Exploring Threads in Swift

VINODH KUMAR · Follow

Published in Stackademic · 9 min read · Feb 20, 2024

Photo by Austin Distel on Unsplash

Concurrency, or doing multiple things at the same time, is an important part of programming. In the world of Apple software development, one key aspect of concurrency is threads. But what exactly are threads, and how do they work in Swift? Let's break it down in simpler terms with examples.

Why Threads Matter in Swift?

In 2007, something called "operation queues" and later in 2009, "Grand Central Dispatch (GCD)" made it easier to do multiple things at once in Apple programming. Before that, developers often used threads directly to handle these tasks.

What's a Thread Anyway?

Think of a thread as a small worker inside your computer that helps it do different tasks. Imagine you're playing a game while listening to music — each activity is like a thread working independently.

The `Thread` class in Swift acts as a layer over "POSIX threads" (pthreads), representing the parallel execution model utilized by iOS, macOS, and various other platforms. Although it's possible to create pthreads directly in Swift using C functions, it tends to be cumbersome. Apple offers the `Thread` class to streamline this process, making thread management more accessible and convenient for developers.

How to create a Thread in swift?

Start by importing the `Foundation framework` for thread management and then use `Thread.detachNewThread` and enclose your task or code block within the curly braces `{}`. For instance:

```
import Foundation

Thread.detachNewThread {
 // Your asynchronous code goes here
}
```

This method instantly spawns a new thread to execute the code enclosed within the `detachNewThread` closure asynchronously, separate from the main thread.

```
import Foundation
// Create a new thread
Thread.detachNewThread {
 // Simulate work by making the thread sleep for 1 second
 Thread.sleep(forTimeInterval: 1)
 // Print details of the current thread
 print(Thread.current)
}
// Pause the main thread for 1.1 seconds
Thread.sleep(forTimeInterval: 1.1)
// Print an exclamation mark in the main thread
print("!")
```

In this code, a new thread is created using `Thread.detachNewThread`. Inside the detached thread, there's a simulated task — sleeping for 1 second and then printing the details of the thread using `Thread.current`. However, because the detached thread doesn't pause the main thread, the main thread proceeds to the next line after starting the detached thread.

Running this code prints the details of the newly created thread and the exclamation mark in the main thread. The output showcases that the thread details printed come from the detached thread

`(NSThread: {number = 2, name = (null)})`,

while the exclamation mark is printed from the main thread

`(_NSMainThread: {number = 1, name = main})`.

The behavior of `Thread.current` changes based on the context it's accessed from: when used within `Thread.detachNewThread`, it displays information about that specific detached thread, while its behavior differs when called from the global file scope, reflecting details about the main thread. This behavior is peculiar but consistent across various concurrency models in Swift.

Threads and Their Order

Threads can be a bit unpredictable! If you create multiple threads, they might not start in the order you expect. For example:

```
Thread.detachNewThread { print("1", Thread.current) }
Thread.detachNewThread { print("2", Thread.current) }
Thread.detachNewThread { print("3", Thread.current) }
Thread.detachNewThread { print("4", Thread.current) }
Thread.detachNewThread { print("5", Thread.current) }
```

When you run this, the order of the numbers and thread details might surprise you.

```
1 <NSThread: 0x100710880>{number = 2, name = (null)}
3 <NSThread: 0x101231e90>{number = 3, name = (null)}
```

```
5 <NSThread: 0x10150d1d0>{number = 4, name = (null)}
2 <NSThread: 0x10112a9e0>{number = 5, name = (null)}
4 <NSThread: 0x101606470>{number = 6, name = (null)}
```

If you run it again it would execute in different order.

```
1 <NSThread: 0x106214f20>{number = 2, name = (null)}
3 <NSThread: 0x1007a6910>{number = 3, name = (null)}
2 <NSThread: 0x1007a6cf0>{number = 4, name = (null)}
4 <NSThread: 0x1063040e0>{number = 6, name = (null)}
5 <NSThread: 0x1007a6e60>{number = 5, name = (null)}
```

Threads aren't guaranteed to start in the order they're created. The operating system's allocation of execution time to threads is complex, so assuming their execution order isn't reliable. If sequential thread execution is necessary, additional coordination logic is needed, a topic we'll explore further later.

In multithreaded programming, `Thread.detachNewThread` lets us create separate threads for concurrent work without blocking others. We can create numerous threads theoretically, relying on the operating system to manage their execution and pauses. Threads can interleave computations, allowing multiple to run in intervals, a fundamental aspect of multithreading.

Priority And Cancellation

The `Thread` class provides tools to manage threads. You can create a thread with specific work using an initializer, like so:

```
let thread = Thread {
  Thread.sleep(forTimeInterval: 1)
  print(Thread.current)
}
```

But remember, this code doesn't start the thread immediately. You have to explicitly kick off the thread's execution using `thread.start()`. This detachment lets the thread work independently from the main application. This is a lazy operation.

Once we've obtained the thread handle, we can adjust its priority using a scale from 0 to 1:

```
thread.threadPriority = 0.75
```

This value tells the operating system whether the thread should be considered low or high priority. It may influence the time allocated for this thread's execution compared to others, but it's essential to note that there are no absolute guarantees.

Using the thread handle, we can cancel the thread's execution at any point:

```
thread.start()
thread.cancel()
```

However, it's important to note that canceling doesn't instantly stop the thread. If canceled immediately after starting, the thread might not execute the provided closure. Yet, if you wait a short time after starting and then cancel:

```
thread.start()
Thread.sleep(forTimeInterval: 0.01)
thread.cancel()
```

You'll observe that the thread's closure still executes, highlighting that cancellation doesn't immediately halt the ongoing work.

Thread cancellation isn't about abruptly halting the thread's execution, as that could lead to various issues. Imagine if the thread opened a file or network connection, canceling midway could leave resources improperly managed. Hence, thread cancellation operates cooperatively.

By invoking `cancel()` on a thread, it toggles a boolean flag within the thread. You can utilize this flag to interrupt thread logic:

```
let thread = Thread {
  Thread.sleep(forTimeInterval: 1)
  guard !Thread.current.isCancelled else {
    print("Cancelled")
    return
  }
  print(Thread.current)
}
```

Using `isCancelled`, you can gracefully halt unnecessary work when cancellation is requested, as in complex tasks like web crawling, where you'd check between steps.

However, not all operations, like `Thread.sleep`, respond cooperatively to cancellation. For instance:

```
let thread = Thread {
  let start = Date()
  defer { print("Finished in", Date().timeIntervalSince(start)) }

  Thread.sleep(forTimeInterval: 1)
  guard !Thread.current.isCancelled else {
    print("Cancelled")
    return
  }
  print(Thread.current)
}
```

Here, `Thread.sleep` won't stop if cancelled, waiting for the full duration despite a cancellation request. This behavior isn't ideal, but it's likely to prevent excessive CPU polling. Overall, thread priority and cooperative cancellation offer glimpses of collaboration in managing thread resources effectively.

Thread Dictionaries

Threads offer a unique feature — the `threadDictionary`, a thread-isolated state accessible globally within a thread. This dictionary proves invaluable for passing data through a system without explicitly passing it to every function, method, or initializer.

In server-side applications handling numerous requests concurrently, threads are often employed for executing tasks. Consider this example:

```
func response(for request: URLRequest) -> HTTPURLResponse {
  Thread {
    // Perform tasks for the request
  }
}
```

However, handling logs and unique identifiers per request can become cumbersome. For instance, associating logs with request IDs throughout a request's lifecycle can be challenging if manually passed around.

The `threadDictionary` simplifies this process. Setting a value, like a `requestId`, within this dictionary when a request starts allows universal access across all code running in that thread, enhancing isolation for parallel threads handling separate requests:

```
let thread = Thread {
  thread.threadDictionary["requestId"] = UUID()
  thread.start()
}
```

This `requestId` is now readily accessible throughout the application for that specific thread:

```
let requestId = Thread.current.threadDictionary["requestId"] as! UUID
```

```
// Use requestId across various functions and methods
```

Even deeply nested functions or object constructions within a thread can access this global-like but thread-isolated `threadDictionary`. This feature proves particularly beneficial in logging, where logs are tagged with request IDs for easy debugging and comprehension.

Limitations:

No Thread Inheritance

When you create a new thread, it doesn't automatically inherit data from its parent thread. For example:

```
let thread = Thread {
  // Creating a new thread
  // Doesn't inherit cancellation state from the outer thread
  Thread.detachNewThread {
    print("Inner thread isCancelled", Thread.current.isCancelled)
  }
  // Outer thread's cancellation status won't affect the inner thread
  guard !Thread.current.isCancelled else {
    print("Cancelled")
    return
  }
}
```

Coordinating Threads

There's no easy way to make a thread wait for others to finish their tasks. Instead, you often end up using loops with pauses to repeatedly check if other threads have completed their work:

```
while !databaseQueryThread.isFinished || !networkRequestThread.isFinished {
  Thread.sleep(forTimeInterval: 0.1)
}
```

Child Thread Cancellation

When a parent thread is canceled, its child threads don't automatically stop. This lack of automatic cancellation in nested threads requires explicit handling and coordination between threads.

Resource intensiveness

For instance, creating a massive number of concurrent threads, as illustrated with a hypothetical web crawler loading numerous web pages, incurs significant overhead and competition among threads for CPU time.

```
let workCount = 1_000
for n in 0..<workCount {
 Thread.detachNewThread {
 print(n, Thread.current)
 // Simulating intense work to load and index a webpage
 while true {}
 }
}
Thread.sleep(forTimeInterval: 3)
```

Executing this code results in 1,000 threads created simultaneously, hogging resources and jostling for CPU time. Threads, by design, continuously run computations on CPU cores. Creating numerous threads for tasks that might involve waiting periods, like network requests or timers, leads to

inefficiencies. Threads, once initiated, continuously consume resources, even during idle times, making them unsuitable for non-blocking tasks.

Attempting to compute the 50,000th prime number illustrates the impact of excessive thread competition:

```
let workCount = 1_000 // Alteration to demonstrate competition's effect
Thread.detachNewThread {
 print("Starting prime thread")
 nthPrime(50_000)
}
```

The performance disparity between running this code with and without the concurrent work threads highlights how excessive thread creation slows down tasks:

> *Without too many threads: Takes about 0.025 seconds to find the 50,000th prime.*
>
> *With too many threads: Takes about 2.93 seconds, over 100x slower!*

Creating and managing thread pools attempts to mitigate these issues by limiting thread counts and sharing available threads among tasks. However, it's a local solution in a global context, and while it can alleviate some resource competition, it doesn't offer a comprehensive solution for cooperative concurrency. Without cooperative solutions provided by threading tools, managing threads becomes a nuanced challenge, impacting system performance and resource utilization.

Data Races

When multiple threads try to access and modify the same data simultaneously, issues arise. These are known as data races. Apple's tools for managing threads don't entirely solve this problem. Let's demonstrate this issue with a simple scenario.

```
class Counter {
 var count = 0
}
let counter = Counter()
for _ in 0..<1_000 {
 Thread.detachNewThread {
 Thread.sleep(forTimeInterval: 0.01)
 counter.count += 1
 }
}
```

Despite spinning up 1,000 threads, the count doesn't reach 1,000 reliably due to the nature of how threads work.

Understanding Data Race Scenarios

Imagine the process of incrementing the count as three steps: get the current count, increment it, and set it back. When multiple threads perform these steps concurrently, their actions might overlap unpredictably, resulting in incorrect counts.

Introducing Locks for Synchronization

To solve this, we introduce locks to synchronize access to the shared data. Using `NSLock`, we can ensure that only one thread at a time can modify the count.

```
class Counter {
 let lock = NSLock()
 var count = 0
func increment() {
 self.lock.lock()
 defer { self.lock.unlock() }
 self.count += 1
 }
}
```

Using the `increment()` method instead of directly accessing the `count`, we consistently achieve a count of 1,000. But, this method feels cumbersome.

Improving Lock Management

We could create a more versatile method that accepts closures, enabling safe modifications:

```
func modify(work: (Counter) -> Void) {
 self.lock.lock()
 defer { self.lock.unlock() }
 work(self)
}
```

Now, we can safely modify the count within a closure:

```
counter.modify {
 $0.count += 1
}
```

This approach ensures thread safety while simplifying the process.

Challenges with Property Synchronization
Trying to make direct property access thread-safe using Swift's `_read` and `_modify` features doesn't solve all problems. For complex mutations or interactions involving the property itself, direct synchronization becomes complex or impossible.

The challenge with locks is their disconnection from the concurrency tool, such as threads. An ideal locking mechanism should intimately understand how multiple units of work run concurrently to ensure synchronization. Swift's latest concurrency tools offer this, but there are other aspects we will cover those in the next part of this series.

Summary

Apple's Thread class was a fundamental tool for enabling concurrency on their platforms. It offered priority, cancellation, and thread dictionaries but had significant limitations:

- Lack of Thread Inheritance: Child threads don't inherit features like priority or cancellation from their parent threads, making coordination challenging.
- Easy Thread Proliferation: Threads could easily multiply, causing inefficiency and resource contention.
- Difficulty in Coordination: Coordinating tasks between threads was complex, requiring explicit synchronization.
- Divergent Code: The code structured for threaded operations starkly differed from non-threaded code.

- Basic Synchronization Tools: Tools for synchronizing between threads were basic and lacked sophistication.

In summary, while Apple's Thread class provided essential concurrency features, it also presented limitations in inheritance, coordination, code structure, and synchronization tools.

In this article, we've explored the basics of threading in Swift. Ready to dive deeper into concurrency? Check out the next part of our series where we explore Swift's Operation Queues: Exploring Swift's Operation Queues: Enhancing Threading with Limitations.

Series Navigation

- **Part 1**: Exploring Threads in Swift
- **Part 2**: Exploring Swift's Operation Queues: Enhancing Threading with Limitations
- **Part 3**: Mastering Dispatch Queues in Swift: Understanding, Implementation, and Limitations
- **Part 4**: Mastering Concurrency: Task
- **Part 5**: Swift Concurrency: Safeguarding Data with @Sendable and Actors

Stackademic 🎓

Thank you for reading until the end. Before you go:

- Please consider **clapping** and **following** the writer! 👏

- Follow us **X** | **LinkedIn** | **YouTube** | **Discord**
- Visit our other platforms: **In Plain English** | **CoFeed** | **Venture** | **Cubed**
- More content at **Stackademic.com**

Swift Swiftui IOS IOS App Development Swift Programming

Written by VINODH KUMAR

Follow

50 Followers · Writer for Stackademic

📱 Senior iOS Developer | Swift Enthusiast | Tech Blogger 🖥️

More from VINODH KUMAR and Stackademic

VINODH KUMAR

Swift Concurrency: Safeguarding Data with @Sendable and Actors

Before delving deeper, let's recall the issue of data synchronization and data races that we...

6 min read · Feb 20, 2024

100

Walid LARABI in Stackademic

Top 20 AI buzzwords in 2024

Explore AI's world! Understand 20 buzzwords easily with simple explanations and visuals.

12 min read · Feb 19, 2024

998 6

Josef Cruz in Stackademic

The Best Code I've Seen as a Professional Programmer

And what I noticed.

· 3 min read · Apr 15, 2024

957 12

VINODH KUMAR

Understanding Memory Allocation in Swift: Stack vs. Heap

Memory management is a fundamental aspect of programming, critical for efficient...

5 min read · Apr 17, 2024

28 2

See all from VINODH KUMAR

See all from Stackademic

Recommended from Medium

VINODH KUMAR

Mastering Concurrency: Task

Swift 5.5 brought in powerful concurrency tools that simplify and strengthen our...

6 min read · Feb 20, 2024

105

Itsuki in Stackademic

Swift: Create Network Layer

I am really overwhelmed with all the API calls in my iOS app, so in this article, I will share...

6 min read · May 3, 2024

16 3

Lists

Apple's Vision Pro
7 stories · 68 saves

Tech & Tools
16 stories · 237 saves

Icon Design
36 stories · 310 saves

Productivity
241 stories · 434 saves

Vladyslav Shkodych

Mastering SwiftUI: Are You Really as Good as You Think?

How deep do you think you know what SwiftUI is and how it works? Have you ever wondere...

16 min read · May 10, 2024

413 6

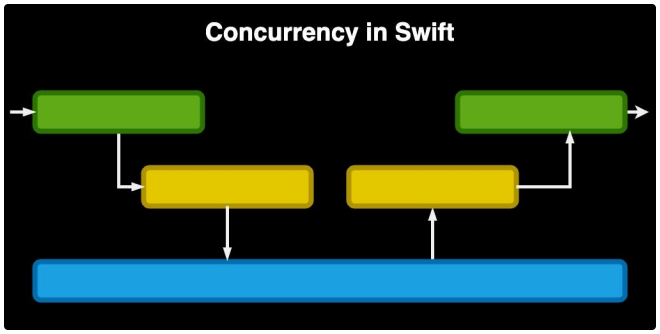
Harsh Vishwakarma

Using Swift Data and The Composable Architecture (TCA)

Hey, fellow SwiftUI enthusiasts! Today, I'm excited to share some hard-earned wisdom...

5 min read · Apr 16, 2024

65 2



Uladzislau Volchyk

Making things glow and shine with SwiftUI

Explore path animation and run a snake around arbitrary views using SwiftUI.

7 min read · Apr 14, 2024

337 4

Mahdi Saedi

Concurrency in Swift

Swift includes support for concurrency, letting you write code that can handle multiple...

7 min read · May 4, 2024

108

See more recommendations