orphan:

Swift Memory and Concurrency Model

> **Warning**
>
> This is a very early design document discussing the features of a possible Swift concurrency model. It should not be taken as a plan of record.

The goal of this writeup is to provide a safe and efficient way to model, design, and implement concurrent applications in Swift. It is believed that it will completely eliminate data races and reduce deadlocks in swift apps, and will allow for important performance wins as well. This happens by eliminating shared mutable data, locks, and the need for most atomic memory accesses. The model is quite different from what traditional unix folks are used to though. :-)

As usual, Swift will eventually support unsafe constructs, so if it turns out that the model isn't strong enough for some particular use case, that coder can always fall back to the hunting for concurrency with arrows and bone knives instead of using the native language support. Ideally this will never be required though.

This starts by describing the basic model, then talks about issues and some possible extensions to improve the model.

Language Concepts

The **unit of concurrency** in Swift is an Actor. As an execution context, an actor corresponds to a lightweight thread or dispatch queue that can respond to a declared list of async messages. Each actor has its own private set of mutable data, can communicate with other actors by sending them async messages, and can lock each other to perform synchronous actions.

Only one message can be active at a time (you can also extend the model to allow read/writer locks as a refinement) in an actor. Through the type system and runtime, we make it impossible for an actor to read or change another actor's mutable data: all mutable data is private to some actor. When a message is sent from one actor to another, the argument is deep copied to ensure that two actors can never share mutable data.

To achieve this, there are three important kinds of memory object in Swift. Given a static type, it is obvious what kind it is from its definition. These kinds are:

1. **Immutable Data** - Immutable data (which can have a constructor, but whose value cannot be changed after it completes) is shareable across actors, and it would make sense to unique them where possible. Immutable data can (transitively) point to other immutable data, but it isn't valid (and the compiler rejects) immutable data that is pointing to mutable data. For example:

   ```
   struct [immutable, inout] IList { data : int, next : List }
   ...
   var a = IList(1, IList(2, IList(3, nil)))
   ...
   a.data = 42 // error, can't mutate field of immutable 'IList' type!
   a.next = nil // error, same deal
   a = IList(2, IList(3, nil)) // ok, "a" itself is mutable, it now points to something new.
   a = IList(1, a) // ok
   ```

   Strings are a very important example of (inout) immutable data.

2. **Normal mutable data** - Data (objects, structs, etc) are mutable by default, and are thus local to the containing actor. Mutable data can point to immutable data with no problem, and since it is local to an actor, no synchronization or atomic accesses are ever required for any mutable data. For example:

   ```
   struct [inout] MEntry { x : int, y : int, list : IList }
   ...
   var b = MEntry(4, 2, IList(1, nil))
   b.x = 4 // ok
   b.y = 71 // ok
   b.list = nil // ok
   b.list = IList(2, nil) // ok
   b.list.data = 42 // error, can't change immutable data.
   ```

   As part of mutable data, it is worth pointing out that mutable "global variables" in swift are not truly global, they are local to the current actor (somewhat similar to "thread local storage", or perhaps to "an ivar on the actor"). Immutable global variables (like lookup tables) are simple immutable data just like today. Global variables with "static constructors / initializers" in the C++ sense have their constructor lazily run on the first use of the variable.

   Not having mutable shared state also prevents race conditions from turning into safety violations. Here's a description of the go issue which causes them to be unsafe when running multiple threads: <http://research.swtch.com/gorace>

   Note that a highly desirable feature of the language is the ability to take an instance of a mutable type and *make it immutable* somehow, allowing it to be passed around by-value. It is unclear how to achieve this, but perhaps typestate can magically make it possible.

3. **Actors** - In addition to being the unit of concurrency, Actors have an address and are thus memory objects. Actors themselves can have mutable fields, one actor can point to another, and mutable data can point to an actor. However, any mutable data in an actor can only be directly accessed by that actor, it isn't possible for one actor to access another actor's mutable data. Also note that actors and objects are completely unrelated: it is not possible for an object to inherit from an actor, and it is not possible for an actor to inherit from an object. Either can contain the other as an ivar though.

   Syntax for actors is TBD and not super important for now, but here is a silly multithreaded mandelbrot example, that does each pixel in "parallel", to illustrate some ideas:

   ```
   func do_mandelbrot(_ x : float, y : float) -> int {
     // details elided
   }

   actor MandelbrotCalculator {
     func compute(_ x : float, y : float, Driver D) {
       var num_iters = do_mandelbrot(x, y)
       D.collect_point(x, y, num_iters)
     }
   ```

```
}

actor Driver {
  var result : image; // result and numpoints are mutable per-actor data.
  var numpoints : int;
  func main() {
    result = new image()
    foreach i in -2.0 ... 2.0 by 0.001 {
      // Arbitrarily, create one MandelbrotCalculator for each row.
      var MC = new MandelbrotCalculator()
      foreach j in -2.0 ... 2.0 by 0.001 {
        MC.compute(i, j, self)
        ++numpoints;
      }
    }
  }

  func collect_point(_ x : float, y : float, num_iters : int) {
    result.setPoint(x, y, Color(num_iters, num_iters, num_iters))
    if (--numpoints == 0)
    draw(result)
  }
}
```

Though actors have mutable data (like 'result' and 'numpoints'), there is no need for any synchronization on that mutable data.

One of the great things about this model (in my opinion) is that it gives programmers a way to reason about granularity, and the data copy/sharing issue gives them something very concrete and understandable that they can use to make design decisions when building their app. While it is a common pattern to have one class that corresponds to a thread in C++ and ObjC, this is an informal pattern -- baking this into the language with actors and giving a semantic difference between objects and actors makes the tradeoffs crisp and easy to understand and reason about.

Communicating with Actors

As the example above shows, the primary and preferred way to communicate with actors is through one-way asynchronous messages. Asynchronous message sensed are nice because they cannot block, deadlock, or have other bad effects. However, they aren't great for two things: 1) invoking multiple methods on an actor that need to be synchronized together, and 2) getting a value back from the actor.

Sending multiple messages asynchronously

With the basic approach above, you can only perform actions on actors that are built into the actor. For example, if you had an actor with two methods:

```
actor MyActor {
  func foo() {...}
  func bar() {...}
  func getvalue() -> double {... }
}
```

Then there is no way to perform a composite operation that needs to "atomically" perform foo() and bar() without any other operations getting in between. If you had code like this:

```
var a : MyActor = ...
a.foo()
a.bar()
```

Then the foo/bar methods are both sent asynchronously, and (while they would be ordered with respect to each other) there is no guarantee that some other method wouldn't be run in between them. To handle this, the async block structure can be used to submit a sequence of code that is atomically run in the actor's context, e.g.:

```
var a : MyActor = ...
async a {
  a.foo()
  a.bar()
}
```

This conceptually submits a closure to run in the context of the actor. If you look at it this way, an async message send is conceptually equivalent to an async block. As such, the original example was equivalent to:

```
var a : MyActor = ...
async a { a.foo() }
async a { a.bar() }
```

which makes it pretty clear that the two sends are separate from each other. We could optionally require all accesses to an actor to be in an async block, which would make this behavior really clear at the cost of coding clarity.

It is worth pointing out that you can't asynchronously call a message and get its return value back. However, if the return value is ignored, a message send can be performed. For example, "a.getvalue()" would be fine so long as the result is ignored or if the value is in an explicit async block structure.

From an implementation perspective, the code above corresponds directly to GCD's dispatch_asynch on a per-actor queue.

Performing synchronous operations

Asynchronous calls are nice and define away the possibility of deadlock, but at some point you need to get a return value back and async programming is very awkward. To handle this, a 'synch' block is used. For example, the following is valid:

```
var x : double
synch a {
  x = a.getvalue();
}
```

but this is not:

```
    var x = a.getvalue();
```

A synch block statement is directly related to dispatch_sync and conceptually locks the specified actor's lock/queue and performs the block within its context.

Memory Ownership Model

Within an actor there is a question of how ownership is handled. It's not in the scope of this document to say what the "one true model" is, but here are a couple of interesting observations:

1. **Automated reference counting** would be much more efficient in this model than in ObjC, because the compiler statically knows whether something is mutable data or is shared. Mutable data (e.g. normal objects) can be ref counted with non-atomic reference counting, which is 20-30x faster than atomic adjustments. Actors are shared, so they'd have to have atomic ref counts, but they should be much much less common than the normal objects in the program. Immutable data is shared (and thus needs atomic reference counts) but there are optimizations that can be performed since the edges in the pointer graph can never change and cycles aren't possible in immutable data.
2. **Garbage collection** for mutable data becomes a lot more attractive than in ObjC for four reasons: 1) all GC is local to an actor, so you don't need to stop the world to do a collection. 2) actors have natural local quiescent points: when they have finished servicing a message, if their dispatch queue is empty, they go to sleep. If nothing else in the CPU needs the thread, it would be a natural time to collect. 3) GC would be fully precise in swift, unlike in ObjC, no conservative stack scanning or other hacks are needed. 4) If GC is used for mutable data, it would make sense to still use reference counting for actors themselves and especially for immutable data, meaning that you'd have *no* "whole process" GC.
3. Each actor can use a **different memory management policy**: it is completely fine for one actor to be GC and one actor to be ARC, and another to be manually malloc/freed (and thus unsafe) because actors can't reach each other's pointers. However, realistically, we will still have to pick "the right" model, because different actors can share the same code (e.g. they can instantiate the same objects) and the compiled code has to implement the model the actor wants.

Issues with this Model

There are two significant issues with this model: 1) the amount of data copying may be excessive if you have lots of messages each passing lots of mutable data that is deep copied, and 2) the awkward nature of async programming for some (common) classes of programming. For example, the "branch and rejoin" pattern in the example requires a counter to know when everyone rejoined, and we really want a "parallel for loop".

I'd advocate implementing the simple model first, but once it is there, there are several extensions that can help with these two problems:

No copy is needed for some important cases: If you can prove (through the type system) that an object graph has a single (unique) pointer to it, the pointer value can be sent in the message and nil'd out in the sender. In this way you're "transferring" ownership of the subgraph from one actor to the other. It's not fully clear how to do this though. Another similar example: if we add some way for an actor to self destruct along with a message send, then it is safe for an actor to transfer any and all of its mutable state to another actor when it destroys itself, avoiding a copy.

Getters for trivial immutable actor fields: If an actor has an ivar with an immutable type, then we can make all stores to it atomic, and allow other actors to access the ivar. Silly example:

```
    actor Window {
      var title : string; // string is an immutable by-ref type.
      ...
    }

    ...
    var x = new Window;
    print(x.title) // ok, all stores will be atomic, an (recursively) immutable data is valid in all actors, so this is
    ...
```

Parallel for loops and other constructs that don't guarantee that each "thread" has its own non-shared mutable memory are very important and not covered by this model at all. For example, having multiple threads execute on different slices of the same array would require copying the array to temporary disjoint memory spaces to do operations, then recopy it back into place. This data copying can be awkward and reduce the benefits of parallelism to make it non-profitable.

There are multiple different ways to tackle this. We can just throw it back into the programmer's lap and tell them that the behavior is undefined if they get a race condition. This is fine for some systems levels stuff, but defeats the purpose of having a safe language and is clearly not good enough for mobile code.

Another (more aggressive) approach is to provide a parallel for loop, and use it as a hint that each iteration can be executed in parallel. It would then be up to the implementation to try to prove the safety of this (e.g. using dependence analysis), and if provable everything is good. If not provable, then the implementation would have to compile it as serial code, or use something like an STM approach to guarantee that the program is correct or the error is detected. There is much work in academia that can be tapped for this sort of thing. One nice thing about this approach is that you'd always get full parallel performance if you "disable checking", which could be done in a production build or something.

Some blue sky kinds of random thoughts

Distributed Programming - Since deep copy is part of the language and "deep copy" is so similar to "serialization", it would be easy to do a simple implementation of something like "Distributed Objects". The primary additional thing that is required is for messages sent to actors to be able to fail, which is required anyway. The granularity issues that come up are similar in these two domains.

Immutable Data w/Synch and Lazy Faulting - Not a fully baked idea, but if you're heavily using immutable data to avoid copies, a "distributed objects" implementation would suffer because it would have to deep copy all the immutable data that the receiver doesn't have, defeating the optimization. One approach to handling this is to treat this as a data synch problem, and have the client fault pieces of the immutable data subgraph in on demand, instead of eagerly copying it.

OpenCL Integration with this model could be really natural: the GPU is an inherently async device to talk to.

UNIX processes: Actors in a shared address space with no shared mutable data are related to processes in a unix app that share by communicating with mmap etc.