orphan: *Note*: This document is **not an accepted Swift proposal**. It does not describe Swift's concurrency mechanisms as they currently exist, nor is it a roadmap for the addition of concurrency mechanisms in future versions of Swift.

Swift Thread Safety

This document describes the Swift thread-safety layer. It includes the motivation for allowing users to write thread-safe code and a concrete proposal for changes in the language and the standard library. This is a proposal and not a plan of record.

The low-level thread-safety layer allows third-party developers to build different kinds of high-level safe concurrency solutions (such as Actors, Coroutines, and Async-Await) in a library. This document describes three different high-level concurrency solutions to demonstrate the completeness and efficacy of the thread-safe layer. Designing an official high-level concurrency model for Swift is outside the scope of this proposal.

Motivation and Requirements

Multi-core processors are ubiquitous and most modern programming languages, such as Go, Rust, Java, C#, D, Erlang, and C++, have some kind of support for concurrent, parallel or multi-threaded programming. Swift is a safe programming language that protects the user from bugs such as integer overflow and memory corruption by eliminating undefined behavior and by verifying some aspects of the program's correctness at runtime. Multi-threaded programs in Swift should not break the safety guarantees of the language.

Swift Memory Model

This section describes the guarantees that unsupervised Swift provides when writing multi-threaded code and why the minimal guarantees of atomicity that Java provide cannot be implemented in Swift.

Let's start by looking at reference counting operations. Swift objects include a reference-count field that holds the number of references that point to that object. This field is modified using atomic operations. Atomic operations ensure that the reference count field is not corrupted by data races. However, using atomic operations is not enough to ensure thread safety. Consider the multi-threaded program below.

```
import Foundation

let queue = DispatchQueue.global(qos: .default)

class Bird {}
var single = Bird()

queue.async {
  while true { single = Bird() }
}
while true { single = Bird() }
```

This program crashes very quickly when it tries to deallocate an already deallocated class instance. To understand the bug try to imagine two threads executing the SIL code below in lockstep. After they both load the same value they both try to release the object. One thread succeeds and deallocates the object while another thread attempts to read the memory of a deallocated object:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\proposals\[swift-main][docs][proposals]Concurrency.rst, line 68)

Cannot analyze code. No Pygments lexer found for "sil".

```
.. code-block:: sil

  %10 = global_addr @singleton : $*Bird

  bb:
    %49 = alloc_ref $Bird
    %51 = load %10 : $*Bird
    store %49 to %10 : $*Bird
    strong_release %51 : $Bird
    br bb
```

Next, we'll look into the problem of sliced values. Intuitively, it is easy to see why sharing memory between two threads could lead to catastrophic bugs. Consider the program below:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\proposals\[swift-main][docs][proposals]Concurrency.rst, line 83)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  Thread #1:              Thread #2:
```

```
A.first = "John"        A.first = "Paul"
A.last  = "Lennon"      A.last  = "McCartney"
```

If thread #1 goes to sleep after executing the first statement and resumes execution after thread #2 runs, then the value of the struct would be "Paul Lennon", which is not a valid value. The example above demonstrates how data races can introduce an inconsistent state.

The Java memory model ensures that pointers and primitive types, such as ints and floats, are never sliced, even when data races occur. It would be nice if Swift had a similar guarantee. Intuitively we would want all struct or class members of primitive types to be aligned to ensure atomic access to the field. However, this is not possible in Swift. Consider the Swift code below:

```
enum Fruit {
  case Apple(Int64),
  case Grape(MyClass)
}
```

The size of the struct above is 65 bits, which means that even on 64bit processors the tag of the enum can't be updated at the same time as the payload. In some race conditions we could accidentally interpret the payload of the struct as pointer using the value stored into the integer.

To summarize, Swift, just like C++, does not make any guarantees about unsynchronized memory, and the semantic of programs with races is undefined. When race conditions occur pointers and primitive data types could be sliced, enums may contain the wrong tag, protocols may refer to invalid dispatch tables, references may point to deallocated objects.

Achieving thread safety

This section describes a set of rules that ensure thread safety in programs that embrace them despite the inherit lack of thread safety in general multi-threaded Swift code.

Safe concurrency is commonly implemented by eliminating shared mutable memory. Go, Erlang and Rust ensure some level of program safety by providing mechanisms for eliminating shared mutable memory. Erlang provides the strongest model by ensuring complete logical address space separation between threads. Rust provides powerful abstraction and rely on the type system to ensure that objects are owned by a single entity. Go provides channels that allow threads to communicate instead of sharing memory (but allows user to pass pointers in channels!). It is not necessary to disallow all sharing of mutable data between threads and it is not necessary to enforce a hermetic separation between the address spaces. It is very useful to be able to share large data structures without copying them around. Mutable data can be shared between threads as long as the access to the data is synchronized and some program properties are verified by the compiler. In Swift thread safety is implemented by preventing threads from sharing mutable memory.

Proposal

In Swift, new threads are created in a new memory enclave that is separate from the parent thread. Values can be copied in and out of the new thread context, but the child thread must never obtain a reference that points to the outside world. Non-reentrant code needs to be explicitly marked as such. Swift enforces these rules statically. The rest of this section describes how Swift ensures safety and deals with global variables and unsafe code.

The three basic elements of thread safety

The Swift language has three features that allow it to ensure thread safety and enforce it at compile time:

1. Copyable Protocol

The **Copyable protocol** marks types of instances that can be copied from one thread context to another.

Instances of some types, such as Int, can be copied safely between threads because they do not contain references that allow threads to access memory that they do not own. Some types, such as String and Array (with copyable elements) can be copied between thread context because they have value semantics and the internal reference is not exposed.

The compiler derives the conformance of POD types and trivial enums to the Copyable protocol automatically. Library designers need to manually mark types with value semantics as Copyable.

Value-semantic types are not the only category of types that can be copied. Library designers can implement thread-safe or lockless data structures and manually mark them as Copyable:

```
// Optionals are copyable if the payload type is copyable.
extension Optional : Copyable where T : Copyable  {}
```

2. Reentrant code

We ensure thread-safety by requiring that code that's executed from a worker thread to only access logical copies of data that belongs to other threads. One way for user code to break away from the memory enclave is to access **global variables**. The Swift compiler must verify that threaded code does not access global variables or unsafe code that it can't verify. There are exceptions to this rule and the compiler provides special annotations for code that performs I/O or calls unsafe code.

Reentrant code is code that only accesses memory that is accessible from the passed arguments. In other words, reentrant code does not access global variables or shared resources.

The thread verifier needs to be able to analyze all of the code that could potentially be executed by a work thread and ensure that it is reentrant. Dynamically dispatched calls, file and module boundary limit the efficacy of the thread-verifier. This means that the information of whether a function is reentrant or not needs to be a part of the **function signature**.

The **unsafe** attribute is used to denote code that is allowed to access global variables and unsafe code. Objective-C methods are automatically marked as **'unsafe'** unless they are explicitly marked with the **safe** attribute. The *safe* and *unsafe* attributes provide a migration path for large bodies of code that do not explicitly mark the APIs as reentrant or non-reentrant.

In the example program below the method *fly* may access the global variable because it is marked with the attribute *unsafe*. The compiler won't allow this method to be executed from a worker-thread.

```
var glob : Int = 1

class Bird {
  unsafe func fly() { glob = 1}
}
```

In the example program below the *issafe* wrapper is used to explicitly mark a region as safe. The developer is pacifying the compiler and explicitly marking the code as safe.

The function `logger` is still considered by the compiler as reentrant and can be called by worker-threads.

```
func logger(_ x : Int) {

  // I know what I'm doing!
  issafe {
    glob = x
  }
}
```

Most protocols in the standard library, like *Incrementable* and *Equatable* are annotated as safe by default.

3. Gateways annotation

Gateway annotation is a special semantics annotation that marks functions that create new threads. This allows the compiler to verify that all of the arguments that are passed to the thread conform to the Copyable protocol and that the code that is executed by the worker thread is reentrant.

The compiler also verifies a few requirements that are special to the thread creation site, like making sure that the closure to be executed does not capture local mutable variables.

Library developers who implement high-level concurrency libraries can use the gateway annotation to mark the functions that launch new threads.

```
@ semantics("swift.concurrent.launch")
public func createTask<ArgsTy>(args : ArgsTy, callback : (ArgsTy) -> Void) {
  ...
}
```

Summary

Together, the thread verifier, the Copyable protocol, and the gateway annotation allow us to implement the thread-safety layer. The rest of this document demonstrates how these features are used for the implementation of high-level concurrency systems.

The implementations of the thread-safety layer, the thread verifier, and programs that use the three concurrency libraries are available in the `concurrency` git branch.

Implementing safe Go-lang style concurrency

In this section, we describe how the proposed thread-safety layer can be used for implementing go-lang style concurrency. Go supports concurrency using coroutines and channels. We are going to demonstrate how to implement go-style concurrency using verified code, Copyable protocol and gateway annotations.

Let's start by implementing Streams, which are analogous to go channels. A stream is simply a blocking queue with restrictions on the types that can be passed. Streams are generic data structures where the queue element type is `Copyable` (and conforms to the relevant protocol, discussed above). Streams are the only legitimate channel of communication between threads.

Streams can be shared by multiple tasks. These tasks can read from and write into the stream concurrently. Reads from streams that contain no data and writes into full streams will be blocked, meaning that the operating system will put the calling thread to sleep and wait for new data to arrive to wake the sleeping thread. This property allows the Stream to be used as a synchronization mechanism.

The second half of the go concurrency feature is coroutines. In Swift lingo, we'll call them Tasks. Tasks are functions that are executed by threads asynchronously. Tasks could have their own stack (this is an implementation detail that is not important at this point) and can run indefinitely. Tasks are created using gateways (see above) that ensure thread safety.

Together tasks and streams create a thread-safe concurrency construct. Let's delve into this claim. Tasks are created using gateways that ensure that all arguments being passed into the closure that will be executed are Copyable. In other words, all of the arguments are either deep-copied or implemented in a way that will forbid sharing of memory. The gateway also ensures that the closure that will be executed by the task is verified, which means that it will not access global variables or unsafe code, and it will not capture any variable that is accessible by the code that is creating the task. This ensures a hermetic separation between the newly created thread and the parent thread. Tasks can communicate using streams that ensure that information that passes between threads, just like the task's closure arguments, does not leak references and keeps the hermetic separation between the tasks. Notice that Streams themselves are Copyable because they can be copied freely between tasks without violating thread safety.

Stream and Tasks provide safety and allow users to develop server-like tasks easily. Reading requests from a queue, processing the request and writing it into another queue are easy, especially since the queues themselves provide the synchronization mechanisms. Deadlocks manifest themselves as read requests from an empty queue, which makes debugging and reasoning about these bugs trivial.

Usage Example

This is an example of a tiny concurrent program that uses Tasks and Streams.

```
let input  = Stream<String>()
let output = Stream<String>()

func echoServer(_ inp : Stream<String>,
                out : Stream<String>) {
  while true { out.push(inp.pop()) }
}

createTask((input, output), callback: echoServer)

for val in ["hello","world"] {
  input.push(val)
  print(output.pop())
}
```

The program above creates a server task that accepts an input stream and an output stream that allows it to communicate with the main thread. The compiler verifies that the task does not access any disallowed memory locations (as described below).

It is entirely possible to remove the manual declaration of the streams and the argument types and define a single endpoint for communication with the new task. In the example below the type declaration of the endpoint helps the type checker to deduct the type of the stream arguments and allows the developer to omit the declaration of the streams in the closure.

```
let comm : _Endpoint<String, Int> = createTask {
  var counter = 0
  while true {
    $0.pop()
    $0.push(counter)
    counter += 1
  }
}

// CHECK: 0, 1, 2,
for ss in ["","",""] {
  comm.push(ss)
  print("\(comm.pop()), ", terminator: "")
}
```

Stream utilities

The Swift library can implement a few utilities that will allow users and library designers to build cool things:

- The `Funnel` class accepts multiple incoming streams and weaves them into a single outgoing stream.
- The `Fan-out` class accepts a single incoming stream and duplicates the messages into multiple outgoing streams.
- The `waitForStream` function accepts multiple Streams and returns only when one or more of the streams are ready to be read.

It is entirely possible to implement MPI-like programs that broadcast messages or send messages to a specific task. It is also very easy to implement barriers for SPMD-like programs using fan-out stream.

Implementing Async - Await

Async-Await is one of the most popular and effective concurrency solutions. In this section we describe how the proposed thread-safety layer can be used for implementing Async-Await style concurrency.

Async calls are function calls that return a Future, which is a mechanism that allows the caller of asynchronous procedures to wait for the results. The async call execute the callback closure in a secure enclave to ensure thread safety.

Example

Example of a concurrent program using Futures in Swift.

```
func mergeSort<T : Comparable>(array: ArraySlice<T>) -> [T] {

  if array.count <= 16  { return Array(array).sorted() }

  let mid = array.count / 2
  let left  = array[0..<mid]
  let right = array[mid..<array.count]

  let lf = async(left,  callback: mergeSort)
  let lr = async(right, callback: mergeSort)

  return merge(lf.await(), lr.await())
}
```

The program above uses async to execute two tasks that sorts the two halves of the array in parallel. Notice that the arrays in the example above are not copied when they are sent to and from the async task. Swift arrays are copy-on-write value types and when an array is copied the underlying storage is not copied with it. This feature of arrays allows swift to share arrays between threads in a safe manner without copying data.

Here is another example of async calls using trailing closures and enums.

```
enum Shape {
  case circle, oval, square, triangle
}

let res = async(Shape.oval) { (c: Shape) -> String in
  switch c {
    case .circle:   return "Circle"
    case .oval:     return "Oval"
    case .square:   return "Square"
    case .triangle: return "Triangle"
  }
}

//CHECK: Shape: Oval
print("Shape: \(res.await())")
```

Notice that the swift compiler infers that `Shape` and *String* can be sent between the threads.

UI programming with Async

One of the goals of this proposal is to allow users to develop multi-threaded UI applications that are safe.

At the moment Swift users that use GCD are advised to start a new block in a new thread. Once the task finishes the recommendation is to schedule another block that will be executed by the main event loop.

Notice that the Async call returns a Future, and the callee needs to block on the result of the Future. In this section we describe the extension to the Async call that allows it to execute code on the main event loop asynchronously.

One possible solution would be to add an async call that accepts two closures. One that's executed asynchronously, and another one that will be executed synchronously after the task is finished. F# provides a similar API (with StartWithContinuations).

One possible implementation is one where the task creation call return an object that allows the users to register callbacks of different kinds. The destructor of the task object would execute the work callback for convenience. The two useful callbacks are "on completion" that would execute code in the main UI thread and "on error" that would be executed in case of an exception in the work closure.

This is a small example from an app that counts the number of prime numbers between one and million concurrently. The first closure is the worker closure that does all the work in a separate thread (and is verified by the thread safety checker), and the second closure is executed by the UI main loop and is free to make unsafe calls capture locals and access globals.

```
@IBAction func onClick(_ sender: AnyObject) {

  progress.startAnimating()
  label!.text = ""

  asyncWith (1_000_000) { (num: Int) -> Int in
    var sum = 0
    for i in 1..<num {
      if isPrime(i) { sum += 1 }
    }
    return sum
  }.setOnComplete { (x: Int) in
    self.label!.text = "Found \(x) primes.\n"
    self.progress.stopAnimating()
  }

}
```

Unsafe Concurrency with unsafeAsync

In many cases iOS users would need to use unsafe code such as code written in Objective-C, or code that has access to shared mutable state. In the previous section we mentioned that it is possible to mark some functions with a special annotation that will signal to the verifier to stop the verification. For example, the `print` function call would have to be marked with such an annotation if we want people to be able to use it from thread-safe code. This feature is useful for library developers, but not for app developers.

Some people may wish to skip the safety checks that the compiler provides and write unsafe asynchronous code. The `unsafeAsync` can allow users to run asynchronous code using Futures and async calls but without the safety checks.

The *async* call is actually a wrapper around unsafeAsync, except that it contains the annotation that tells the verifier to verify that the code is thread-safe (explained in the previous section). For example:

```
@_semantics("swift.concurrent.async")
// This annotation tells the compiler to verify the closure and the passed arguments at the call site.
public func async<RetTy, ArgsTy>(args: ArgsTy, callback: @escaping (ArgsTy) -> RetTy) -> Future<RetTy>
  return unsafeAsync(args, callback: callback)
}
```

Example of shared data structures

In the example below the class PrimesCache is explicitly marked by the user as a Copyable. The user implemented a thread-safe class that allows concurrent access to the method `isPrime`. To implement a critical section the user inherit the class `Sync` that contains a lock and a method that implements a critical section. The user also had to annotate the shared method as safe because the verifier has no way of knowing if the call is safe. Notice that the critical section itself is not enough to ensure thread safety because the critical section could be accessing memory that is shared between threads that are not synchronized on the same lock.

```
final class PrimesCache : Sync, Copyable {
  var cache: [Int : Bool] = [:]

  @_semantics("swift.concurrent.safe")
  func isPrime(_ num: Int) -> Bool {
    return self.critical {
      if let r = self.cache[num] { return r }
      let b = calcIsPrime(num)
      self.cache[num] = b
      return b
    }
  }
}

func countPrimes(_ p: PrimesCache) -> Int {
  var sum = 0
  for i in 2..<10_000 where p.isPrime(i) { sum += 1 }
  return sum
}

let shared = PrimesCache()
let r1 = async(shared, callback: countPrimes)
let r2 = async(shared, callback: countPrimes)

// CHECK: [1229, 1229]
print([r1.await(), r2.await()])
```

Example of parallel matrix multiply using Async

This is a small example of the parallel matrix multiplication algorithm using async and futures. The slices of the matrix are not copied when they are moved between the threads because ContiguousArray has value semantics and the parallel code runs significantly faster.

```
func ParallelMatMul(_ a: Matrix, _ b: Matrix) -> Matrix {
  assert(a.size == b.size, "size mismatch!")

  // Handle small matrices using the serial algorithm.
  if a.size < 65 { return SerialMatMul(a, b) }

  var product = Matrix(a.size)
  // Extract 4 quarters from matrices a and b.
  let half = a.size/2
  let a11 = a.slice(half, 0,    0)
  let a12 = a.slice(half, 0,    half)
  let a21 = a.slice(half, half, 0)
  let a22 = a.slice(half, half, half)
  let b11 = b.slice(half, 0,    0)
  let b12 = b.slice(half, 0,    half)
  let b21 = b.slice(half, half, 0)
  let b22 = b.slice(half, half, half)

  // Multiply each of the sub blocks.
```

```
  let c11_1 = async((a11, b11), callback: ParallelMatMul)
  let c11_2 = async((a12, b21), callback: ParallelMatMul)
  let c12_1 = async((a11, b12), callback: ParallelMatMul)
  let c12_2 = async((a12, b22), callback: ParallelMatMul)
  let c21_1 = async((a21, b11), callback: ParallelMatMul)
  let c21_2 = async((a22, b21), callback: ParallelMatMul)
  let c22_1 = async((a21, b12), callback: ParallelMatMul)
  let c22_2 = async((a22, b22), callback: ParallelMatMul)

  // Add the matching blocks.
  let c11 = c11_1.await() + c11_2.await()
  let c12 = c12_1.await() + c12_2.await()
  let c21 = c21_1.await() + c21_2.await()
  let c22 = c22_1.await() + c22_2.await()

  // Save the matrix slices into the correct locations.
  product.update(c11, 0,    0)
  product.update(c12, 0,    half)
  product.update(c21, half, 0)
  product.update(c22, half, half)
  return product
}
```

Implementing Actors

In this section we describe how the proposed thread-safety layer can be used for implementing Actor-based concurrency.

Actors communicate using asynchronous messages that don't block. Systems that use actors can scale to support millions of concurrent actors because actors are not backed by a live thread or by a stack.

In Swift actors could be implemented using classes that inherit from the generic `Actor` class. The generic parameter determines the type of messages that the actor can accept. The message type needs to be of `Copyable` to ensure the safety of the model. The actor class exposes two methods: `send` and `accept`. Messages are sent to actors using the `send` method and they never block the sender. Actors process the message using the `accept` method.

At this point it should be obvious to the reader of the document why marking the `accept` method as thread safe and allowing the parameter type to be `Copyable` will ensure the safety of the system (this is discussed at length in the previous sections).

The `accept` method is executed by a user-space scheduler and not by live thread and this allows the system to scale to tens of thousands of active actors.

The code below depicts the famous prime numbers sieve program using actors. The sieve is made of a long chain of actors that pass messages to one another. Finally, a collector actor saves all of the messages into an array.

```
// Simply collect incoming numbers.
class Collector : Actor<Int> {

  var numbers = ContiguousArray<Int>()

  override func accept(_ x: Int) { numbers.append(x) }
}

// Filter numbers that are divisible by an argument.
class Sieve : Actor<Int> {
  var div: Int
  var next: Actor<Int>

  init(div d: Int, next n: Actor<Int>) {
    div = d
    next = n
  }

  override func accept(_ x: Int) {
    if x != div && x % div == 0 { return }
    next.send(x)
  }
}

var col = Collector()
var head: Actor<Int> = col

// Construct the Sieve
for i in 2..<limit { head = Sieve(div: i, next: head) }

// Send all of the integers
for i in 2..<(limit * limit) { head.send(i) }

// CHECK: [1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
print(col.numbers.sort())
```