Scala Concurrency Notes

Parallelism and Concurrency

Background

- Modern CPUS are getting faster by increasing the number of cores instead of increasing clock speed
    - This is largely because power dissipation requirements of a chip increase dramatically with clock speed, known as the “power wall”
- Simultaneous computation by separate/independent physical processing elements is called “Parallel computation”
- Concurrent computation refers to time-multiplexed computation
    - Concurrent processing is generally focused on improving modularity/maintainability and perhaps response latency
    - Parallel is about using the available hardware to complete the work sooner
- Scala collections are not intrinsically threadsafe, so avoid concurrent writes to mutable versions (immutable ones are only ever read, so are safe!)
    - Never write to a collection that’s being concurrently traversed
    - Never read from a collection that’s being concurrently modified
    - Immutable collections, and Java concurrent collections may be used from multiple threads

Parallel computation forms

- Parallel computation forms include:
    - ***Bit parallel*** (4, 8, 16, 32, 64 bit words)

    - ***Instruction parallel*** (independent operations in a sequence run at the same time)
    - ***Task parallelism*** (separate “threads” making progress on related sub-tasks that contribute progress to the overall task)
    - ***Data parallel*** (the same computations performed in parallel on subsets of an overall dataset)

Data parallelism

- The “data parallel” approach depends on the ability to subdivide data, process the subsets independently, and then merge the results back.
    - Splitting and rejoining are the essence of the “map-reduce” concept.
    - Data parallel may be constructed in an ad-hoc fashion but is supported directly by Scala’s parallel collections
    - The operations must (generally) be associative and independent, since the order of operations is unpredictable, and data cannot easily be shared among processing threads
    - Preparing a parallel collection from a non-parallel one involves compute costs. For List, this is expensive, for arrays and trees the cost is much lower.
- Parallel trees:
    - Can be used in a purely functional fashion (they’re immutable) so threadsafety is good.
    - Typically exhibit poor “locality” which means they behave poorly with respect to the CPU cache.)
    - Involve lots of small objects to construct a large tree. This can behave poorly with respect to garbage collection
    - Combine efficiently

- Arrays:
    - Individual cells must be protected from concurrent modifications
    - Have excellent index-based access
    - Have good locality
    - Are expensive to concatenate
- Scala provides several structures for parallel computation
    - ParIterable, ParSet, ParMap, ParSeq
    - There are also “parent” traits that are agnostic regarding parallelism, these are GenIterable, GenSet, GenMap, GenSeq
    - Monadic operations (`flatMap`, `map`, `withFilter`, etc.) on parallel collections shard the data and run in parallel. Because of this, for expressions built on parallel collections also run parallel automatically
- Manual splitting / combining
    - Divide source data into groups, and provide “thread-private” destination structures (because these are thread-private, they can be mutable)
    - Recombine the partial results after processing all the individual sub-groups
    - The processing requirements for splitting and recombination are crucial to good performance
    - The `Splitter` trait (which extends `Iterator`) allows dividing an `Iterator` into sub-chunks
    - The `Combiner`, which extends `Builder`, is useful for merging things back together.

General concurrency problems

- Three basic problems must be addressed in concurrent or parallel computation
    - Visibility

    - “Transactional” correctness
    - Timing
- In addition, performance and responsiveness can be lost from other problems, such as:
    - Deadlock
    - Memory Bottleneck (trivial computations may be memory bound)
    - Context switch overhead
- Overview of JVM memory model
    - Platform independent rules defined using “Happens-Before” relationships
    - If hb(a,b) and b observes an effect of a, then b will see the effect of a
    - Must not make assumptions about implementation
    - Happens-before can be quite difficult to reason about in any but very simple cases--avoid temptation to be clever
    - ***CRITICALLY IMPORTANT Happens-before describes “visibility of effect”, not actual order of occurrence.***
- 8 Language defined relationships:
    1. Lines of code ***in a single thread*** have happens-before relationships consistent with the meaning of sequence, iteration, and selection in the source language.
    2. Happens before relationships are transitive; that is if hb(a,b) and hb(b,c) then hb(a,c) -- *this rule can propagate relationships across threads and is therefore very important*
    3. Start a thread by one thread happens before the target thread begins
    4. End of a thread happens before another thread determines the thread ended

5. Write to a volatile variable happens before a subsequent read of that variable
6. Release of a monitor (exiting a synchronized region) happens before a subsequent acquisition of that monitor (entry to a region synchronized on the same object)
7. Interrupting a thread by one thread happens before the target thread notices it has been interrupted
8. End of a constructor happens before start of finalization

- In Scala the JVM's notion of volatile variables and monitor locks are provided through:
  - `@volatile` annotation
  - `AnyRef.synchronized` method
- In general, using the primitive mechanisms for interacting with the Java Memory Model is complex and error prone. It's almost always better to use higher level library and API features instead.

Actor Model

- A simple, well-established model for parallel and concurrent computation is known variously as the pipeline, producer-consumer, or actor model.
- Earlier versions of Scala provided an actor implementation as part of the core libraries, but these have been deprecated, removed from the API, and instead the Akka framework is recommended.
- An actor receives messages through a mailbox (similar to a queue), which solves the three major data sharing problems of visibility, timing, and transactional correctness
  - When the message is taken from the queue the infrastructure guarantees that all the contents of the message are visible to the receiver

    - The software must be written in a way that ensure that message contents are never mutated after the message is sent. If this rule is broken, the changes might not be seen, and the protections of the actor model are lost. Generally, all messages should be entirely immutable as this avoids the risk of error. Scala case classes are excellent for messages.
    - Messages sent by one source to one receiver are received in the order they were sent. However, messages from other sources may be interleaved in unpredictable ways
    - Messages are received after they are written, and no CPU time is spent polling for them. This provides a solution to the timing problem
- Akka's actor implementation uses a callback type mechanism for processing messages in actors. This scales better than creating many threads

Configuring Akka

- To include Akka into an SBT project add these lines:

```
lazy val akkaVersion = "2.5.3"

libraryDependencies ++= Seq("com.typesafe.akka"
%% "akka-actor" % akkaVersion)
```

Foundations of the Akka API

- Ensure the necessary imports are available, e.g.:

```
import akka.actor.{Actor, ActorRef, ActorSystem,
Props}
```

- Configure an Actor infrastructure

```
val system = ActorSystem("MyActors")
```

- Define the actor's class as a subclass of Actor, and give it a `receive` method. This is a partial function that processes the messages that the actor expects. Body will usually be implemented using case clauses

```
class MyActor extends Actor {
  override def receive = {
    case "Hello" => sender ! "Nice to meet you"
    case x => println(s"received $x")
  }
}
```

- Use the actor system to create and start the actor, storing an actor reference to use for sending messages. The first argument is a recipe for creating the actor (the system might re-create it if it crashes), the second is an optional name.

```
val act = system.actorOf(Props[MyActor], "act")
```

- Then use the exclamation point operator (also the two-argument `tell` method) to send a message to the actor through its reference

```
act ! (oven, "Hello")
```

- Note: this snippet doesn't show the sender receiving the response that the actor defined above would return. Such a returned message is simply handled, asynchronously, by the sender's receive method

Creating the Actor

- Actors can be instantiated in a number of ways, but in all cases the actor system must do the work. There are two reasons for this:

    - An actor must never be accessible to another, all interaction takes place by sending messages. The ActorRef is therefore the only generally accessible element of the actor, and that is what is returned by the Actor system when it creates the actor
    - If an actor throws an exception the actor system will remove and replace it with a new instance
- To support the creation, the actorOf method takes an argument of type Props. This can constructed in several ways optionally supplying constructor arguments:

```
Props[MyActor]
Props(classOf[MyActor], "ArgOne", "ArgTwo")
Props(new MyActor("ArgOne", "ArgTwo"))
```

    - Note that the third form uses a pass by name argument, allowing the construction to be performed by the actor system when needed.
- An actor can create a “child” actor, over which it can have a supervisory relationship. This approach can also build subsystems of related actors that cooperate. This can be done using the `actorOf` method in the parent actor’s `context`. Context is available as a member of the Actor base:

```
// In a receive method:
val child: ActorRef = this.context.actorOf(
  Props[MyChildActor], "child1")
```

- Such a child actor may be “watched”. This means that if the actor is stopped cleanly--for example by sending it a `PoisonPill` message, or calling `stop()` on it--then the parent will receive a `Terminated(actorRef)` message.
    - Watch an an actor with: `context.watch(actorRef)`

Actor Paths

- Actors can be described using a URL-like format. That format reflects the parent-child hierarchy, and includes prefix elements.
- In effect, the path describes a service endpoint in a relatively abstract way.
- Note that an actor path can describe a path that does not exist, or has not yet been created.
- A “root” actor, created with the name “myRootActor” in an ActorSystem named “MyActorSystem” would likely be represented as:

```
akka://MyActorSystem/user/myRootActor
```

- If this actor created a child called “myChild”, that would be represented as:

```
akka://MyActorSystem/user/myRootActor/myChild
```

- Actor paths can be used to find and communicate with actors. This can be useful for building loosely coupled subsystems.
- To build an ActorPath:

```
val p = ActorPath
  .fromString("akka://system/user")
```

- An ActorPath can be extended using the forward slash operator:

```
val p2 = p / "actTwo"
```

- To send a message directly an actor that is believed to be on a particular path, use an ActorSelection built from the path:
  ```
  system.actorSelection(p2) ! "Message "
  ```

Broadcasting

- An ActorPath can represent wildcard paths:

```
val p = ActorPath
  .fromString("akka://system/user/*")
```

- A message sent to such a path will be broadcast to all the actors on matching paths
- A router can be constructed that allows sending a message to all the configured recipients. The message should be wrapped in a Broadcast object, which is stripped off before the message reaches the recipient(s):

```
val mr1 = sys.actorOf(Props[MyRoutee], "one")
val mr2 = sys.actorOf(Props[MyRoutee], "two")
val r = new Router(BroadcastRoutingLogic())
  .addRoutee(mr1).addRoutee(mr2)
r.route(new Broadcast("Listen up!"),
  Actor.noSender)
```

- Notes:
  - The message is sent through the router using the `route` method, not the `tell` (`!`) method.
  - The `route` method requires a sender; `Actor.noSender` is a reference to the dead letter message box, and can be used when no reply makes sense
  - The router is immutable; calls to `addRoutee` return a new router. Take care to send broadcast messages to the right instance!

Actor fields

- The actor trait includes several elements that are useful in the body of the implementation of an actor.
- Since `this` refers to the actor as a whole, and it's common to need to access the actor's own `ActorRef`, the `Actor` trait provides the `self` field. Remember that the actor's

implementation (`this`) should not be shared, but the corresponding `ActorRef` is intended for that purpose.
- The `sender()` method allows the actor to determine the `ActorRef` that originated the message that is currently being handled. This can be used to send a reply message, but note that such replies are not directly tied to the original message, and are asynchronous.
- The `context` field provides access to the `ActorContext`

The ActorContext

- The ActorContext also provides several useful features including:
    - Create a child actor, with `actorOf`
    - Change the behavior of the actor with `become`
    - Find child and parent actors
    - Obtain the actor's `ExecutionContext` using the `dispatcher` method. This allows asynchronous execution of arbitrary tasks, represented as `Runnable` objects
    - Stop a child actor using the `stop` method. When an actor is stopped, it will no longer receive messages, and if another actor is watching it, that actor will receive a `Terminated` message.
    - Access the `ActorSystem` using the `system` method
    - Request a Terminated message if a specific child actor is stopped cleanly. This is done using the `watch` method

Scheduled Execution

- The ActorSystem can trigger execution of units of work (for example `Runnable` objects) either after a delay, or on a regular schedule. The scheduler is a best effort, low-accuracy system, but it serves well in a great many situations.

- Obtain a reference to the scheduler from the system, and invoke one of the overloaded schedule methods on it. This example uses an initial delay, and an interval between subsequent executions, along with a Runnable job:

```
system.scheduler.schedule(
  FiniteDuration(1, TimeUnit.SECONDS),
  FiniteDuration(1, TimeUnit.SECONDS),
  () => println(System.currentTimeMillis()))
```

- The schedule method takes an implicit parameter for an execution context. This is normally satisfied by the import:

```
scala.concurrent.ExecutionContext.Implicits.global
```

- Single executions are also supported by the scheduler using the `scheduleOnce` method
- Scheduling a job returns a `Cancellable` (note the British spelling, with double-l) which allows the job to be canceled.

Event Bus

- The event bus provides a publish/subscribe type of message delivery system. It is often used for sending timer ticks and for handling otherwise undeliverable messages (known as dead letters).
- Obtain a reference to the event bus from the ActorSystem, and publish any object on it as a message (it's usual to restrict these messages to immutable case class instances, as with all messages in actor systems):

```
case class TickMessage(time: Long)
[...]
system.eventStream.publish(
  TickMessage(System.currentTimeMillis()))
```

- Messages on the event bus are handled by actors that are subscribed to the bus for particular classes of event, that is, for particular object types.

```
system.eventStream.subscribe(
  receiverActor, classOf[TickMessage])
```

- The subscription honors object hierarchy, so subscribing for a parent class will result in receiving events of all the subclass types
- Actors receiving messages from the event bus can receive normal messages too
- Undeliverable messages in the actor system result in events on the event bus of type `DeadLetter` so simply subscribing to `classOf[DeadLetter]` will allow an actor to receive all the dead letters in the system

The Future and Promise API

- A promise, or promise pipeline, is a monad-like concept that handles asynchrony allowing the programmer to specify a series of transformation that are applied to data after they have become available
- The internal implementation of promise libraries generally use a callback mechanism, thereby avoiding the creation of many threads

Scala Promise API

- Future is a monadic structure that allows a computation to be performed in a separate thread (taken from a pool) and the result of that operation to be passed forward along the monad structure
- The execution pool is normally provided using implicits via the import:

`scala.concurrent`
`.ExecutionContext.Implicits.global`

- Future factory with behavior argument
  - isCompleted and value attributes
- To handle callback type situations, use a Promise
  - The promise is used to create and control a future
  - Upon completion of the callback/asynchronous task, inject the result data into the pipeline using any one of `complete(Try)`, `success(value)`, and `failure(exception)`
- Monad-like behaviors for handling problems include `failed`, `fallBackTo`, `recover`, `recoverWith`