

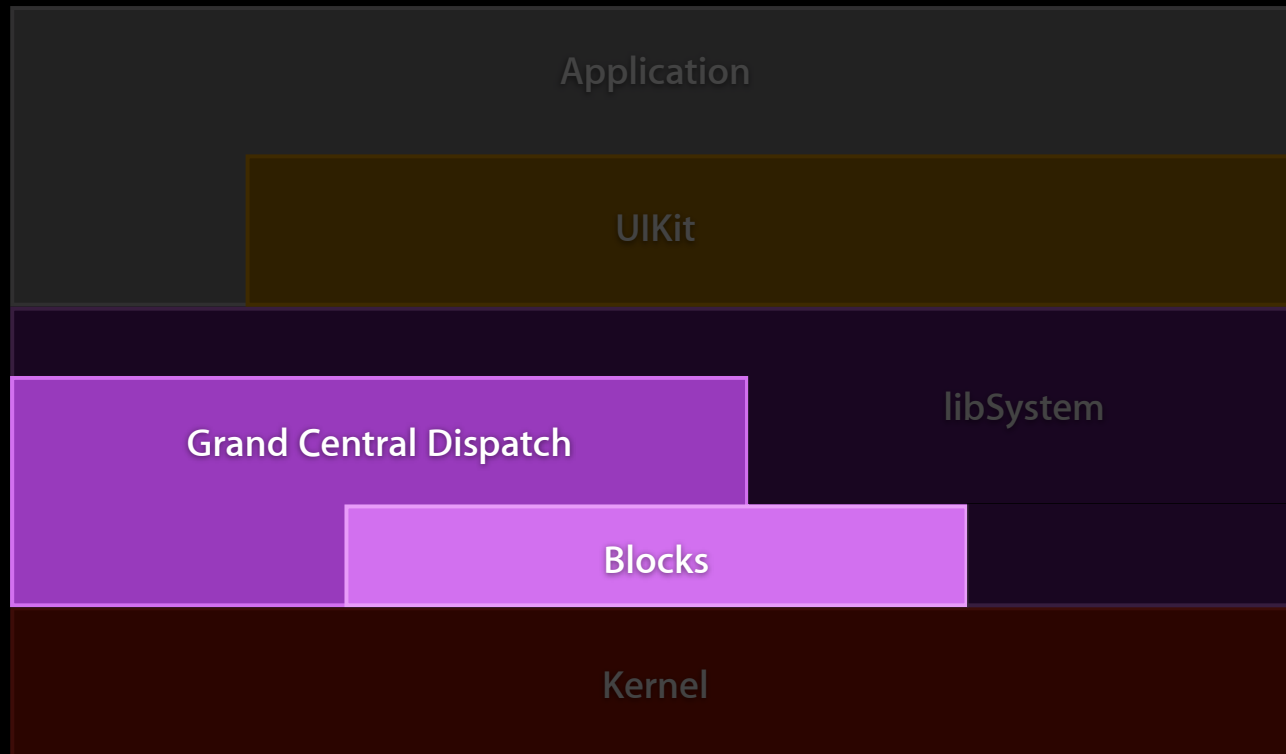
Blocks and Grand Central Dispatch in Practice

Session 308

Dave Zarzycki
Developer Technologies

These are confidential sessions—please refrain from streaming, blogging, or taking pictures

Technology Stack

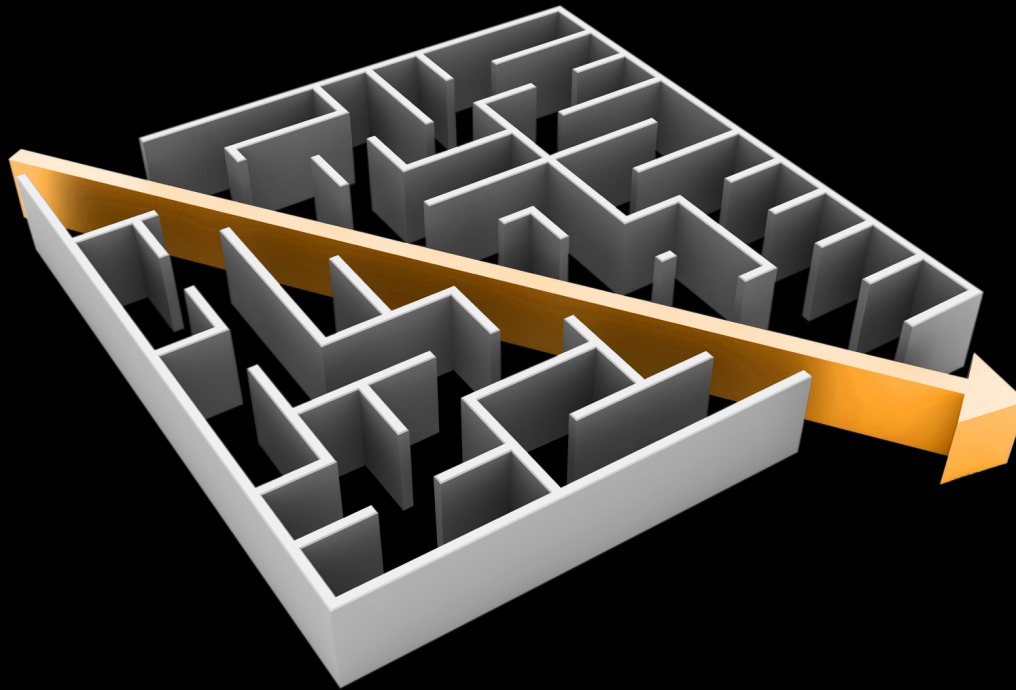


Agenda

- Introduction
 - Blocks
 - Grand Central Dispatch
- Memory management
 - What blocks automate
 - What blocks do not automate

Blocks

Blocks Simplify Function Callbacks



Functions

```
// body of code  
{  
    return a - b;  
}
```

Functions

```
// body of code  
{  
    return a - b;  
}
```

Blocks

```
// body of code  
{  
    return a - b;  
}
```

Functions

```
// body of code  
{  
    return a - b;  
}
```

Blocks

```
// body of code  
{  
    return a - b;  
}
```




Functions

```
// body of code  
{  
    return a - b;  
}
```

```
// pointer to a function  
*
```

Blocks

```
// body of code  
{  
    return a - b;  
}
```

```
// pointer to a block  
^
```

Functions

```
// body of code  
{  
    return a - b;  
}
```

```
// pointer to a function  
int (*cmpr)(int, int);
```

Blocks

```
// body of code  
{  
    return a - b;  
}
```

```
// pointer to a block  
int (^cmpr)(int, int);
```

Functions

```
// body of code  
{  
    return a - b;  
}
```

```
// pointer to a function  
int (*cmpr)(int, int);
```

```
// better style  
typedef int (*func_t)(int, int);
```

Blocks

```
// body of code  
{  
    return a - b;  
}
```

```
// pointer to a block  
int (^cmpr)(int, int);
```

```
// better style  
typedef int (^block_t)(int, int);
```

Functions

```
// body of code
{
    return a - b;
}
```

```
// pointer to a function
int (*cmpr)(int, int);
```

```
// better style
typedef int (*func_t)(int, int);
func_t cmpr = arg;
```

Blocks

```
// body of code
{
    return a - b;
}
```

```
// pointer to a block
int (^cmpr)(int, int);
```

```
// better style
typedef int (^block_t)(int, int);
block_t cmpr = arg;
```

Functions

```
// body of code  
{  
    return a - b;  
}
```

```
// pointer to a function  
int (*cmpr)(int, int);
```

```
// better style  
typedef int (*func_t)(int, int);  
func_t cmpr = arg;  
cmpr(x, y);
```

Blocks

```
// body of code  
{  
    return a - b;  
}
```

```
// pointer to a block  
int (^cmpr)(int, int);
```

```
// better style  
typedef int (^block_t)(int, int);  
block_t cmpr = arg;  
cmpr(x, y);
```

Functions

```
// body of code  
{  
    return a - b;  
}
```

Blocks

```
// body of code  
{  
    return a - b;  
}
```

Functions

```
// body of code  
  
    return a - b;  
}
```

Blocks

```
// body of code  
  
    return a - b;  
}
```


Functions

```
// body of code  
    (int a, int b) {  
        return a - b;  
    }
```

Blocks

```
// body of code  
    (int a, int b) {  
        return a - b;  
    }
```

Functions

```
// body of code  
int my_cmp(int a, int b) {  
    return a - b;  
}
```

Blocks

```
// body of code  
^(int a, int b) {  
    return a - b;  
}
```

Functions

```
// body of code
int my_cmp(int a, int b) {
    return a - b;
}

// callee
void sort(int *, int, func_t);
```

Blocks

```
// body of code
^(int a, int b) {
    return a - b;
}

// callee
void sort(int *, int, block_t);
```

Functions

```
// body of code
int my_cmp(int a, int b) {
    return a - b;
}

// callee
void sort(int *, int, func_t);

// usage
sort(array, 10, my_cmp);
```

Blocks

```
// body of code
^(int a, int b) {
    return a - b;
}

// callee
void sort(int *, int, block_t);
```

Functions

```
// body of code
int my_cmp(int a, int b) {
    return a - b;
}

// callee
void sort(int *, int, func_t);

// usage
sort(array, 10, my_cmp);
```

Blocks

```
// body of code
^(int a, int b) {
    return a - b;
}

// callee
void sort(int *, int, block_t);

// usage
sort(array, 10, ^(int a, int b) {
    return a - b;
});
```

Trivial Blocks vs. Nontrivial Functions

Implementing a Configurable Sort

Functions

Blocks

Functions

Blocks

```
sort(array, 10, ^(int a, int b) {  
    return a - b;  
});
```

Functions

Blocks

```
bool rev = arg;
```

```
sort(array, 10, ^(int a, int b) {  
    return a - b;  
});
```

Functions

Blocks

```
bool rev = arg;
```

```
sort(array, 10, ^(int a, int b) {  
    if (rev) return b - a;  
    else return a - b;  
});
```

Functions

Blocks

```
bool rev = arg;
```

```
sort(array, 10, ^(int a, int b) {  
    if (rev) return b - a;  
    else return a - b;  
});
```

Functions

```
bool rev = arg;
```

```
sort(array, 10, my_cmp);
```

Blocks

```
bool rev = arg;
```

```
sort(array, 10, ^(int a, int b) {  
    if (rev) return b - a;  
    else return a - b;  
});
```

Functions

```
struct data_s d = { arg };
```

```
sort(array, 10, my_cmp);
```

Blocks

```
bool rev = arg;
```

```
sort(array, 10, ^(int a, int b) {  
    if (rev) return b - a;  
    else return a - b;  
});
```

Functions

```
struct data_s d = { arg };
```

```
sort(array, 10, &d, my_cmp);
```

Blocks

```
bool rev = arg;
```

```
sort(array, 10, ^(int a, int b) {  
    if (rev) return b - a;  
    else return a - b;  
});
```

Functions

```
struct data_s d = { arg };
```

```
sort(array, 10, &d, my_cmp);
```

```
////////////////////////////////////
```

Blocks

```
bool rev = arg;
```

```
sort(array, 10, ^(int a, int b) {  
    if (rev) return b - a;  
    else return a - b;  
});
```


Functions

```
struct data_s {  
    bool rev;  
};
```

```
////////////////////////////////////  
struct data_s d = { arg };  
sort(array, 10, &d, my_cmp);
```

Blocks

```
bool rev = arg;  
  
sort(array, 10, ^(int a, int b) {  
    if (rev) return b - a;  
    else return a - b;  
});
```

Functions

```
struct data_s {  
    bool rev;  
};
```

```
int my_cmp(int a, int b)  
{  
    return a - b;  
}
```

```
////////////////////////////////////  
struct data_s d = { arg };  
sort(array, 10, &d, my_cmp);
```

Blocks

```
bool rev = arg;
```

```
sort(array, 10, ^(int a, int b) {  
    if (rev) return b - a;  
    else return a - b;  
});
```

Functions

```
struct data_s {  
    bool rev;  
};  
  
int my_cmp(int a, int b)  
{  
  
    return a - b;  
}  
  
/////////////////////////////////  
struct data_s d = { arg };  
sort(array, 10, &d, my_cmp);
```

Blocks

```
bool rev = arg;  
  
sort(array, 10, ^(int a, int b) {  
    if (rev) return b - a;  
    else return a - b;  
});
```

Functions

```
struct data_s {
    bool rev;
};

int my_cmp(void *ctxt, int a, int b)
{
    struct data_s *d = ctxt;
    if (d->rev) return b - a;
    else return a - b;
}

////////////////////////////////////
struct data_s d = { arg };
sort(array, 10, &d, my_cmp);
```

Blocks

```
bool rev = arg;

sort(array, 10, ^(int a, int b) {
    if (rev) return b - a;
    else return a - b;
});
```

Functions

```
struct data_s {
    bool rev;
};

int my_cmp(void *ctxt, int a, int b)
{
    struct data_s *d = ctxt;
    if (d->rev) return b - a;
    else return a - b;
}

////////////////////////////////////
struct data_s d = { arg };
sort(array, 10, &d, my_cmp);
```

Blocks

```
bool rev = arg;

sort(array, 10, ^(int a, int b) {
    if (rev) return b - a;
    else return a - b;
});
```

Extracting Results

Functions

```
struct data_s {
    bool rev;
};

int my_cmp(void *ctxt, int a, int b)
{
    struct data_s *d = ctxt;
    if (d->rev) return b - a;
    else return a - b;
}

////////////////////////////////////
struct data_s d = { arg };
sort(array, &d, my_cmp);
```

Blocks

```
bool rev = arg;

sort(array, ^(int a, int b) {
    if (rev) return b - a;
    else return a - b;
});
```

Functions

```
struct data_s {
    bool rev;
};

int my_cmp(void *ctxt, int a, int b)
{
    struct data_s *d = ctxt;
    if (d->rev) return b - a;
    else return a - b;
}

////////////////////////////////////
struct data_s d = { arg };
sort(array, &d, my_cmp);
```

Blocks

```
bool rev = arg;
    int cnt = 0;

sort(array, ^(int a, int b) {
    cnt++;
    if (rev) return b - a;
    else return a - b;
});

log("Count: %d", cnt);
```


Functions

```
struct data_s {
    bool rev;
};

int my_cmp(void *ctxt, int a, int b)
{
    struct data_s *d = ctxt;
    if (d->rev) return b - a;
    else return a - b;
}

////////////////////////////////////
struct data_s d = { arg };
sort(array, &d, my_cmp);
```

Blocks

```
bool rev = arg;
__block int cnt = 0;

sort(array, ^(int a, int b) {
    cnt++;
    if (rev) return b - a;
    else return a - b;
});

log("Count: %d", cnt);
```

Functions

```
struct data_s {
    bool rev;
};

int my_cmp(void *ctxt, int a, int b)
{
    struct data_s *d = ctxt;
    if (d->rev) return b - a;
    else return a - b;
}

////////////////////////////////////
struct data_s d = { arg };
sort(array, &d, my_cmp);
```

Blocks

```
bool rev = arg;
__block int cnt = 0;

sort(array, ^(int a, int b) {
    cnt++;
    if (rev) return b - a;
    else return a - b;
});

log("Count: %d", cnt);
```

Functions

```
struct data_s {
    bool rev;
};

int my_cmp(void *ctxt, int a, int b)
{
    struct data_s *d = ctxt;

    if (d->rev) return b - a;
    else return a - b;
}

////////////////////////////////////

struct data_s d = {
    arg,

};

sort(array, &d, my_cmp);
```

Blocks

```
bool rev = arg;
__block int cnt = 0;

sort(array, ^(int a, int b) {
    cnt++;
    if (rev) return b - a;
    else return a - b;
});

log("Count: %d", cnt);
```

Functions

```
struct data_s {
    bool rev;
    int *out_count;
};

int my_cmp(void *ctxt, int a, int b)
{
    struct data_s *d = ctxt;
    (*d->out_count)++;
    if (d->rev) return b - a;
    else return a - b;
}

////////////////////
int cnt = 0;
struct data_s d = {
    arg,
    &cnt
};

sort(array, &d, my_cmp);

log("Count: %d", cnt);
```

Blocks

```
bool rev = arg;
__block int cnt = 0;

sort(array, ^(int a, int b) {
    cnt++;
    if (rev) return b - a;
    else return a - b;
});

log("Count: %d", cnt);
```

Functions

```
struct data_s {
    bool rev;
    int *out_count;
};

int my_cmp(void *ctxt, int a, int b)
{
    struct data_s *d = ctxt;
    (*d->out_count)++;
    if (d->rev) return b - a;
    else return a - b;
}

////////////////////

int cnt = 0;
struct data_s d = {
    arg,
    &cnt
};

sort(array, &d, my_cmp);

log("Count: %d", cnt);
```

Blocks

```
bool rev = arg;
__block int cnt = 0;

sort(array, ^(int a, int b) {
    cnt++;
    if (rev) return b - a;
    else return a - b;
});

log("Count: %d", cnt);
```

Blocks and Apple

Many APIs use them

Examples

Enumeration

```
[dict enumerateKeysAndObjectsUsingBlock: ^(id key, id obj, BOOL *stop) {  
    NSLog(@"%@ = %@", key, object);  
}];
```

```
[array enumerateObjectsUsingBlock: ^(id obj, NSUInteger idx, BOOL *stop) {  
    NSLog(@"%lu = %@", idx, object);  
}];
```

```
// and more!
```

Examples

Callbacks

```
^(void) { ... }
```


Examples

Callbacks

```
^(void) { ... }
```

```
^{ ... }
```

Grand Central Dispatch

Kevin Van Vechten
Core OS

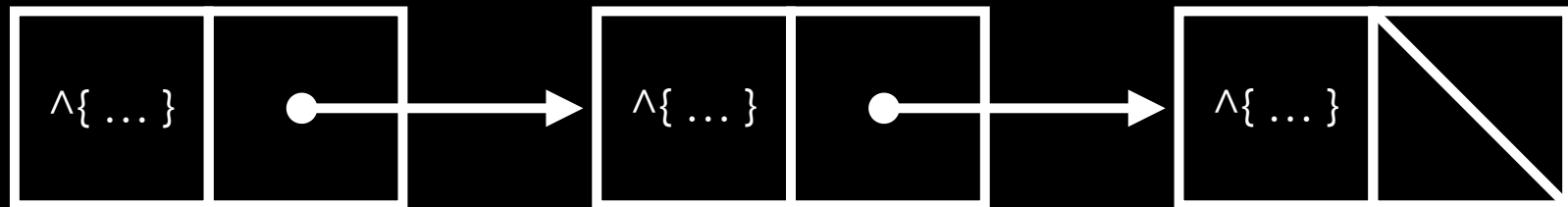
Grand Central Dispatch



- Execute blocks on queues
 - Serialized
 - Concurrent
 - Asynchronous

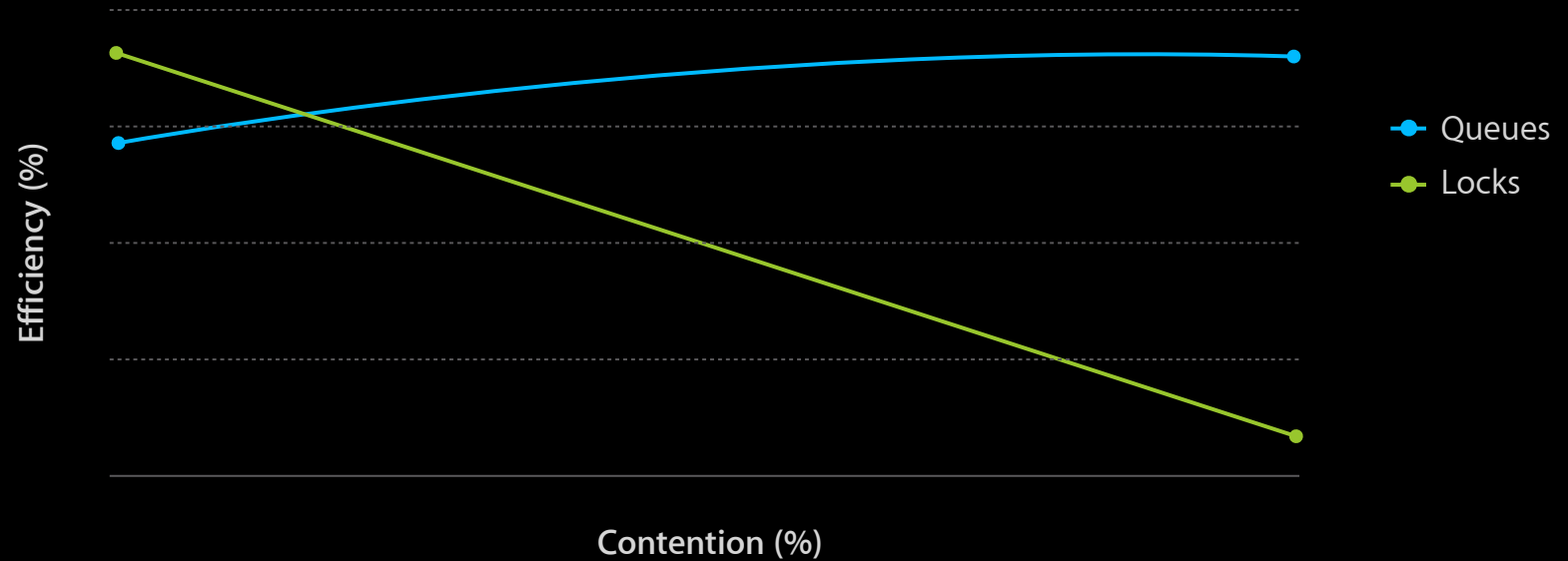
Queues

- FIFO
- Atomic enqueue
- Automatic dequeue



Throughput

Throughput efficiency vs. contention



Scaleability

From one to many cores



Scaleability

From one to many cores

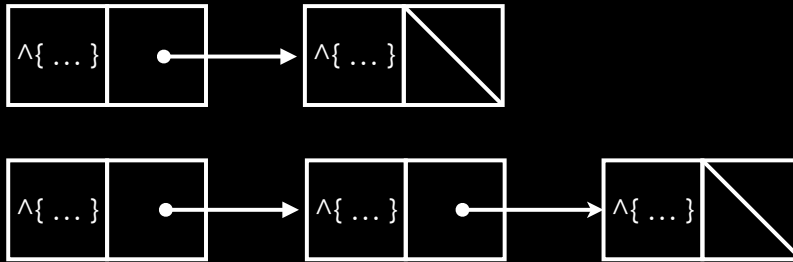


Scaleability

From one to many cores



Automatic Threads



Dispatch Fundamentals

- `dispatch_queue_t`
- `dispatch_sync`
- `dispatch_apply`
- `dispatch_async`

dispatch_queue_t

A dispatch queue object

- Standard create/retain/release semantics

```
dispatch_queue_t q = dispatch_queue_create("com.example.myqueue", NULL);
```

```
// must pair retain & release
```

```
dispatch_retain(q);
```

```
dispatch_release(q);
```

```
// last release deallocates
```

```
dispatch_release(q);
```

`dispatch_sync`

dispatch_sync

Enqueue a block for synchronous execution

- Useful to implement critical sections

```
bool debit_account(Account *account, Transaction *transaction)
{
    dispatch_sync(account->queue, ^{
        account->balance -= transaction->amount;
    });
    return true;
}
```

dispatch_sync

Advanced patterns

- Use __block to modify enclosing scope
- Use return to safely leave critical section

```
bool debit_account(Account *account, Transaction *transaction)
{
    __block bool result = false;
    dispatch_sync(account->queue, ^{
        if (transaction->amount > account->balance) return;
        account->balance -= transaction->amount;
        result = true;
    });
    return result;
}
```

dispatch_sync

Caveat

- Queues are strictly FIFO therefore nested dispatch_sync *will* deadlock

```
dispatch_sync(queue, ^{  
    dispatch_sync(queue, ^{  
        // NOT REACHED: DEADLOCK  
    });  
});
```

`dispatch_apply`

dispatch_apply

Data-level parallelism with blocks

- Scales with number of cores and concurrent dispatch_apply operations

```
// for (index = 0; index < count; index++) {  
dispatch_apply(count, queue, ^(size_t index) {  
    outputs[index] = perform_computation(inputs[index]);  
});
```

```
dispatch_apply(count, queue, ^(size_t index) { ... });
```



dispatch_apply

Caveat

- Watch out for hidden locks that negate performance benefits

```
dispatch_apply(count, queue, ^(size_t index) {  
    printf("%lu\n", index);  
});
```

dispatch_apply

Striding

- Amortize costs for tiny operations
- Avoid false cache sharing

```
#define STRIDE (10 * (CACHE_LINE_SIZE / sizeof(double))) // measure & tune!  
dispatch_apply(count / STRIDE, queue, ^(size_t index) {  
    size_t j = index * STRIDE;  
    size_t j_stop = j + STRIDE;  
    do {  
        outputs[index] = perform_computation(inputs[index]);  
    } while (j < j_stop);  
});
```

`dispatch_async`

dispatch_async

Enqueue a block for *asynchronous* execution

- Useful to implement deferred critical sections
- Returns immediately

```
void calculate_interest(Account *account)
{
    dispatch_async(account->queue, ^{
        account->balance += account->balance * INTEREST_RATE;
    });
}
```

dispatch_async



- Move work off the main thread
 - Stay responsive to UI events
- Deferred execution of tasks
 - Automatic concurrency

dispatch_async

Nested invocations provide asynchronous callbacks

- Communication between subsystems
- Useful pattern to avoid blocking the main thread

```
-(IBAction)onClick:(UIButton *)sender
{
    dispatch_async(account->queue, ^{
        UIImageRep *image = renderAccountStatement(account);
        dispatch_async(dispatch_get_main_queue(), ^{
            [image draw];
        });
    });
}
```


dispatch_async

Advanced patterns

- Use queue—block callback pair as last arguments to async functions

```
void renderAccountStatementAsync(Account *account,  
    dispatch_queue_t queue, my_callback_t block);
```

```
-(IBAction)onClick:(NSButton *)sender  
{  
    renderAccountStatementAsync(account, dispatch_get_main_queue(),  
        ^(UIImageRep *image) {  
            [image draw];  
        });  
}
```

dispatch_async

Caveat

- dispatch_queue_t must be retained in nested blocks
- C dynamic allocations must be manually copied/retained

```
void myAsync(dispatch_queue_t queue, my_callback_t block)
{
    dispatch_retain(queue);
    dispatch_async(background_queue, ^{
        dispatch_async(queue, ^{
            block();
        });
        dispatch_release(queue);
    });
}
```

Blocks and Memory Management

Dave Zarzycki
Developer Technologies

Functions

```
void dispatch_async_f(  
    dispatch_queue_t queue,  
    void *context,  
    dispatch_function_t);
```

Blocks

```
void dispatch_async(  
    dispatch_queue_t queue,  
    dispatch_block_t block);
```

Functions

```
void dispatch_async_f(  
    dispatch_queue_t queue,  
    void *context,  
    dispatch_function_t);
```

Blocks

```
void dispatch_async(  
    dispatch_queue_t queue,  
    dispatch_block_t block)  
{  
    dispatch_async_f(queue,  
        Block_copy(block),  
        _static_helper);  
}
```

Functions

```
void dispatch_async_f(  
    dispatch_queue_t queue,  
    void *context,  
    dispatch_function_t);
```

Blocks

```
void dispatch_async(  
    dispatch_queue_t queue,  
    dispatch_block_t block)  
{  
    dispatch_async_f(queue,  
        Block_copy(block),  
        _static_helper);  
}  
  
void _static_helper(void *ctxt) {  
    dispatch_block_t b = ctxt;  
    b();  
    Block_release(b);  
}
```

Functions

```
void dispatch_async_f(  
    dispatch_queue_t queue,  
    void *context,  
    dispatch_function_t);
```

Blocks

```
void dispatch_async(  
    dispatch_queue_t queue,  
    dispatch_block_t block)  
{  
    dispatch_async_f(queue,  
        Block_copy(block),  
        _static_helper);  
}  
  
void _static_helper(void *ctxt) {  
    dispatch_block_t b = ctxt;  
    b();  
    Block_release(b);  
}
```

Blocks Trivially Wrap Traditional APIs

What Does Block_copy() Do?

- Automatically copies values
 - Integers, floats, pointers, etc
 - Shared variables are forced with __block
- Automatically copies and releases other blocks
- Automatically retains and releases Objective-C objects
- Automatically calls C++ copy constructors and destructors
 - Use the Apple LLVM 3.0 Compiler

What Does Block_copy() **Not** Do?

Read your mind...

```
dispatch_async(queue, ^{  
  
    [_ivar doSomething];  
});
```

What Does Block_copy() **Not** Do?

Read your mind...

```
dispatch_async(queue, ^{  
    // implicitly: self->_ivar  
    // therefore: self is automatically retained  
    [_ivar doSomething];  
});
```

What Does Block_copy() **Not** Do?

Read your mind...

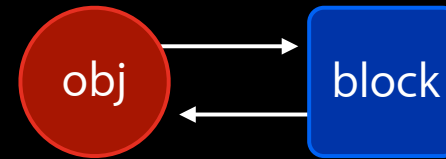
```
dispatch_async(queue, ^{  
    // implicitly: self->_ivar  
    // therefore: self is automatically retained  
    [_ivar doSomething];  
});
```

```
NSString *tmp = _ivar;    // workaround  
dispatch_async(queue, ^{  
    [tmp doSomething];  
});
```

What Does Block_copy() **Not** Do?

Fix retain cycles...

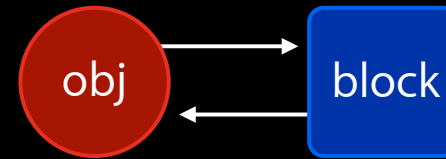
```
[obj setHandler: ^{  
    [obj doSomething];  
}];
```



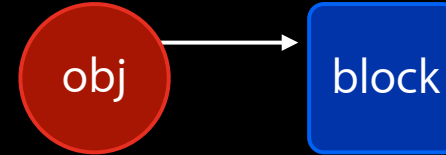
What Does Block_copy() **Not** Do?

Fix retain cycles...

```
[obj setHandler: ^{  
    [obj doSomething];  
}];
```



```
// workaround via __block which  
// does not implicitly retain  
__block NSThingy *tmp = obj;  
[obj setHandler: ^{  
    [tmp doSomething];  
}];
```



What Does Block_copy() **Not** Do?

Retain non-objects...

```
dispatch_async(queue, ^{  
    xyz();  
    dispatch_async(other_queue, ^{  
        abc();  
    });  
});
```

What Does Block_copy() **Not** Do?

Retain non-objects...

```
dispatch_retain(other_queue);
dispatch_async(queue, ^{
    xyz();
    dispatch_async(other_queue, ^{
        abc();
    });
    dispatch_release(other_queue);
});
```


What Does Block_copy() **Not** Do?

Retain non-objects...

```
CFRetain(foo);  
dispatch_async(queue, ^{  
    CFFooDoSomethingAwesome(foo);  
    CFRelease(foo);  
});
```

What Does Block_copy() **Not** Do?

Not implicitly called by non-blocks...

```
block_t array[10];
```

```
for (i = 0; i < 10; i++) {
```

```
    // the block is only valid inside the loop!
```

```
    array[i] = ^{ ... };
```

```
}
```

What Does Block_copy() **Not** Do?

Not implicitly called by non-blocks...

```
block_t array[10];
```

```
for (i = 0; i < 10; i++) {
```

```
    // the block is only valid inside the loop!
```

```
    array[i] = ^{ ... };
```

```
}
```

```
return ^{ ... };
```

```
// code must Block_copy() to outlive scope!
```

Better Blocks

Automatic reference counting

- Many of these challenges are solved by ARC
- Some are not
 - Retain and release of non-objects
- See the ARC talks for more information

Conclusion

- Blocks and Grand Central Dispatch
 - Simpler
 - Safer
- Already patterns you use today
 - Enumeration
 - Callbacks
 - Synchronization
 - Asynchronous code

More Information

Michael Jurewitz

Developer Tools and Performance Evangelist

jurewitz@apple.com

Paul Danbold

Core OS Evangelist

danbold@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

Developer Tools Kickoff	Pacific Heights Monday 3:15PM
Introducing Automatic Reference Counting	Presidio Tuesday 4:30PM
Mastering Grand Central Dispatch	Pacific Heights Thursday 10:15AM
Objective-C Advancements In-Depth	Mission Friday 11:30AM

Labs

Grand Central Dispatch Lab	Core OS Lab A Thursday 2:00PM
Xcode 4 Lab	Developer Tools Lab A Wednesday 11:30AM
Xcode 4 Lab	Developer Tools Lab A Thursday 11:30AM
LLVM Lab	Developer Tools Lab A Wednesday 2:00PM
Objective-C and Automatic Reference Counting Lab	Developer Tools Lab B Thursday 2:00PM

Q&A

