iOS Performance and Concurrency

Patrick Thomson
Performance Matters

• iOS devices are resource-constrained
• Users will notice performance issues
• The deciding factor between a good and an awful app
Demo
Obligatory Documentation Mention

“Concurrency Programming Guide”
Why is This Slow?

- Draw graphics
- Handle input
- Receive input

run loop
Draw windows and process image data

Receive input

Handle input

Receive input
Draw windows

Handle input

Receive input

Process data
Concurrency is Tricky

- Order of execution varies
- Bugs may appear and disappear at random
- Coordinating tasks and stateful data is hard
- Ergo, writing correct concurrent code is complicated
Deadlock
Nondeterministic Execution
Nondeterministic Execution

NSLog(@"bye");
NSLog(@"hi");
Nondeterministic Execution

NSLog(@"bye");
Core 1

NSLog(@"hi");
Core 2
Nondeterministic Execution

Core 1

Core 2
Race Conditions

```java
if (flag == NO) {
    flag = YES;
    launchMissiles();
}
```
Race Conditions

```java
if (flag == NO) {
    flag = YES;
    launchMissiles();
}
```
Race Conditions

```java
if (flag == NO) {
    flag = YES;
    launchMissiles();
}
```
Race Conditions

```java
if (flag == NO) {
    flag = YES;
    launchMissiles();
}
```
Race Conditions

```plaintext
if (flag == NO) {
    flag = YES;
    launchMissiles();
}
```
Race Conditions

if (flag == NO) {
  flag = YES;
  launchMissiles();
}
iOS Concurrency Tools

- iOS is an amalgam of many historical technologies, and it shows
- Too many options
- Luckily, you generally only need one
Use Grand Central Dispatch!

Always, always reach for GCD first!
So What is GCD?

- Set of concurrency primitives
- Based around asynchronicity
- Adjusts itself to leverage the capabilities of the devices it’s running on
Why Use GCD?

- Transparently manages the minutiae of threading details
- Works in C, Objective-C, and C++
- Built on simple, easy-to-understand concepts
- Gorgeous and simple API
- Absurdly fast
How it Works

Thread
How it Works

Task

Thread
How it Works

Task

Task

Task

Thread
How it Works

Task

Thread

Task

Thread

Task
How it Works

Task

Task

Task

Task

Thread

Thread
How it Works

- Task
- Task
- Task
- Task

- Thread
- Thread
- Thread
How it Works

Task

Thread

Thread

Thread
How it Works
How it Works
How it Works

Thread
Tasks

- Implemented in terms of C blocks
- Fundamental units of work
- Tasks have no return value: if you want any computed values to persist, you must do so within the task

```swift
^{
    NSLog(@"Hello!");
    sleep(2);
    NSLog(@"Goodbye!");
}
```
Serial Queues

- Execute tasks one at a time
- First-in, first-out order
- Very lightweight (~160 bytes)
- Can replace locks and mutexes
Serial Queues

- Execute tasks one at a time
- First-in, first-out order
- Very lightweight (~160 bytes)
- Can replace locks and mutexes
Queue

'a'  'b'  'c'
Queue

'a'  'b'  'c'
Queue

'a'  'b'  'c'
dispatch_queue_t ourQueue;
ourQueue = dispatch_queue_create("edu.cmu.dataviz.example", NULL);
Enqueuing Tasks

- `dispatch_async(queue, task)` schedules task for execution on queue and returns immediately.

- `dispatch_sync(queue, task)` schedules task on queue and does not return until task completes.
dispatch_queue_t queue = dispatch_queue_create("edu.cmu.dataviz.example", NULL);

dispach_async(queue, ^{
    NSLog(@"Task beginning execution...");
    sleep(2);
    NSLog(@"Task ending..." Banci);
});

dispatch_release(queue);
dispatch_queue_t queue = dispatch_queue_create("edu.cmu.dataviz.example", NULL);

dispatch_async(queue, ^{
    NSLog(@"Task beginning execution...");
    sleep(2);
    NSLog(@"Task ending...");
});

dispach_release(queue);
Retain and Release

Any dispatch object you create must be managed with dispatch_retain and dispatch_release.

def dispatch_queue_t ourQueue:
ourQueue = dispatch_queue_create("edu.cmu.dataviz.example", NULL);

// blah blah blah

def dispatch_release(ourQueue);
Concurrent Queues

- May execute tasks simultaneously
- No guarantee as to the order of task execution
- Use your own or the provided app-wide concurrent queues
Concurrent Queues

- May execute tasks simultaneously
- No guarantee as to the order of task execution
- Use your own or the provided app-wide concurrent queues
Queue Priorities

DISPATCH_QUEUE_PRIORITY_HIGH
DISPATCH_QUEUE_PRIORITY_DEFAULT
DISPATCH_QUEUE_PRIORITY_LOW

DISPATCH_QUEUE_PRIORITY_BACKGROUND
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, NULL);

versus

dispatch_queue_create("edu.cmu.example", DISPATCH_QUEUE_CONCURRENT);
Main Queue

- Associated with the run loop
- Useful for drawing, UI updating, thread-unsafe frameworks
- Avoid it if possible

Diagram:
- Draw graphics
- Handle input
- Receive input
Main Queue

- Associated with the run loop
- Useful for drawing, UI updating, thread-unsafe frameworks
- Avoid it if possible
Coordinating Tasks

- Often one dispatches asynchronous work and then waits for its completion.
- Groups count when tasks enter and exit for a queue.
- Can wait until all tasks are done and execute a block at that point.
Coordinating Tasks

- Often one dispatches asynchronous work and then waits for its completion.
- Groups count when tasks enter and exit for a queue.
- Can wait until all tasks are done and execute a block at that point.
Coordinating Tasks

- Often one dispatches asynchronous work and then waits for its completion
- Groups count when tasks enter and exit for a queue
- Can wait until all tasks are done and execute a block at that point
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, NULL);
dispatch_group_t group = dispatch_group_create();

dispatch_group_async(group, queue, ^{ /* work goes here */ });
dispatch_group_async(group, queue, ^{ /* more work */ });
dispatch_group_async(group, queue, ^{ /* even more work! */ });
dispatch_group_notify(group, ^{ /* executes when all work is done */ });
// wait until all the work is done
dispatch_group_wait(group, DISPATCH_TIME_FOREVER);
Enumeration

dispatch_apply(size_t iterations, dispatch_queue_t queue, void (^block)(size_t));

- Spawn tasks asynchronously and wait until each completes
- For loops: dispatch_apply
- Foreach loops: enumerateObjectsWithOptions:usingBlock: with NS Enumeration Concurrent
Doing Things Once

```c
static dispatch_once_t onceToken;
dispatch_once(&onceToken, ^{
    launchMissiles();
});
```
Measuring Time

dispatch_time_t dispatch_time(dispatch_time_t base, uint64_t nanoseconds);

dispatch_time_t DISPATCH_TIME_NOW;
dispatch_time_t DISPATCH_TIME_FOREVER;
dispatch_after()

double delayInSeconds = 2.0;
dispatch_time_t delay = dispatch_time(DISPATCH_TIME_NOW, delayInSeconds * NSEC_PER_SEC);

dispatch_after(delay, queue, ^{
    NSLog(@"Hello, from two seconds in the future!");
});
Suspending and Resuming

You can prevent queues from executing blocks by suspending them.

These do not suspend the execution of any running tasks.
Things We Didn’t Cover

• Sources: monitor the file system, running processes, signals and timers, and execute blocks appropriately

• Dispatch I/O: highly efficient, adaptable, non-blocking I/O

• Barriers: enforce invariants on your concurrent queues

• Semaphores: counted, multicore safe, faster than POSIX
Grand Central Dispatch (GCD) Reference

Overview

Grand Central Dispatch (GCD) comprises language features, runtime libraries, and system enhancements that provide systematic, comprehensive improvements to the support for concurrent code execution on multicore hardware in iOS and Mac OS X. The BSD subsystem, CoreFoundation, and Cocoa APIs have all been extended to use these enhancements to help both the system and your application run faster, more efficiently, and with improved responsiveness.

Consider how difficult it is for a single application to use multiple cores effectively, let alone doing it on different computers with different numbers of computing cores or in an environment with multiple applications competing for those cores. GCD, operating at the system level, can better accommodate the needs of all running applications, matching them to the available system resources in a balanced fashion.

This document describes the GCD API, which supports the asynchronous execution of operations at the Unix level of the system. You can use this API to manage interactions with file descriptors, Mach ports, signals, or timers. In Mac OS X 10.7 and later, you can also use GCD to handle general purpose asynchronous I/O operations on file descriptors.

GCD is not restricted to system-level applications, and before you use it for higher-level applications, you should consider whether similar functionality provided in Cocoa (via multithreaded and block objects) would be easier to use or more appropriate for your needs. See Concurrency Programming Guide for more information.

Functions by Task

Creating and Managing Queues

dispatch_get_global_queue

dispatch_get_main_queue

dispatch_queue_create

dispatch_get_current_queue

dispatch_queue_get_label

dispatch_get_target_queue

dispatch_main

Queuing Tasks for Dispatch

GCD provides and manages FIFO queues to which your application can submit tasks in the form of block objects. Blocks submitted to dispatch queues are executed on a pool of threads fully managed by the system. No guarantee is made as to the thread on which a task executes. GCD offers three kinds of queues:

- Main tasks execute serially on your application’s main thread
- Concurrent: tasks are dequeued in FIFO order, but run concurrently and can finish in any order.
- Serial: tasks execute one at a time in FIFO order

“Grand Central Dispatch Reference”
Tips, Best Practices, and Hints
Plan Ahead

- Factor your work into its most discrete units
- Keep it simple
- Instruments is your friend
Favor Immutability

- Avoid sharing mutable data structures
- stick to NSArray, NSDictionary, NSSet rather than their mutable counterparts
- If you absolutely can’t avoid mutable shared state, guard access to it with a serial queue
Working with Graphics

- Don’t be afraid to spawn queues
- Only draw on the main thread
- `dispatch_get_main_queue();`
Other Concurrency Options
NSTimer

- Performs a given selector repeatedly with a provided interval
- Good for simple repeated invokeds
- Doesn’t support blocks
- Supports preemptive cancellation
NSOperationQueue

- An object-oriented take on the notion of queues and tasks
- Precedes GCD; now implemented on top of it
- Heavyweight – requires a lot of subclassing
- Useful if tasks must support cancellation
- Probably won’t need this, outside of API’s such as CMMotionManager
NSThread

- In the past, concurrency meant invoking or subclassing NSThread
- No longer recommended
- Still has a few useful helper functions – [NSThread isMainThread]
performSelector and friends

- performSelector: withObject: afterDelay:
- performSelectorOnMainThread: withObject: waitUntilDone:
- performSelectorInBackground: withObject:
- + cancelPreviousPerformRequestsWithTarget:
Do not use these!

- Since they take selectors, you sacrifice type-checking
- They sweep complexity under the rug, which is a Bad Move in general
- Everything you can do with these methods you can do with GCD
Questions?