```

```
object SeriousActor extends Actor {
  def act() {
    for (i <- 1 to 5) {
      println("To be or not to be")
      Thread.sleep(1000)
    }
  }
}
```

Running Example 1

- SillyActor.start() ; SeriousActor.start()
- Demo
  - From this example we can see that Actor is a class that can be sub-classed (just like Thread in Java)
  - You start an actor by calling start()
  - At some point, the scheduler calls the actor's act() method
    - The actor will be active until that method returns
    - This is just like Thread's run() method, only the name has changed

Processing Messages

- To process a message, an actor must call either receive or react
  - react is a special case of receive that we'll discuss below
- You can think of receive as a "switch" statement that specifies the structure of the different type of messages it wants to receive
  - When an actor calls receive, it looks at the mailbox and attempts to find a waiting message that matches one of the branches of the "switch" statement
    - it processes the first match that it finds

Example

- val echoActor = actor {
  - while (true) {
    - receive {
      - case msg =>
        - println("received message: " + msg)
    - }
  - }
- }
- This actor loops forever and prints out any message it receives

A message is sent with the ! operator:

echoActor ! "hi there"
echoActor ! 25

Demo

Conserve Threads

- When an act() method calls receive(), it tells the scala run-time system that this actor needs its own thread
  - The actor may be spending its time switching between processing messages and performing a long computation
- Since threads in Java are not cheap, scala provides the react keyword to tell the runtime that all this thread does is react to messages
  - This means it spends most of its time blocked
    - Scala uses this information to assign “react actors” to a single thread, thus conserving threads in the overall system

Example

- object NameResolver extends Actor {
  - ...
  - def act() {
    - react {
      - case (name: String, actor: Actor) =>
        - actor ! getIp(name)
        - act()
      - case “EXIT” =>
        - println(“quitting”)
    - }
    - ...

Note: no explicit loop; that’s because react doesn’t return (enables sharing of multiple actors on a single thread)

instead, react must call act() if it wants to keep waiting for messages

Results

- To test Scala's claim that react helps conserve threads
  - I wrote a program that can create a specified number of NameResolvers that either
    - use receive or
    - use react
- Results: when creating 100 NameResolvers
  - using receive: 104 threads created
  - using react: 7 threads created (!)

Past Examples

- With the Agent model of concurrency, you can easily avoid interference problems
  - Here's an example of the ornamental garden problem
    - No need for mutual exclusion: create two agents that act as turnstiles and have them send increment messages to a shared counter agent
- However, it can sometimes be tricky to design interactions
  - Here's an example of the museum problem written in this model

Spawning Actors (I)

- The museum example demonstrates a common design idiom in the Agent model of concurrency
  - An agent can only respond to messages when its not doing anything else
  - makes sense: that's just like a single threaded program
    - Think Web browsers and loading images;
      - if they didn't use multiple threads, web pages would load very slowly indeed!
- So, if an agent needs to perform a long computation, it needs to spawn another agent to do that for them

Spawning Agents (II)

- def reminder() {
  - val mainActor = this
  - Actor.actor {
    - Thread.sleep(1000+generator.nextInt(1000))
    - mainActor ! "reminder"
  - }
- }
- ...
  - receive {
    - case "reminder" =>
      - counter ! "increment"
      - reminder()

Final Example

- Message syntax can be as complex as you need it
  - Here's an example of a network node status monitor
    - Taken from this tutorial
    - It queries a domain to see if its "alive"
  - But first
    - Since this example uses Scala "case classes" to create more complex messages with domain-specific syntax
    - Lets do a quick tutorial on case classes

Case Classes

- abstract class Expr
- case class Var(name : String) extends Expr
- case class Number(num: Double) extends Expr
- case class UnOp(operator: String, arg: Expr) extends Expr
- case class BinOp(operator: String, left: Expr, right: Expr) extends Expr

- To create an expression you can now say
- var op = BinOp(“+”, Number(0), Var(x)) // equals 0 + x

Match Expressions

- Case classes shine when used in match statements
- def simplify(expr : Expr) : Expr = expr match {
  - case UnOp(“-”, UnOp(“-”, e)) => e
  - case BinOp(“+”, Number(0), e) => e
  - case BinOp(“*”, Number(1), e) => e
  - case _ => expr
- }
- simplify(BinOp(“+”, Number(0), Var(x))) returns Var(x)
  - because “0 + x” == “x”

Case classes in Example

- case class NodeStatusRequest(address: InetAddress, a: Actor)
- sealed abstract class NodeStatus
- case class Available(address: InetAddress) extends NodeStatus
- case class Unresponsive(
  - address: InetAddress,
  - reason: Option[String]) extends NodeStatus

- Option[String] is a special type that can either have the value Some(String) or None

Demo

- This demo sets up an actor to check the availability of various domains
- It then passes a few messages to this actor and then waits for the actor to respond
  - It can also handle the case when it gets an unexpected message

Wrapping Up

- We have looked at a few alternative models to the “locks and shared data” model of concurrency that
  - draw on functional programming techniques
  - do not allow threads to share data
  - allow threads to communicate via asynchronous messages
- Deadlock and Race conditions are still possible in this model but harder to achieve
  - However, interference is simply not possible in this model
- Functional techniques seem like a promising method for tackling concurrency on multi-core hardware

Coming Up

- Lecture 27: Dealing with Bugs
  - Chapter 11 of Head First Software Development
- Lecture 28: Software Abstractions
  - Overview of Software Abstractions Optional Textbook