Concurrent Applications

Dr.-Ing. Thomas Springer
M.Sc. Martin Weißbach
the prepared Xcode project for today’s lecture can be downloaded here:

https://www.rn.inf.tu-dresden.de/martin/iOS-Programming-17/Concurrent-Applications-Example.zip
Queues and Threads
• sequential execution of code
• multiple threads of a process execute their code in parallel
• belongs to a process (subunit)
• provided by the OS
• share their process’ address space

• available via NSThread (Foundation.framework)
• direct access to POSIX-Threads (< pthread.h > available)
• do not create threads manually

• create execution queues or use default queues provided by the system

• pieces of codes are scheduled on queues for execution
  • by default all operations are scheduled on the main queue
• the execution of tasks of a given queue is scheduled according to the available system resources and is handled by the system
• a queue and a thread are not necessarily the same unit
• tasks scheduled on the same queue may not necessarily be executed on the same thread
• **Main Queue**
  
  • created when the application launches
  
  • always available
  
  • all UI operations are scheduled to the main queue
    
    • e.g. changing a UILabel’s text property
      
      • if you’re working with multiple queues, make sure the property change is scheduled to the main queue
Queues

• **Serial Queue**
  • all tasks are executed sequentially as according to the schedule
  • *Main Queue*
  • long running operations block all consecutive operations
  • e.g. the UI would not react to user interactions

• **Concurrent Queue**
  • tasks are taken sequentially from the queue according to the schedule
  • tasks can be executed in parallel by utilizing multiple threads
A Brief Summary

Worker Thread | Dispatch Queue

Thread | Run Loop

Main Thread | Run Loop | Dispatch Queue
Grand Central Dispatch
• introduced in iOS 4 and Mac OS X 10.6
• low level C API – much more Swift(y) now

• runs on the OS level and is intended to match system resources to all running applications according to their needs [2]
• manages a pool of threads of each application based on available and required system resources
Queue schema according to [1]
• **Asynchronous Execution of Tasks**

  • dispatch tasks on a random global queue that’s provided by the system
  • dispatch tasks asynchronously
    • closure (task) is assigned for asynchronous execution

```swift
func scheduleTask() {
    DispatchQueue.global().async {
        // Perform some work here
        DispatchQueue.main.async(execute: {
            // Perform some update of the user interface
        })
    }
}
```
• **Asynchronous Execution of Tasks**

• create a custom queue and schedule tasks on that queue

• dispatch tasks asynchronously

```
func scheduleTask2() {
    // get a serial queue
    let queue = DispatchQueue(label: "identifier")
    queue.async {
        // Perform some work here
        DispatchQueue.main.async(execute: {
            // Perform some update of the user interface
        })
    }
}
```
Dispatch Groups

```swift
func dispatchGroups() {
    let queue1 = DispatchQueue(label: "test1")
    let queue2 = DispatchQueue(label: "test2")
    let group = DispatchGroup()

    queue1.async(group: group, execute: {
        // Perform some work here
    })
    queue2.async(group: group, execute: {
        // Perform some work here
    })

    group.notify(queue: DispatchQueue.main, execute: {
        // Do some work here
        // Maybe update the UI (since it's the MainQueue)
    })
}
```

1) create a group object

2) schedule tasks
   - pass the group object as parameter to group tasks of different queues

3) schedule callback
   - invoked all tasks tight to the group object are finished
   - specify queue to execute the callback on
Grand Central Dispatch

- Tasks once dispatched cannot be cancelled
- No status and progress information of tasks available
- only closures can be dispatched with GCD
Work Items

• DispatchWorkItems can be scheduled with GCD
  • wait for completion (synchronization)
  • completion notification
  • cancellation (no preemption)

```swift
let workitem = DispatchWorkItem {
    // Specify the work to be performed here
}
```
func workItems() {

    let workitem = DispatchWorkItem {
        // Specify the work to be performed here
    }

    workitem.wait() // waits indefinitely, setting timeout possible

    let queue = DispatchQueue(label: "test")
    queue.async(execute: workitem)
}

func workItems() {

    let workitem = DispatchWorkItem {
        // Specify the work to be performed here
    }

    let queue = DispatchQueue(label: "test")
    queue.async(execute: workitem)

    // do some work, reconsider

    workitem.cancel()

}


func workItems() {

    let workitem = DispatchWorkItem {
        // Specify the work to be performed here
    }
    workitem.notify(queue: DispatchQueue.main, execute: {
        // Do some work when the workitem finished
        // Maybe update UI (since you're on the MainQueue
    })

    let queue = DispatchQueue(label: "test")
    queue.async(execute: workitem)
}

• Tasks once dispatched cannot be cancelled

• No status and progress information of tasks available

• only closures can be dispatched with GCD
OperationQueue & Operation
OperationQueue & Operation

- **OperationQueue**
  - manages the execution of a set of Operation objects
  - operations cannot be removed from a queue, but they can be cancelled
  - execution of objects is scheduled according priority-levels and inter-operation dependencies
    - inter-operation dependencies provide an absolute order of execution even if operations are added to different queues
OperationQueue & Operation

- **Operation**
  - represents a single task incl. data and code to be executed
  - abstract class, but contains built-in logic to safely execute the task
  - task can only be executed once within this object

- **Operation Dependencies**
  - operation is considered ready for execution if all dependencies have finished
  - `addDependency`, `removeDependency`
  - dependencies are considered finished regardless of execution result
    - valid for successful execution as well as cancellation
• subclass Operation
• start the execution of the custom task in the -start() method
• observe the state of the operation using KVO
  • cancelled (KVO-Key: isCancelled)
  • executing (KVO-Key: isExecuting)
  • ready (KVO-Key: isReady)
• make sure to emit the correct KVO-Notifications when implementing a subclass of Operation yourself
• never refer to super within the subclass
Quality of Service Classes
• no sound priority support for queues and threads of the framework

• QoS Classes intended to define default priority setups for different use cases

• priority inversion not prevented by the QoS classes
Quality of Service Classes

• **User-Interactive** (*QualityOfService.userInteractive*)
  - tasks are interacting with the user
    - e.g. refresh the user interface or perform animations
  - executed on the main-thread
  - work duration is virtually instantaneous

• **User-Initiated** (*QualityOfService.userInitiated*)
  - tasks are user initiated and require an immediate result to proceed with the user interaction
    - e.g. opening a saved file or performing an action when something in the UI is clicked
  - work duration is nearly instantaneous, a few seconds or less

• **Utility** (*QualityOfService.utility*)
  - tasks that take some time to finish and have usually a progress indication visible to the user
    - e.g. download tasks or data imports
  - work duration can take up to a few minutes

• **Background** (*QualityOfService.background*)
  - work that operates in the background without the user's knowledge
    - e.g. indexing or backups
  - work duration can take up to hours
Quality of Service Classes

- applicable to threads, dispatch queues & blocks, NSOperations and processes

- GCD runs on `QOS_CLASS_DEFAULT` (default QoS Class)
References
