5 Concurrency Problems

Written by Scott Grosch

Unfortunately, for all the benefits provided by dispatch queues, they're not a panacea for all performance issues. There are three well-known problems that you can run into when implementing concurrency in your app if you're not careful:

- Race conditions
- Deadlock
- Priority inversion

Race conditions

Threads that share the same process, which also includes your app itself, share the same address space. What this means is that each thread is trying to read and write to the same shared resource. If you aren't careful, you can run into **race conditions** in which multiple threads are trying to write to the same variable at the same time.

Consider the example where you have two threads executing, and they're both trying to update your object's `count` variable. Reads and writes are separate tasks that the computer cannot execute as a single operation. Computers work on **clock cycles** in which each tick of the clock allows a single operation to execute.

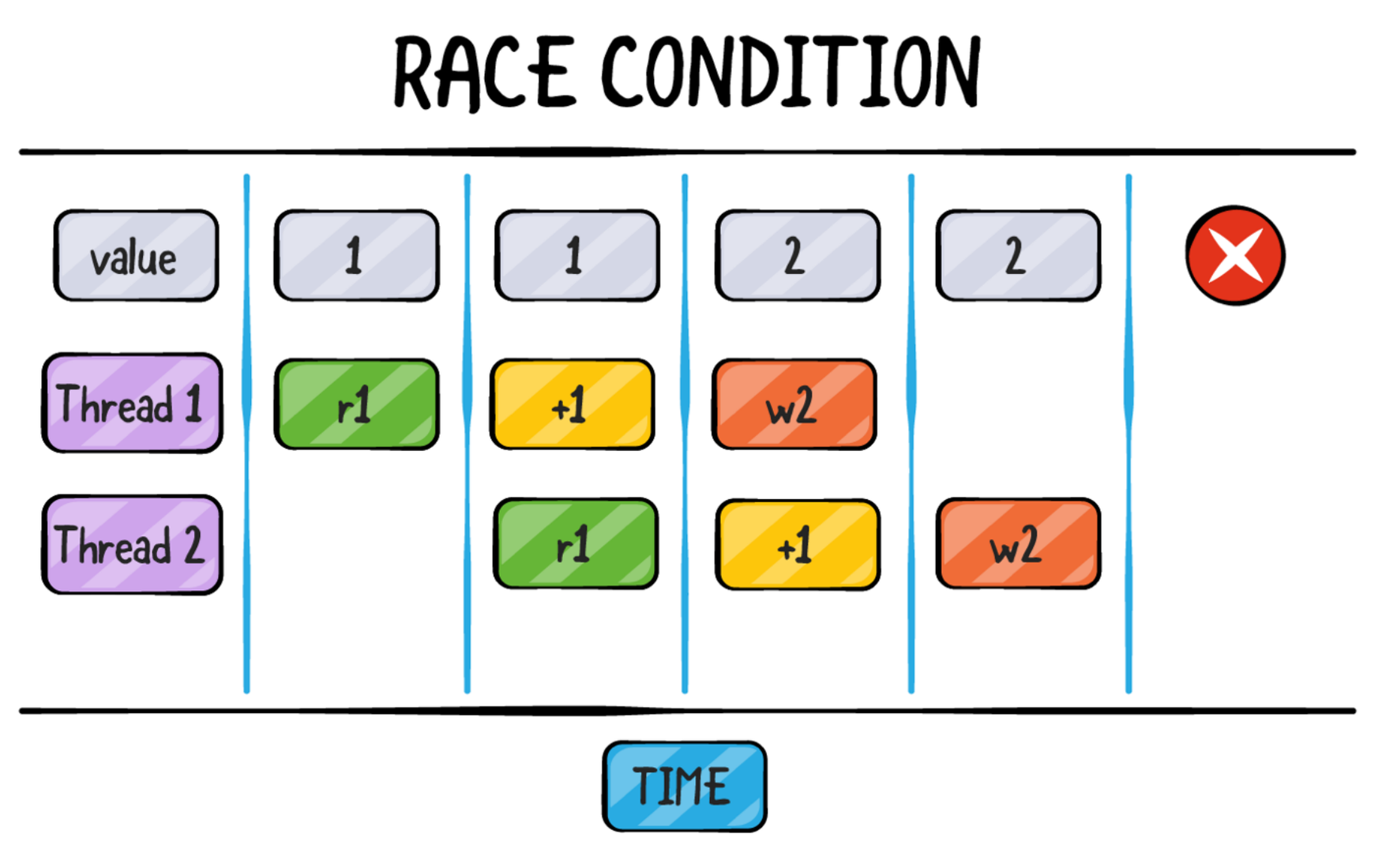
> **Note**: Do not confuse a computer's clock cycle with the clock on your watch. An iPhone XS has a 2.49 GHz processor, meaning it can perform 2,490,000,000 clock cycles per second!

Thread 1 and thread 2 both want to update the count, and so you write some nice clean code like so:

```
count += 1
```

Seems pretty innocuous, right? Break that statement down into its component parts, add a bit of hand-waving, and what you end up with is something like this:

1. Load value of variable `count` into memory.
2. Increment value of `count` by one in memory.
3. Write newly updated `count` back to disk.



The above graphic shows:

- Thread 1 kicked off a clock cycle before thread 2 and read the value `1` from `count`.
- On the second clock cycle, thread 1 updates the *in-memory* value to `2` and thread 2 reads the value `1` from `count`.
- On the third clock cycle, thread 1 now writes the value `2` back to the `count` variable. However, thread 2 is just now updating the *in-memory* value from `1` to `2`.
- On the fourth clock cycle, thread 2 now also writes the value `2` to `count`... except you expected to see the value `3` because two separate threads both updated the value.

This type of race condition leads to incredibly complicated debugging due to the non-deterministic nature of these scenarios.

If thread 1 had started just two clock cycles earlier you'd have the value `3` as expected, but don't forget how many of these clock cycles happen per second.

You might run the program 20 times and get the correct result, then deploy it and start getting bug reports.

You can usually solve race conditions with a serial queue, as long as you know they are happening. If your program has a variable that needs to be accessed concurrently, you can wrap the reads and writes with a private queue, like this:

```
private let threadSafeCountQueue = DispatchQueue(label: "...")
private var _count = 0
public var count: Int {
  get {
    return threadSafeCountQueue.sync {
      _count
    }
  }
  set {
    threadSafeCountQueue.sync {
      _count = newValue
    }
  }
}
```

Because you've not stated otherwise, the `threadSafeCountQueue` is a serial queue.

Remember, that means that only a single operation can start at a time. You're thus controlling the access to the variable and ensuring that only a single thread at a time can access the variable. If you're doing a simple read/write like the above, this is the best solution.

> **Note**: You can implement the same private queue sync for lazy variables, which might be run against multiple threads. If you don't, you could end up with two instances of the lazy variable initializer being run. Much like the variable assignment from before, the two threads could attempt to access the same lazy variable at nearly identical times. Once the second thread tries to access the lazy variable, it wasn't initialized *yet*, but it is about to be created by the access of the first thread. A classic race condition.

Thread barrier

Sometimes, your shared resource requires more complex logic in its getters and setters than a simple variable modification. You'll frequently see questions related to this online, and often they come with solutions related to locks and semaphores. Locking is very hard to implement properly. Instead, you can use Apple's **dispatch barrier** solution from GCD.

If you create a concurrent queue, you can process as many read type tasks as you want as they can all run at the same time.

When the variable needs to be written to, then you need to lock down the queue so that everything already submitted completes, but no new submissions are run until the update completes.

You implement a dispatch barrier approach this way:

```
private let threadSafeCountQueue = DispatchQueue(label: "...",
                                                 attributes: .concurrent)
private var _count = 0
public var count: Int {
  get {
    return threadSafeCountQueue.sync {
      return _count
    }
  }
  set {
    threadSafeCountQueue.async(flags: .barrier) { [unowned self] in
      self._count = newValue
    }
  }
}
```

Notice how you're now specifying that you want a concurrent queue and that the writes should be implemented with a barrier. The barrier task won't occur until all of the previous reads have completed.

Once the barrier hits, the queue pretends that it's serial and only the barrier task can run until completion. Once it completes, all tasks that were submitted after the barrier task can again run concurrently.

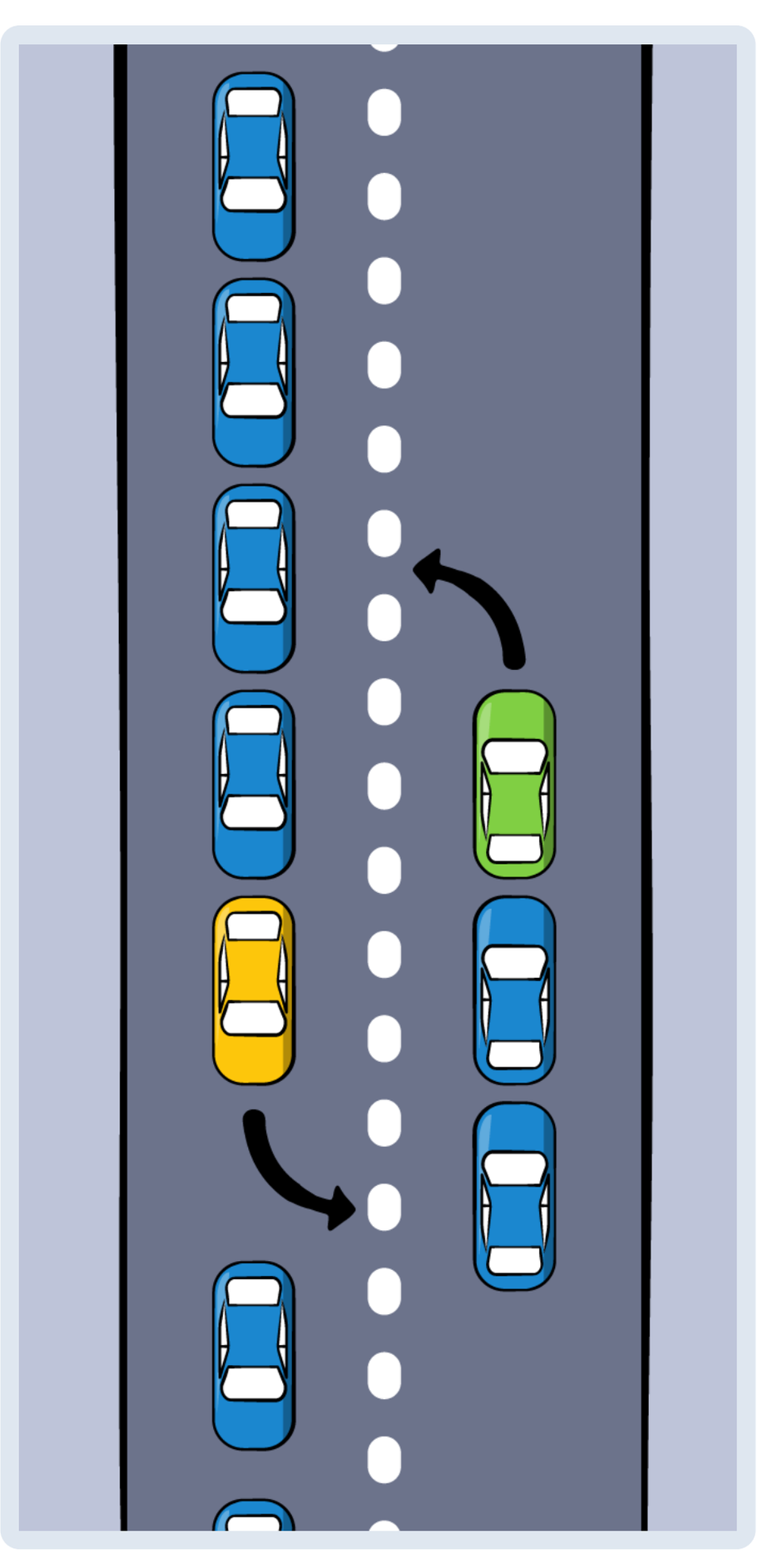
Deadlock

Imagine you're driving down a two-lane road on a bright sunny day and you arrive at your destination. Your destination is on the other side of the road, so you turn on the car's turn signal. You wait as tons of traffic drives in the other direction.

While waiting, five cars line up behind you. You then notice a car coming the other way with its turn signal on and it also stops a few cars past you, but it is now blocked by your backup.

Unfortunately, there are more cars behind the oncoming car and so the other lane backs up as well, blocking your ability to turn into your destination and clear the lane.

You've now reached **deadlock**, as you are both waiting on another task that can never complete. Neither of you can turn as cars are blocking the entrances to your destinations.



Deadlock is a pretty rare occurrence in Swift programming, unless you are using something like semaphores or other explicit locking mechanisms. Accidentally calling `sync` against the current dispatch queue is the most common occurrence of this that you'll run into.

If you're using semaphores to control access to multiple resources, be sure that you ask for resources in the same order. If thread 1 requests a hammer and then a saw, whereas thread 2 requests a saw and a hammer, you can deadlock. Thread 1 requests and receives a hammer at the same time thread 2 requests and receives a saw. Then thread 1 asks for a saw — without releasing the hammer — but thread 2 owns the resource so thread 1 must wait. Thread 2 asks for a saw, but thread 1 still owns the resource, so thread 2 must wait for the saw to become available. Both threads are now in deadlock as neither can progress until their requested resources are freed, which will never happen.

Priority inversion

Technically speaking, **priority inversion** occurs when a queue with a lower quality of service is given higher **system priority** than a queue with a higher **quality of service**, or **QoS**. If you've been playing around with submitting tasks to queues, you've probably noticed a constructor to `async`, which takes a `qos` parameter.

Back in Chapter 3, "Queues & Threads," it was mentioned that the QoS of a queue is able to change, based on the work submitted to it. Normally, when you submit work to a queue, it takes on the priority of the queue itself. If you find the need to, however, you can specify that a specific task should have higher or lower priority than normal.

If you're using a `.userInitiated` queue and a `.utility` queue, and you submit multiple tasks to the latter queue with a `.userInteractive` quality of service (having a higher priority than `.userInitiated`), you could end up in the situation in which the latter queue is assigned a higher priority by the operating system. Suddenly all the tasks in the queue, most of which are really of the `.utility` quality of service, will end up running *before* the tasks from the `.userInitiated` queue. This is simple to avoid: If you need a higher quality of service, use a different queue!

The more common situation wherein priority inversion occurs is when a higher quality of service queue shares a resource with a lower quality of service queue. When the lower queue gets a lock on the object, the higher queue now has to wait. Until the lock is released, the high-priority queue is effectively *stuck* doing nothing while low-priority tasks run.

To see priority inversion in practice, open up the playground called **PriorityInversion.playground** from the **starter** project folder in this chapter's project materials.

In the code, you'll see three threads with different QoS values, as well as a semaphore:

```
let high = DispatchQueue.global(qos: .userInteractive)
let medium = DispatchQueue.global(qos: .userInitiated)
let low = DispatchQueue.global(qos: .background)

let semaphore = DispatchSemaphore(value: 1)
```

Then, various tasks are started on all queues:

```
high.async {
    // Wait 2 seconds just to be sure all the other tasks have enqueued
    Thread.sleep(forTimeInterval: 2)
    semaphore.wait()
    defer { semaphore.signal() }

    print("High priority task is now running")
}

for i in 1 ... 10 {
    medium.async {
        let waitTime = Double(exactly: arc4random_uniform(7))!
        print("Running medium task \(i)")
        Thread.sleep(forTimeInterval: waitTime)
    }
}

low.async {
    semaphore.wait()
    defer { semaphore.signal() }

    print("Running long, lowest priority task")
    Thread.sleep(forTimeInterval: 5)
}
```

If you display the console (⇧ + ⌘ + Y) and then run the playground, you'll see a different order each time you run:

```
Running medium task 7
Running medium task 6
Running medium task 1
Running medium task 4
Running medium task 2
Running medium task 8
Running medium task 5
Running medium task 3
Running medium task 9
Running medium task 10
Running long, lowest priority task
High priority task is now running
```

The end result is always the same. The high-priority task is always run *after* the medium and low-priority tasks due to priority inversion.

Where to go from here?

Throughout this chapter, you explored some common ways in which concurrent code can go wrong. While deadlock and priority inversion are much less common on iOS than other platforms, race conditions are definitely a concern you should be ready for.

If you're still eager to learn more about these common concurrency issues, the internet is full of excellent and surprising examples. In Chapter 12, "Thread Sanitizer," you'll lean more about how to track down race conditions in threading issues.

Completed

6. Operations →

← 4. Groups & Semaphores