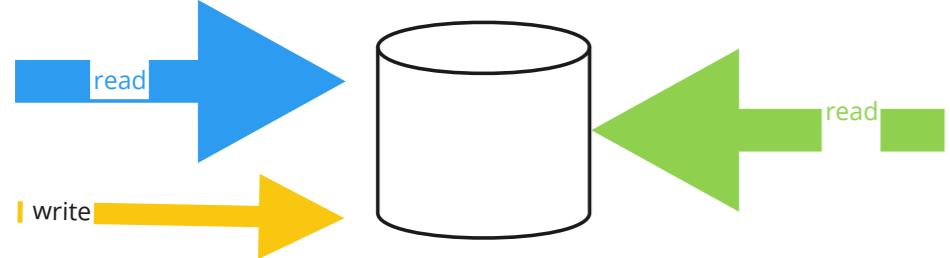


Синхронизация в iOS

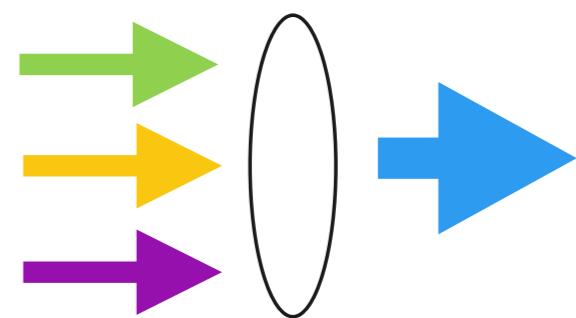
Задачи

не поломать
ничего при
работе с данными

Как синхронизировать запись и чтение из нескольких потоков?



СИНХРОНИЗАЦИЯ



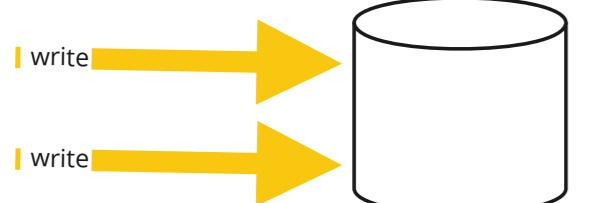
```
19 let firstQueue = DispatchQueue(label: "first queue") 🦇  
20 var value = "😈"  
21  
22 func setValue(_ value: String) {  
23     sleep(1)  
24     self.value = value  
25     print("did set \(value)")  
26 }  
27  
28 func run() {  
29     print(value)  
30     firstQueue.async {  
31         self.setValue("😊")  
32     }  
33     print(value)  
34 }  
35 }
```

did set 😊

а должен быть 😊

Annotations: A purple devil emoji is above the code at line 21. A red box highlights the line "self.setValue("😊")" with the text "а должен быть 😊" above it. Below the code, the text "did set 😊" is written.

Multiple writers



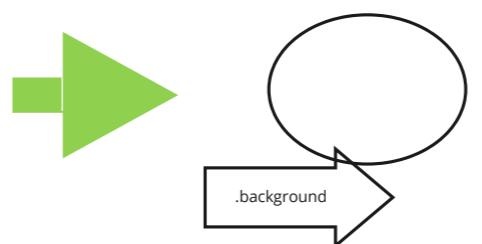
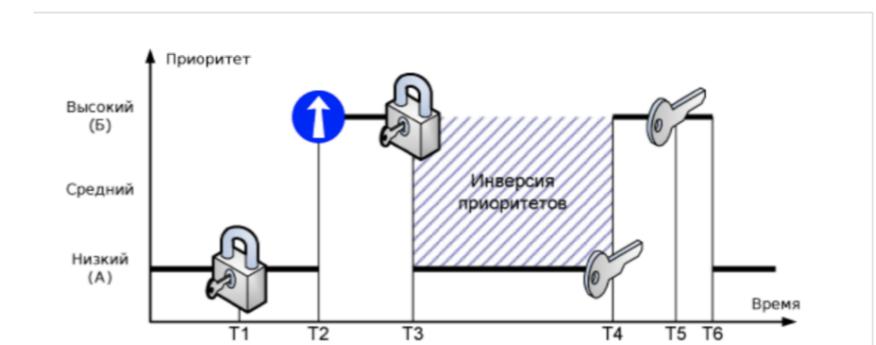
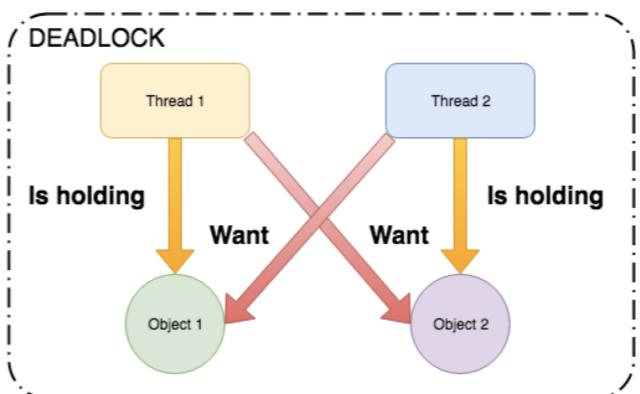
```
36  
37 var values = ["😈"]  
38  
39  
40 func run() {  
41     print(values)  
42     firstQueue.async {  
43         self.values.append("😊")  
44 //         self.val = Thread 3: EXC_BAD_ACCESS (c  
45         print(self.values)  
46     }  
47  
48     secondQueue.async {  
49         self.values.append("😊")  
50         print(self.values)  
51     }  
52  
53     }  
54 }
```

Их мы и будем решать разными методами

Что может пойти не так?

- **Условия гонки [Race condition]**: С несколькими потоками, при работе с одними данными, в результате чего сами данные становятся непредсказуемыми и зависят от порядка выполнения потоков.
- **Конкуренция за ресурс [Resource contention]**: Несколько потоков, выполняющих разные задачи, пытаются получить доступ к одному ресурсу, тем самым увеличивая время необходимое для безопасного получения ресурса. Эта задержка может привести к непредвиденному поведению.
- **Вечная блокировка [Deadlock]**: Несколько потоков блокируют друг друга.
- **Голодание [Starvation]** : Поток не может получить доступ к ресурсу и безуспешно пытается сделать это снова и снова.
- **Инверсия приоритетов [Priority Inversion]**: Поток с низким приоритетом удерживает ресурс, которые требуется другому потоку с более высоким приоритетом.
- **Неопределенность и справедливость [Non-deterministic and Fairness]**: Мы не можем делать предположений, когда и в каком порядке поток сможет получить ресурс, эта задержка не может быть определена априори и в значительной степени зависит от количества конфликтов. Однако, примитивы синхронизации могут обеспечивать справедливость, гарантируя доступ всем потокам которые ожидают, также учитывая порядок.

```
46
47 var values = ["😊"]
48
49 func run() {
50     print(values)
51     firstQueue.async {
52         sleep(1)
53         self.values.append("😎")
54         print(self.values)
55     }
56
57     secondQueue.async {
58         self.values.removeAll()
59         print(self.values)
60     }
61
62 }
63
64 }
```



medium.com

Всё о многопоточности в Swift: Часть 1: Настоящее

Данная статья является вольным переводом с некоторыми дополнениями отличного материала от Umberto Raimondi по многопоточности с Playground с материалами есть на GitHub. www.uraimo.com В настоящее время Swift не включает какой-либо нативный функционал д...

Задача про данные

Предоставляется языкоком

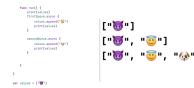
атомарность - Atomicity

An operation is atomic if it appears to the rest of the system to occur at a single instant without being interrupted. An atomic operation can either complete or return to its original state.

Atomicity is a safety measure which enforces that operations do not complete in an unpredictable way when accessed by multiple threads or processes simultaneously.

Гарантии языка

- Гарантия: переменные в процессе инициализируются атомарно.
- Логика синхронизации: если вы не можете избежать атомарности при инициализации. И так как предложенное никакой атомарной и межпотоковой чистоты не имеет, давите пальцы.
- Доступ к состоянию также не атомарен.



in the future

Инструменты

Локи

Kinds of Locks

There are two main types of locks, which are often combined to form more complex and stronger locks, which are the main building blocks in multithreaded OS applications.

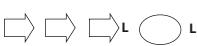
Lock is an abstract concept for thread synchronization. The basic idea is to protect access to a given piece of code at a time. Different kinds exist:

1. **Mutex** – mutexes allow only one thread to enter a given region of code at a time. You can think of it as a semaphore with a maximum value of 1.

2. **Reentrant** – reentrant mutexes allow a thread to re-enter a lock in a loop while checking if the lock is still held by the same thread.

3. **Read-write lock** – provides concurrent access for read-only operations, but exclusive access to write operations. Thread when reading is a reader and writing is a writer.

4. **Recursive lock** – allows a thread to be acquired by the same thread many times.



Overview of Apple Locking APIs

Lock

Locks are C-compatible `NSLock`s, i.e. Objective-C-like classes. They correspond to Mutex and recursive lock and don't have Swift counterparts.

A lower-level C `pthread_mutex_t` is also available in Swift. It can be configured both as a mutex and a recursive lock.

Semaphore

`OSSemaphore` has been deprecated in iOS 10 and now there is no exact matrix to a semaphore in Swift. The closest class is `DispatchSemaphore`, which doesn't look as convenient, but it's much easier to work with to be honest.

Thus, if a lower-level API report that the semaphore does not make iteration of writer's possibility.

Read-write lock

`pthread_rwlock_t` is a lower-level read/write lock that can be used in Swift.

Semaphore

`DispatchSemaphore` provides an implementation of a semaphore. It is intended for the sake of completeness, as it makes it very easy to use semaphores for debugging atomic properties.

(пример на objc)



GCD

semaphore



Semaphore →



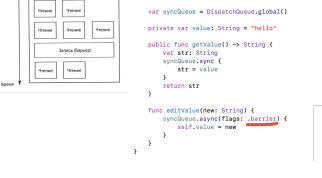
очереди



let x = Atomic<Int>(5)
x.mutate { \$0 += 1 }

барьеры

DISPATCH BARRIER



хороший подход

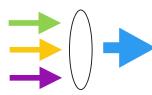
Single writer multiple readers

есть объект
у него есть свойство
в данном объекте можно обращаться с множеством потоков
обращение должно быть скооперировано
читать можно параллельно, а писать только последовательно
при записи читатели блокируются

секция синхронизации

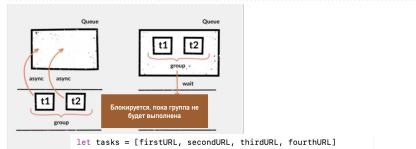
запись

Задача про синхронизацию нескольких потоков



группы

DISPATCH GROUP



```
let tasks = [firstURL, secondURL, thirdURL, fourthURL]
queue.async {
    let myGroup = DispatchGroup()
    for task in tasks {
        myGroup.enter()
        print(task)
        // что-то делаем. Когда сделали - выходим из группы
    }
    myGroup.wait()
    // делаем что-то, для чего мы всех ждали
}
```



```
let tasks = [firstURL, secondURL, thirdURL, fourthURL]
let myGroup = DispatchGroup()
for task in tasks {
    myGroup.enter()
    print(task)
    // что-то делаем. Когда сделали - выходим из группы
    myGroup.leave()
}
myGroup.notify(queue: DispatchQueue.main) {
    // делаем что-то, для чего мы всех ждали
    // сразу на нужной очереди! и не блокируем поток!
}
```

Дз: прикреплю

Вопросы?