Mastering Grand Central Dispatch

Session 210

Daniel Steffen
Core OS

Grand Central Dispatch

- Introduced in Mac OS X Snow Leopard and iOS 4
- Core technology for asynchrony and concurrency
- Identical API and functionality across platforms and hardware

Grand Central Dispatch

Overview

- Brief introduction to GCD
- What is new in GCD on Mac OS X Lion and iOS 5
- Advanced usage of GCD API

Introduction to GCD

Blocks and Queues

Blocks

Encapsulate units of work

```
id obj = [Example new];
int arg = 5;
```

```
later(^{
      [obj doSomething:arg];
});
```

```
arg = 6;
[obj doSomething:arg];
[obj release];
```

Queues

Serialization

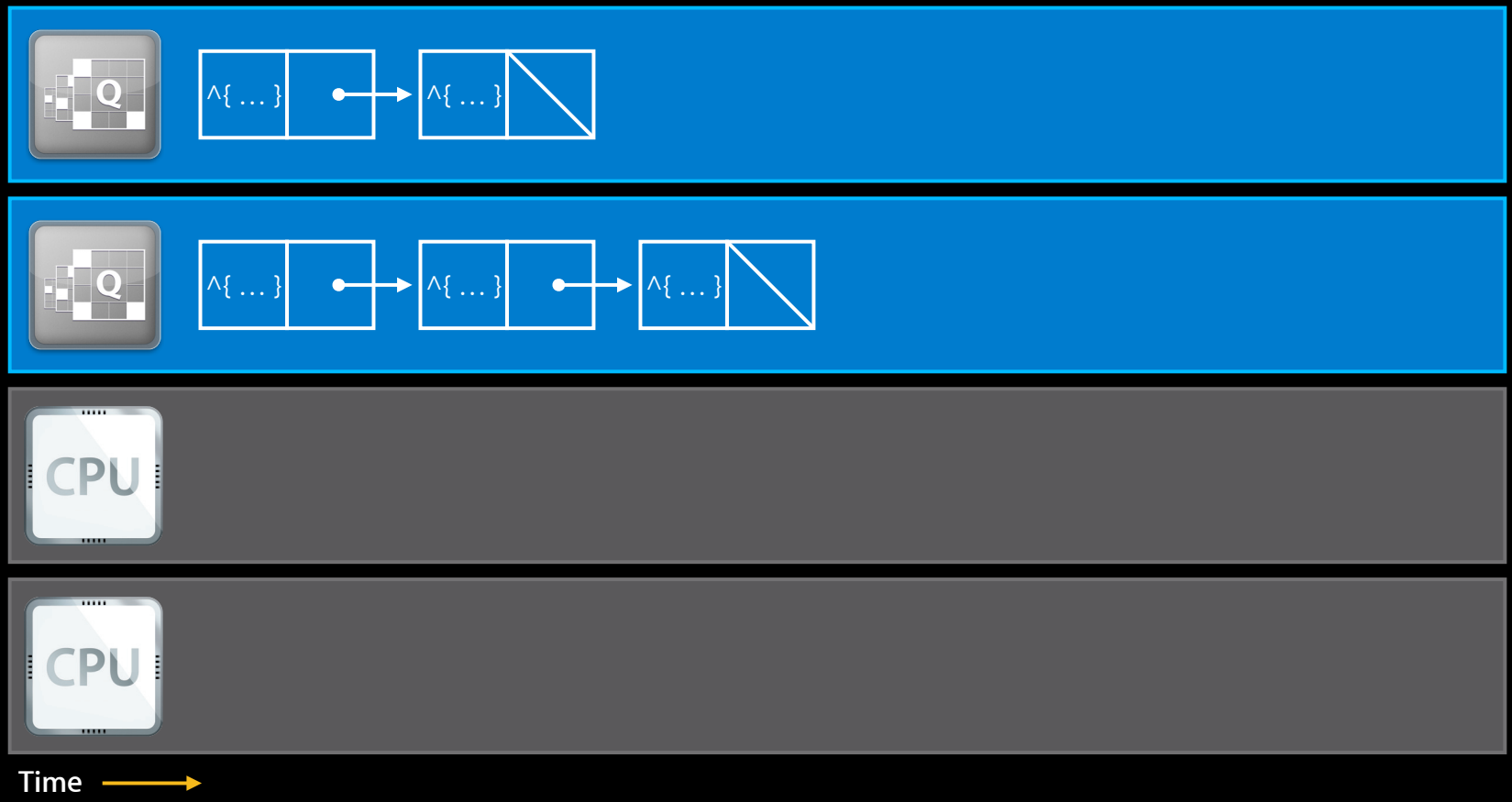
- Lightweight list of blocks
- Enqueue and dequeue are FIFO
- Serial queues execute blocks one at a time

Queues

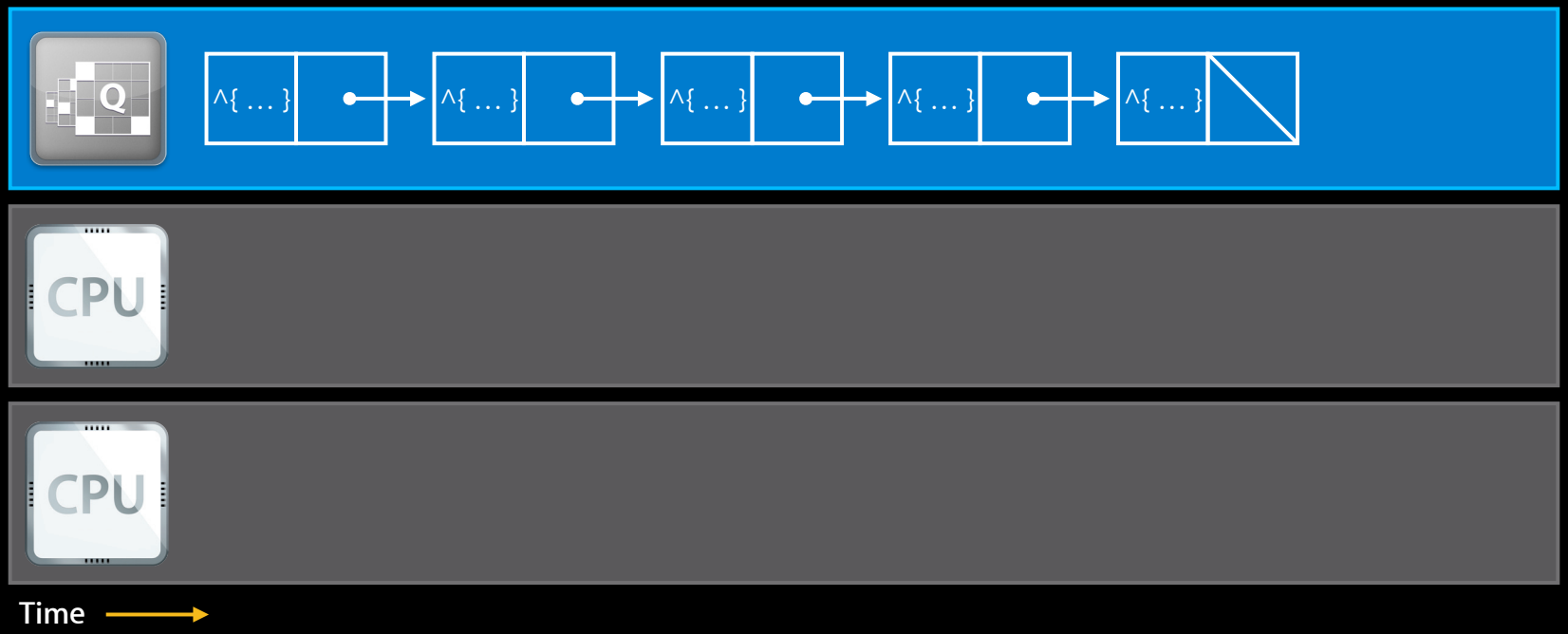
Concurrency

- Concurrent queues execute multiple blocks at the same time
- Concurrently executed blocks may complete out of order
- Queues execute concurrently with respect to other queues

Serial Queues



Concurrent Queue



Queues

API

- Submitting blocks to queues

```
dispatch_async(queue, ^{ /* Block */ });
dispatch_sync(queue,  ^{ /* Block */ });
```

- Submitting blocks later

```
dispatch_after(when, queue, ^{ /* Block */ });
```

- Concurrently executing one block many times

```
dispatch_apply(iterations, queue, ^(size_t i){ /* Block */ });
```

Queues

API

- Suspending and resuming execution

```
dispatch_suspend(queue);
dispatch_resume(queue);
```

- Managing queue lifetime

```
dispatch_retain(queue);
dispatch_release(queue);
```

Queues

Pitfalls

- Intended for flow control, not as general-purpose data structures
- Once a block is submitted to a queue, it will execute
- Be careful with synchronous API

Queues

Types

- Global queue

```
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
```

- Main queue

```
dispatch_get_main_queue();
```

- Serial queue

```
dispatch_queue_create("com.company.app.task", DISPATCH_QUEUE_SERIAL);
```

Concurrent Queues

Concurrent Queues

- Execute multiple blocks at the same time

```
dispatch_queue_create("com.company.app.task", DISPATCH_QUEUE_CONCURRENT);
```

- Can be suspended and resumed like serial queues
- Support barrier blocks

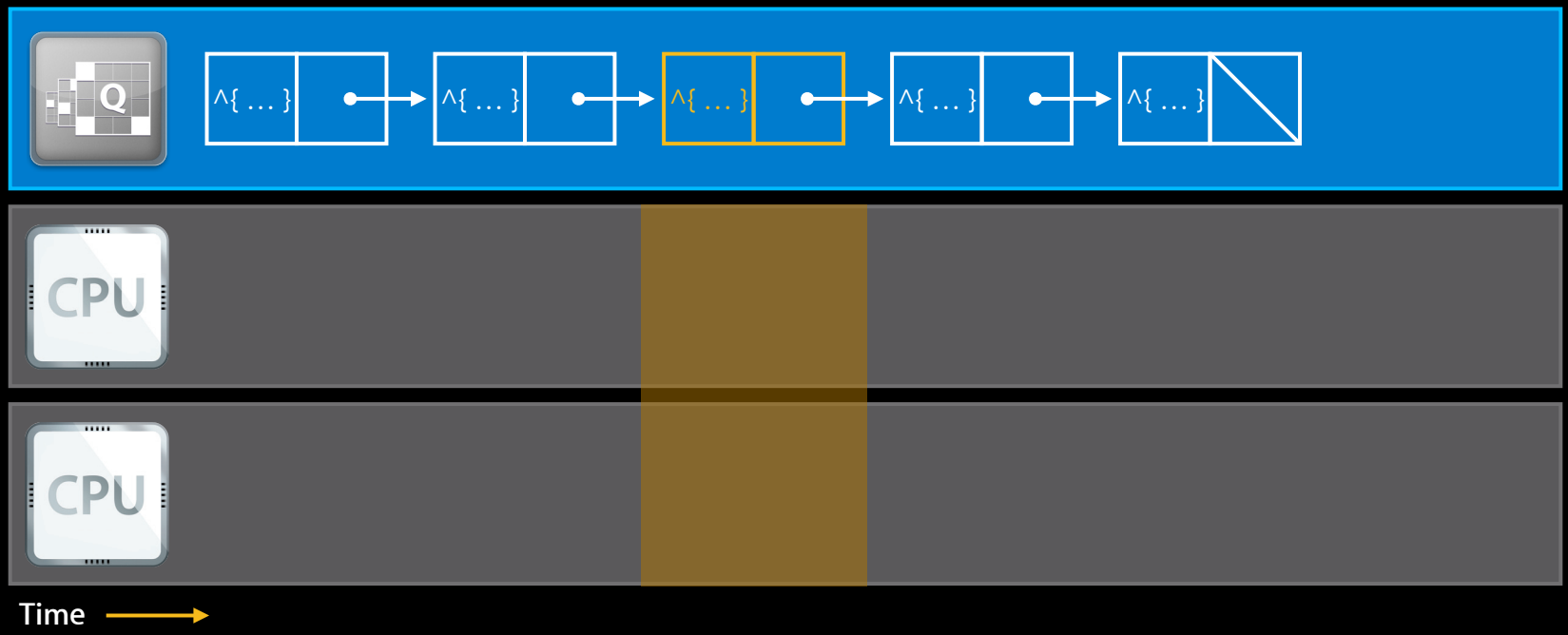
Concurrent Queues

Barrier Block

- Will not run until all blocks submitted earlier have completed
- Blocks submitted later will not run until barrier block has completed
- Submitting barrier blocks to concurrent queues

```
dispatch_barrier_async(concurrent_queue, ^{ /* Barrier */ });
dispatch_barrier_sync(concurrent_queue,  ^{ /* Barrier */ });
```

Concurrent Queue



Concurrent Queues

Implement efficient reader/writer schemes

- Many concurrent readers or a single writer (barrier)

```
dispatch_sync(concurrent_queue, ^{ /* Read */ });
dispatch_barrier_async(concurrent_queue, ^{ /* Write */ });
```

- Readers enqueued while write is in progress
- Writer enqueued while reads are in progress

Concurrent Queues

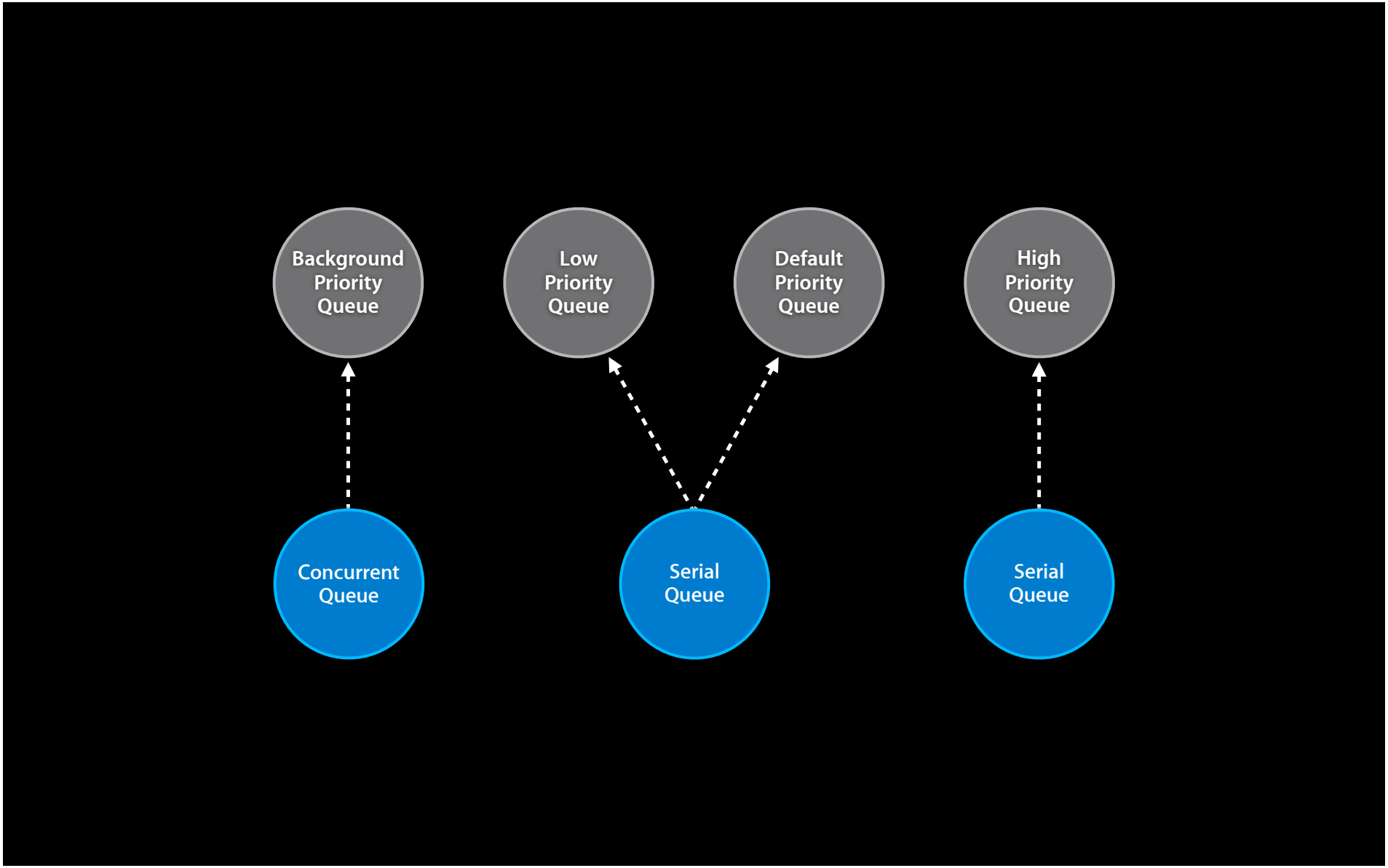
Implement efficient reader/writer schemes

```
- (id)objectAtIndex:(NSUInteger)index {
    __block id obj;
    dispatch_sync(self.concurrent_queue, ^{
        obj = [self.array objectAtIndex:index];
    });
    return obj;
}
- (void)insertObject:(id)obj atIndex:(NSUInteger)index {
    dispatch_barrier_async(self.concurrent_queue, ^{
        [self.array insertObject:obj atIndex:index];
    });
}
```

Target Queues

Target Queues

- Where the dequeue operation for a queue is executed
- Global queues are at the bottom of target queue hierarchy
- Determine scheduling and dequeue priority


Background Priority Queue
Low Priority Queue
Default Priority Queue
High Priority Queue
Concurrent Queue
Serial Queue
Serial Queue

Target Queues

Background priority

- Additional global queue priority

```
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0);
```

- Worker threads with lowest scheduling priority
- Throttled I/O

Target Queues

Hierarchy

- Setting target queue is an asynchronous barrier operation

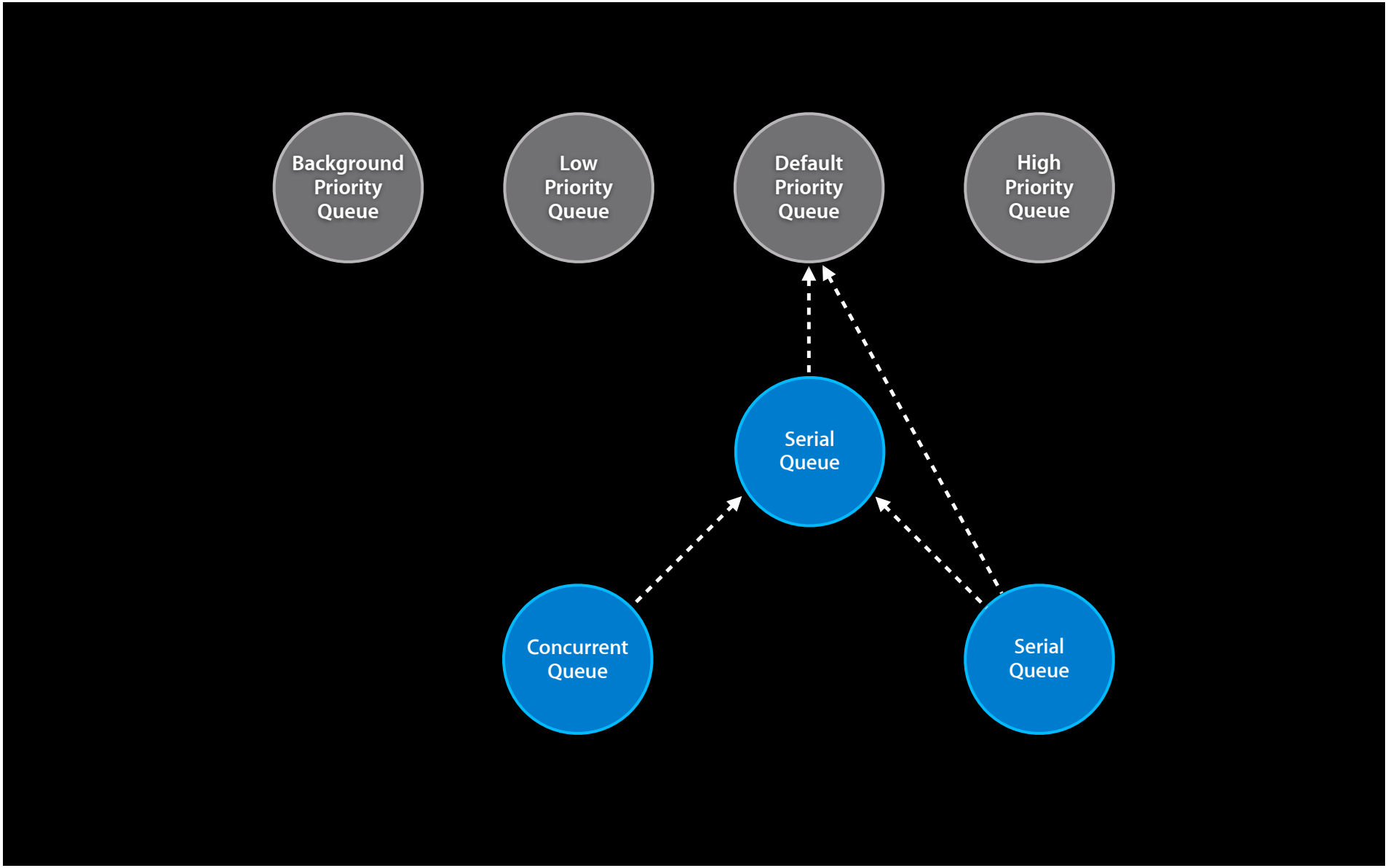
```
dispatch_set_target_queue(queue, target);
```

- Arbitrarily deep hierarchies are supported
- Loops are undefined

Target Queues

Idioms

- Setting target queue to a serial queue synchronizes with that queue
- No implicit ordering between queues


Background Priority Queue
Low Priority Queue
Default Priority Queue
High Priority Queue
Serial Queue
Concurrent Queue
Serial Queue

Target Queues

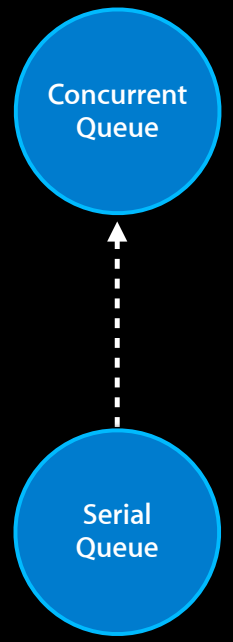
Make a concurrent queue serial

- Promote reader-writer lock to exclusive lock
- Set target queue to a serial queue
- Everything in hierarchy above a serial queue becomes serial

Target Queues

Serialize callbacks to a caller-supplied queue

- Caller's queue might be concurrent
- Setup serial queue targeting the caller-supplied queue
- Submit callbacks to this serial queue



Target Queues

Jumping the queue

- Enqueueing at the front of a serial queue?
- High priority item needs to jump ahead of already enqueued items
- Combine queue suspension with target queues

Target Queues

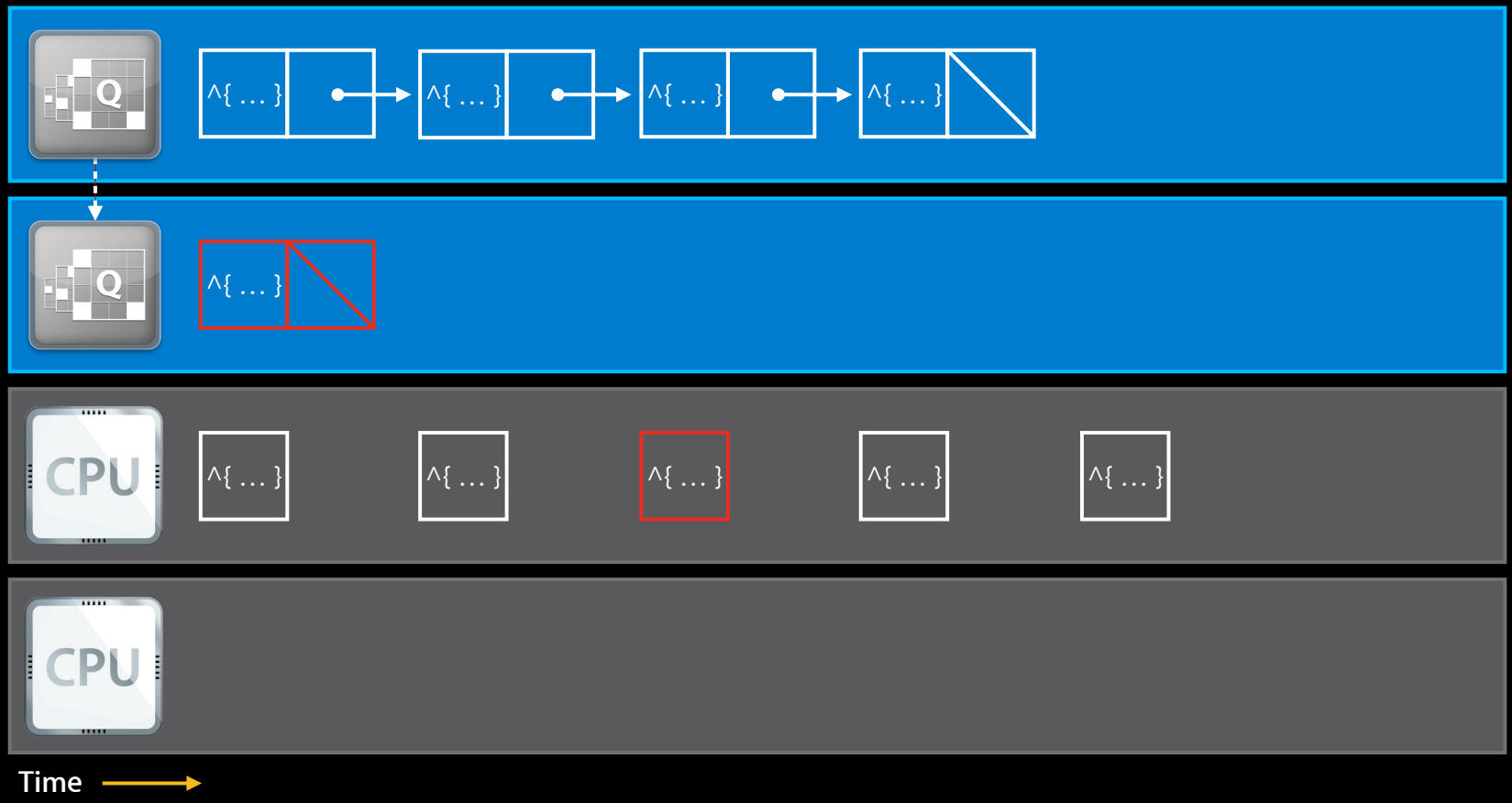
Jumping the queue

```
low  = dispatch_queue_create(“low”,  DISPATCH_QUEUE_SERIAL);
high = dispatch_queue_create(“high”, DISPATCH_QUEUE_SERIAL);
dispatch_set_target_queue(low, high);
```

```
dispatch_async(low, ^{ /* Low priority block */ });
dispatch_async(low, ^{ /* Low priority block */ });
```

```
dispatch_suspend(low);
dispatch_async(high, ^{
  /* High priority block */
  dispatch_resume(low);
});
```

Jumping the Queue



Queue-Specific Data

Queue-Specific Data

- Per-queue key-value storage

```
dispatch_queue_set_specific(queue, &key, value, destructor);
value = dispatch_queue_get_specific(queue, &key);
```

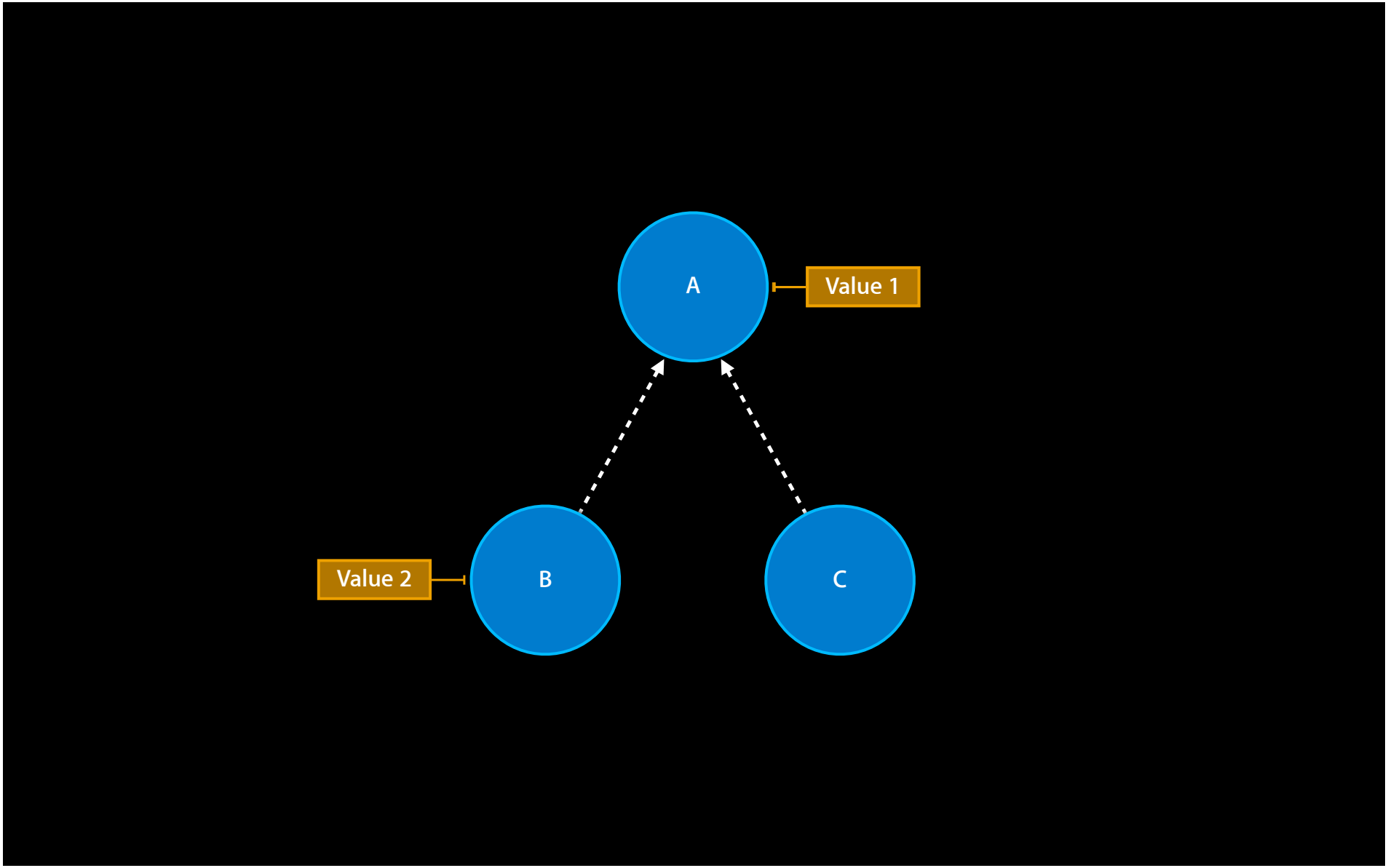
- Keys are compared as pointers
- Destructor called when value unset or at queue destruction

Queue-Specific Data

- Current value for key

```
value = dispatch_get_specific(&key);
```

- Aware of target queue hierarchy
  - Value for target queue if current queue has no value set


A
Value 1
Value 2
B
C

Dispatch Data

Dispatch Data

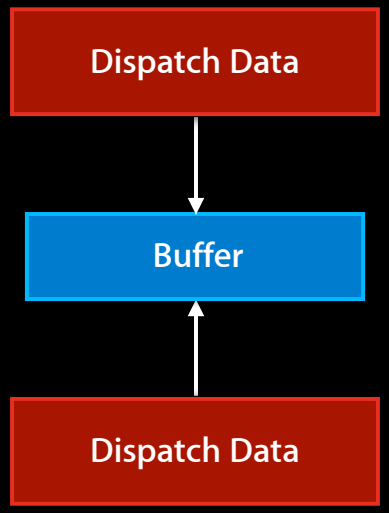
- Container object for multiple discontiguous memory buffers
- Container and represented buffers are immutable
- Avoids copying buffers as much as possible

Dispatch Data

Creation

- From app-owned buffer

```
dispatch_data_t data = dispatch_data_create(buffer, size,
                                            queue, ^{ /* destructor */ });
```



Dispatch Data

Destructors

- Copy buffer

```
DISPATCH_DATA_DESTRUCTOR_DEFAULT
```

- Malloc'd buffer

```
DISPATCH_DATA_DESTRUCTOR_FREE
  ≈ ^{ free(buffer); }
```

- Custom

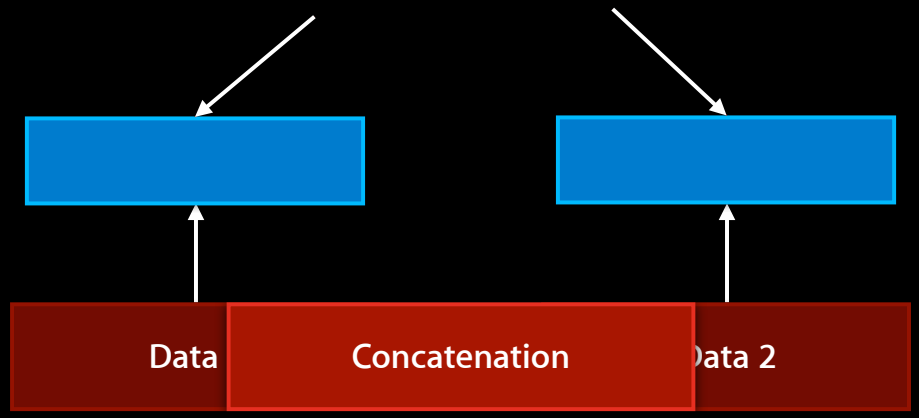
```
^{ CFRelease(cfdata); }
```

Dispatch Data

Creation

- Concatenation of data objects

```
concat = dispatch_data_create_concat(data1, data2);
```

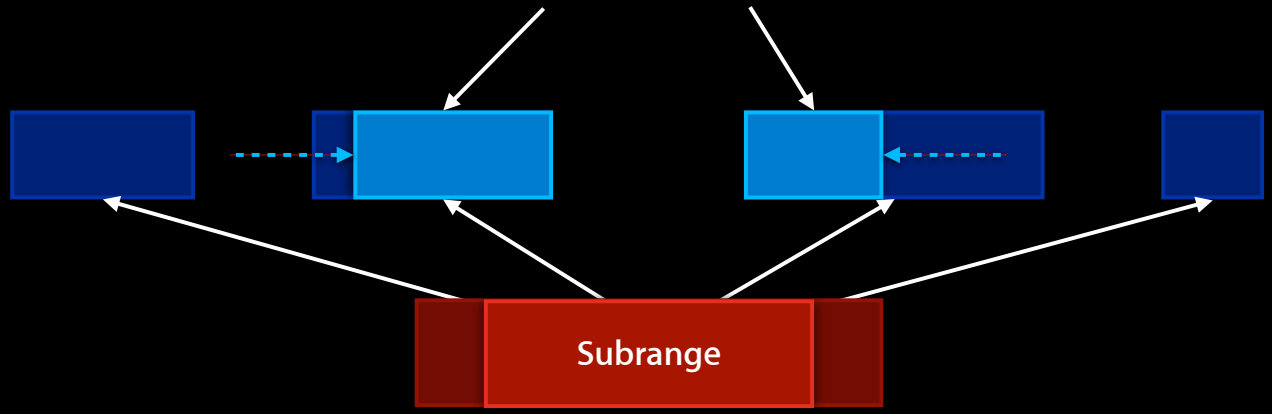


Dispatch Data

Creation

- Subrange of data object

```
subrange = dispatch_data_create_subrange(data, offset, length);
```



Dispatch Data

- Total size of represented buffers

```
size = dispatch_data_get_size(data);
```

- Singleton object for zero-sized buffer

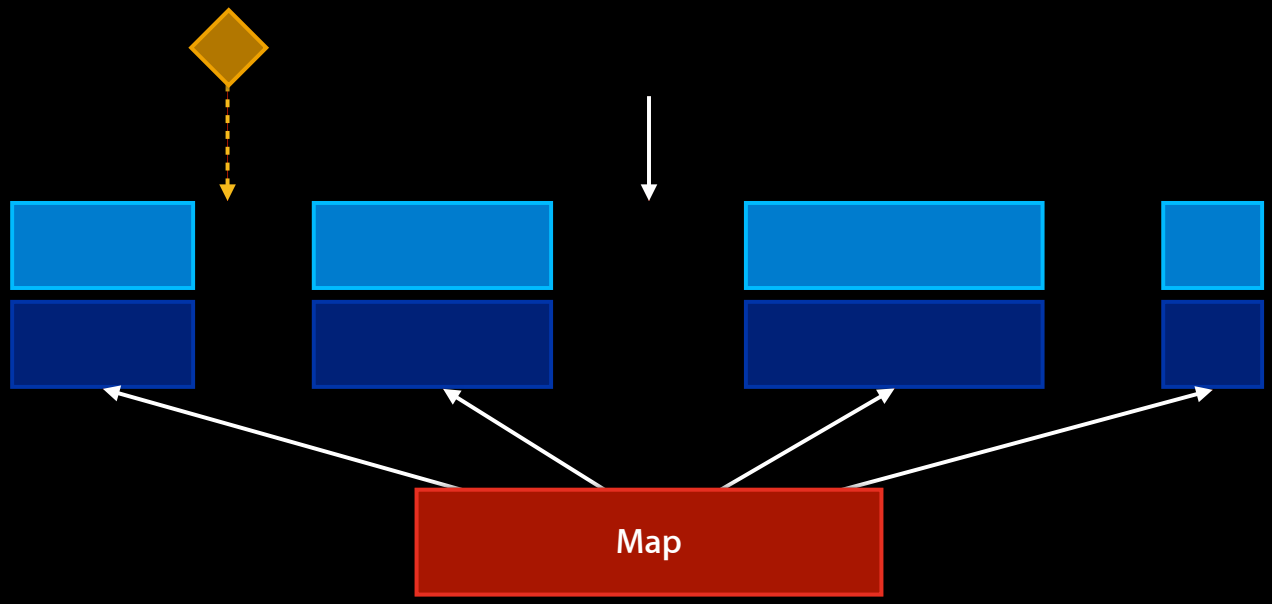
```
dispatch_data_t dispatch_data_empty;
```

Dispatch Data

Buffer access

- Copy buffers into single contiguous map

```
map = dispatch_data_create_map(data, &buffer, &size);
```

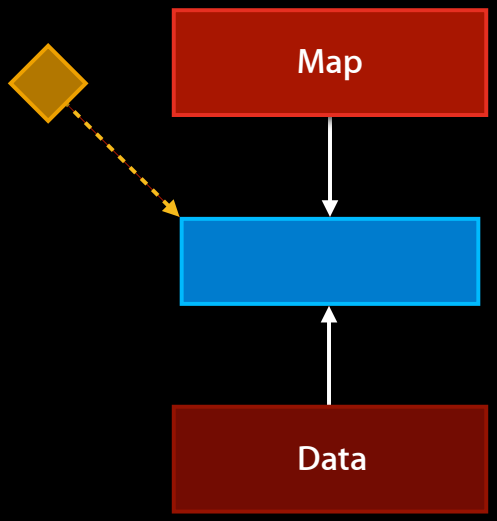


Dispatch Data

Buffer access

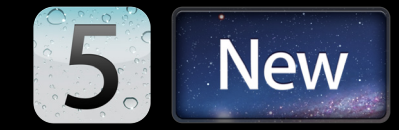
- No copy if buffer is already contiguous

```
map = dispatch_data_create_map(data, &buffer, &size);
```



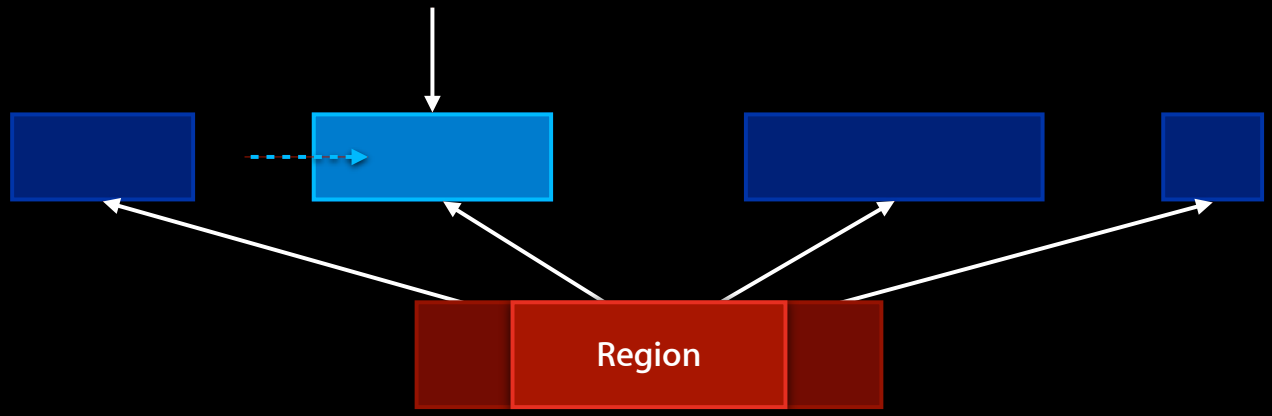
Dispatch Data

Buffer access



- Find contiguous buffer at location

```
region = dispatch_data_copy_region(data, location, &offset);
```

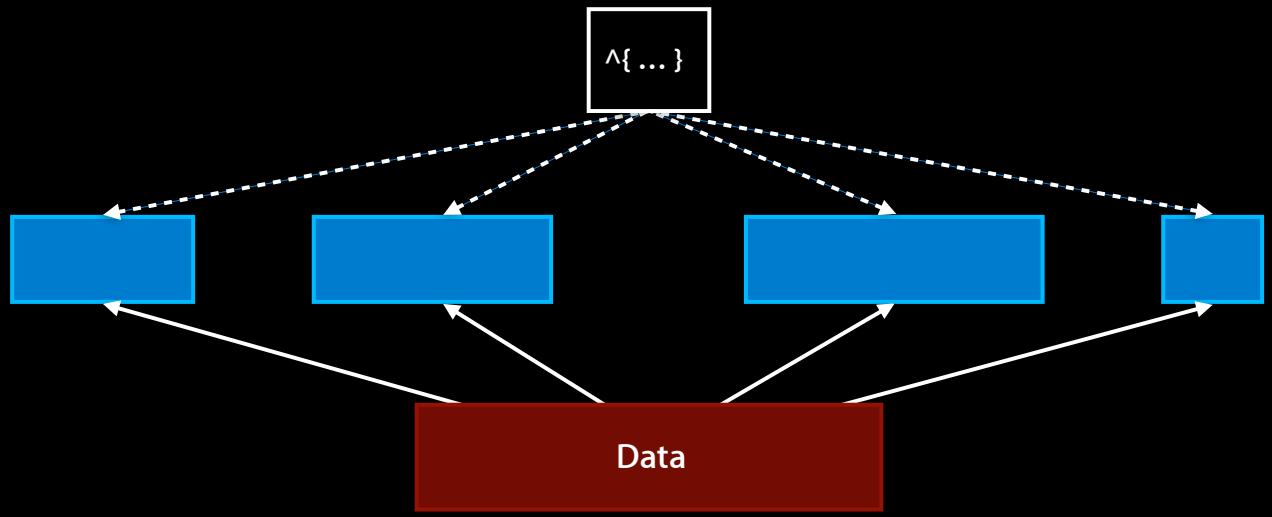


Dispatch Data

Buffer traversal

- Serially apply a block to each contiguous buffer

```
dispatch_data_apply(data, ^(dispatch_data_t region, size_t offset,
                        const void *buffer, size_t size){ /* Iterator */ });
```



Dispatch Data

Buffer traversal


```
dispatch_data_t data = acquire_data(), header;
__block size_t position = SIZE_MAX;

dispatch_data_apply(data, ^(dispatch_data_t region, size_t offset,
                                const void *buffer, size_t size){
     void *location = memchr(buffer, 0x1a, size); /* find ^Z */
     if (location) position = offset + (location - buffer);
     return (bool)!location;
});

header = dispatch_data_create_subrange(data, 0, position);
dispatch_release(data);
```

Dispatch I/O

Dispatch I/O

Goals

- Asynchronous I/O from file descriptors and paths
- Extend GCD patterns to POSIX-level file and network I/O
- Optimize I/O process-wide across subsystems

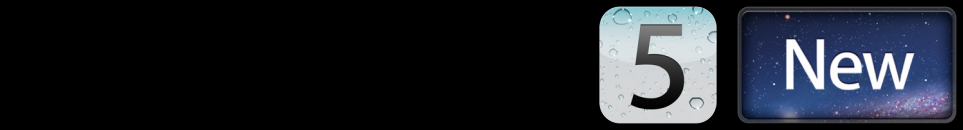
Dispatch I/O

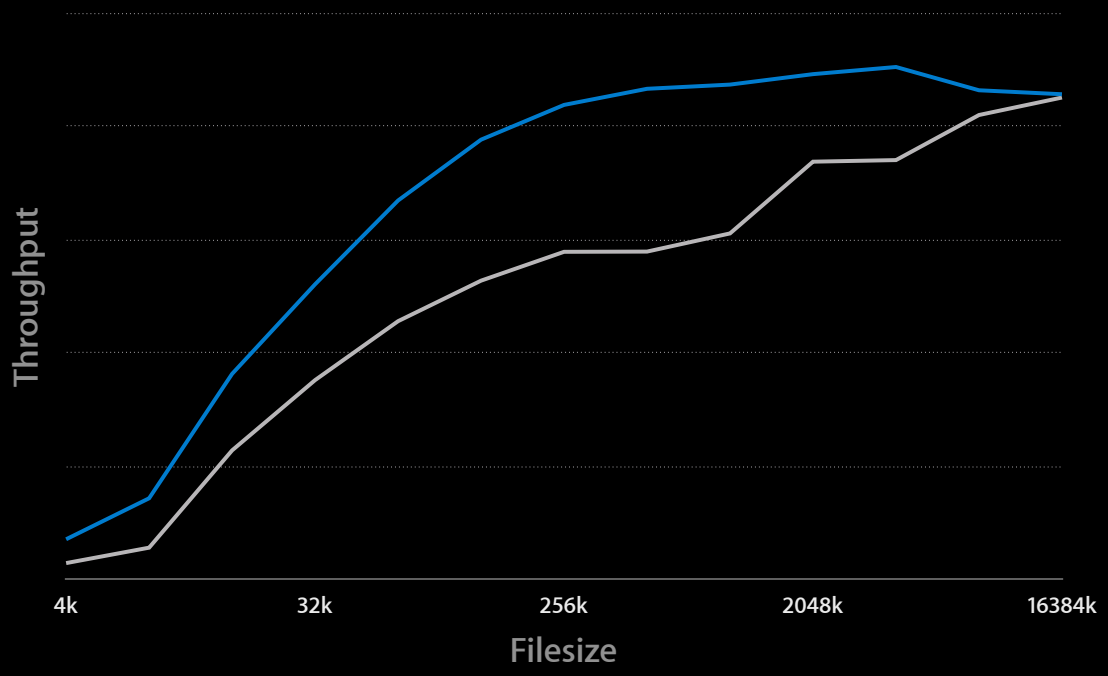
Optimizations

- Non-blocking network I/O
- Concurrent I/O to different physical devices
- Pipelining of I/O requests to single device

Dispatch I/O

Optimized throughput with advisory reads





iPad 2 (16 GB) reading 640 files

Dispatch I/O

Advantages

- Avoid threads blocked in I/O syscalls
- Manage I/O buffers with dispatch data objects
- Incremental processing of partial data

Dispatch I/O

Channels

- Encapsulate I/O policy on file descriptor or path
- Track file descriptor ownership
- Specify access type at creation

```
DISPATCH_IO_STREAM
DISPATCH_IO_RANDOM
```

Dispatch I/O

Stream-access channels

- I/O operations start at (and advance) file pointer position
- Asynchronous I/O operations are performed serially
- Reads may be performed concurrently with writes

Dispatch I/O

Random-access channels

- I/O operations start at specified offset to initial file pointer position
- Asynchronous I/O operations are performed concurrently
- File descriptor must be seekable

Dispatch I/O

Channel creation

- With file descriptor

```
dispatch_io_create(type, fd, queue, ^(…){ /* Cleanup */ });
```

- With path

```
dispatch_io_create_with_path(type, path, oflag, mode, queue,
                                                    ^(…){ /* Cleanup */ });
```

- With channel

```
dispatch_io_create_with_io(type, channel, queue, ^(…){ /* Cleanup */ });
```

Dispatch I/O

Channel cleanup

- File descriptor is under system control until cleanup handler is called
  - Must not modify file descriptor directly during this time
- Occurs once all pending I/O operations have completed and channel has been released or closed

```
^(int error){
      if (error) { /* Handle error */ }
      close(fd);
}
```

Dispatch I/O

I/O operations

- Asynchronous read at file pointer or offset

```
dispatch_io_read(channel, offset, length, queue, ^(…){ /* Handler */ });
```

- Asynchronous write at file pointer or offset

```
dispatch_io_write(channel, offset, data,  queue, ^(…){ /* Handler */ });
```

Dispatch I/O

I/O handlers

- Incremental processing
  - Must retain data if needed after handler returns
- Read operations are passed data read since last handler invocation
- Write operations are passed data not yet written

```
^(bool done, dispatch_data_t data, int error){
      if (data)  { /* Process partial data */ }
      if (error) { /* Handle error */ }
      if (done)  { /* Complete processing */ }
}
```

Dispatch I/O

Barrier operations

- Executes once pending I/O operations have completed
- Exclusive access to file descriptor
- Non-destructive modifications of file descriptor allowed

```
dispatch_io_barrier(channel, ^{
      int fd = dispatch_io_get_descriptor(channel);
      if (fsync(fd) == -1) {
         handle_error(errno);
     }
});
```

Dispatch I/O

Closing channels

- Close to new operations but let existing operations complete

```
dispatch_io_close(channel, 0);
```

- Close and interrupt existing operations

```
dispatch_io_close(channel, DISPATCH_IO_STOP);
```

Dispatch I/O

Transliteration


```
dispatch_io_t in, out;
dispatch_queue_t q = dispatch_get_global_queue(
                                DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
```

```
in = dispatch_io_create(DISPATCH_IO_RANDOM, fd, q, ^(int err){
     if (err) handle_error(err);
     close(fd);
});
out = dispatch_io_create_with_path(DISPATCH_IO_STREAM,
            path, O_WRONLY|O_CREAT, 0600, q, ^(int err){
     if (err) handle_error(err);
});
```

Dispatch I/O

Transliteration


```
dispatch_io_read(in, 0, SIZE_MAX, q,
            ^(bool done, dispatch_data_t data, int err){
      if (data) {
         dispatch_data_t tdata = transliterate(data);
         dispatch_io_write(out, 0, tdata, q,
                  ^(bool wdone, dispatch_data_t wdata, int werr){
            if (werr) handle_error(werr); });
         dispatch_release(tdata);
      }
      if (done) dispatch_release(out);
      if (err) handle_error(err); });
dispatch_release(in);
```

Summary

- Target queues
- Concurrent queues
- Queue-specific data
- Dispatch data
- Dispatch I/O

More Information

Michael Jurewitz
Developer Tools and Performance Evangelist
jurewitz@apple.com

Documentation
Concurrency Programming Guide
http://developer.apple.com

Open Source
Mac OS Forge > libdispatch
http://libdispatch.macosforge.org

Apple Developer Forums
http://devforums.apple.com

Related Sessions

Blocks and Grand Central Dispatch in Practice	Pacific Heights Wednesday 10:15AM
Introducing XPC	Russian Hill Wednesday 11:30AM

Introducing Blocks and Grand Central Dispatch on iPhone	WWDC10 iTunes
Simplifying iPhone App Development with Grand Central Dispatch	WWDC10 iTunes

Labs

Grand Central Dispatch Lab	CoreOS Lab A Thursday 2:00PM - 3:45PM


WWDC2011