Avoiding race conditions in Swift

Thoughts about concurrency, race conditions, locks and GCD queues

Mostafa Abdellateef

Photo by marcos mayer on Unsplash

I recently had to debug an issue in a code written in Swift, a variable whose value is unexpectedly changing by different threads, and as

someone who has been in a similar situation before, but back then it was an Objective-C code, I used to follow the traditional approach to solve these kinds of problems, using locks to synchronize access to this shared resource. In Objective-C I used `@synchronized` directive to accomplish that task. So I started searching how to write `@synchronized` in Swift, but I couldn't find it.

There is still a way to create mutex locks without `@synchronized` in swift but while I was getting a little bit deeper into that topic, I found other approaches to handle these type of issues even without using locks at all. In this article we will explore some these approaches but let's first have a quick refresher about concurrency.

Concurrency 101

Concurrency means running more than one task at the same time. It is a major topic that you will find almost everywhere in modern software programming. As everything else in software, the concept of concurrency appears at different levels of abstractions, you can find it in distributed-computing architectures, database systems, CPU architectures, low-level and high-level software programs. Each of these abstractions has different challenges and different approaches for solving them but across all of these abstractions there is always one common challenge, ***managing access to shared resources***. In the rest of this article we will be addressing Concurrency from high-level programming perspective.

> once you encounter a shared resource in a concurrent software, you have to be careful while accessing this resource especially if you are writing to it.

At the heart of concurrency lies the concept of ***threads***, think of a thread as a path of execution, so in order for concurrency to be possible, we need to be able to execute multiple threads at the same time. For

instance, in the same program (Process) there may be a thread for playing audio, another thread for accepting user input, and a thread for downloading a file from a remote server. All this threads needs to execute in the same time seamlessly without any conflict.

> *Multi-Threading concurrency is possible even on hardware that is limited to a single-core processor, using techniques like time-slicing which keeps all running tasks in a tasks pool and give each one of them a specific amount of time to execute then switch to the next task even if the first one is still not finished, giving the illusion that these tasks are running simultaneously.*

Sounds great!! let's start creating a new thread for every concurrent task we need to execute. Unfortunately writing concurrent code is not that straight forward, but the good news is, if you used the right tools for the job, things will not be quite hard also.

Example: Building our own ATM

Imagine we had established our own bank — our bank had only one branch as a start. One day one of the founders came with this brilliant idea of deploying an ATM machine in our bank's single branch. Our ATM works with the following sequence:

1. First we need to check if the current balance has sufficient amount of money for the withdrawal.
2. If the balance is sufficient, start the withdrawal process, The withdrawal process needs to connect to our bank's servers to do some validations, a process that normally takes up to two seconds to finish.
3. After the validation process is completed, we will deduce the amount of withdrawn money from the deisgnated balance.

x

Everything is going as expected, our customers are happy and our bank had a massive breakthrough. Our ATM was handling more than 100 transactions per day without any issue so in order to be able to process more transactions in less time, we added another ATM. Everything went well for the first few days, until our accountant found something strange. A customer was able to withdraw an amount of money more than his total balance!!

After reviewing the logs we were be able to find the exact scenario of how this happened, The two ATMs withdrawn from the balance at the same exact time.

Let’s simulate the two ATMs in code to be able to understand more about how this issue did happen. Don’t worry about syntax for now, we will explain how queues work later. Just know that these two ATMs are working independently in parallel and started the withdrawal process almost at the same time.

The output of this code will be :

To understand more, let’s plot what happened in a timeline:

ATM1 changed the balance after AMT2 has already checked if it is sufficent

That’s it, The problem is that, ***when ATM2 checked the balance it was sufficient but at the time of balance deduction, the balance was***

already altered by ATM1 and hence it became insufficient.

There are three main issues that can happen in concurrent programs, **Race Conditions**, **Priority inversion**, and **Deadlocks**. A **race condition** happens when two or more threads access a shared data and change it's value at the same time. Our ATM is an example of how can race conditions affect the correctness of program results. **Priority inversion** happens when low priority tasks lock a resource needed by a higher priority task. This problem is hard to identify because it is probabilistic. **Deadlocks** happens when two threads are waiting for each other to release a shared resource, ending up blocked for infinity. We will explore deadlocks later in this article. let's rephrase a paragraph from Apple's docs descriping why concurrency issues are tricky.

> What's bad about concurrency issues is that they might not be obvious, if you are lucky the corrupted resource might cause obvious performance problems or crashes that are relatively easy to track down and fix. If you are unlucky, however, the corruption may cause subtle errors that do not manifest themselves until much later, or the errors might require a significant overhaul of your underlying coding assumptions.

In the rest of this article we will focus on how to improve our ATM to prevent such issues from happening.

Attempt 1: A good design is your best shot

When it comes to thread safety, a good design is the best protection you have. Avoiding shared resources and minimizing the interactions between your threads makes it less likely for those threads to interfere with each other. Let's see how can we prevent race conditions in our ATM with a simple tweak:

Double checking balance will fix our issue

Using the same exact code we used before to execute two withdrawal operations at the same time, the output of this code will be :

The trick here is double checking the balance, one time before validation to avoid the overhead if balance is insufficient, the other time is after the long validation process has completed. This was a trivial tweak just to prove our point, **careful design can protect you from race conditions**.

A lot of patterns and best practices exist to minimize the risk in concurrency code, including **avoiding shared global states**, favor **immutable over mutable values**, using **pure functions** to minimize side-effects and adopting strategies like **shared-nothing policy** and **Copy On Write (COW)**.

Using these best practices makes your code better in general and less vulnerable to concurrency issues, but in real-world scenarios a completely interference-free design is not always possible, however. Using other tools beside good design is inevitable.

Attempt 2: Using Locks

Using locks is the traditional way to prevent race conditions. The idea behind locks is that, you want to synchronize access to a part of your code, you create a lock (a token) that can be accessed by different threads, once a thread accesses that specific part of the code, it acquires that lock, and as long as the lock is acquired by a thread, other threads are blocked from accessing this code waiting for that lock to be released. **The first step to use locks is to identify the critical section in your code**. A critical section is the part of the code is considered to be thread safe if it is only accessed by a single thread at any point of time. In another words, this is the part of the code that accesses shared resources.

In our ATM, the critical section is the part doing the acutal withdrawal

operation. In order to use locks in our ATM we will use NSLock class.

the output of this code will be :

Note how this output differs from attempt 1, using locks ATM1 didn't start the withdrawal until ATM2 finished and released the lock.

Locks need to be used with Cautious, **they can cause deadlocks if you are not careful**. Assume we have two concurrent threads running, consider the following scenario:

1. While Thread1 is executing, it acquired a lock to access Function1(Lock1)
2. Another thread (Thread2) is executing in parallel, it acquired another lock synchronizing access to Function2 (Lock2)
3. Before releasing Lock1, Thread1 tries to access Function2, so it needs to wait for Lock2 to be released.
4. Before releasing Lock2, Thread2 tries to access Function1, so it needs to wait for Lock1 to be released.
5. Now Thread1 has Lock1 and waiting for Thread2 to release Lock2, but Thread2 will not release Lock2 until Lock1 is released first. Resulting in both threads waiting for each other forever. **A deadlock.**

Attempt 3: Using Serial Queues

Now that we understood locks can be dangerous, here comes another solution. We will use GCD serial queues to organize access to shared resources, As we did before using locks, we are restricting access to the critical section to only one thread at a time, the difference however is instead of locks we will use queues. Let's first understand how GCD queues work.

GCD Queues 101

Grand Central Dispatch (GCD) is Apple's framework for executing concurrent code, GCD abstracts concurrent code with Queues and Operations rather than letting you directly manage threads. The normal workflow is you dispatching blocks of code to queues and the framework is responsible of figuring out how many threads to create and how to manage the created threads lifecycle.

When using queues, you have to decide two things: the type of the queue, and how will you push (dispatch) the task to that queue. GCD queues can be either concurrent or serial and pushing tasks to the queues can be synchronous or asynchronous. That leaves us with for possibilities for tasks execution:

1. async dispatch to serial queue
2. async dispatch to concurrent queue
3. sync dispatch to serial queue
4. sync dispatch to concurrent queue.

Serial queue means tasks are executed one after the other in a FIFO (First In First Out) order. Serial queue needs only one thread to execute it's tasks because it **guarantees only one task is running at any point of time**.

Concurrent queues however creates as many threads as the tasks dispatched to it, however as we mentioned before, we don't have control over these threads and it totally depends on GCD to create them. The exact number of tasks executing at any given point is variable and depends on system conditions. In Concurrent queues tasks are still started in the order in which they were added to the queue.

Regardless of queue type, Async and sync differs only in the caller site. Async means non-blocking, the control returns immediately to thread which dispatched the block. On the other side, Sync means blocking, the dispatching thread loose control and waits till the dispatched block of code finished execution.

The following diagram illustrates the workflow of dispatching sync and async to serial queue.

Async and Sync dispatching to a Serial Queue

The left-hand side of the diagram illustrates sync and async dispatches and the right-hand side illustrates how serial queues work. **If we will to replace this diagram with a one for concurrent queues the LHS will remain exactly the same** while the RHS will change to reflect parallel execution of different blocks at the same time.

GCD provides 5 global concurrent queues as well as the main queue which is a serial queue for the main thread. You can also create your custom queues which can be serial or concurrent. Queus priority levels are abstracted to a concept called quality of service (QoS):

Now let’s see how can queues help us avoiding race conditions? Instead

of using locks we will use a serial queue to fulfill mutual exclusion by making critical section code accessible by only one thread at any point of time. As we did before, **we will first identify the critical section then all we have to do is just dispatch the critical section to a serial queue.**

And the output will be exact as the one using locks (This time first ATM executed before second ATM but this is totally random, if you re-run the code again ATM2 may be faster and execute first)

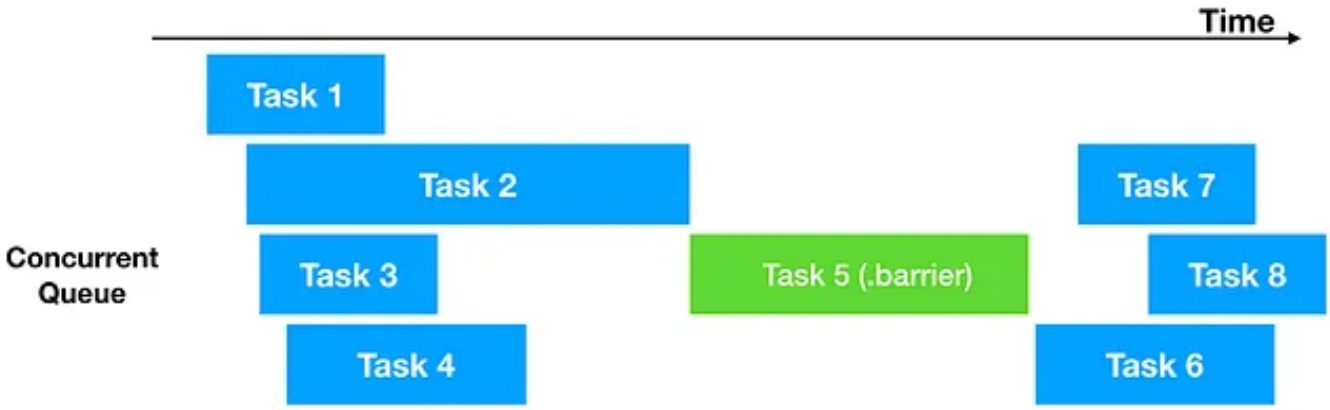
Replacing your lock-based code with queues eliminates many of the penalties associated with locks and also simplifies your remaining code. For example, It lowers the risk of deadlocks and Dispatching async can protect a shared resource without blocking the caller thread. Using queues makes code more readable, and easier to reason about.

> *Be careful that using queues doesn't totally prevent deadlocks too. For example, `sync` dispatching a task from a serial queue to the same serial queue, will immediately cause a deadlock because the caller thread is waiting for the dispatched block to execute but the dispatched block will never execute because the serial queue is blocked executing the caller code.*

Attempt 4: Using Dispatch Barrier with Concurrent Queues

What If the critical section was already dispatched to a concurrent queue, and you want to synchronize it's access to prevent race conditions. Do we have dispatch it again to a serial queue?

When dispatching a code block to a concurrent queue, you can assign a flag to it indicating that it is a barrier task, meaning that **when it is time to execute this task, it should be the only executing item on the specified queue.**



So all we will need to do is to dispatch the critical section code to a concurrent queue but this time with the `.barrier` flag

The output will be exactly as using locks and serial queues.

Wrapping up

At this article we talked quickly about concurrency and threads, introduced the three main challenges embedded into concurrency; race conditions, priority inversion, and deadlocks. Then we explored how good design can help you avoid race conditions, mentioned some patterns and strategies that might help. We came to a conclusion that although avoiding synchronization issues altogether is preferable, it is not always possible. So we started exploring other solutions including locks, serial queues and barrier tasks. **While these solutions differ in execution details, the idea remains the same, identify the critical section and make it accessible by only one thread at a time.**

Thank you for reaching the end of such a long article, If you found this article helpful please leave some claps and share it with your friends so they can benefit from it to. 👏👏