Concurrency

CMSC 436

Deadlock

Let's say we have two semaphores `semA` and `semB`

Common when need exclusive access to multiple resources

We also have two concurrent tasks `task1` and `task2`

Avoiding Deadlock

This is *hard*

- Imposing a global order on locks will avoid it
- "Always acquire `semA` *before* `semB`"
- What if you only *need* `semB`?

Avoiding Deadlock

Try to avoid holding multiple locks:

1. Acquire first lock
2. Copy data you need from first resource to local variable
3. Release first lock
4. Acquire second lock
5. Do what you need to with second resource
6. Release second lock

Take **CMSC433 or CMSC424** if you want to know (a *lot*) more!

Priority Inversion

Asynchronous tasks assigned *priorities*

Higher-priority tasks should run before lower-priority tasks

Semaphores can mess this up!

Priority Inversion Example

```
let high = DispatchQueue.global(qos: .userInteractive)
let medium = DispatchQueue.global(qos: .userInitiated)
let low = DispatchQueue.global(qos: .background)

let semaphore = DispatchSemaphore(value: 1)

high.async {
    Thread.sleep(forTimeInterval: 2)
    semaphore.wait()
    defer { semaphore.signal() }
    print("Executing high")
}

for i in 1...5 {
    medium.async {
        let waitTime = Double.random(in:0...7)
        semaphore.wait()
        defer { semaphore.signal() }
        print("Executing medium \(i)")
        Thread.sleep(forTimeInterval: waitTime)
    }
}

low.async {
    semaphore.wait()
    defer { semaphore.signal() }
    print("Executing low")
    Thread.sleep(forTimeInterval: 5)
}
```

Priority Inversion Output

```
Executing medium 2
Executing medium 3
Executing medium 1
Executing medium 4
Executing medium 5
Executing low
Executing high
```

All low- and medium-priority tasks enter semaphore's FIFO before high-priority task

Operations

Concurrency without GCD (but built on top of it)

Allows for dependencies between operations

Functional-programming-style passing of results from one to another

Reusable, cancellable

Must subclass `Operation`

Convenience subclass `BlockOperation`

`BlockOperation`

Construct an operation (with potentially multiple tasks)

Schedule it with `start()`

- Can provide a single closure during initialization
- Can add closures later
- Can wait for completion

BlockOperation Example

```
import Foundation

var countOperation = BlockOperation()
for i in 0...10 {
    countOperation.addExecutionBlock {
        print("count \(i)")
    }
}

countOperation.completionBlock = {
    print("All done!")
}

countOperation.start()
```

```
count 5
count 1
count 0
count 6
count 3
count 2
count 4
count 7
count 8
count 10
count 9
All done!
```

Operation Queues

- Multiple operations, like tasks in a `DispatchQueue`
- Supports *dependencies* between operations
- Can *cancel* dependent operations from an individual operation
- Can limit amount of concurrency
- Lots of other functionality

OperationQueue Example

The code:

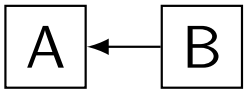
```
let opQueue = OperationQueue()
opQueue.maxConcurrentOperationCount = 2

let op1 = BlockOperation {
    print("op1")
}
let op2 = BlockOperation {
    print("op2")
}
let op3 = BlockOperation {
    print("op3")
}

op2.addDependency(op1)
op2.addDependency(op3)

op3.addDependency(op1)
op3.addDependency(op2)

opQueue.addOperation(op1)
opQueue.addOperation(op2)
opQueue.addOperation(op3)

opQueue.waitUntilAllOperationsAreFinished()
print("All done!")
```

The output:

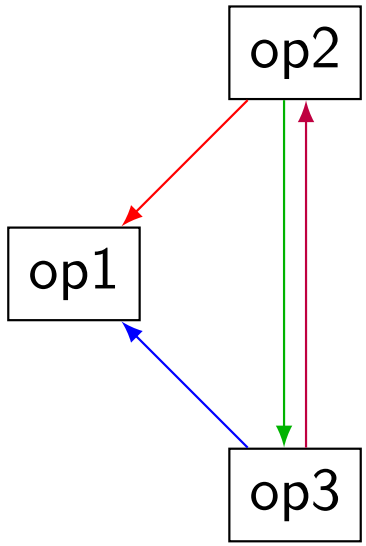
```
op1
```

What happened?

Dependency Graphs



B *depends on* A




```
op2.addDependency(op1)
op3.addDependency(op1)
op2.addDependency(op3)
op3.addDependency(op2)
```

`op2` and `op3` have *circular dependency* $\Rightarrow$ Deadlock!

For large `OperationQueues`, constructing dependency graph can reveal circular dependencies