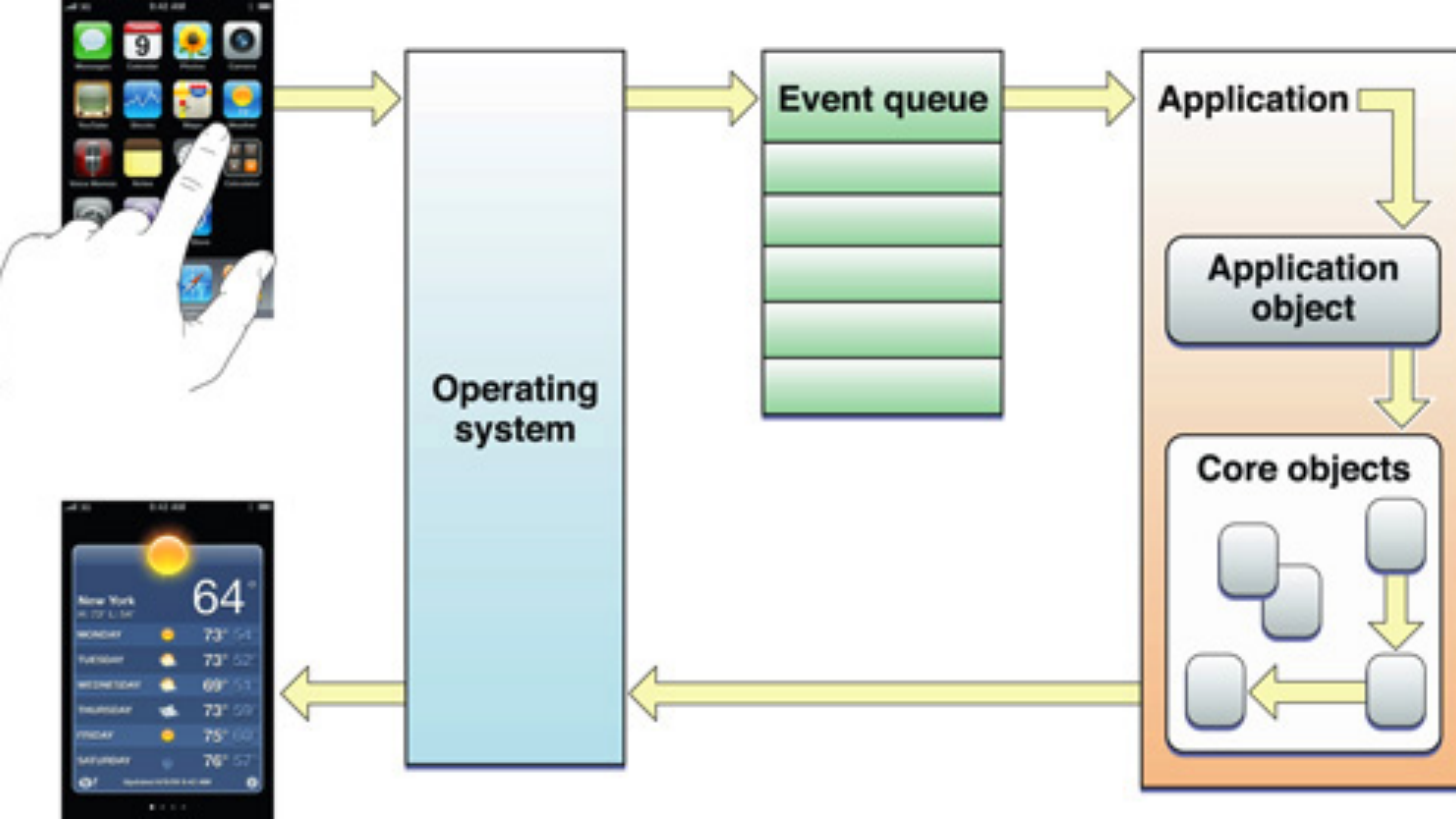
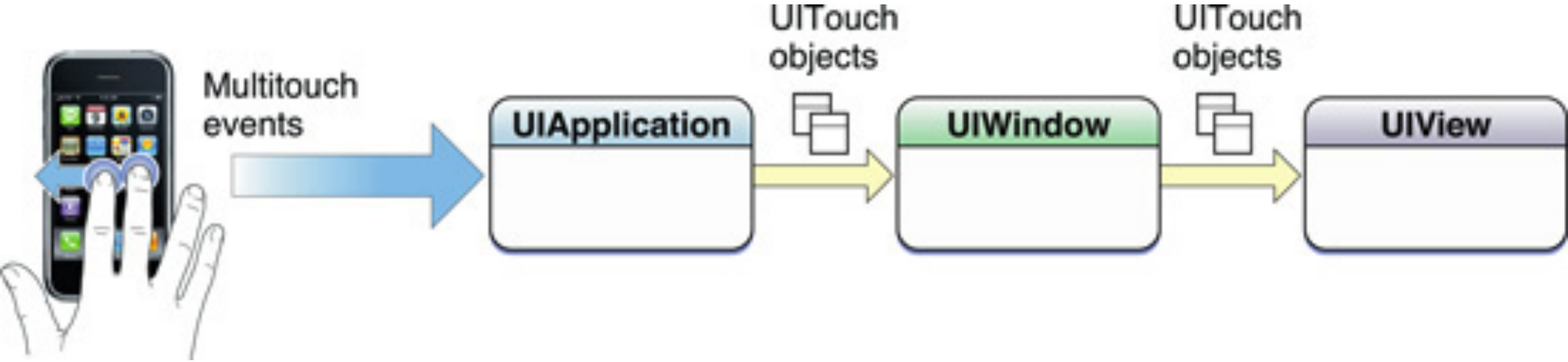
Promises 🎁

UI should be responsive at all times

How does iOS enforces that UI is performant at all times?



Event queue
Application
Application object
Operating system
Core objects


Multitouch events
UIApplication
UITouch objects
UIWindow
UITouch objects
UIView

The tradeoff is that all *event-handling* must be done on the main thread

- Creation of *all* UIKit objects
- Drawing
- Presentation/dismissal of **`UIViewControllers`**
- Layout of the View's frames
  - Autolayout
  - Manual based layout

What about…

- Networking
- Data Base
- File I/O
- Computations
- Image rendering

iOS is *Unix*-based, so it's a completely multithreaded environment

	4º Generation	5º Generation	6º Generation	7º Generation	8º Generation
2 Cores					
3 Cores					

Ideally, you'd want to move all those time-consuming operations to the background thread, right?

A programmer had a problem. He thought to himself, "I know, I'll solve it with threads!". has Now problems. two he

Common pitfalls

- Deadlocks
- Priority inversion
- Data corruption
- and more!

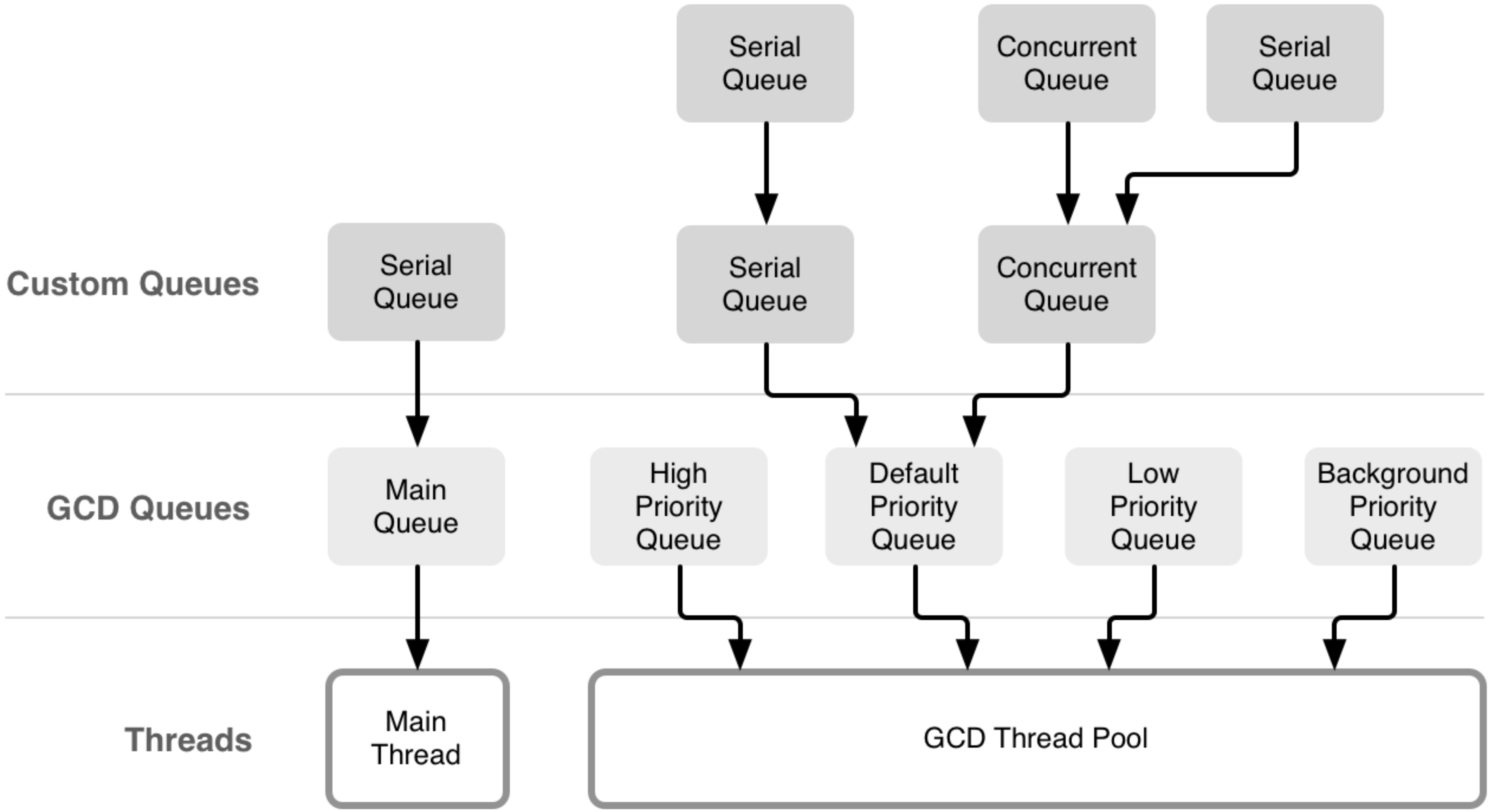

rmann
0 m – 110 yds/vgs
100% Polies
100m

Threading options:

- `NSOperationQueue`
- Grand Central Dispatch
- `NSThread`
- `pthread`

Grand Central Dispatch

- Introduced in iOS 4
- Thread pool is managed by the OS, not the developer.
- Introduces the Queue concept
  - Work is added with Closures/Blocks
  - Thread Pool is managed by the OS according to system resources.


Serial Queue
Concurrent Queue
Serial Queue
Custom Queues
Serial Queue
Serial Queue
Concurrent Queue
GCD Queues
Main Queue
High Priority Queue
Default Priority Queue
Low Priority Queue
Background Priority Queue
Threads
Main Thread
GCD Thread Pool

Serial vs Concurrent Queues

- Serial queues finish executing one work item before moving to the next.
- Concurrent queues could potentially execute more than work item at a time.

Schedule work

```swift
let serialQueue = DispatchQueue(label: "queuename")
serialQueue.async {
    //Do async work here
}

serialQueue.sync {
    //Do sync work here
}
```

Queue creation

```
let concurrentQueue = DispatchQueue(label: "queuename", attributes: .concurrent)

let backgroundQueue = DispatchQueue(
 label: "queuename",
 qos: .background,
 attributes: [],
 autoreleaseFrequency: .workItem,
 target: nil
)

let global = DispatchQueue.global(qos: .background)
```

QoS

```
typedef NS_ENUM(NSInteger, NSQualityOfService) {
    NSQualityOfServiceUserInteractive = 0x21,
    NSQualityOfServiceUserInitiated = 0x19,
    NSQualityOfServiceUtility = 0x11,
    NSQualityOfServiceDefault = -1
} API_AVAILABLE(macos(10.10), ios(8.0), watchos(2.0), tvos(9.0));
```

Cancel support

```swift
let workItem = DispatchWorkItem {
    // Do some exciting work
}
workerQueue.async(execute: workItem)
workItem.cancel()
```

Ok, now some real world examples:

```swift
self.apiClient.requestProducts { (data, error) in
    guard error == nil else {
        handler(nil, error!)
        return
    }
    self.parser.parseData(data) { (products, error) in
        guard error == nil else {
            handler(nil, error!)
            return
        }

        self.coreDataStack.storeProducts(products) { (managedProducts, error) in
            guard error == nil else {
                handler(nil, error!)
                return
            }

            handler(managedProducts, nil)
        }
    }
}
```

```swift
func fetchProductsAndUsers(handler: @escaping (Void) -> Void) {

    var productsFetchReady: Bool = false
    var usersFetchReady: Bool = false

    self.apiClient.requestProducts {
        productsFetchReady = true

        if productsFetchReady && usersFetchReady {
            handler()
        }
    }

    self.apiClient.requestUsers {
        usersFetchReady = true

        if productsFetchReady && usersFetchReady {
            handler()
        }
    }
}
```

Let's add some Swift

```
let fetchProducts =
self.apiClient.fetchProducts()
    .then(self.parser.parseData)
    .then(self.coreDataStack.storeProducts)

fetchProducts
.onSuccess { products in
    // Do stuff with products
}.onFailure { error
    // Do stuff with error
}
```

```swift
let fetchProducts = self.apiClient.fetchProducts()
let fetchUsers = self.apiClient.fetchUsers()

let combined = fetchProducts.and(fetchUsers)

combined.onSuccess { products in
    // Do stuff with products
}.onFailure { error
    // Do stuff with error
}
```

Promise<T>[1]

- Describes an object that acts as a proxy for a result that is initially unknown, usually because the computation of its value is yet incomplete.
- Also known as *future*, *delay* and *deferred*
- Implementations available for Java, JavaScript, C++, Phyton…
- Makes it easier to implement the Actor Model.

---

[1] Futures and promises, Wikipedia

`Promise<T>`

- Only available to Swift via 3rd Party libraries:
  - FutureKit
  - Deferred
  - PromiseKit

Promise<T>

```
func getAnImageFromServer(url : URL) -> Future<UIImage> {
    let p = Promise<UIImage>()

    DispatchQueue.global().async {
        let i = UIImage()
        p.completeWithSuccess(i)
    }
    return p.future
}
```

Demo

Takeaways:

- Any completionBlock based API is easy to wrap using Promises.
- Delegate-based APIs are harder, but not impossible.
- Wrap from top to bottom, always leaving old APIs available for callers.
  - Easier integration.
  - Real improvement will come when the full stack is adapted.
- Don't forget to unit-test.

Promises vs Rx:

Similarities

- Both are monads:
  - `map/flatMap`
  - `reduce`
  - `combine`
- Have support for success and failure scenarios.
- Abstracts underlying threading system.

Promises vs Rx:

Differences

- Promises are for one-off uses.
- Rx has the concept of stream.
  - The data is continously changing value.

Promises vs Rx:

When to use each

- Promises are far more suited for REST API clients.
- Rx are better for document editors/real time networking.