АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ АКТОРНЫХ МЕХАНИЗМОВ И GCD В IOS С ИСПОЛЬЗОВАНИЕМ ЯЗЫКА ПРОГРАММИРОВАНИЯ SWIFT

Григорян Д.А.

Григорян Давид Арамович – старший разработчик мобильных приложений, компания Ozon, г. Москва

Аннотация: *в статье анализируется производительность доступа к объектам памяти с помощью механизмов акторной модели и библиотеки libdispatch в языке программирования Swift.*
Ключевые слова: *Swift, iOS, macOS, RAM, GCD, Actors, multithreading, barriers.*

В общепринятой модели операционной системы, в многопоточной среде, используется различное множество подходов для обеспечения целостности, корректности чтения и записи данных в оперативную память. Как правило, для обеспечения доступа к критической секции используются такие примитивы как **TSL**, **XCHG** и их аналоги. В операционной системе iOS для обеспечения подобного рода поведения, можно использовать как низкоуровневые механизмы POSIX стандарта, так и более высокоуровневые решения, которые включают в себя: библиотеку **libdispatch**, класс **NSLock**, а также, новый встроенный ссылочный тип - **actor**.

Рассмотрим следующий пример кода, представляющий из себя интерфейс корзины, который используется в многопоточной среде.

```
import Foundation

struct CartItem {
    let id: String
    let name: String
}

final class Cart {
    private var items = [String: CartItem]()

    func addItem(cartItem: CartItem) {
        items[cartItem.id] = cartItem
    }

    func getItem(by id: String) -> CartItem? {
        guard let item = items[id] else { return nil }
        return item
    }
}
```

Рис. 1. Интерфейс и реализация корзины

В данном случае очевидно, что программа завершится аварийно, если вызывать методы данного класса из разных потоков, так как коллекция, которая представляет из себя реализацию ассоциативного массива в iOS - не является потокобезопасной.

```
import Foundation

struct CartItem {
    let id: String
    let name: String
}

final class Cart {
    private var items = [String: CartItem]()

    func addItem(cartItem: CartItem) {
        items[cartItem.id] = cartItem                    Thread 4: EXC_BAD_ACCESS (code=1, address=0x10)
    }

    func getItem(by id: String) -> CartItem? {
        guard let item = items[id] else { return nil }
        return item
    }
}
```

Рис. 2. Аварийное завершение программы при обращении к методам корзины в многопоточной среде

Для предотвращения подобного рода патологий, можно использовать барьеры, как один из встроенных решений синхронного доступа к объектам памяти в iOS SDK. Барьеры, согласно классической POSIX схеме, не позволяют процессу перейти к следующей фазе, пока все процессы не будут готовы перейти к следующей фазе. Когда процесс достигает барьера, он блокируется до тех пор, пока этого барьера не достигнут все остальные процессы. Это позволяет синхронизировать группы процессов. Принцип работы барьера изображен на рис. 3.

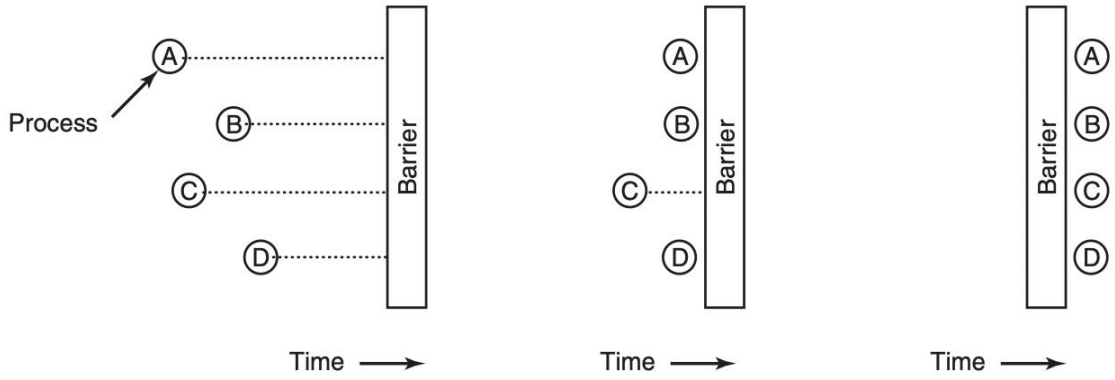

Рис. 3. Принцип работы барьера. Процесс достигает барьера, далее все процессы, кроме одного, заблокированы на барьере, и наконец последний процесс достигает барьера, и все процессы преодолевают этот барьер

Если воспользоваться данным подходом и обеспечить доступ к коллекции с помощью барьера, то можно тем самым обеспечить безопасный доступ к критической секции. Барьеры в iOS реализованы в стандартной библиотеке libdispatch, поставляемой iOS SDK. Libdispatch является надстройкой над низкоуровневыми POSIX потоками и предоставляет дополнительные возможности в плане управления очередями и их приоритетами. Используя барьеры в iOS, можно также добиться подхода блокировки чтения-записи (Single Writer Multiple Reader).

```
import Foundation

struct CartItem {
    let id: String
    let name: String
}

final class Cart {
    private var items = [String: CartItem]()
    private let syncQueue = DispatchQueue(label: "com.concurrent.cart.queue", attributes: .concurrent)

    func addItem(cartItem: CartItem) {
        syncQueue.async(flags: .barrier) { [weak self] in
            self?.items[cartItem.id] = cartItem
        }
    }

    func getItem(by id: String) -> CartItem? {
        return syncQueue.sync {
            guard let item = items[id] else { return nil }
            return item
        }
    }
}
```

Рис. 4. Реализация блокировки чтения-записи с помощью барьера

Данный подход обеспечивает безопасный доступ к критической секции в многопоточной среде и программа успешно и без аварийных ситуаций завершает свою работу.

Начиная с версии языка Swift 5.5, в iOS появилась возможность применять подобного рода механизмы с помощью акторов. Акторы автоматически сериализуют все вызовы к методам и свойствам, тем самым гарантируя, что только один вызывающий клиент, в заданный момент времени, может взаимодействовать с актором. Это в свою очередь, дает гарантию от возникновений data-race, так как все вызовы будут выполнятся последовательно, одна за другой. Для того чтобы реализовать подобный функционал достаточно изменить тип корзины с **class** на **actor**.

```swift
import Foundation

struct CartItem {
    let id: String
    let name: String
}


actor Cart {
    private var items = [String: CartItem]()

    func addItem(cartItem: CartItem) {
        items[cartItem.id] = cartItem
    }

    func getItem(by id: String) -> CartItem? {
        guard let item = items[id] else { return nil }
        return item
    }
}
```

Рис. 5. Корзина в виде актора

Для анализа производительности рассмотренных механизмов, существует готовый инструмент **signpost** от Apple, который позволяет фиксировать интересующийся промежуток времени или события.

С помощью специальной разметки **signpost** необходимо отметить те участки, которые будут выступать в качестве оценки итогового времени выполнения операций чтения и записи. Для этого необходимо создать объекта класса **OSLog** указав наименование подсистемы и категорию. Далее перед началом выполнения асинхронных операций вызывается функция **os_signpost** с типом **begin**, а по окончанию всех операций, функция с типом **end**.

```swift
let asyncWorkPost = OSLog(subsystem: "com.critical.section.test", category: "AsyncOperations")

os_signpost(.begin, log: asyncWorkPost, name: "Total time")
/*
 async work
 */
// await
os_signpost(.end, log: asyncWorkPost, name: "Total time")
```

*Рис. 6. Разметка и вызов функций **signpost***

В качестве тестовой среды используется цикл из 50 операций, внутри которого используются глобальные очереди. Внутри этих очередей происходит асинхронный вызов чтения и записи. Для получения итогового результата используется **dispatch_group**, внутри которого вызывается функция **os_signpost** с типом **end**. Далее, в инструментах среды разработки **Xcode** создается инструмент с типом **os_signpost**.

Для объекта корзины, доступ к операциям чтения и записи которой закрыты с помощью барьера, результаты следующие:

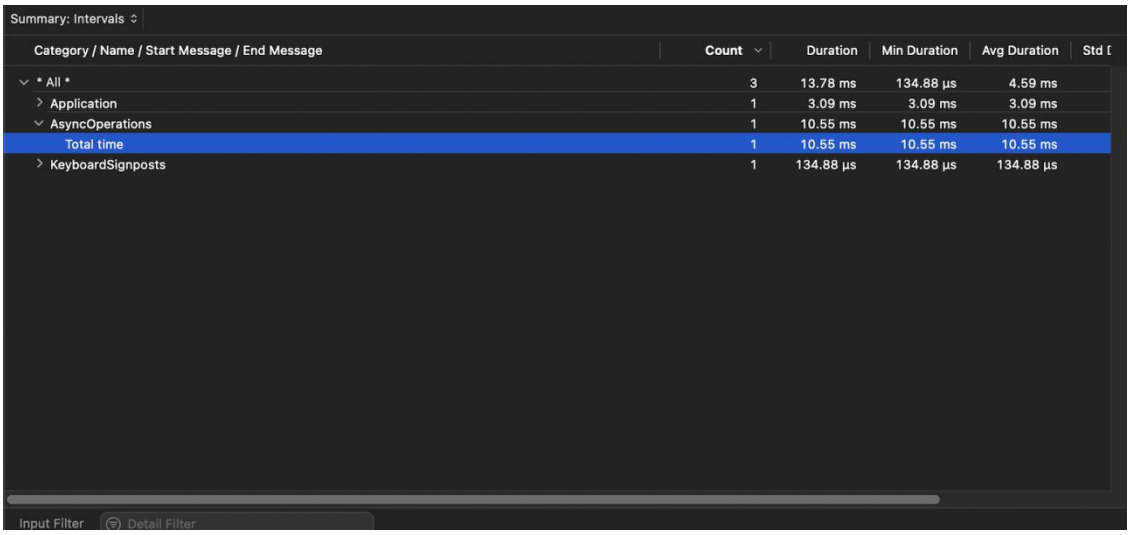

Рис. 7. Среднее время выполнения всех операций чтения и записи для объекта корзины на основе ***dispatch_barrier***

Для объекта корзины, которая реализована с помощью актора, результаты следующие:

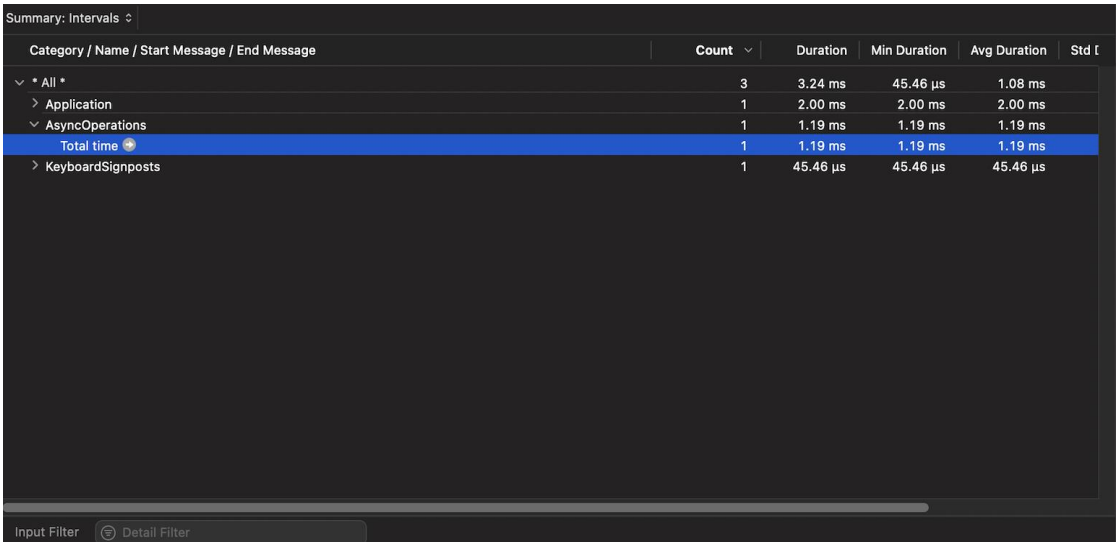

Рис. 8. Среднее время выполнения всех операций чтения и записи для объекта корзины на основе актора

Как видно из результатов, корзина работающая на основе конкурентной очереди с использованием барьера работает на порядок медленнее чем тот же самый объект на основе акторного подхода.

Заключение

В статье были рассмотрены основные механизмы обеспечения безопасности операций чтения и записи. Барьеры, являются одним из основных механизмов обеспечения безопасного доступа к критической секции. Язык программирования Swift и Apple SDK предоставляют весь необходимый спектр инструментов для применения подходов обеспечивающих корректный доступ к данным в многопоточной среде. Акторы, которые были представлены в языке программирования Swift начиная с версии 5.5, предоставляют доступ к своим методам и свойствам в сериализованном виде, выполняя все операции последовательно. Несмотря на это, скорость выполнения операций чтения и записи в объектах реализованных с помощью акторов на порядок быстрее, чем обычные классы которые инкапсулируют данную работу с помощью барьеров. При отладке проблем и временных промежутков в приложении существует возможность записывать точную последовательность произошедших событий вместе с дополнительными данными об этих событиях. OSLog обеспечивают непрерывную запись поведения приложения во время выполнения и упрощают выявление проблем и процесс измерения времени различного рода операций, которые не могут быть обнаружены с помощью других методов.

Список литературы

1. Современные операционные системы. 4-е издание. Таненбаум Эндрю, Бос Херберт, 2015.
2. Официальная документация компании Apple, Swift book. [Электронный ресурс], 2021. Режим доступа: https://docs.swift.org/swift-book/LanguageGuide/Concurrency.html/ (дата обращения: 21.04.2022).
3. Конференция Apple WWDC, Protect mutable state with Swift actors, 2021. [Электронный ресурс]. Режим доступа: https://developer.apple.com/videos/play/wwdc2021/10133/ (дата обращения: 21.04.2022).