Modernizing GCD Usage

How to stay on core

Session 706

Daniel Chimene, Core Darwin
Daniel A. Steffen, Core Darwin
Pierre Habouzit, Core Darwin
Efficiency Through Observation

Going off core during an operation reduces efficiency

![Graph showing efficiency over observation time](attachment:graph.png)
1.3x faster after combining queue hierarchies
Parallelism and concurrency
Parallelism and concurrency

Using GCD for concurrency
Parallelism and concurrency
Using GCD for concurrency
Unified Queue Identity
Parallelism and concurrency
Using GCD for concurrency
Unified Queue Identity
Finding problem spots
Parallelism
Simultaneous execution of closely related computations

Concurrency
Composition of independently executed tasks
Parallelism
Parallelism
Simultaneous execution of closely related computations
Parallelism
Simultaneous execution of closely related computations
Parallelism
Simultaneous execution of closely related computations
Take Advantage of System Frameworks

Accelerate  Metal 2  Core ML  Core Animation
Parallelism with GCD

Express explicit parallelism with `DispatchQueue.concurrentPerform`

Parallel for-loop—calling thread participates in the computation

More efficient than many asyncs to a concurrent queue

`DispatchQueue.concurrentPerform(1000) { i in /* iteration i */ }`
Parallelism with GCD

Express explicit parallelism with `DispatchQueue.concurrentPerform`

Parallel for-loop—calling thread participates in the computation

More efficient than many asyncs to a concurrent queue

```swift
DispatchQueue.concurrentPerform(1000) { i in /* iteration i */ }```

Parallelism with GCD

Express explicit parallelism with `DispatchQueue.concurrentPerform`

Parallel for-loop—calling thread participates in the computation

More efficient than many asyncs to a concurrent queue

```swift
DispatchQueue.concurrentPerform(1000) { i in /* iteration i */ }

dispatch_apply(DISPATCH_APPLY_AUTO, 1000, ^(size_t i){ /* iteration i */ })
```

`DISPATCH_APPLY_AUTO` deploys back to macOS 10.9, iOS 7.0
Dynamic Resource Availability
Choosing an iteration count

DispatchQueue.concurrentPerform(3) { i in /* iteration i */ }
Dynamic Resource Availability
Choosing an iteration count

DispatchQueue.concurrentPerform(3) { i in /* iteration i */ }
Dynamic Resource Availability

Choosing an iteration count

```
DispatchQueue.concurrentPerform(3) { i in /* iteration i */ }
```
Dynamic Resource Availability
Choosing an iteration count

DispatchQueue.concurrentPerform(6) { i in /* iteration i */ }
Dynamic Resource Availability

Choosing an iteration count

DispatchQueue.concurrentPerform(6) { i in /* iteration i */ }

UI Rendering
Dynamic Resource Availability

Choosing an iteration count

DispatchQueue.concurrentPerform(11) { i in /* iteration i */ }
Dynamic Resource Availability

Choosing an iteration count

```
DispatchQueue.concurrentPerform(11) { i in /* iteration i */ }
```
Dynamic Resource Availability

Choosing an iteration count

DispatchQueue.concurrentPerform(1000) { i in /* iteration i */ }
Parallelism

Leverage system frameworks

Use `DispatchQueue.concurrentPerform`

Consider dynamic availability
Concurrency
Concurrency
Composition of independently executed tasks
Concurrency
Composition of independently executed tasks
Concurrency
Composition of independently executed tasks

Diagram:
- User Interface
- Networking
- Database
Concurrency
Composition of independently executed tasks

User Interface
Networking
Database
Concurrency
Composition of independently executed tasks

User Interface
Concurrency
Composition of independently executed tasks
Concurrency
Composition of independently executed tasks

CPU
User Interface
Database
Networking
Concurrency
Composition of independently executed tasks

Touch!
Concurrency
Composition of independently executed tasks

Touch!
Concurrency
Context switching

- User Interface
- Database
- Networking
Context Switching
The power of concurrency

The OS can choose a new thread at any time
Context Switching
The power of concurrency

The OS can choose a new thread at any time
• A higher priority thread needs the CPU
Context Switching
The power of concurrency

The OS can choose a new thread at any time
- A higher priority thread needs the CPU
- A thread finishes its current work
Context Switching
The power of concurrency

The OS can choose a new thread at any time
- A higher priority thread needs the CPU
- A thread finishes its current work
- Waiting to acquire a resource
Context Switching
The power of concurrency

The OS can choose a new thread at any time
• A higher priority thread needs the CPU
• A thread finishes its current work
• Waiting to acquire a resource
• Waiting for an asynchronous request to complete
Excessive Context Switching
Too much of a good thing

Repeatedly bouncing between contexts can become expensive
Excessive Context Switching
Too much of a good thing

Repeatedly bouncing between contexts can become expensive
Excessive Context Switching
Too much of a good thing

Repeatedly bouncing between contexts can become expensive
• CPU runs less efficiently
Excessive Context Switching
Too much of a good thing

Repeatedly bouncing between contexts can become expensive
• CPU runs less efficiently
Excessive Context Switching
Too much of a good thing

Repeatedly bouncing between contexts can become expensive
• CPU runs less efficiently
• There may be others ahead in line for CPU access
Excessive Context Switching
Too much of a good thing

Repeatedly bouncing between contexts can become expensive

• CPU runs less efficiently
• There may be others ahead in line for CPU access
Excessive Context Switching
Too much of a good thing

Repeatedly bouncing between contexts can become expensive
• CPU runs less efficiently
• There may be others ahead in line for CPU access
Excessive Context Switching
Too much of a good thing

Repeatedly waiting for exclusive access to contended resources
Repeatedly switching between independent operations
Repeatedly bouncing an operation between threads
Excessive Context Switching
Too much of a good thing

Repeatedly waiting for exclusive access to contended resources
Repeatedly switching between independent operations
Repeatedly bouncing an operation between threads
Too much of a good thing

Repeatedly waiting for exclusive access to contended resources

Repeatedly switching between independent operations

Repeatedly bouncing an operation between threads
Lock Contention
Lock Contention
Visualization in Instruments
Lock Contention
Visualization in Instruments
Lock Contention
Visualization in Instruments
Lock Contention
Visualization in Instruments

Frequent context-switching

10\(\mu\)s

10\(\mu\)s
Lock Contention
Lock Contention
Lock Contention

Fair locks

- Owned
- CPU
Lock Contention

Fair locks
Lock Contention

Fair locks

- owned
- reserved
Lock Contention
Fair locks

Diagram showing lock contention with CPU, owned, reserved, and owned sections.
Lock Contention

Fair locks
Lock Contention

Unfair locks

[Diagram showing lock contention with 'CPU', 'owned', and other symbols]
Lock Contention

Unfair locks
Lock Contention

Unfair locks
Lock Contention
Use the right lock for the job

<table>
<thead>
<tr>
<th></th>
<th>Unfair</th>
<th>Fair</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available types</td>
<td>os_unfair_lock</td>
<td>pthread_mutex_t, NSLock</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DispatchQueue.sync</td>
</tr>
<tr>
<td>Contended lock re-acquisition</td>
<td>Can steal the lock</td>
<td>Context switches to next waiter</td>
</tr>
<tr>
<td>Subject to waiter starvation</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Lock Ownership
Lock Ownership

Ownership helps resolve priority inversion
• High priority waiter
• Low priority owner
# Lock Ownership

## Single Owner

<table>
<thead>
<tr>
<th>Serial queues</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DispatchWorkItem.wait</code></td>
</tr>
<tr>
<td><code>os_unfair_lock</code></td>
</tr>
<tr>
<td><code>pthread_mutex, NSLock</code></td>
</tr>
</tbody>
</table>
## Lock Ownership

<table>
<thead>
<tr>
<th>Single Owner</th>
<th>No Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial queues</td>
<td>dispatch_semaphore</td>
</tr>
<tr>
<td>DispatchWorkItem.wait</td>
<td>dispatch_group</td>
</tr>
<tr>
<td>os_unfair_lock</td>
<td>pthread_cond, NSCondition</td>
</tr>
<tr>
<td>pthread_mutex, NSLock</td>
<td>Queue suspension</td>
</tr>
</tbody>
</table>
## Lock Ownership

<table>
<thead>
<tr>
<th>Single Owner</th>
<th>No Owner</th>
<th>Multiple Owners</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial queues</td>
<td>dispatch_semaphore</td>
<td>Private concurrent queues</td>
</tr>
<tr>
<td>DispatchWorkItem.wait</td>
<td>dispatch_group</td>
<td>pthread_rwlock</td>
</tr>
<tr>
<td>os_unfair_lock</td>
<td>pthread_cond, NSCondition</td>
<td></td>
</tr>
<tr>
<td>pthread_mutex, NSLock</td>
<td>Queue suspension</td>
<td></td>
</tr>
</tbody>
</table>
Optimizing Lock Contention

Inefficient behaviors are often emergent properties

Visualize your app’s behavior with Instruments

Use the right lock for the job
Too Much of a Good Thing

Repeatedly waiting for exclusive access to contended resources
Repeatedly switching between independent operations
Repeatedly bouncing an operation between threads
Using GCD for Concurrency

Daniel A. Steffen, Core Darwin
### Grand Central Dispatch

<table>
<thead>
<tr>
<th>Topic</th>
<th>WWDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplifying iPhone App Development with Grand Central Dispatch</td>
<td>2010</td>
</tr>
<tr>
<td>Asynchronous Design Patterns with Blocks, GCD, and XPC</td>
<td>2012</td>
</tr>
<tr>
<td>Power, Performance, and Diagnostics: What's new in GCD and XPC</td>
<td>2014</td>
</tr>
<tr>
<td>Building Responsive and Efficient Apps with GCD</td>
<td>2015</td>
</tr>
<tr>
<td>Concurrent Programming with GCD in Swift 3</td>
<td>2016</td>
</tr>
</tbody>
</table>
Serial Dispatch Queue

Fundamental GCD primitive
Serial Dispatch Queue

**Fundamental GCD primitive**

- Mutual exclusion
- FIFO ordered
Serial Dispatch Queue

Fundamental GCD primitive
• Mutual exclusion
• FIFO ordered
• Concurrent atomic enqueue
• Single dequeuer
let queue = DispatchQueue(label: "com.example.queue")
queue.async { /* 1 */ }
queue.async { /* 2 */ }
queue.sync { /* 3 */ }
let queue = DispatchQueue(label: "com.example.queue")
queue.async { /* 1 */ }
queue.async { /* 2 */ }
queue.sync { /* 3 */ }
let queue = DispatchQueue(label: "com.example.queue")
queue.async { /* 1 */ }
queue.async { /* 2 */ }
queue.sync { /* 3 */ }

let queue = DispatchQueue(label: "com.example.queue")
queue.async { /* 1 */ }
queue.async { /* 2 */ }
queue.sync { /* 3 */ }

Serial Dispatch Queue
let queue = DispatchQueue(label: "com.example.queue")

queue.async { /* 1 */ }

queue.async { /* 2 */ }

queue.sync { /* 3 */ }

let queue = DispatchQueue(label: "com.example.queue")
queue.async { /* 1 */ }
queue.async { /* 2 */ }
queue.sync { /* 3 */ }
let queue = DispatchQueue(label: "com.example.queue")

queue.async { /* 1 */ }
queue.async { /* 2 */ }
queue.sync { /* 3 */ }
Dispatch Source

Event monitoring primitive

```swift
let source = DispatchSource.makeReadSource(fileDescriptor: fd, queue: queue)
source.setEventHandler { read(fd) }
source.setCancelHandler { close(fd) }
source.activate()
```
Dispatch Source

Event monitoring primitive

```swift
let source = DispatchSource.makeReadSource(fileDescriptor: fd, queue: queue)
source.setEventHandler { read(fd) }
source.setCancelHandler { close(fd) }
source.activate()
```
Dispatch Source

Event monitoring primitive

- Event handler executes on target queue

```swift
let source = DispatchSource.makeReadSource(fileDescriptor: fd, queue: queue)
source.setEventHandler { read(fd) }
source.setCancelHandler { close(fd) }
source.activate()
```
Dispatch Source

Event monitoring primitive

• Event handler executes on target queue
• Invalidation pattern with explicit cancellation

```swift
let source = DispatchSource.makeReadSource(fileDescriptor: fd, queue: queue)
source.setEventHandler { read(fd) }
source.setCancelHandler { close(fd) }
source.activate()
```
Dispatch Source

Event monitoring primitive
- Event handler executes on target queue
- Invalidation pattern with explicit cancellation
- Initial setup followed by activate

```swift
let source = DispatchSource.makeReadSource(fileDescriptor: fd, queue: queue)
source.setEventHandler { read(fd) }
source.setCancelHandler { close(fd) }
source.activate()
```
Target Queue Hierarchy

Serial queues and sources can form a tree
Target Queue Hierarchy

Serial queues and sources can form a tree
Serial queues and sources can form a tree

```
let Q1 = DispatchQueue(label: "Q1", target: EQ)
let Q2 = DispatchQueue(label: "Q2", target: EQ)
```
Target Queue Hierarchy

Serial queues and sources can form a tree

Shared single mutual exclusion context

Independent individual queue order

```swift
let Q1 = DispatchQueue(label: "Q1", target: EQ )
let Q2 = DispatchQueue(label: "Q2", target: EQ )
```
Target Queue Hierarchy
Target Queue Hierarchy

Q1

1 2 3 4 5 6

Q2

A B C D
Target Queue Hierarchy

Q1

Q2

EQ

1 2 A 3 B 4 5 6 C D
Target Queue Hierarchy
Quality of Service

Abstract notion of priority
Provides explicit classification of your work
Affects various execution properties

User Interactive
User Initiated
Utility
Background

Power, Performance, and Diagnostics: What's new in GCD and XPC

WWDC 2014
Quality of Service

Abstract notion of priority

Provides explicit classification of your work

Affects various execution properties

User Interactive

User Initiated

Utility

Background

Power, Performance, and Diagnostics: What's new in GCD and XPC

WWDC 2014
Quality of Service

Abstract notion of priority

Provides explicit classification of your work

Affects various execution properties
Quality of Service

Abstract notion of priority
Provides explicit classification of your work
Affects various execution properties

User Interactive (UI)
User Initiated (IN)
Utility (UT)
Background
Quality of Service

Abstract notion of priority

Provides explicit classification of your work

Affects various execution properties

User Interactive (UI)
User Initiated (IN)
Utility (UT)
Background (BG)
QoS and Target Queue Hierarchy
QoS and Target Queue Hierarchy
QoS and Target Queue Hierarchy
QoS and Target Queue Hierarchy
QoS and Target Queue Hierarchy
QoS and Target Queue Hierarchy
QoS and Target Queue Hierarchy

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

Diagram:
- IN
- S1
- Q1
- EQ
- Q2
- UT
- S2
- UI
QoS and Target Queue Hierarchy

queue.async { ... }

EQ

IN

S1

Q1

EQ

UT

S2

Q2

UI
QoS and Target Queue Hierarchy

```python
queue.async { ... }
```
QoS and Target Queue Hierarchy

Priority Inversion
QoS and Target Queue Hierarchy

Priority Inversion Resolved
Granularity of Concurrency
Event Monitoring Setup
Event Monitoring Setup

Network Connection
Event Monitoring Setup

1. Network Connection
2. Dispatch Source
3. Dispatch Queue
Event Monitoring Setup
Event Monitoring Setup
Event Handling on Many Independent Queues
Event Handling on Many Independent Queues
Event Handling on Many Independent Queues
Event Handling on Many Independent Queues
Event Handling on Many Independent Queues
Single Mutual Exclusion Context

Diagram showing three processes (S) accessing a shared resource (Q).
Single Mutual Exclusion Context
Single Mutual Exclusion Context

[Diagram showing a network of nodes connected by arrows, with labels 'S', 'Q', and 'EQ'.]
Single Mutual Exclusion Context
Too Much of a Good Thing

Repeatedly waiting for exclusive access to contended resources
Repeatedly switching between independent operations
Repeatedly bouncing an operation between threads
Avoid Unbounded Concurrency
Repeatedly switching between independent operations
Avoid Unbounded Concurrency
Repeatedly switching between independent operations
Avoid Unbounded Concurrency
Repeatedly switching between independent operations

Many queues becoming active at once
• Independent per-client sources
• Independent per-object queues
Avoid Unbounded Concurrency
Repeatedly switching between independent operations

Many workitems submitted to global concurrent queue
Avoid Unbounded Concurrency
Repeatedly switching between independent operations

Many workitems submitted to global concurrent queue
• If workitems block, more threads will be created
• May lead to thread explosion
Avoid Unbounded Concurrency
Repeatedly switching between independent operations

Many workitems submitted to global concurrent queue
• If workitems block, more threads will be created
• May lead to thread explosion
One Queue per Subsystem
One Queue per Subsystem

- User Interface
- Main Queue
- Networking
- Database
One Queue per Subsystem

- User Interface
  - Main Queue
- Networking
  - Serial Queue
- Database
  - Serial Queue
One Queue Hierarchy per Subsystem

- User Interface
- Main Queue
- Networking
- Serial Queue
- Database
- Serial Queue
One Queue Hierarchy per Subsystem

- NQ
  - Serial Queue
  - Networking
    - Serial Queue
- DQ
  - User Interface
    - Main Queue
  - Database
    - Serial Queue
Good Granularity of Concurrency

Fixed number of serial queue hierarchies
Good Granularity of Concurrency

Fixed number of serial queue hierarchies
Good Granularity of Concurrency

- Fixed number of serial queue hierarchies
- Coarse workitem granularity between hierarchies
Good Granularity of Concurrency

Fixed number of serial queue hierarchies

Coarse workitem granularity between hierarchies
Good Granularity of Concurrency

Fixed number of serial queue hierarchies
Coarse workitem granularity between hierarchies
Finer workitem granularity inside a hierarchy
Using GCD for Concurrency

Organize queues and sources into serial queue hierarchies

Use a fixed number of serial queue hierarchies

Size your workitems appropriately
Introducing Unified Queue Identity

Pierre Habouzit, Core Darwin
Mutual Exclusion Context

Deep dive

S1 → Q1 → EQ → UT → S2

S1 → Q1 → EQ → UT → UI → S2
Mutual Exclusion Context
Deep dive
let EQ = DispatchQueue(label: "com.example.exclusion-context")
Unified Queue Identity
Asynchronous workitems
Unified Queue Identity
Asynchronous workitems

EQ

EQ.async { ... }
Unified Queue Identity
Asynchronous workitems

EQ

EQ.async { ... }
Unified Queue Identity
Asynchronous workitems

Kernel
Owner
EQ

Application
EQ

EQ.async { ... }
Unified Queue Identity
Asynchronous workitems

Kernel

Owner

EQ

Application

EQ

NEW
Unified Queue Identity
Asynchronous workitems

EQ.async { ... }
Unified Queue Identity
Asynchronous workitems

```
EQ.async { ... }
```
Unified Queue Identity
Asynchronous workitems

Kernel

Owner

UT

EQ

Application

EQ

NEW
Unified Queue Identity
Synchronous workitems

Kernel
Owner
UT
EQ

Application
IN
EQ.sync { ... }
EQ
Unified Queue Identity
Synchronous workitems

Kernel

Owner

UT

EQ

Application

IN

EQ.sync { ... }

EQ
Unified Queue Identity
Synchronous workitems

Kernel
- Owner
- Sync Waiters
- EQ

Application
- EQ
- EQ.sync { ... }

NEW
Unified Queue Identity
Synchronous workitems

Kernel

Owner

Sync Waiters

EQ

Application

EQ

EQ.sync { ... }
One Identity to Find Them All

... and in the kernel bind them
let S1 = DispatchSource.makeReadSource(
  fileDescriptor: fd, queue: EQ)
S1.setEventHandler { ... }
S1.activate()
let S1 = DispatchSource.makeReadSource(
    fileDescriptor: fd, queue: EQ)
S1.setEventHandler { ... }
S1.activate()
let S1 = DispatchSource.makeReadSource(
    fileDescriptor: fd, queue: EQ)
S1.setEventHandler { ... }
S1.activate()
let S1 = DispatchSource.makeReadSource(
    fileDescriptor: fd, queue: EQ)
S1.setEventHandler { ... }
S1.activate()
let S2 = DispatchSource.makeReadSource(
    fileDescriptor: fd, queue: EQ)
S2.setEventHandler(qos: .UserInteractive) {
...
}
S2.activate()
let S2 = DispatchSource.makeReadSource(
    fileDescriptor: fd, queue: EQ)
S2.setEventHandler(qos: .UserInteractive) { ... }
S2.activate()
let S2 = DispatchSource.makeReadSource(
    fileDescriptor: fd, queue: EQ)
S2.setEventHandler(qos: .UserInteractive) { ... }
S2.activate()
Too Much of a Good Thing

Repeatedly waiting for exclusive access to contended resources
Repeatedly switching between independent operations
Repeatedly bouncing an operation between threads
Without Unified Identity
In macOS Sierra and iOS 10
Without Unified Identity
In macOS Sierra and iOS 10
Without Unified Identity

In macOS Sierra and iOS 10
Without Unified Identity
In macOS Sierra and iOS 10
Without Unified Identity
In macOS Sierra and iOS 10
Without Unified Identity
In macOS Sierra and iOS 10
Without Unified Identity
In macOS Sierra and iOS 10
Without Unified Identity
In macOS Sierra and iOS 10
Without Unified Identity

In macOS Sierra and iOS 10
Without Unified Identity

In macOS Sierra and iOS 10
Without Unified Identity
In macOS Sierra and iOS 10

EQ
S2
S1
UT
S2 Handler
E1
E2
S1 Handler
E1
E2
EQ
UT
S1
S2
UI
S2 Handler
E1
Without Unified Identity
In macOS Sierra and iOS 10
Leveraging Ownership and Unified Identity
Leveraging Ownership and Unified Identity
Leveraging Ownership and Unified Identity
Leveraging Ownership and Unified Identity
Leveraging Ownership and Unified Identity
Leveraging Ownership and Unified Identity
Leveraging Ownership and Unified Identity
Leveraging Ownership and Unified Identity
The runtime uses every possible hint to optimize behavior.
Modernizing Existing Code
Modernizing Existing Code

No dispatch object mutation after activation

Protect your target queue hierarchy
No Mutation Past Activation

Set the properties of inactive objects before activation
• Source handlers
• Target queues
No Mutation Past Activation

Set the properties of inactive objects before activation

• Source handlers
• Target queues

```swift
let mySource = DispatchSource.makeReadSource(fileDescriptor: fd, queue: myQueue)
```
No Mutation Past Activation

Set the properties of inactive objects before activation

• Source handlers
• Target queues

```swift
let mySource = DispatchSource.makeReadSource(fileDescriptor: fd, queue: myQueue)
mySource.setEventHandler(qos: .userInteractive) { ... }
mySource.setCancelHandler { close(fd) }
```
No Mutation Past Activation

Set the properties of inactive objects before activation
  - Source handlers
  - Target queues

```swift
let mySource = DispatchSource.makeReadSource(fileDescriptor: fd, queue: myQueue)

mySource.setEventHandler(qos: .userInteractive) { ... }
mySource.setCancelHandler { close(fd) }

mySource.activate()
```
No Mutation Past Activation

Set the properties of inactive objects before activation

• Source handlers
• Target queues

```swift
let mySource = DispatchSource.makeReadSource(fileDescriptor: fd, queue: myQueue)

mySource.setEventHandler(qos: .userInteractive) { ... }
mySource.setCancelHandler { close(fd) }

mySource.activate()

mySource.setTarget(queue: otherQueue)
```
Effects of Queue Graph Mutation

Priority and ownership snapshots can become stale

• Defeats priority inversion avoidance
• Defeats direct handoff optimization
• Defeats event delivery optimization
Effects of Queue Graph Mutation

Priority and ownership snapshots can become stale
• Defeats priority inversion avoidance
• Defeats direct handoff optimization
• Defeats event delivery optimization

System frameworks may create sources on your behalf
• XPC connections are like sources
Protecting the Target Queue Hierarchy
Protecting the Target Queue Hierarchy

Build your queue hierarchy bottom to top
Protecting the Target Queue Hierarchy

Build your queue hierarchy bottom to top
Protecting the Target Queue Hierarchy

Build your queue hierarchy bottom to top
Opt into “static queue hierarchy”
Protecting the Target Queue Hierarchy

Build your queue hierarchy bottom to top

Opt into “static queue hierarchy”
Protecting the Target Queue Hierarchy

Build your queue hierarchy bottom to top

Opt into “static queue hierarchy”

```c
Q1 = dispatch_queue_create("Q1",
    DISPATCH_QUEUE_SERIAL)
dispatch_set_target_queue(Q1, EQ)
```
Protecting the Target Queue Hierarchy

Build your queue hierarchy bottom to top

Opt into “static queue hierarchy”

Q1 = dispatch_queue_create("Q1",
        DISPATCH_QUEUE_SERIAL)
dispatch_set_target_queue(Q1, EQ)

Q1 = dispatch_queue_create_with_target("Q1",
        DISPATCH_QUEUE_SERIAL, EQ)
Demo
Finding problem spots

Daniel A. Steffen, Core Darwin
iPad Pro
Anything you can do, you can do better.

Newsroom
PRESS RELEASE
JUNE 5, 2017
HomePod reinvents music in the home

Developer Insights
See how developers approach finding success on the App Store.

WWDC17. Now Live Streaming
June 5, 2017
See the latest advancements in macOS, iOS, watchOS, and tvOS with over 100 sessions presented by Apple engineers at this year’s conference. We’re live streaming all sessions daily and posting videos and related sample code throughout the week.

Watch WWDC session videos.
-(void)createConnections:(int)numberOfConnections serverPort:(int)port {

    struct sockaddr_in serverAddr = [self server];
    conns = (struct client_connection *)malloc
        (numberOfConnections * sizeof(struct client_connection));

    for (int i = 0; i < numberOfConnections; i++) {
        int sock = socket(PF_INET, SOCK_STREAM, 0);
        int ret = connect(sock, (struct sockaddr *) &serverAddr, sizeof(serverAddr));
        assert(ret >= 0);

        int flags = fcntl(sock, F_GETFL, 0);
        fcntl(sock, F_SETFL, flags | O_NONBLOCK);

        char queue_name[1024];
        snprintf(queue_name, 1024, "com.apple.client-queue-%d", i);
        dispatch_queue_t queue = dispatch_queue_create(queue_name, DISPATCH_QUEUE_SERIAL);

        dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ, sock, 0, NULL);

        dispatch_block_t block = dispatch_block_create(DISPACH_BLOCK.Assign_Current, ^{
            /* Drop the data read block start signpost */
            kdebug_signpost_start(MYNEWS_CONN_DATA_RECV, i, sock, 0, 0);

            /* Re-initialize the buffer for the connection */

            /* Call the read block */
            
            /* Re-initialize the buffer for the connection */
        }
    }
}
-(void)createConnections:(int)numberOfConnections serverPort:(int)port {
    struct sockaddr_in serverAddr = [self server];
    conns = (struct client_connection *)malloc(numberOfConnections * sizeof(struct client_connection));

    for (int i = 0; i < numberOfConnections; i++) {
        int sock = socket(PF_INET, SOCK_STREAM, 0);
        int ret = connect(sock, (struct sockaddr *)&serverAddr, sizeof(serverAddr));
        assert(ret >= 0);

        int flags = fcntl(sock, F_GETFL, 0);
        fcntl(sock, F_SETFL, flags | O_NONBLOCK);

        char queue_name[1024];
        snprintf(queue_name, 1024, "com.apple.client-queue-%d", i);
        dispatch_queue_t queue = dispatch_queue_create(queue_name, DISPATCH_QUEUE_SERIAL);

        dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ, sock, 0, NULL);

        dispatch_block_t block = dispatch_block_create(DISPATCH_BLOCK_ASSIGN_CURRENT, ^{
            /* Drop the data read block start signpost */
            kdebug_signpost_start(MYNEWS_CONN_DATA_RECV, i, sock, 0, 0);

            /* Re-initialize the buffer for the connection */
            // Initialize buffer
        });

        dispatch_source_set_target(source, queue);
        dispatch_source_set_callback(source, block);
        dispatch_source أكد

        close(sock);
    }
    free(conns);
}
if (err == 0 || (err < 0 && errno != EAGAIN && errno != EINTR)) {
    dispatch_source_cancel(source);
    break;
}
if (err < 0 && errno == EAGAIN) {
    break;
}
conns[i].index += err;

/* Add URL to global Set */
[self processURL:conns[i].buffer];

/* Drop the data read block end signpost */
kdebug_signpost_end(MYNEXT_CONN_DATA_RECV, i, sock, 0, 0);
'});

dispatch_activate(source);

dispatch_source_set_event_handler(source, block);
dispatch_source_set_cancel_handler(source, ^{
    close(sock);
});

dispatch_set_target_queue(source, queue);
if (err == 0 || (err < 0 && errno != EAGAIN && errno != EINTR)) {
    dispatch_source_cancel(source);
    break;
}

if (err < 0 && errno == EAGAIN) {
    break;
}

conns[i].index += err;

/* Add URL to global Set */
[self processURL:conns[i].buffer];

/* Drop the data read block end signpost */
kdebug_signpost_end(MYNEWS_CONN_DATA_RECV, i, sock, 0, 0);
}

dispatch_activate(source);

dispatch_source_set_event_handler(source, block);
dispatch_source_set_cancel_handler(source, ^{
    close(sock);
});

dispatch_set_target_queue(source, queue);
-(void)createConnections:(int)numberOfConnections serverPort:(int)port
{
    struct sockaddr_in serverAddr = [self server];
    conns = (struct client_connection *)malloc
             (numberOfConnections * sizeof(struct client_connection));

    for (int i = 0; i < numberOfConnections; i++) {
        int sock = socket(PF_INET, SOCK_STREAM, 0);
        int ret = connect(sock, (struct sockaddr *) &serverAddr, sizeof(serverAddr));
        assert(ret >= 0);

        int flags = fcntl(sock, F_GETFL, 0);
        fcntl(sock, F_SETFL, flags | O_NONBLOCK);

        char queue_name[1024];
        snprintf(queue_name, 1024, "com.apple.client-queue-%d", i);
        dispatch_queue_t queue = dispatch_queue_create(queue_name, DISPATCH_QUEUE_SERIAL);

        dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ, sock, 0, NULL);

        dispatch_block_t block = dispatch_block_create(DISPATCH_BLOCK_ASSIGN_CURRENT, ^{
            /* Drop the data read block start signpost */
            kdebug_signpost_start(MYNEWS_CONN_DATA_RECV, i, sock, 0, 0);

            /* Re-initialize the buffer for the connection */
        });
-(void)createConnections:(int)numberOfConnections serverPort:(int)port{

    struct sockaddr_in serverAddr = [self server];
    conns = (struct client_connection *)malloc(numberOfConnections * sizeof(struct client_connection));

    for (int i = 0; i < numberOfConnections; i++) {
        int sock = socket(PF_INET, SOCK_STREAM, 0);
        int ret = connect(sock, (struct sockaddr *) &serverAddr, sizeof(serverAddr));
        assert(ret == 0);

        int flags = fcntl(sock, F_GETFL, 0);
        fcntl(sock, F_SETFL, flags | O_NONBLOCK);

        char queue_name[1024];
        snprintf(queue_name, 1024, "com.apple.client-queue-%d", i);
        dispatch_queue_t queue = dispatch_queue_create(queue_name, DISPATCH_QUEUE_SERIAL);

        dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ, sock, 0, NULL);

        dispatch_block_t block = dispatch_block_create(DISPATCH_BLOCK_ASSIGN_CURRENT, ^{
            /* Drop the data read block start signpost */
            kdebug_signpost_start(MYNEWS_CONN_DATA_RECV, i, sock, 0, 0);

            /* Re-initialize the buffer for the connection */
        });

    }
}

if (err == 0 || (err < 0 && errno != EAGAIN && errno != EINTR)) {
    dispatch_source_cancel(source);
    break;
}
if (err < 0 && errno == EAGAIN) {
    break;
}
conns[i].index += err;

/* Add URL to global Set */
[self processURL:conns[i].buffer];

/* Drop the data read block end signpost */
kdebug_signpost_end(MYNEWS_CONN_DATA_RECV, i, sock, 0, 0);
}

dispatch_source_set_event_handler(source, block);
dispatch_source_set_cancel_handler(source, ^(void){
    close(sock);
});
dispatch_set_target_queue(source, queue);
dispatch_activate(source);
if (err == 0 || (err < 0 && errno != EAGAIN && errno != EINTR)) {
    dispatch_source_cancel(source);
    break;
}
if (err < 0 && errno == EAGAIN) {
    break;
}
    conns[i].index += err;

/* Add URL to global Set */
[self processURL:conns[i].buffer];

/* Drop the data read block end signpost */
kdebug_signpost_end(MYNEWS_CONN_DATA_RECV, i, sock, 0, 0);
});

dispatch_source_set_event_handler(source, block);
dispatch_source_set_cancel_handler(source, ^{
    close(sock);
});
dispatch_set_target_queue(source, queue);
dispatch_activate(source);
-(void)createConnections:(int)numberOfConnections serverPort:(int)port
{
    struct sockaddr_in serverAddr = [self server];
    conns = (struct client_connection *)malloc
        (numberOfConnections * sizeof(struct client_connection));

    for (int i = 0; i < numberOfConnections; i++) {
        int sock = socket(PF_INET, SOCK_STREAM, 0);
        int ret = connect(sock, (struct sockaddr *) &serverAddr, sizeof(serverAddr));
        assert(ret >= 0);

        int flags = fcntl(sock, F_GETFL, 0);
        fcntl(sock, F_SETFL, flags | O_NONBLOCK);

        char queue_name[1024];
        snprintf(queue_name, 1024, "com.apple.client-queue-%d", i);
        dispatch_queue_t queue = dispatch_queue_create(queue_name, DISPATCH_QUEUE_SERIAL);

        dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ, sock, 0, NULL);

        dispatch_block_t block = dispatch_block_create(DISPATCH_BLOCK_ASSUME_CURRENT, ^{
            /* Drop the data read block start signpost */
            kdebug_signpost_start(MYNEWS_CONN_DATA_RECV, i, sock, 0, 0);

            /* Re-initialize the buffer for the connection */
            bzero(&conns[i].buffer, CONNECTION_BUFFER_SIZE);

            for (int j = 0; j < MAX_CONNECTIONS; j++) {
                /* Signpost Buffer Read */
                kdebug_signpost_signpost(MYNEWS_CONN_DATA_READ, i, sock, j);
            }

            dispatch_source_set_cancel_callback(source, ^{
                if (i > 0)
                    kdebug_signpost_end(MYNEWS_CONN_DATA_RECV, i, sock, 0);
                dispatch_source_cancel(source);
            });
        });
- (void)createConnections:(int)numberOfConnections serverPort:(int)port
{
    struct sockaddr_in serverAddr = [self server];
    conns = (struct client_connection *)malloc
            (numberOfConnections * sizeof(struct client_connection));

    for (int i = 0; i < numberOfConnections; i++) {
        int sock = socket(PF_INET, SOCK_STREAM, 0);
        int ret = connect(sock, (struct sockaddr *) &serverAddr, sizeof(serverAddr));
        assert(ret >= 0);

        int flags = fcntl(sock, F_GETFL, 0);
        fcntl(sock, F_SETFL, flags | O_NONBLOCK);

        char queue_name[1024];
        snprintf(queue_name, 1024, "com.apple.client-queue-%d", i);
        dispatch_queue_t queue = dispatch_queue_create(queue_name, DISPATCH_QUEUE_SERIAL);

        dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ, sock, 0, NULL);

        dispatch_block_t block = dispatch_block_create(DISPATCH_BLOCK.Assign kotlinx
        /* Drop the data read block start signpost */
        kdebug_signpost_start(MYNEWS_CONN_DATA_RECV, i, sock, 0, 0);

        /* Re-initialize the buffer for the connection */
        bzero(&conns[i].buffer, CONNECTION_BUFFER_SIZE);
}
Demo
Finding problem spots

Daniel A. Steffen, Core Darwin
Summary

Not going off-core is ever more important
Size your work appropriately
Choose good granularity of concurrency
Modernize your GCD usage
Use tools to find problem spots
More Information

<table>
<thead>
<tr>
<th>Related Sessions</th>
<th>Location</th>
<th>Date/Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introducing Core ML</td>
<td></td>
<td>WWDC 2017</td>
</tr>
<tr>
<td>Accelerate and Sparse Solvers</td>
<td>Grand Ballroom A</td>
<td>Thursday 10:00AM</td>
</tr>
<tr>
<td>Using Metal 2 for Compute</td>
<td>Grand Ballroom A</td>
<td>Thursday 4:10PM</td>
</tr>
<tr>
<td>Writing Energy Efficient Apps</td>
<td>Executive Ballroom</td>
<td>Friday 9:00AM</td>
</tr>
<tr>
<td>App Startup Time: Past, Present, and Future</td>
<td>Hall 2</td>
<td>Friday 10:00AM</td>
</tr>
<tr>
<td>Lab Name</td>
<td>Location</td>
<td>Time</td>
</tr>
<tr>
<td>--------------------------------------------------------</td>
<td>-------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Kernel &amp; Runtime Lab</td>
<td>Technology Lab D</td>
<td>Wed 1:50PM–4:10PM</td>
</tr>
<tr>
<td>Kernel &amp; Runtime Lab</td>
<td>Technology Lab J</td>
<td>Thu 10:00AM–12:00PM</td>
</tr>
<tr>
<td>Performance Profiling and Runtime Analysis Tools Lab</td>
<td>Technology Lab K</td>
<td>Thu 1:00PM–4:10PM</td>
</tr>
<tr>
<td>Optimizing App Startup Time Lab</td>
<td>Technology Lab E</td>
<td>Fri 11:00AM–12:30PM</td>
</tr>
</tbody>
</table>