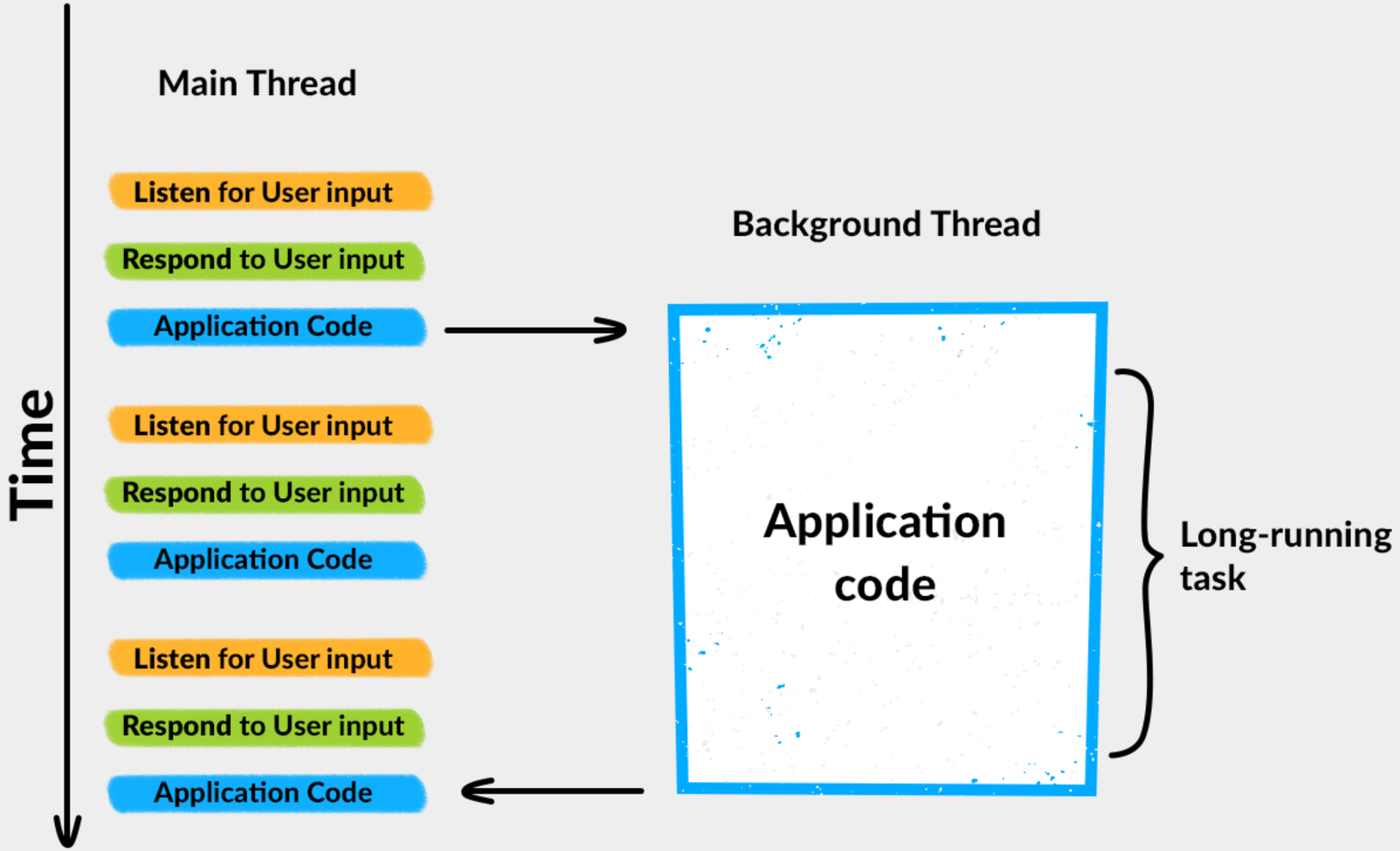
Multithreading in iOS

Alexey Shchukin

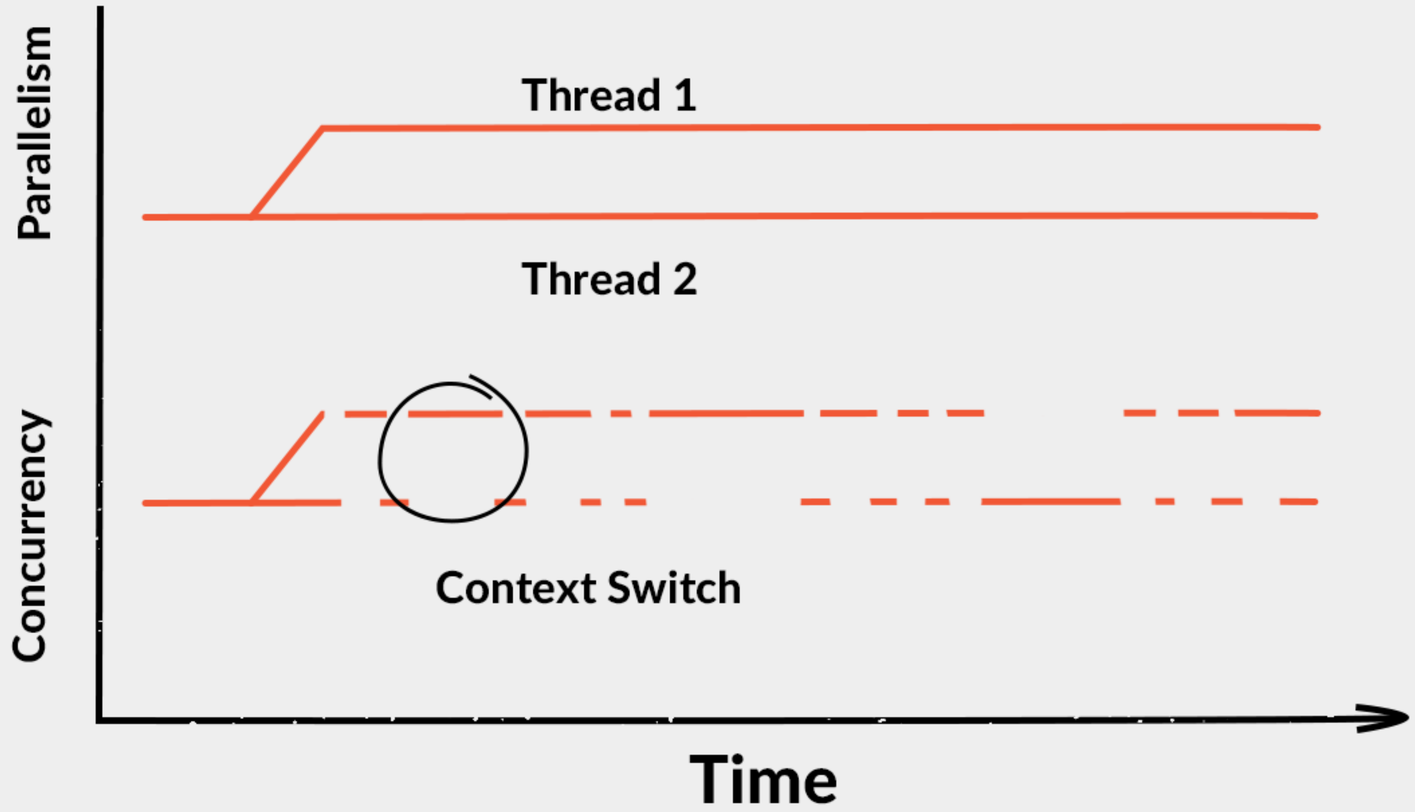
Lecture plan

- Multithreading basics
- Quality of service
- Synchronization
- Recursive mutex
- Condition
- Read write lock
- Spin lock
- Synchronized
- Problems
- Atomic operations

Multithreading basics



Concurrency vs Parallelism



pthread

```
var thread = pthread_t(bitPattern: 0)
var attr = pthread_attr_t()

pthread_attr_init(&attr)
pthread_create(&thread, &attr, { pointer in

    print("test")

    return nil
}, nil)
```

Thread

```
var nsthread = Thread(block: {
    print("test")
})
nsthread.start()
```

Quality of service

```
@available(iOS 8.0, *)
public enum QualityOfService : Int {

    case userInteractive

    case userInitiated

    case utility

    case background

    case `default`
}
```

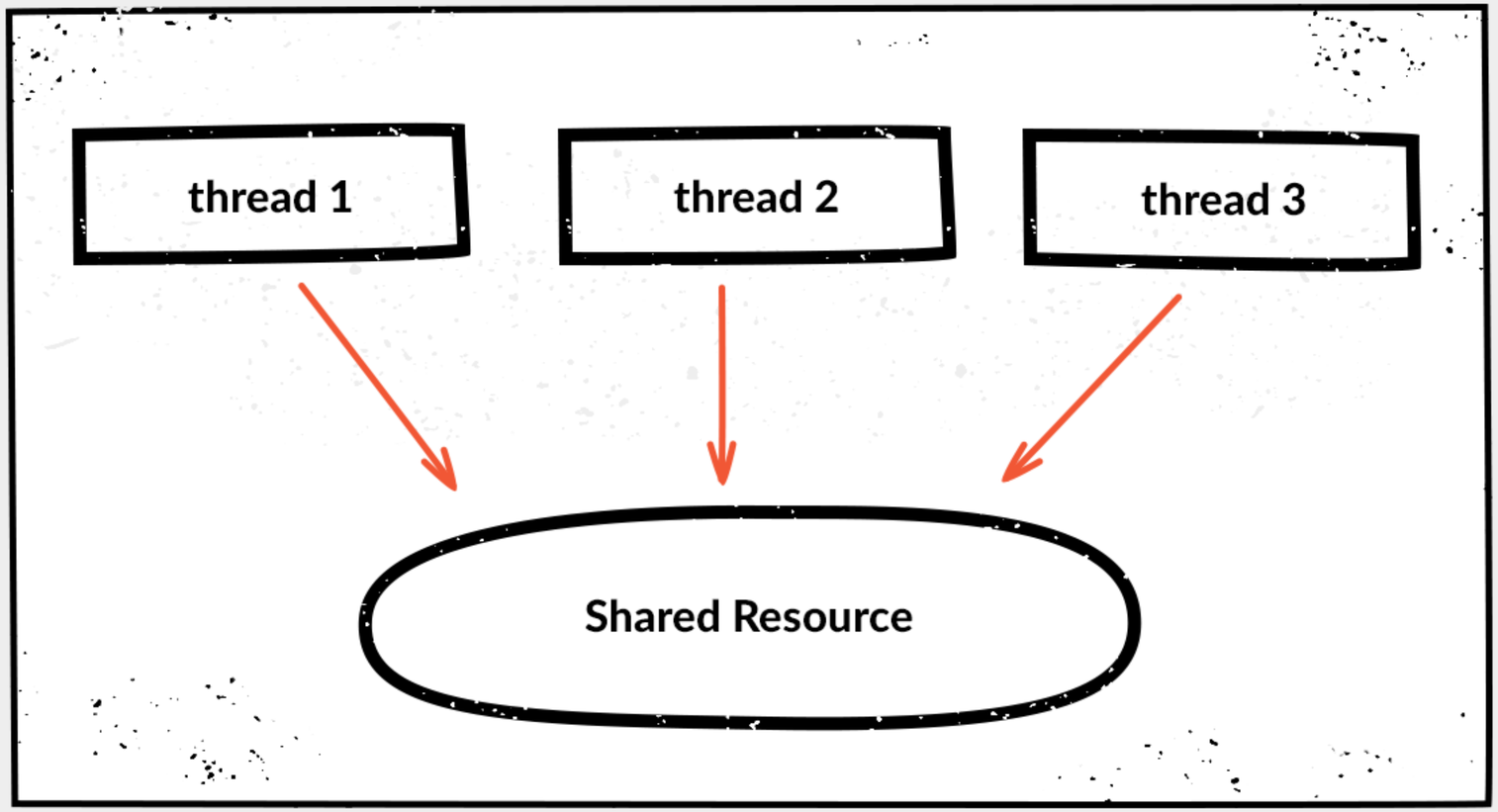
pthread qos

```
class PthreadQosTest {

    func test() {
        var thread = pthread_t(bitPattern: 0)
        var attribute = pthread_attr_t()
        pthread_attr_init(&attribute)
        pthread_attr_set_qos_class_np(&attribute, QOS_CLASS_USER_INITIATED, 0)
        pthread_create(&thread, &attribute, { pointer in

            print(“test")
            pthread_set_qos_class_self_np(QOS_CLASS_BACKGROUND, 0)

            return nil
        }, nil)
    }
}
```

Thread qos

```
class QosThreadTest {

    func test() {

        let thread = Thread {

            print("test")
            print(qos_class_self())
        }
        thread.qualityOfService = .userInteractive
        thread.start()

        print(qos_class_main())
    }
}
```

Synchronization



pthread_mutex

```
class Test {

    private var mutex = pthread_mutex_t()

    init() {
        pthread_mutex_init(&mutex, nil)
    }

    func test() {
        pthread_mutex_lock(&mutex)
        //Do something
        pthread_mutex_unlock(&mutex)
    }
}
```

NSLock

```
public class NSLockTest {

    private let lock = NSLock()

    func test(i: Int) {
        lock.lock()
        //Do something
        lock.unlock()
    }
}
```

Recursive pthread_mutex

```
class RecursiveMutexTest {

    private var mutex = pthread_mutex_t()
    private var attr = pthread_mutexattr_t()

    init() {
        pthread_mutexattr_init(&attr)
        pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE)
        pthread_mutex_init(&mutex, &attr)
    }

    func test1() {
        pthread_mutex_lock(&mutex)
        test2()
        pthread_mutex_unlock(&mutex)
    }

    func test2() {
        pthread_mutex_lock(&mutex)
        //Do something
        pthread_mutex_unlock(&mutex)
    }
}
```

NSRecursiveLock

```
class RecursiveLockTest {

    private let lock = NSRecursiveLock()

    public func test1() {
        lock.lock()
        test2()
        lock.unlock()
    }

    func test2() {
        lock.lock()
        //Do something
        lock.unlock()
    }
}
```

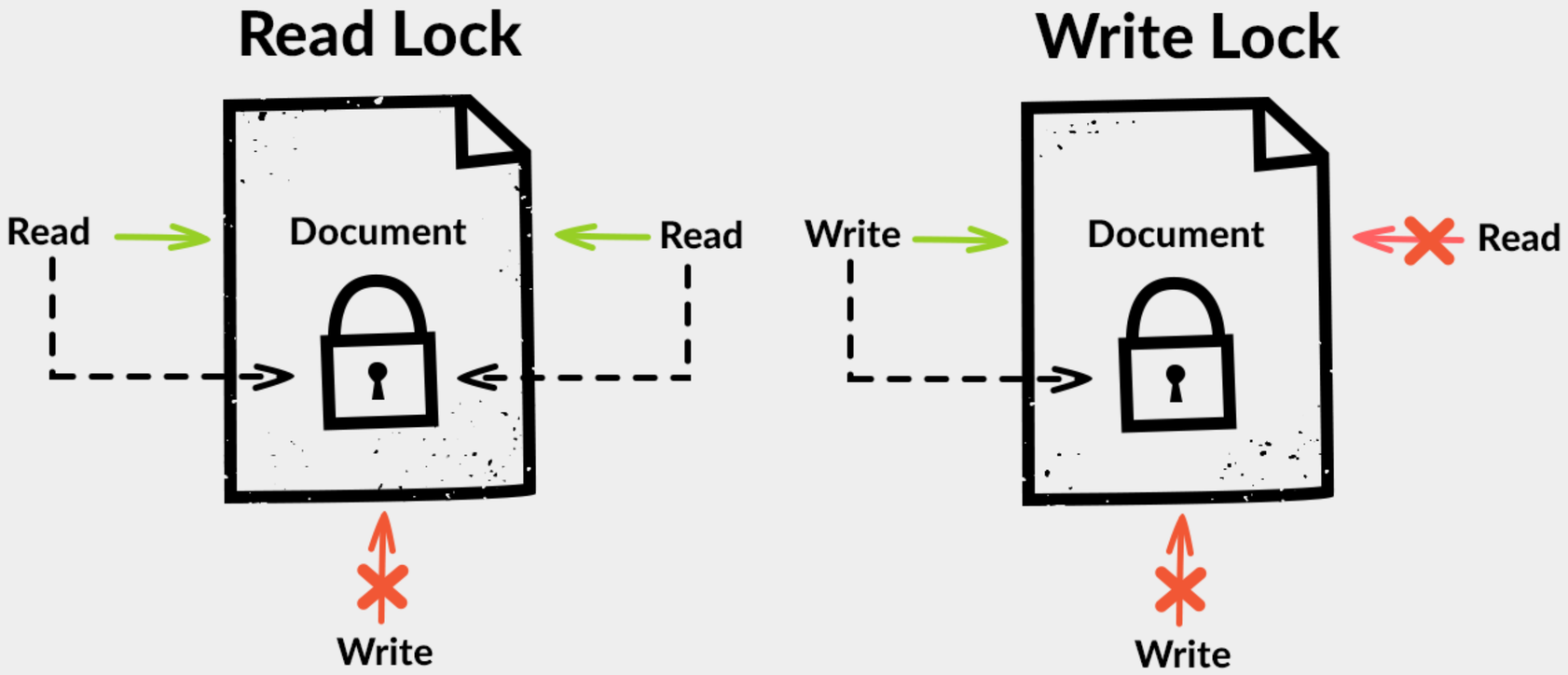
pthread_cond

```
class MutexConditionTest {
    private var condition = pthread_cond_t()
    private var mutex = pthread_mutex_t()
    private var attr = pthread_mutexattr_t()
    private var check = false

    init() {
        pthread_cond_init(&condition, nil)
        pthread_mutexattr_init(&attr)
        pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE)
        pthread_mutex_init(&mutex, &attr)
    }

    func test1() {
        pthread_mutex_lock(&mutex)
        while !check {
            pthread_cond_wait(&condition, &mutex)
        }
        //Do something
        pthread_mutex_unlock(&mutex)
    }

    func test2() {
        pthread_mutex_lock(&mutex)
        check = true
        pthread_cond_signal(&condition)
        pthread_mutex_unlock(&mutex)
    }
}
```

NSCondition

```
class ConditionTest {
    private let condition = NSCondition()
    private var check: Bool = false

    func test1() {
        condition.lock()
        while(!check) {
            condition.wait()
        }
        condition.unlock()
    }

    func test2() {
        condition.lock()

        check = true
        condition.signal()
        condition.unlock()
    }
}
```

Read Write Lock



Read Write Lock

```
class ReadWriteLockTest {
    private var lock = pthread_rwlock_t()
    private var attr = pthread_rwlockattr_t()

    private var test: Int = 0

    init() {
        pthread_rwlock_init(&lock, &attr)
    }

    var testProperty: Int {
        get {
            pthread_rwlock_rdlock(&lock)
            let tmp = test
            pthread_rwlock_unlock(&lock)
            return tmp
        }
        set {
            pthread_rwlock_wrlock(&lock)
            test = newValue
            pthread_rwlock_unlock(&lock)
        }
    }
}
```

Spin lock

```
class SpinLockTest {

    private var lock = OS_SPINLOCK_INIT

    func test() {

        OSSpinLockLock(&lock)
        //Do something ...
        OSSpinLockUnlock(&lock)
    }
}
```

Unfair Lock

```
class UnfairLockTest {

    private var lock = os_unfair_lock_s()

    func test() {
        os_unfair_lock_lock(&lock)
        //Do something...
        os_unfair_lock_unlock(&lock)
    }
}
```

Synchronized

```
class SynchronizedTest {

    private let lock = NSObject()

    func test() {
        objc_sync_enter(lock)
        //Do something...
        objc_sync_exit(lock)
    }
}
```

Deadlock

```
class DeadLockTest {
    private let lock1 =
NSLock()
    private let lock2 =
NSLock()
    var resourceA = false
    var resourceB = false

}
```

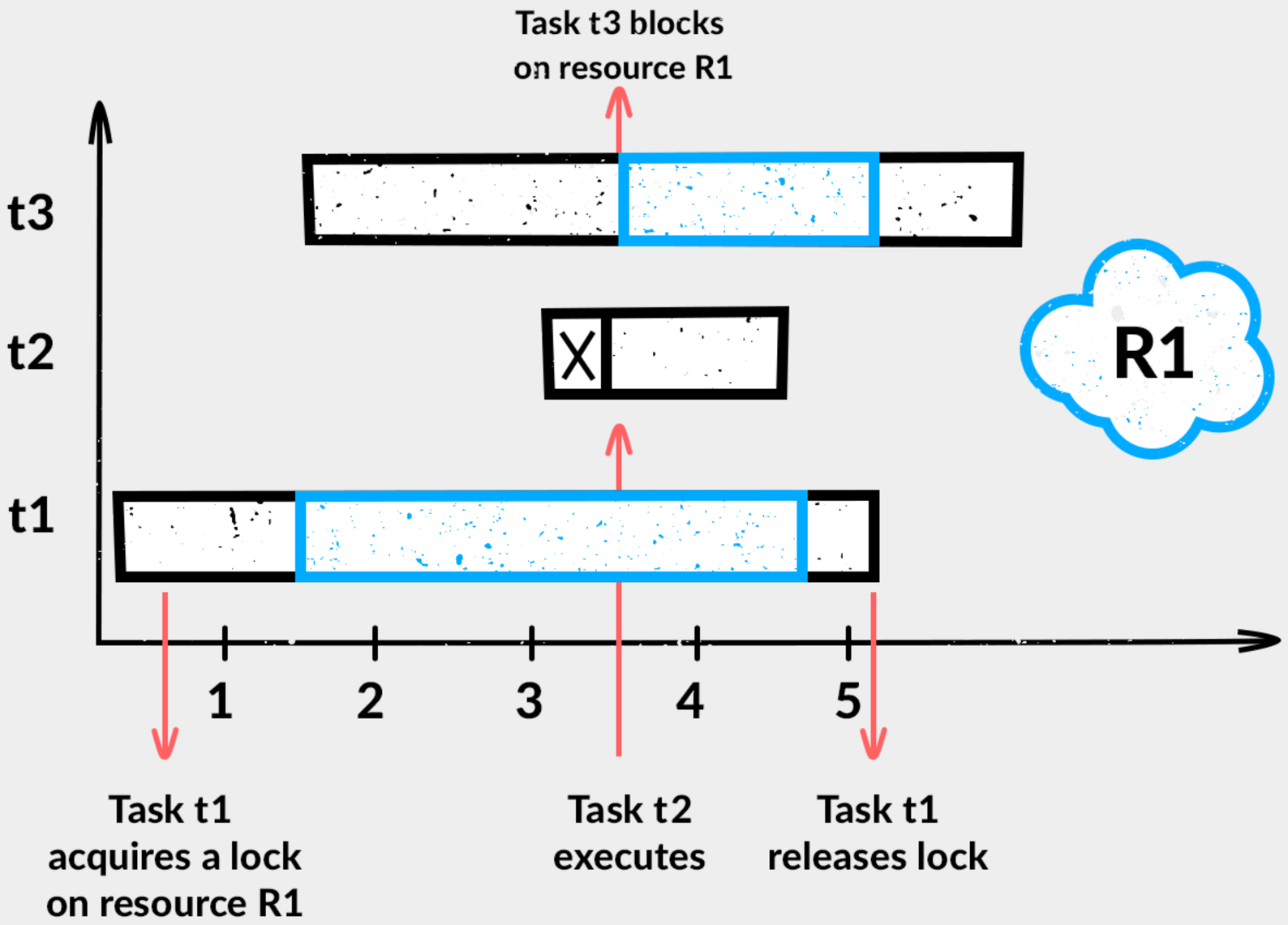
```
let thread1 = Thread(block: {
    self.lock1.lock()

    self.resourceA = true

    self.lock2.lock()
    self.resourceB = true
    self.lock2.unlock()

    self.lock1.unlock()
})
thread1.start()
```

```
let thread2 = Thread(block: {
    self.lock2.lock()

    self.resourceB = true

    self.lock1.lock()
    self.resourceA = true
    self.lock1.lock()

    self.lock2.lock()
})
thread2.start()
```

Livelock ???

```
class LivelockTest {

    var flag1 = false
    var flag2 = false

    func test() {

        let thread1 = Thread(block: {
            self.flag1 = true
            while self.flag2 {
                self.flag1 = false
                sleep(1) //Delay
                self.flag1 = true
            }
            self.flag1 = true
        })
        thread1.start()

        let thread2 = Thread(block: {
            self.flag2 = true
            while self.flag1 {
                self.flag2 = false
                sleep(1) //Delay
                self.flag2 = true
            }
            self.flag2 = true
        })
        thread2.start()
    }
}
```

Priority inversion



Atomic operations

```
class AtomicOperationsPseudoCodeTest {
    func compareAndSwap(old: Int, new: Int, value: UnsafeMutablePointer<Int>) -> Bool {
        if(value.pointee == old) {
            value.pointee = new
            return true
        }
        return false
    }

    func atomicAdd(amount: Int, value: UnsafeMutablePointer<Int>) -> Int {
        var success = false
        var new: Int = 0

        while !success {
            let original = value.pointee
            new = original + amount
            success = compareAndSwap(old: original, new: new, value: value)
        }

        return new
    }
}
```

Atomic operations usage

```
class AtomicOperationsTest {

    private var i: Int64 = 0

    func test() {

        OSAtomicCompareAndSwap64Barrier(i, 10, &i)
        OSAtomicAdd64Barrier(20, &i)
        OSAtomicIncrement64Barrier(&i)
    }
}
```

Memory Barrier

```
class MemoryBarrierTest {
    class TestClass {
        var t1: Int?
        var t2: Int?
    }
    var testClass: TestClass?

    func test() {
        let thread1  = Thread {
            let tmp = TestClass()
            tmp.t1 = 100
            tmp.t2 = 500
            OSMemoryBarrier()
            self.testClass = tmp
        }
        thread1.start()

        let thread2 = Thread {
            while self.testClass == nil { }
            OSMemoryBarrier()
            print(self.testClass?.t1)
        }
        thread2.start()
    }
}
```

Multithreading in swift

```
class SwiftTest {
    private let queue = DispatchQueue(label: "SwiftTest")

    struct TestStruct {

        var i: Int
        var y: Int
    }

    let testStruct = TestStruct(i: 0, y: 0)

    func test() {

        queue.async {
            print(self.testStruct.i)
        }

        queue.async {
            print(self.testStruct.y)
        }
    }
}
```

Multithreading with collection

```
class CollectionTest {
    private let queue = DispatchQueue(label: "CollectionTest")
    private let cache = NSCache<NSString, NSNumber>()

    func test() {
        queue.async {
            let number = NSNumber(integerLiteral: 1)
            let key = NSString(string: "test")
            self.cache.setObject(number, forKey: key)
        }

        queue.async {
            let number = NSNumber(integerLiteral: 2)
            let key = NSString(string: "test")
            self.cache.setObject(number, forKey: key)
        }

        queue.async {
            let number = NSNumber(integerLiteral: 3)
            let key = NSString(string: "test")
            self.cache.setObject(number, forKey: key)
        }
    }
}
```

Multithreading in UI

```
class BackgroundDrawingView: UIView {
    private let queue = DispatchQueue(label: "BackgroundDrawingView", attributes: .concurrent)

    override init(frame: CGRect) {
        super.init(frame: frame)
        drawBackground(rect: frame)
    }

    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    func drawBackground(rect: CGRect) {
        queue.async {
            let test = "test"
            let attributes = [NSAttributedStringKey.foregroundColor: UIColor.green,
                              NSAttributedStringKey.kern: 20,
                              NSAttributedStringKey.font: UIFont.systemFont(ofSize: 30)
                ] as [NSAttributedStringKey : Any]

            UIGraphicsBeginImageContextWithOptions(rect.size, false, 0)
            test.draw(in: rect, withAttributes: attributes)
            let image = UIGraphicsGetImageFromCurrentImageContext()
            UIGraphicsEndImageContext()

            DispatchQueue.main.async {
                self.layer.contents = image?.cgImage
            }
        }
    }
}
```