**Concurrency Using GCD**

**Concurrency Options**

Concurrency is extremely important on iOS because it allows an app to stay fluid during resource-intensive computations. There are four main concurrency options on iOS:

* **Grand Central Dispatch (GCD)**

The easiest option to use is Grand Central Dispatch, or GCD. GCD is the best option for simple parallelization, where some operation needs to be run in the background. Other common uses for GCD include serializing data to disk or performing simple background computations.

* **Operation**

The Operation class provides a layer of complex functionality above GCD, and in fact is built on top of GCD. Operations allow for subclassing, complex dependency graphs, and cancellation.

* **NSThread**

NSThreads are the best option when direct control of a thread is needed – for example, when thread priorities need to be manually controlled. This is particularly useful in real-time applications.

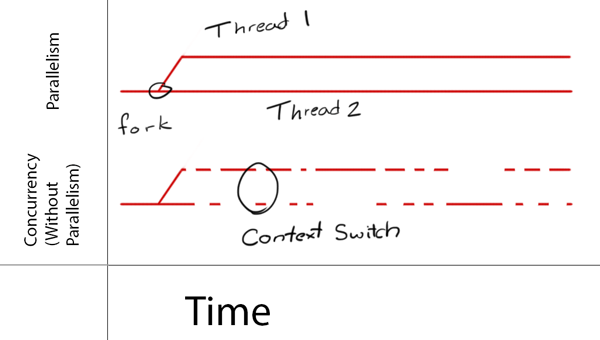
* async / await

The new async / await syntax introduced in Swift 5.5 / iOS 15 enables us to annotate asynchronous functions (or computed properties) with the async keyword, which in turn requires us to use the await keyword when calling them. At that point, the system will automatically manage all of the complexity that’s involved in waiting for such an asynchronous call to complete, without blocking any other outside code from executing.

**Concurrency**

In iOS, a process or application is made up of one or more threads. The threads are managed independently by the operating system scheduler. Each thread can execute concurrently but it’s up to the system to decide if this happens and how it happens.

Single-core devices can achieve concurrency through time-slicing. They would run one thread, perform a context switch, then run another thread.



Multi-core devices on the other hand, execute multiple threads at the same time via parallelism.

GCD is built on top of threads. Under the hood it manages a shared thread pool. With GCD you add blocks of code or work items to **dispatch queues** and GCD decides which thread to execute them on.

As you structure your code, you’ll find code blocks that can run simultaneously and some that should not. This then allows you to use GCD to take advantage of concurrent execution.

Note that GCD decides how much parallelism is required based on the system and available system resources. It’s important to note that parallelism *requires* concurrency, but concurrency does not *guarantee* parallelism.

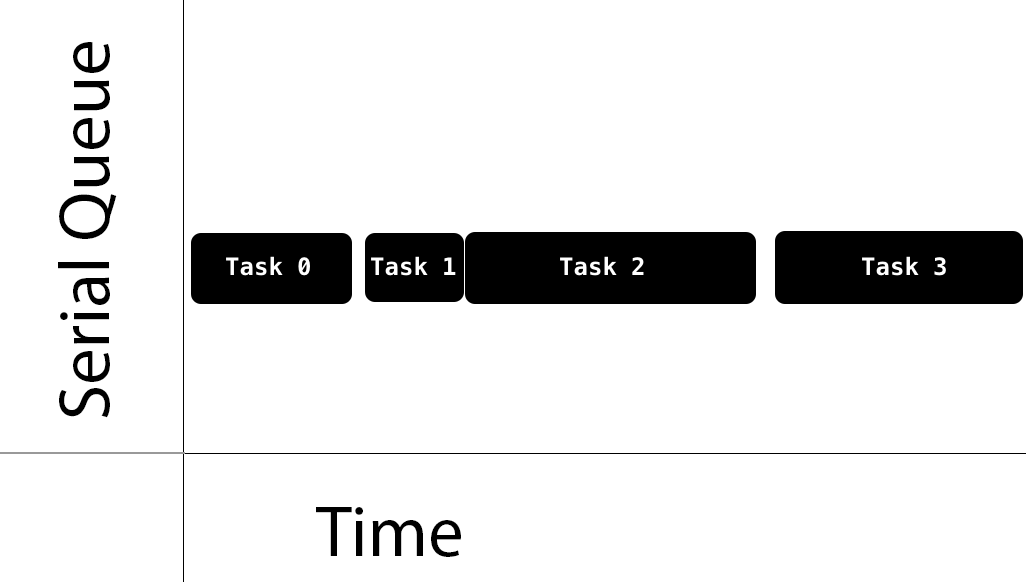
Basically, concurrency is about *structure* while parallelism is about *execution*.

**Grand Central Dispatch (GCD)**

GCD provides dispatch queues represented by the class DispatchQueue to manage tasks you submit and execute them in a FIFO order guaranteeing that the first task submitted is the first one started.

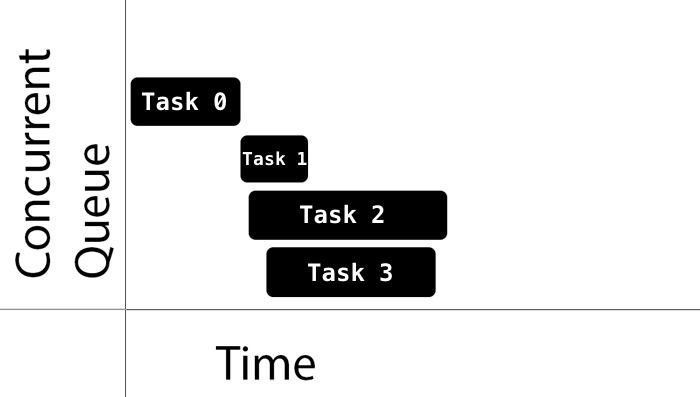
Dispatch queues are thread-safe which means that you can access them from multiple threads simultaneously. The benefits of GCD are apparent when you understand how dispatch queues provide thread safety to parts of your own code. The key to this is to choose the right *kind* of dispatch queue and the right *dispatching function* to submit your work to the queue.

Queues can be either *serial* or *concurrent*. Serial queues guarantee that only one task runs at any given time. GCD controls the execution timing. You won’t know the amount of time between one task ending and the next one beginning:



Concurrent queues allow multiple tasks to run at the same time. Tasks are guaranteed to start in the order they were added. Tasks can finish in any order and you have no knowledge of the time it will take for the next task to start, nor the number of tasks that are running at any given time.

See the sample task execution below:



Notice how Task 1, Task 2, and Task 3 start quickly one after the other. On the other hand, Task 1 took a while to start after Task 0. Also notice that while Task 3 started after Task 2, it finished first.

The decision of when to start a task is entirely up to GCD. If the execution time of one task overlaps with another, it’s up to GCD to determine if it should run on a different core, if one is available, or instead to perform a context switch to run a different task.

GCD provides three main types of queues:

1. **Main queue**: runs on the main thread and is a serial queue.
2. **Global queues**: concurrent queues that are shared by the whole system. There are four such queues with different priorities: high, default, low, and background. The background priority queue is I/O throttled.
3. **Custom queues**: queues that you create which can be serial or concurrent. These actually trickle down into being handled by one of the global queues.

When setting up the global concurrent queues, you don’t specify the priority directly. Instead you specify a Quality of Service (QoS) class property. This will indicate the task’s importance and guide GCD into determining the priority to give to the task.

The QoS classes are:

* **User-interactive**: This represents tasks that need to be done immediately in order to provide a nice user experience. Use it for UI updates, event handling and small workloads that require low latency. The total amount of work done in this class during the execution of your app should be small. This should run on the main thread.
* **User-initiated**: The represents tasks that are initiated from the UI and can be performed asynchronously. It should be used when the user is waiting for immediate results, and for tasks required to continue user interaction. This will get mapped into the high priority global queue.
* **Utility**: This represents long-running tasks, typically with a user-visible progress indicator. Use it for computations, I/O, networking, continuous data feeds and similar tasks. This class is designed to be energy efficient. This will get mapped into the low priority global queue.
* **Background**: This represents tasks that the user is not directly aware of. Use it for prefetching, maintenance, and other tasks that don’t require user interaction and aren’t time-sensitive. This will get mapped into the background priority global queue.

### Synchronous vs. Asynchronous

With GCD, you can dispatch a task either synchronously or asynchronously.

* A *synchronous* function returns control to the caller after the task is completed.
* An *asynchronous* function returns immediately, ordering the task to be done but not waiting for it. Thus, an asynchronous function does not block the current thread of execution from proceeding on to the next function.

### Managing Tasks

For our purposes, a task is effectively a **closure**. Closures are self-contained, callable blocks of code that can be stored and passed around.

Tasks that you submit to a DispatchQueue are encapsulated by a DispatchWorkItem. You can configure the behavior of a DispatchWorkItem such as its QoS class or whether to spawn a new detached thread.

In a Swift program, GCD is easy to use because of the closure-based interface. In general, you’ll want to use the DispatchQueue method async when you need to perform a network-based or CPU intensive task in the background and not block the current thread.

Here’s a quick guide of how and when to use the various queues with async:

* ***Main Queue***: This is a common choice to update the UI after completing work in a task on a concurrent queue. To do this, you’ll code one closure inside another. Targeting the main queue and calling async guarantees that this new task will execute sometime after the current method finishes.
* ***Global Queue***: This is a common choice to perform non-UI work in the background.
* ***Custom Serial Queue***: A good choice when you want to perform background work serially and track it. This eliminates resource contention since you know only one task at a time is executing. Note that if you need the data from a method, you must inline another closure to retrieve it or consider using sync.

The Swift syntax for using async is shown below. Simply embed the code you want to run asynchronously inside the curly braces. See the **Primes** app example for an illustration of these two techniques in use.

**Swift**

// Dispatch an asynchronous operation from some thread to the main thread

DispatchQueue.main.async {

...

}

// Dispatch an asynchronous operation with background QOS on the global background queue

// There are four commonly-used values for qos, as described above:

//

// - DispatchQoS.QoSClass.userInteractive

// - DispatchQoS.QoSClass.userInitiated

// - DispatchQoS.QoSClass.utility

// - DispatchQoS.QoSClass.background

DispatchQueue.global(qos: .background).async {

...

}