TODAY

- Semaphores
- Race conditions / deadlocks / priority inversions
- Operation queues

SEMAPHORES

- Assume downloading a lot of data from the network
  - dispatch queues to offload work
  - dispatch groups for completion notification
  - but what if only want four simultaneous downloads?
- Straightforward semaphores
  - create w/ number of resources
  - wait on it to get access
RACE CONDITIONS

- Threads 1 and 2 both execute:
  - `count += 1`

- Very difficult to debug ("Heisenbug")
COULD SOLVE W/ A SERIAL Q

private let threadSafeCountQueue = DispatchQueue(label: "...")
private var _count = 0
public var count: Int {
    get {
        return threadSafeCountQueue.sync {
            _count
        }
    }
    set {
        threadSafeCountQueue.sync {
            _count = newValue
        }
    }
}

• queues serial by default

private let threadSafeCountQueue = DispatchQueue(label: "...") // serial Q
public var count: Int
public func incr(inout anInt: Int) {
    return threadSafeCountQueue.sync {
        anInt += 1
    }
}

incr(&count)

• queues serial by default
private let threadSafeCountQueue = DispatchQueue(label: "...", attributes: .concurrent)

private var _count = 0
public var count: Int {
    get {
        return threadSafeCountQueue.sync {
            return _count
        }
    }
    set {
        threadSafeCountQueue.async(flags: .barrier) { [unowned self] in
            self._count = newValue
        }
    }
}

private let q = DispatchQueue(label: "436", .concurrent)
public var count: Int

public func incr(inout anInt: Int) {
    return q.async(flags: .barrier) {
        anInt += 1
    }
}
incr(&count)
incr(&count)
DEADLOCK

- With semaphores:
  - Thread 1 acquires A, waits to get B
  - Thread 2 acquires B, waits to get A

- How to prevent?
  - Always possible? Why or why not?

- Most common in iOS with `.sync()` against serial dispatch queue

PRIORITY INVERSION

- When low-priority job gets higher priority than high-priority job…

- Dispatch queues have different QOS levels (basically priorities)
  - tasks added with explicit QOS levels
  - system adapts queue QOS to highest job QOS
  - low-priority tasks on same queue might end up running with high priority

- Happens if resources shared among high-/low-priority queues
  - and low priority queue gets the resource first
PRIORITY INVERSION

Demo
ALTERNATIVE TO GCD: OPERATIONS

- Advantages:
  - inter-operation dependencies
  - clean way to pass results from one to next (func. prog.)
  - reusability
  - cancelling
  - KVO notifications

- Details
  - on top of GCD
  - object-oriented
  - Operation is abstract class
    - must be subclassed

OPERATION QUEUES

- BlockOperation works by:
  - subclassing an Operation
  - adding a closure
  - adding an array of closures

- Useful operation queue methods / properties
  - .maxConcurrentOperationCount = 4
  - .cancelAllOperations()
  - .waitUntilAllOperationsAreFinished()
  - etc.
BlockOperation is a convenience subclass of Operation

```swift
let operation = BlockOperation {
    print("2 + 3 = \((2 + 3)\)"
}
operation.start()

or

let sentence = "Draymond Green is back, baby!"
let wordOperation = BlockOperation()
for word in sentence.split(separator: " ") {
    wordOperation.addExecutionBlock {
        print(word)
    }
}
wordOperation.start()
```