Concurrency

CMSC 436

Overview

Concurrency is a big topic

We will look at:

- How Swift handles conflicting accesses
- Dispatch Queues and Groups
- Barriers
- Race Conditions and Semaphores
- URL Sessions
- Deadlock
- Priority Inversion
- Operation Queues

Conflicting Accesses

A *conflicting access* occurs when

- two (or more) threads access the same location
- the accesses overlap in time
- *at least* one is a write

Most accesses are *effectively* synchronous

Cases when they are not:

- longer-term access
- `inout` parameters
- value-based data

An Example of a Conflict

Consider the following example:

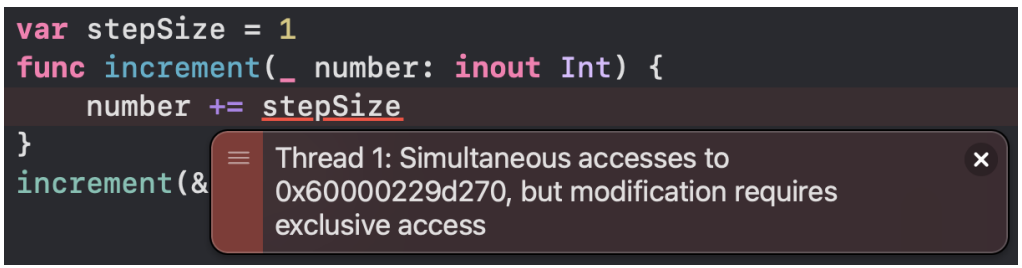
```
struct Item {
   var name: String
   var cost: Int
}

struct Order {
   var items: [Item] = []
   var total: Int = 0

   mutating func addItems(_ newItems: [Item]) {
      items += newItems
      total = items.reduce(0) { res,item in res+item.cost }
   }
}
```

While we're adding items, the total is incorrect!

A Single-Threaded Conflict

```
var stepSize = 1

func increment(_ number: inout Int) {
    number += stepSize
}

increment(&stepSize)
```

This will generate a runtime error!



We can fix this with explicit copies:

```
var copy = stepSize
increment(&copy)
stepSize = copy
```

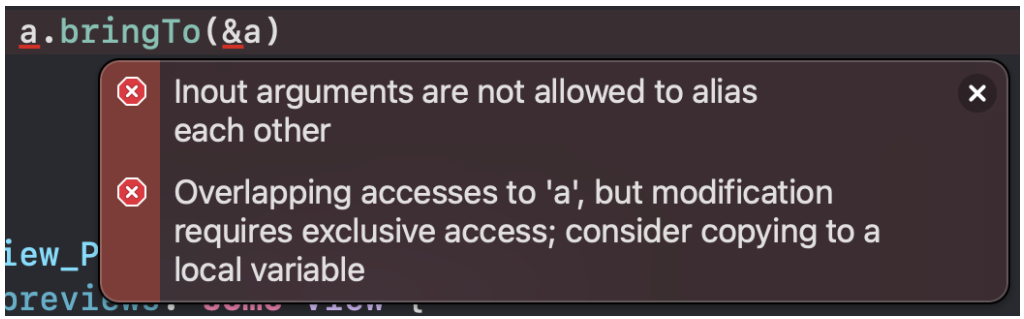
Another Single-Threaded Conflict

If we have multiple `inout` parameters, we can't duplicate them!

```
func balance(_ x: inout Int, _ y: inout Int) {
   let sum = x+y
   x = sum/2
   y = sum - x
}

var a = 20;
var b = 13;
balance(&a,&b) // a=>16, b=>17
balance(&a,&a) // Conflicting accesses!
```

Conflicting Access to `self`

```
struct Point {
    var x: Double
    var y: Double

     func average(_ a: inout Double, _ b: inout Double) {
         let sum = a + b
         a = sum/2
         b = sum - a
     }
     mutating func bringTo(_ other: inout Point) {
         average(&self.x, &other.x)
         average(&self.y, &other.y)
     }
}

var a = Point(x:3.0,y:5.0)
a.bringTo(&a)
```



Conflicting Access to Properties

Mutating part of a struct, enum, or tuple (value types) mutates the *entire* thing

```
var info = (a: 3,b: 5)
balance(&info.a,&info.b)
```

```
Simultaneous accesses to 0x10a9c5090, but modification requires exclusive access.
Previous access (a modification) started at  (0x10a9c6397).
Current access (a modification) started at:
0    libswiftCore.dylib                 0x00007fff2ff7be50 swift_beginAccess + 568
2    Untitled Page 2                    0x0000000108163150 main + 0
3    CoreFoundation                     0x00007fff20390114 __CFRUNLOOP_IS_CALLING_OUT_TO_A_BL
4    CoreFoundation                     0x00007fff2038f382 __CFRunLoopDoBlocks + 434
5    CoreFoundation                     0x00007fff20389bc1 __CFRunLoopRun + 899
6    CoreFoundation                     0x00007fff2038949f CFRunLoopRunSpecific + 567
7    GraphicsServices                   0x00007fff2c257d28 GSEventRunModal + 139
8    UIKitCore                          0x00007fff24696967 -[UIApplication _run] + 912
9    UIKitCore                          0x00007fff2469bb43 UIApplicationMain + 101
10   Untitled Page 2                    0x0000000108163150 main + 194
11   libdyld.dylib                      0x00007fff2025a3e8 start + 1
Fatal access conflict detected.
```

Preventing Conflicting Accesses

Swift *proves* memory safety

Locals are less restricted than globals

```
func foo() {
   var info = (a: 3,b: 5)
   balance(&info.a,&info.b)
}

foo()
```

Compiler can prove this is safe!

Compiler Safety Proofs

Compiler only allows accesses it can prove are safe

Exclusive access is stronger than *memory safety*

- Easier to prove, though

Compiler can prove exclusive access iff:

- Accessing only stored properties
- Local variables
- Not captured by *escaping closure*
  - Passed to a function
  - Stored or returned as function value.

Multi-threading

How do we run multiple things at once?

Usually threads!

Great for making effective use of multiple CPUs/cores

Expensive (kernel threads)

In mobile environment, we want *fewer* threads

In iOS, we manage threads with **Grand Central Dispatch**

Grand Central Dispatch

We've already encountered this:
`DispatchQueue.global(qos: .background).async`

In addition to queues, we have

- Lock-based synchronization
- Barriers
- etc.

Dispatch Queues Revisited

Tasks can be *synchronous* or *asynchronous*

Main queue

- Serial execution
- UI activity should go here
- Anything long-running or non-UI should *not* go here!

Global queues

- Concurrent execution
- This is where non-UI stuff goes
- Specify a *quality of service* (QoS)

Global Queue QoS

`DispatchQoS.QoSClass` determines the priority

From highest to lowest:

`userInteractive` things the user interacts with (eg, animations)

`userInitiated` for tasks that prevent user actions from occuring (eg, reading a document)

`default` for active tasks on user's behalf

`utility` for tasks the user doesn't actively track (system stuff, eg networking or continuous data feeds)

`background` for maintenance/cleanup tasks

`unspecified` effectively `nil` — there is no QoS class (system determines the priority)

Synchrony, Asynchrony, Serial, and Concurrent

Synchronous tasks block the caller until they complete

```
queue.sync { ... }
```

Asynchronous tasks return to the caller immediately; run in a separate thread

```
queue.async { ... }
```

Serial execution processes tasks one at a time, in-order

```
DispatchQueue.main
```

Concurrent execution processes tasks in separate threads (or otherwise interleaved) at the "same" time

```
DispatchQueue.global
```

Custom Queues

You can create your own queues

```
let serialQ = DispatchQueue(label: "MySerialQueue")

let concurrentQ = DispatchQueue(label: "MyConcurrentQueue",
                                attributes: .concurrent)
```

Serial queues will execute in a single thread

Concurrent queues will execute in multiple threads

Creating a Task

We've seen task creation before:

```
myQueue.async { [weak self] in
    doSomething()
}
```

Closure will be run in queue-servicing thread

Capturing references weakly is almost always a good idea!

Tasks Creating Tasks

Tasks can enqueue other tasks

- Break up larger task into smaller units (be careful with task boundaries!)
- Parts of task might need to be on different queues

```
DispatchQueue.global(qos: .utility).async {
    [weak self] in
    guard let self = self else { return }
    doSomethingLowPriority()
    DispatchQueue.main.async { [weak self] in
        guard let self = self else { return }
        self.updateUI()
    }
}
```

Dispatch Groups

`DispatchGroup` creates a group of asynchronous tasks

Allows you to wait for all of them to complete

- `init()` Creates a new group
- `enter()` Adds subsequent statements to a task (increments task count)
- `leave()` Ends a block of statements (decrements task count)
- `wait()` Block until all group tasks complete
- `notify(queue: DispatchQueue, work: DispatchWorkItem)` Schedules a task to run when all group tasks complete

`wait()` and `notify()` have versions that take other parameters

Dispatch Group Example

```
var group = DispatchGroup()
var count = 0

for _ in 0..<10 {
   group.enter()


   sleep(5)
   count += 1
   group.leave()

}

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

Takes 50 seconds!

```
var group = DispatchGroup()
var count = 0

for _ in 0..<10 {
   group.enter()
   DispatchQueue.global(
      qos: .background).async {
         sleep(5)
         count += 1
         group.leave()
      }
}

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

Takes about 5 seconds

`DispatchGroup` creates a *barrier*