Basics

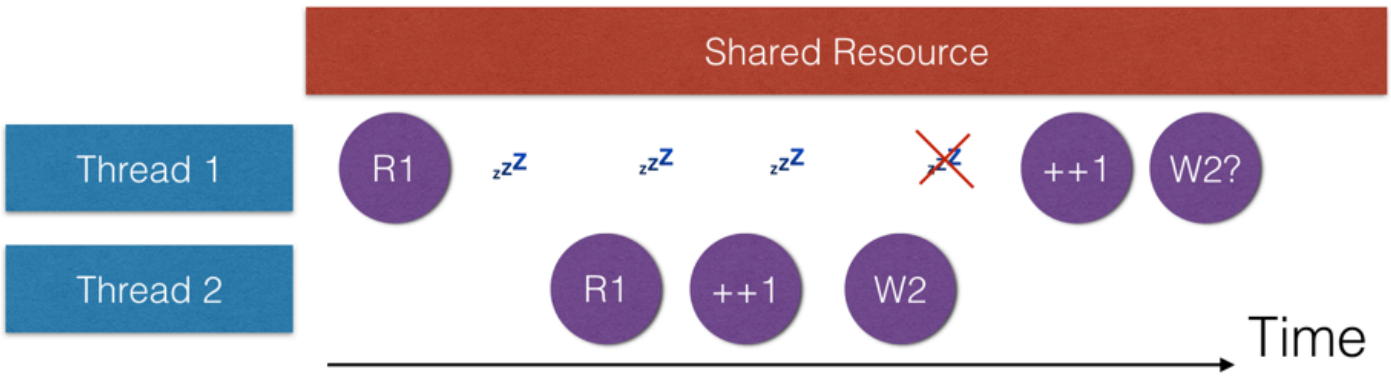
- When code executes the path of execution is singular. (What does this mean?)
- **Threads** allows the OS and our apps to have multiple separate paths of execution.
- This means we can run tasks at the same time (with multicore).
- Every line of code runs on *some thread*.
- You will almost never want to deal *directly* with threads in iOS because dealing with threads directly is incredibly complex.
- Instead you will use either the C API *GCD* (Grand Central Dispatch) or (NS)OperationQueue to interact *indirectly* with threads by using queues.
- Recall that Queues are actually a data structure.
- What qualities do queues have again?

3 Threading Problems

- There are 3 infamous problems that can arise when dealing directly with threads. These can also arise when dealing with Queues.

1) Race Conditions

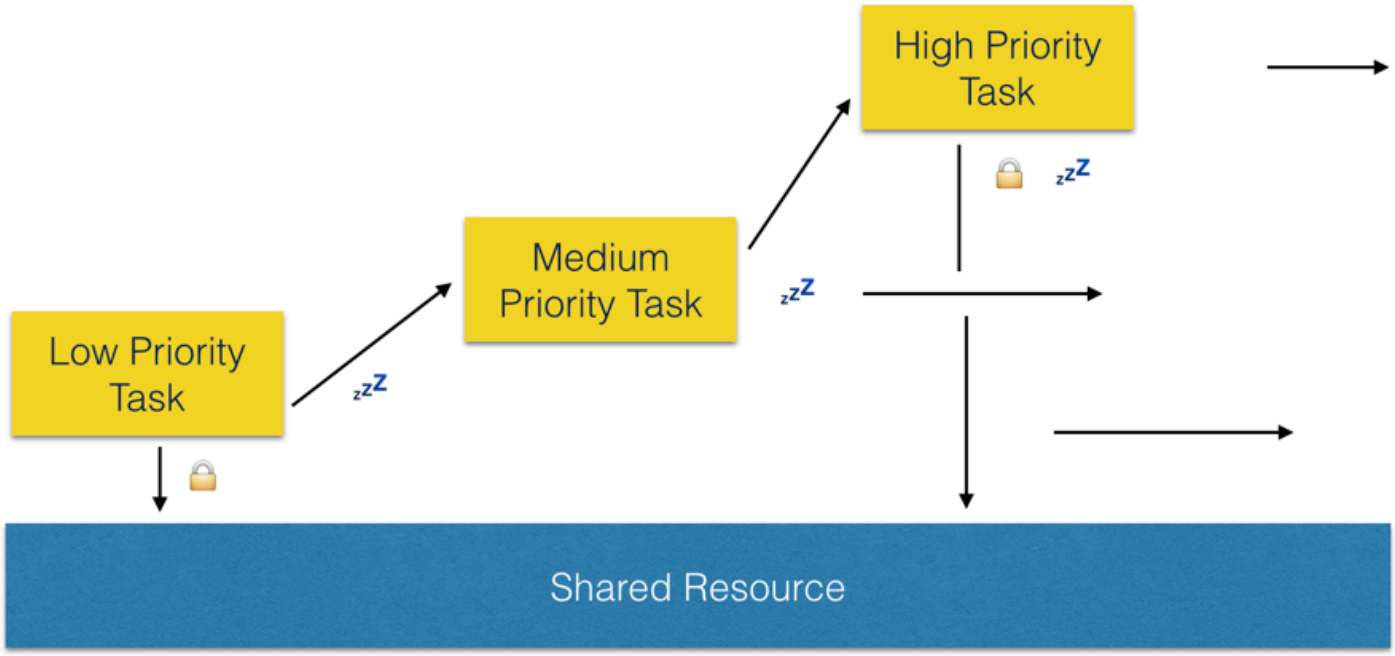
- Can happen when 2 or more threads try to *modify* a shared resource



- Hard to debug because probabilistic
- Traditional solution is to *lock* the shared resource while it is being mutated to prevent another process from writing to it.
- Serial Q's are actually a much better solution than locks when you have different threads writing to the same resource (more on serial vs concurrent Q's below)

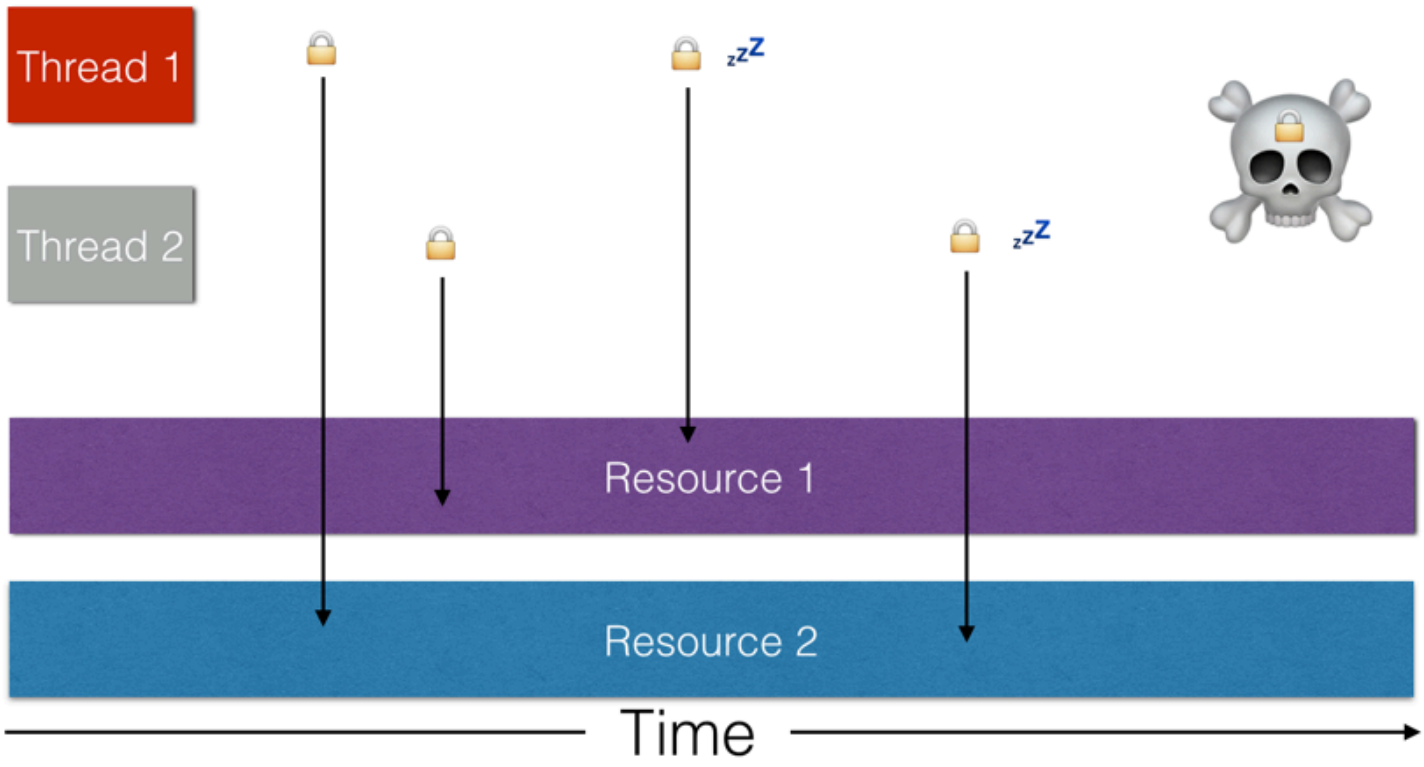
2) Priority Inversion

- priority inversion is when a lower priority task preempts a higher priority task



- GCD solves this by boosting the priority of lower priority tasks to speed them up.

3) Dead Lock
- When two threads never execute because they are both waiting for the other to release a shared resource.



- This is why solving race conditions using locks can be problematic.
- Better to use Serial Q's when writing to shared resources from different Q's.

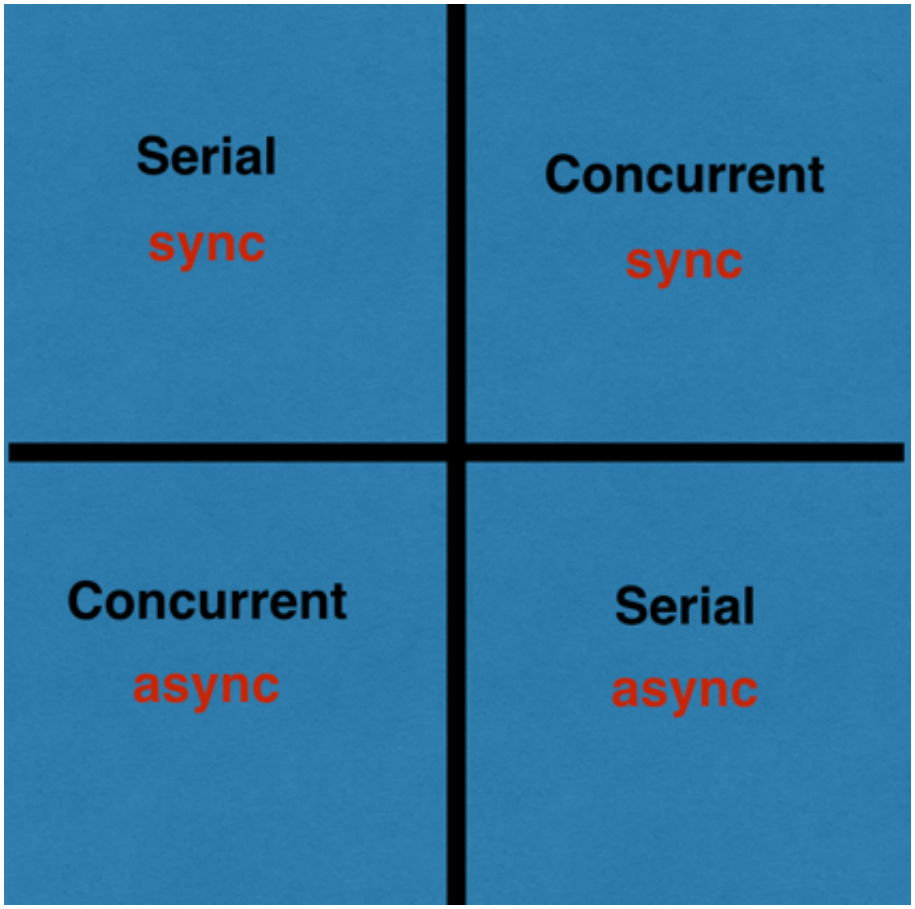
Terminology

- Concurrency allows us to run multiple paths of execution at the same time.
- Single core devices do this by task switching very quickly.
- Since 2011 iPad and iPhone have been multicore which allows true concurrency.
- Instead of dealing directly with threads GCD/(NS)OperationQueue use queues.

- Queues are not the same thing as threads.
- Queues are abstractions that employ threads.
- Queues are used to execute *blocks/closures.*
- Queues can be *serial* or *concurrent*.
- A *serial queue* **executes** blocks FIFO.
- Serial queues wait until the currently executed block completes before the next block is executed.
- A *concurrent queue* **dequeues** (starts) blocks in a FIFO manner.
- The execution of blocks on a concurrent queue happen on different underlying threads concurrently (depending on “resources”).
- Concurrent queues do not wait for the currently dequeued block to return before executing the next one. They dequeue in order, but immediately.
- This means the completion of blocks on a concurrent queue is essentially random. Never rely on the order of completion.
- **Main Queue:**
  - The main queue is the interface queue.
  - The main queue uses the applications *main thread*.
  - The main queue is where all interface interaction events are received (like touch events).
  - **When your app interacts with the view layer it *must* do so on the main queue.**
  - This is because the main queue is ***not thread safe***. (What does it mean to be “thread safe”?)
  - Example of tapping a button
    - Touch event arrives on main thread.
    - Button’s action runs on the main thread.
    - No other interface events can be handled while this is happening.

        - Once the code finishes running, the main thread is again ready to receive events or run code.
    - The main thread is ***blocked*** while your code runs on it, and this can make the interface unresponsive if your code is a time hog.
    - You need to worry about creating background queues in at least 2 situations
        - 1. Your code does something that ***might*** take a long time. (Even if you have a fast network and you're not doing an expensive request, why should you ***never ever*** do a network request on the main Q?)
        - 2. Some framework/library you're using calls you back on a background queue. E.g. NSURLSession calls you back on a background queue.
            - So, you must get a reference to the main queue in order to update the view. Note: only come back to the main queue at the moment you need to update the interface, never before. Coming back before you actually need to update the view may result in intermittent weirdness.
- We have concurrent and serial queues.
- We also have sync and async execution.
- You can run an async task on a concurrent or serial Q.
- sync/async tells you whether the current Q you're calling from needs to wait for the task to complete before continuing.
- serial/concurrent tells you whether a Q has 1 or many threads (whether it can run only 1 or many tasks

simultaneously).
- sync/async is about what happens to the execution of code at the source of the task (does it wait or keep moving).
- concurrent/serial is about the execution of the task (does it execute serially ie, on 1 thread, or concurrently on multiple threads).



(NS)OperationQueue Vs GCD

- Apple offers three different (!) API's for concurrency: C GCD, Swift GCD wrapper and (NS)OperationQueue.
- (NS)OperationQueue is an OO wrapper around GCD available in Swift and Objc.
- Swift 3 now adds another related API that is a struct

based wrapper around GCD.
- For simple concurrency you will most likely use GCD (called DispatchQueue in Swift >= 3).
- Use (NS)OperationQueue if you need to do more complex concurrency, like communicating between tasks, and monitoring execution.
- You need your concurrent operations to be objects (Objc). (Why might you need to do this?)
- You need to cancel operations, or you need other kinds of control, like scheduling.
- You need to be notified about the state of operations. ((NS)OperationQueue uses KVO).
- I'm mostly going to focus on GCD (C API in Objc, DispatchQueue Swift 3), since this will be what you will use most often, but we will also look at some simple examples of (NS)OperationQueue.

Creating Or Getting Queues in GCD:

- Note: To see everything GCD can do a search in **Dash** with "*dispatch_*" with ObjC selected as the language.
- 2 ways to get a queue:
  - 1) User created (named).
  - 2) System created (global).
- You are rarely going to use the first option in Objc (can't control the priority), but Swift 3+ has made user created Q's a much more useful option.
- User created background queues look like this:

```

// C Definition of User Created Q's
```

```
3  dispatch_queue_t dispatch_queue_create( const char
   *label, dispatch_queue_attr_t attr);
4
5  // Creation Objc
6  dispatch_queue_t userCreatedBackgroundQ1 =
   dispatch_queue_create("com.lighthouse.threading.1",
   DISPATCH_QUEUE_CONCURRENT);
7
8  dispatch_queue_t userCreatedBackgroundQ2 =
   dispatch_queue_create("com.lighthouse.threading.2",
   DISPATCH_QUEUE_SERIAL);
9
10 // Creation Swift 1.x + 2.x
11
12 let userCreatedBackgroundQ1: dispatch_queue_t  =
   dispatch_queue_create("com.lighthouse.threading.1",
   DISPATCH_QUEUE_CONCURRENT);
13
14 let userCreatedBackgroundQ2: dispatch_queue_t =
   dispatch_queue_create("com.lighthouse.threading.2",
   DISPATCH_QUEUE_SERIAL);
15
16 // Creation Swift 3
17
18 let bgQ1 = DispatchQueue(label: "com.steve.queue")
   // serial by default
19 let bgQ2 = DispatchQueue(label: "com.steve.queue",
   qos: .userInitiated, attributes: .concurrent)
20
```

- Apple has added QOS to the GCD Swift 3 API.
- I will return to this below.

- System created queues (**the one you will mostly use**) look like this:

```
1
2  // Objc Definition
3  dispatch_queue_t dispatch_get_global_queue( long
   identifier, unsigned long flags);
4
5  // Objc Creation
6  dispatch_queue_t sysCreatedbackgroundQ1 =
   dispatch_get_global_queue(QOS_CLASS_USER_INTERACTIVE
   , 0);
7
8  // Swift 1.x + 2.x Creation
9  let sysCreatedbackgroundQ1: dispatch_queue_t =
   dispatch_get_global_queue(QOS_CLASS_USER_INTERACTIVE
   , 0);
10
11 // Swift 3
12 let userInitiatedQOS =
   DispatchQueue.global(qos:.userInitiated)
13 let userInteractiveQOS =
   DispatchQueue.global(qos:.userInteractive)
14 let defaultQOS = DispatchQueue.global() // default
15 let utilityQOS = DispatchQueue.global(qos:.utility)
16 let backgroundQOS =
   DispatchQueue.global(qos:.background)
17 let unspecifiedQOS =
   DispatchQueue.global(qos:.unspecified)
18
```

- dispatch_queue_t is the return type (_t is the way C indicates this is a type).
- dispatch_get_global_queue() takes 2 parameters.
- Always pass 0 for the second param (it's reserved for future use)
- The first param is an enum value that specifies the relative priority, called *Quality Of Service (QOS).*
- There are 4 possible options in Objc:

QOS_CLASS_USER_INTERACTIVE // highest priority
QOS_CLASS_USER_INITIATED // high priority, used when user initiates an action
QOS_CLASS_UTILITY // used for long running tasks
QOS_CLASS_BACKGROUND // used when user doesn't need result

- Swift 3 adds *unspecified* to the QOS.

- You can get ahold of the main queue like this:

```
1
2 // Getting Ref To Main Queue Objc
3 dispatch_queue_t mainQ = dispatch_get_main_queue();
4
5 // Getting Ref To Main Queue Swift 1.x + 2.x
6 let mainQ2: dispatch_queue_t =
  dispatch_get_main_queue();
7
8 // Swift 3
9 let mainQ3 = DispatchQueue.main
10
11
```

Adding Tasks To Queues in GCD:

```

// C Definition
void dispatch_async( dispatch_queue_t queue,
dispatch_block_t block);

// Objc Example
dispatch_queue_t backgroundQ1 =
dispatch_get_global_queue(QOS_CLASS_USER_INTERACTIVE
, 0);

dispatch_async(backgroundQ1, ^{
    printf("%d: 1\n", __LINE__);
});
printf("%d: 2\n", __LINE__);

// Swift 1.x + 2.x Example

let backgroundQ1: dispatch_queue_t =
dispatch_get_global_queue(QOS_CLASS_USER_INTERACTIVE
, 0);

dispatch_async(backgroundQ1){
    print(#line, "1")
}
print(#line, "2");

// Swift 3

DispatchQueue.global(qos:.userInteractive).async {
```

```
25   print(#line, "3")
26 }
27
28 print(#line, "4")
29
```

- `dispatch_async` takes 2 paramaters: the queue and the block to dispatch
- `dispatch_async` returns *immediately*
- What does the above code print and why?
- `dispatch_sync` does not return until the block has run
- You will almost never use `dispatch_sync` except in very advanced situations. (Don't try to handle dependencies using `dispatch_sync`. Use (NS)OperationQueue instead).

- `dispatch_once` is used to execute a block only once during the lifetime of the app.
- `dispatch_once` is used in the singleton pattern in Objc.

```
1
2 // C Definition
3 void dispatch_once( dispatch_once_t *predicate,
  dispatch_block_t block);
4
5 // Objc Creation
6 static dispatch_once_t onceToken = 0;
7
8 // This waits for completion
9 dispatch_once(&onceToken, {
10     printf("this will only execute once")
```

```
});

dispatch_once(&onceToken, {
    printf("this will never execute!")
})

// Swift 1.x + 2.x Creation

var onceToken: dispatch_once_t = 0;

// This waits for completion
dispatch_once(&onceToken, {
    print("this will only execute once")
});

dispatch_once(&onceToken, {
    print("this will never execute!")
})

// NB. xCode includes a code snippet with this.

// Swift 3

???

```

Never Do This

- Never call sync on a background Q. Why?

```swift
// Swift 3
DispatchQueue.global().sync {
    print("💀")
}
```

- Never call sync on the mainQ since this will deadlock.

```swift
// Swift 3
DispatchQueue.main.sync {
    print("☠️")
}
```

Questions:

1) Is the mainQ serial or concurrent?
2) What's the difference between serial/concurrent and sync/async?
3) Since a call to a background Q will return at some unpredictable time in the future what are the different techniques you can use to update a view once the task is completed?