Concurrency
CMSC 436
Dispatch Group Example

```swift
var group = DispatchQueueGroup()
var count = 0

for _ in 0..<10 {
    group.enter()
    sleep(5)
    count += 1
    group.leave()
}

group.wait()
print(count)
```

Takes 50 seconds!

(note `.background` vs `.utility`)

```swift
var group = DispatchQueueGroup()
var count = 0

for i in 0..<10 {
    group.enter()
    if i % 2 == 0 {
        DispatchQueue.global(qos: .background).async {
            sleep(5)
            count += 1
            group.leave()
        }
    } else {
        DispatchQueue.global(qos: .utility).async {
            sleep(5)
            count += 1
            group.leave()
        }
    }
}

group.wait()
print(count)
```

Takes about 5 seconds

DispatchGroup creates a logical barrier.
Barriers

DispatchGroup creates **cross-thread/cross-queue** sync

Execution waits for all group tasks

Sometimes we want a barrier in a **single queue**:

1. Start a bunch of concurrent tasks
2. Wait for them to complete
3. Start more concurrent tasks

"Use a barrier to synchronize the execution of one or more tasks in your dispatch queue. When you add a barrier to a concurrent dispatch queue, the queue delays the execution of the barrier block (and any tasks submitted after the barrier) until all previously submitted tasks finish executing. After the previous tasks finish executing, the queue executes the barrier block by itself. Once the barrier block finishes, the queue resumes its normal execution behavior."

**Note:** does not work on global queues.
import Foundation

var count = 0
var queue = DispatchQueue(label: "queue", attributes: .concurrent)
for _ in 0..<10 {
    queue.async {
        sleep(UInt32.random(in: 0..<5))
        let x = count
        sleep(UInt32.random(in: 0..<5))
        count = x + 1
        print("counting up: \(count)")
    }
}
queue.async(flags: .barrier) {
    print("after first barrier: \(count)")
}
for _ in 0..<10 {
    queue.async {
        sleep(UInt32.random(in: 0..<5))
        let x = count
        sleep(UInt32.random(in: 0..<5))
        count = x-1
        count -= 1
        print("counting down: \(count)")
    }
}
print("not in queue: \(count)")
queue.async(flags: .barrier) {
    print("after second barrier: \(count)")
}
Running our Example

counting up: 1
not in queue: 1
counting up: 2
counting up: 3
counting up: 2
counting up: 3
counting up: 1
counting up: 2
counting up: 3
counting up: 3
after first barrier: 3
counting down: 1
counting down: 1
counting down: -1
counting down: -1
counting down: -1
counting down: -1
after second barrier: -1
not in queue: 0
counting up: 1
counting up: 1
counting up: 2
counting up: 1
counting up: 2
counting up: 1
counting up: 3
counting up: 1
counting up: 2
counting up: 3
after first barrier: 3
counting down: 1
counting down: 1
counting down: -1
counting down: -3
counting down: -5
counting down: -1
counting down: 1
counting down: -1
counting down: -1
counting down: -3
after second barrier: -1

Why are these different?!

Previous version of these slides in class was wrong.
Race Conditions

- Execute tasks concurrently
- Reading/writing a shared value

Some writes might have stale values!

```
x = read(count) \Rightarrow 0
   y = x + 1 \Rightarrow 1

x = read(count) \Rightarrow 0
   y = x + 1 \Rightarrow 1
   write(count, y) \Rightarrow 1

x = read(count) \Rightarrow 1
   y = x + 1 \Rightarrow 2
   x = read(count) \Rightarrow 1
   y = x + 1 \Rightarrow 2
   write(count, y) \Rightarrow 1
   write(count, y) \Rightarrow 2
```
Semaphores and Mutual Exclusion

A *mutual exclusion* (or *mutex*) excludes all but one process/thread from accessing a resource.

Swift uses *semaphores* to implement this (they can do other things, too).

`DispatchSemaphore(value: count)` creates a semaphore with count *copies* of a *resource*.

- `wait()` Requests a semaphore, returns when value > 0
- `signal()` Releases a resource, incrementing value
- `wait(timeout: t)` Returns whether resource acquired in t:
  - `DispatchTime`
Adding Semaphores

```swift
import Foundation
var count = 0
var queue = DispatchQueue(label: "queue", attributes: .concurrent)
var sem = DispatchSemaphore(value: 1)
for _ in 0..<100000 {
    queue.async {
        sem.wait()
        count += 1
        sem.signal()
    }
}
queue.async(flags: .barrier) {
    sem.wait()
    print("after first barrier: \(count)")
    sem.signal()
}
for _ in 0..<100000 {
    queue.async {
        sem.wait()
        count -= 1
        sem.signal()
    }
}
print("not in queue: \(count)")
queue.async(flags: .barrier) {
    sem.wait()
    print("after second barrier: \(count)")
    sem.signal()
}
```
Running With and Without Semaphores

- As expected, w/o semaphores you get random final values, from -146 to 123 in my three runs.
- W/ semaphores all is well.
URL Sessions

We often need remote resources

We often need a bunch of remote resources

We don’t want to update the UI until we’ve received them all

URLSession groups URL requests

- Does not run in main queue
- Caches responses
- Supports cookies and credentials
- Creates tasks for uploading/downloading
URL Session Tasks

URLSession creates URLSessionTasks:

- **dataTask**  Retrieve the contents of a URL
- **downloadTask**  Like dataTask, but stores in a file
- **uploadTask**  Uploads data to a URL from memory or file
- **streamTask**  Retrieves streaming data

These tasks have several variants

- URL (simple, default values)
- **URLRequest** (required for customizing, using for *POST* etc.)
- With or without a *completion handler* (a closure)
URL Session Tasks

Tasks are run asynchronously

Not on main queue

Not run on creation!

resume() Run the task now
suspend() Temporarily suspend the task
cancel() Permanently cancel the task

If your completion handler affects UI, it should schedule a new task on DispatchQueue.main
URL Session Example

The code:

```swift
var session = URLSession(configuration: .default)
let task = session.dataTask(with: URL(string: "https://www.cs.umd.edu/~keleher")!) {
    data, response, err in
    if let data = data {
        print(String(data: data, encoding: .ascii)!) // This will print the HTML content from the URL
    }
}
task.resume()
```

The result:

```html
<html lang="en">
<head>
    <title>Pete Keleher</title>
</head>
<body>
    <h1>Pete Keleher</h1>
    <p>
        Prof<br/>
        Department of Computer Science<br/>
        5146 Iribe Center<br/>
        keleher AT umd.edu<br/>
        ...
    </p>
</body>
</html>
```