4 Groups & Semaphores

Written by Scott Grosch

Sometimes, instead of just tossing a job into a queue, you need to process a group of jobs. They don't all have to run at the same time, but you need to know when they have all completed. Apple provides **Dispatch Groups** for this exact scenario.

DispatchGroup

The aptly named `DispatchGroup` class is what you'll use when you want to track the completion of a *group* of tasks.

You start by initializing a `DispatchGroup`. Once you have one and want to track a task as part of that group, you can provide the group as an argument to the `async` method on any dispatch queue:

```
let group = DispatchGroup()

someQueue.async(group: group) { ... your work ... }
someQueue.async(group: group) { ... more work .... }
someOtherQueue.async(group: group) { ... other work ... }

group.notify(queue: DispatchQueue.main) { [weak self] in
  self?.textLabel.text = "All jobs have completed"
}
```

As seen in the example code above, groups are not hardwired to a single dispatch queue. You can use a single group, yet submit jobs to multiple queues, depending on the priority of the task that needs to be run. `DispatchGroup`s provide a `notify(queue:)` method, which you can use to be notified as soon as every job submitted has finished.

> **Note**: The notification is itself asynchronous, so it's possible to submit more jobs to the group after calling notify, as long as the previously submitted jobs have not already completed.

You'll notice that the `notify` method takes a dispatch queue as a parameter. When the jobs are all finished, the closure that you provide will be executed in the indicated dispatch queue. The `notify` call shown is likely to be the version you'll use most often, but there are a couple other versions which allow you to specify a quality of service as well, for example.

Synchronous waiting

There be dragons here!

If, for some reason, you can't respond asynchronously to the group's completion notification, then you can instead use the `wait` method on the dispatch group. This is a *synchronous* method that will block the current queue until all the jobs have finished. It takes an optional parameter which specifies how long to wait for the tasks to complete. If not specified then there is an infinite wait time:

```
let group = DispatchGroup()

someQueue.async(group: group) { ... }
someQueue.async(group: group) { ... }
someOtherQueue.async(group: group) { ... }

if group.wait(timeout: .now() + 60) == .timedOut {
  print("The jobs didn't finish in 60 seconds")
}
```

> **Note**: Remember, this *blocks* the current thread; never **ever** call `wait` on the main queue.

In the above example, you're giving the tasks up to 60 seconds to complete their work before `wait` returns.

It's important to know that the jobs *will* still run, even after the timeout has happened. To see this in practice, go to the starter projects in this chapter's download materials and open the playground named **DispatchGroup.playground**.

In the playground, the code adds two jobs to a dispatch group: one that takes 10 seconds (job 1) and another one that takes two seconds to complete:

```
let group = DispatchGroup()
let queue = DispatchQueue.global(qos: .userInitiated)

queue.async(group: group) {
  print("Start job 1")
  Thread.sleep(until: Date().addingTimeInterval(10))
  print("End job 1")
 }

queue.async(group: group) {
  print("Start job 2")
  Thread.sleep(until: Date().addingTimeInterval(2))
  print("End job 2")
 }
```

It then synchronously waits for the group to complete:

```
if group.wait(timeout: .now() + 5) == .timedOut {
  print("I got tired of waiting")
} else {
  print("All the jobs have completed")
}
```

Run the playground and look at the output on the right side of the Xcode window. You'll immediately see messages telling you that jobs 1 and 2 have started. After two seconds, you'll see a message saying job 2 has completed, and then three seconds later a message saying, "I got tired of waiting."

You can see from the sample that job 2 only sleeps for two seconds and that's why it can complete. You specified five total seconds of time to wait, and that's not enough for job 1 to complete, so the timeout message was printed.

However, if you wait another five seconds — you've already waited five and job 1 takes ten seconds — you'll see the completion message for job 1.

At this point, calling a synchronous wait method like this should be a **code smell** to you, potentially pointing out other issues in your architecture. Sure, it's much easier to implement synchronously, but the entire reason you're reading this book is to learn how to make your app perform as fast as possible. Having a thread just spin and continually ask, "Is it done yet?" isn't the best use of system resources.

Wrapping asynchronous methods

A dispatch queue natively knows how to work with dispatch groups, and it takes care of signaling to the system that a job has completed for you. In this case, **completed** means that the code block has run its course. Why does that matter? Because if you call an asynchronous method inside of your closure, then the closure will *complete* before the internal asynchronous method has completed.

You've got to somehow tell the task that it's not done until those internal calls have completed as well. In such a case, you can call the provided `enter` and `leave` methods on `DispatchGroup`. Think of them like a simple count of running tasks. Every time you `enter`, the count goes up by `1`. When you `leave`, the count goes down by `1`:

```
queue.dispatch(group: group) {
  // count is 1
  group.enter()
  // count is 2
  someAsyncMethod {
    defer { group.leave() }

    // Perform your work here,
    // count goes back to 1 once complete
  }
}
```

By calling `group.enter()`, you let the dispatch group know that there's another block of code running, which should be counted towards the group's overall completion status. You, of course, have to pair that with a corresponding `group.leave()` call or you'll never be signaled of completion. Because you have to call `leave` even during error conditions, you will usually want to use a `defer` statement, as shown above, so that, no matter how you exit the closure, the `group.leave()` code executes.

In a simple case similar to the previous code sample, you can simply call the `enter` / `leave` pairs directly. If you're going to use `someAsyncMethod` frequently with dispatch groups, you should wrap the method to ensure you never forget to make the necessary calls:

```
func myAsyncAdd(
  lhs: Int,
  rhs: Int,
  completion: @escaping (Int) -> Void) {
  // Lots of cool code here
  completion(lhs + rhs)
}

func myAsyncAddForGroups(
  group: DispatchGroup,
  lhs: Int,
  rhs: Int,
  completion: @escaping (Int) -> Void) {
    group.enter()

    myAsyncAdd(first: first, second: second) { result in
      defer { group.leave() }
      completion(result)
    }
}
```

The wrapper method takes a parameter for the group that it will count against, and then the rest of the arguments should be exactly the same as that of the method you're wrapping. There's nothing special about wrapping the async method other than being 100% sure that the group `enter` and `leave` methods are properly handled.

If you write a wrapper method, then testing — you do test, right? — is simplified to a single location to validate proper pairing of `enter` and `leave` calls in all utilizations.

Downloading images

Performing a network download should always be an asynchronous operation. This book's technical editor once had an assignment that required him to download all of the player's avatars before presenting the user with a list of players and their images. A dispatch group is a perfect solution for that task.

Please switch to the playground named **Images.playground** in this chapter's startup folder. Your task is to download each image from the provided `names` array in an asynchronous manner. When complete, you should show at least one of the images and terminate the playground. Take a moment to try and write the code yourself before continuing.

How'd you do? Clearly you're going to have to loop through the images to generate a URL, so start with that:

```
for id in ids {
  guard let url = URL(string: "\(base)\(id)-jpeg.jpg") else { continue }
```

Now that you've got a valid URL, call `URLSession`'s `dataTask` method. That's asynchronous already, so you'll need to handle the group's entry and exit:

```
group.enter()

let task = URLSession.shared.dataTask(with: url) {
  data, _, error in
```

As always, with asynchronous code, the `defer` statement is going to be your friend. Now that you've started the asynchronous task, regardless of how it exits, you've got to tell the dispatch group that the task has completed. If you don't, the app will hang forever waiting for completion:

```
defer { group.leave() }
```

After that, it's just a matter of converting the image and adding it to the array:

```
    if error == nil, let data = data, let image = UIImage(data: data) {
      images.append(image)
    }
  }

  task.resume()
}
```

Due to properly handling the `enter` and `leave` pairs, you no longer have to spin and wait synchronously for the groups to enter. Use the `notify(queue:)` callback method to be informed when all image downloads have completed. Add this code outside the `for` loop:

```
group.notify(queue: queue) {
  images[0]

  PlaygroundPage.current.finishExecution()
}
```

If you run the playground now and watch the sidebar, you'll see each job starting, the images downloading, and eventually the notification triggering with the first image of the bunch.

Semaphores

There are times when you really need to control how many threads have access to a shared resource. You've already seen the read/write pattern to limit access to a single thread, but there are times when you can allow more resources to be used at once while still maintaining control over the total thread count.

If you're downloading data from the network, for example, you may wish to limit how many downloads happen at once.

You'll use a dispatch queue to offload the work, and you'll use dispatch groups so that you know when all the downloads have completed. However, you only want to allow four downloads to happen at once because you know the data you're getting is quite large and resource-heavy to process.

By using a `DispatchSemaphore`, you can handle exactly that use case. Before any desired use of the resource, you simply call the `wait` method, which is a synchronous function, and your thread will pause execution until the resource is available. If nothing has claimed ownership yet, you immediately get access. If somebody else has it, you'll wait until they `signal` that they're done with it.

When creating a **semaphore**, you specify how many concurrent accesses to the resource are allowed. If you wish to enable four network downloads at once, then you pass in `4`. If you're trying to lock a resource for exclusive access, then you'd just specify `1`.

Open up the playground named **Semaphores.playground** and you'll find some simple boilerplate code to set up the group and queue. After the line that assigns the dispatch queue, create a semaphore that allow four concurrent accesses:

```
let semaphore = DispatchSemaphore(value: 4)
```

You'll want to simulate performing 10 network downloads, so create a loop that dispatches onto the queue, using the group. Right after creating the semaphore, implement the loop:

```
for i in 1...10 {
  queue.async(group: group) {

  }
}
```

There shouldn't be anything surprising there as it's the same type of code you just saw. On each download thread, you now want to ask for permission to use the resource. To simulate the network download, you can just have the thread sleep for three seconds. Insert this code inside the `async` block:

```
semaphore.wait()
defer { semaphore.signal() }

print("Downloading image \(i)")

// Simulate a network wait
Thread.sleep(forTimeInterval: 3)

print("Downloaded image \(i)")
```

Just as you had to be sure to call `leave` on a dispatch group, you'll want to be sure to signal when you're done using the resource. Using a `defer` block is the best option as there's then no way to leave without letting go of the resource.

If you run the playground, you should immediately see that four **downloads** happen, then, three seconds later, another four occur. Finally, three seconds after that, the final two complete.

That's a useful example just to prove that the semaphores are doing their job of limiting access to the network. However, you need to actually download something!

Remove everything in the playground after the creation of the `semaphore` variable and then copy the code from the **Images.playground** starting with the `let base` statement, and paste it immediately after creating the semaphore. The resource you're trying to control is the network, so you can let the URL get created in the `for` loop but, before you enter the group, you'll need to wait for an available semaphore, so add in a semaphore call just before `group.enter()`:

```
semaphore.wait()
```

You need to use both elements because the semaphore controls access to the resource, and the dispatch group is how you are tracking completion. Modify the `defer` statement as well to handle the release of the semaphore:

```
defer {
  group.leave()
  semaphore.signal()
}
```

The order of the lines doesn't really matter; I just like to have the semaphore be the **outer** element that starts and ends the task. Update the `DispatchSemaphore` to have a value of `2` instead of `4` and then run the playground. You should see it work as before, albeit slower due to the limitation of just two downloads happening at once.

Think of the semaphore itself as being some type of resource. If you have three hammers and four saws available, you'd want to create two semaphores to represent them:

```
let hammer = DispatchSemaphore(value: 3)
let saw = DispatchSemaphore(value: 4)
```

Where to go from here?

Modify the various values in the playgrounds to be sure you understand how both groups and semaphores work.

Can you think of cases in your previous or current apps that might have benefited from either one? Don't be concerned if you can't think of a use case for semaphores. They're an advanced topic that very rarely comes up in daily programming, but it's good to know they exist when you need them.

Now that you've seen how great concurrency with GCD can be, it's time to talk about some of the negative aspects.

Completed