iOS Threading and Concurrency

- iOS offers a host of options that support concurrency.

Table 1: iOS Concurrency Technologies

<table>
<thead>
<tr>
<th>Technology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSThread class</td>
<td>Cocoa implements threads using the NSThread class. Cocoa also provides methods on NSObject for spawning new threads and executing code on already-running threads.</td>
</tr>
<tr>
<td>Grand Central Dispatch (GCD)</td>
<td>An alternative to threads that lets you focus on the tasks you need to perform rather than on thread management. With GCD, you define the task you want to perform and add it to a work queue, which handles the scheduling of your task on an appropriate thread. Work queues take into account the number of available cores and the current load to execute your tasks more efficiently than you could do yourself using threads.</td>
</tr>
<tr>
<td>Operation objects</td>
<td>Another alternative to threads, an operation object is a wrapper for a task that would normally be executed on a secondary thread. This wrapper hides the thread management aspects of performing the task, leaving you free to focus on the task itself. You typically use these objects in conjunction with an operation queue object, which actually manages the execution of the operation objects on one or more threads.</td>
</tr>
<tr>
<td>Idle-time notifications</td>
<td>For tasks that are relatively short and very low priority, idle time notifications let you perform the task at a time when your application is not as busy. Cocoa provides support for idle-time notifications using the NSNotificationQueue object. To request an idle-time notification, post a notification to the default NSNotificationQueue object using the NSPostWhenIdle option. The queue delays the delivery of your notification object until the run loop becomes idle.</td>
</tr>
<tr>
<td>Asynchronous functions</td>
<td>The system interfaces include many asynchronous functions that provide automatic concurrency for you. These APIs may use system daemons and processes or create custom threads to perform their task and return the results to you. (The actual implementation is irrelevant because it is separated from your code.) As you design your application, look for functions that offer asynchronous behavior and consider using them instead of using the equivalent synchronous function on a custom thread.</td>
</tr>
<tr>
<td>Timers</td>
<td>You can use timers on your application’s main thread to perform periodic tasks that are too trivial to require a thread, but which still require servicing at regular intervals.</td>
</tr>
</tbody>
</table>
Synchronization Tools

- To prevent different threads from changing data unexpectedly, you can either design your application to not have synchronization issues or you can use synchronization tools.
- Although avoiding synchronization issues altogether is preferable, it is not always possible.
- The following sections describe some of the synchronization tools available in iOS.

Atomic Operations

- Atomic operations are a simple form of synchronization that work on simple data types.
- The advantage of atomic operations is that they do not block competing threads. For simple operations, such as incrementing a counter variable, this can lead to much better performance than taking a lock.
- iOS includes numerous operations to perform basic mathematical and logical operations. Among these operations are atomic versions of the compare-and-swap, test-and-set, and test-and-clear operations.

Volatile Variables

- The compiler often optimizes code by loading the values for variables into registers. For local variables, this is usually not a problem.
- If the variable is visible from another thread however, such an optimization might prevent the other thread from noticing any changes to it.
- Applying the `volatile` keyword to a variable forces the compiler to load that variable from memory each time it is used. You might declare a variable as `volatile` if its value could be changed at any time by an external source that the compiler may not be able to detect.
- Because `volatile` variables decrease the number of optimizations the compiler can perform, they should be used sparingly and only where needed to ensure correctness.
**NSLock**

- **NSLock** can be used to protect a **critical section** of your code, which is a segment of code that only one thread at a time is allowed access.
- **NSLock** implements a basic **mutually exclusive lock** (or **mutex**) for Cocoa applications.
- A mutex is a type of semaphore that grants access to only one thread at a time. If a mutex is in use and another thread tries to acquire it, that thread blocks until the mutex is released by its original holder. If multiple threads compete for the same mutex, only one at a time is allowed access to it.
- The interface for all locks (including **NSLock**) is actually defined by the **NSLocking** protocol, which defines the **lock** and **unlock** methods. You use these methods to acquire and release the lock.
- In addition to the standard locking behavior, the **NSLock** class adds the **tryLock** and **lockBeforeDate:** methods. The **tryLock** method attempts to acquire the lock but does not block if the lock is unavailable; instead, the method simply returns **NO**.
- The **lockBeforeDate:** method attempts to acquire the lock but unblocks the thread (and returns **NO**) if the lock is not acquired within the specified time limit.
- The following example shows how you could use an **NSLock** object to coordinate the updating of a visual display, whose data is being calculated by several threads. If the thread cannot acquire the lock immediately, it simply continues its calculations until it can acquire the lock and update the display.
**NSLock**

```c
BOOL moreToDo = YES;
NSLock *theLock = [[NSLock alloc] init];

...

while (moreToDo)
{
    /* Do another increment of calculation */
    /* until there's no more to do. */

    if ([theLock tryLock])
    {
        /* Update display used by all threads. */

        [theLock unlock];
    }
}
```

**NSCondition**

- **Conditions** are a special type of lock that you can use to synchronize the order in which operations must proceed. They differ from mutex locks in a subtle way. A thread waiting on a condition remains blocked until that condition is signaled explicitly by another thread.
- The [NSCondition](https://developer.apple.com//documentation/coreFoundation/NSCondition) class lets you create an object that you can lock like a mutex and then wait on like a condition.
The @synchronized Directive

- The @synchronized directive is a convenient way to create mutex locks on the fly in Objective-C code.
- The @synchronized directive does what any other mutex lock would do – it prevents different threads from acquiring the same lock at the same time.
- In this case, however, you do not have to create the mutex or lock object directly. Instead, you simply use any Objective-C object as a lock token, as shown in the following example:

```objective-c
- (void)myMethod:(id)anObj
{
    @synchronized(anObj)
    {
        // Everything between the braces is protected by the @synchronized
        // directive.
    }
}
```

- The object passed to the @synchronized directive is a unique identifier used to distinguish the protected block.
  - If you execute the preceding method in two different threads, passing a different object for the anObj parameter on each thread, each would take its lock and continue processing without being blocked by the other.
  - If you pass the same object in both cases, however, one of the threads would acquire the lock first and the other would block until the first thread completed the critical section.
- As a precautionary measure, the @synchronized block implicitly adds an exception handler to the protected code. This handler automatically releases the mutex in the event that an exception is thrown.
- This means that in order to use the @synchronized directive, you must also enable Objective-C exception handling in your code. If you do not want the additional overhead caused by the implicit exception handler, you should consider using the lock classes.
- By default, the setter and getter code generated for declared properties will include the @synchronized directive to protect the instance variable. Coding the attribute nonatomic on the property will eliminate the synchronization code.
Threading

NSThread

- You can subclass NSThread and override the main method to implement your thread’s main entry point. If you override main, you do not need to invoke the inherited behavior by calling super. You can later call the instance method start to execute your thread object’s code.
- There is also a class method called detachNewThreadSelector:toTarget:withObject: that can create a new NSThread instance to execute a specified method:

```objective-c
+ (void)detachNewThreadSelector:(SEL)aSelector toTarget: (id)aTarget withObject: (id)anArgument
```

Parameters

aSelector
The selector for the message to send to the target. This selector must take only one argument and must not have a return value.

aTarget
The object that will receive the message aSelector on the new thread.

anArgument
The single argument passed to the target. May be nil.

Discussion
For iOS applications, the method aSelector is responsible for setting up an autorelease pool for the newly detached thread and freeing that pool before it exits.
**NSObject Threading Methods**

- You can use the NSObject instance method `performSelectorOnMainThread:withObject:waitUntilDone:` to let a background thread do something on the main thread (update the user interface, for example).

```objc
-(void)performSelectorOnMainThread:(SEL)aSelector withObject:(id)arg
   waitUntilDone:(BOOL)wait
```

**Parameters**

*aSelector*
A selector that identifies the method to invoke. The method should not have a significant return value and should take a single argument of type `id`, or no arguments.

*arg*
The argument to pass to the method when it is invoked. Pass `nil` if the method does not take an argument.

*wait*
A Boolean that specifies whether the current thread blocks until after the specified selector is performed on the receiver on the main thread. Specify `YES` to block this thread; otherwise, specify `NO` to have this method return immediately.

If the current thread is also the main thread, and you specify `YES` for this parameter, the message is delivered and processed immediately.

**Discussion**
You can use this method to deliver messages to the main thread of your application. The message in this case is a method of the current object that you want to execute on the thread.

As part of its normal run loop processing, the main thread dequeues the message and invokes the desired method. Multiple calls to this method from the same thread cause the corresponding selectors to be queued and performed in the same order in which the calls were made.
Example

- (IBAction)runTasks:(id)sender {

    taskCount = 5;    // instance variable

    for (int i = 0; i < taskCount; ++i) {
        [NSThread detachNewThreadSelector:@selector(runTaskAtIndex:)
            toTarget:self
            withObject:[NSNumber numberWithInt:i]];
    }
}

- (void)runTaskAtIndex:(NSNumber *)taskIndex {

    @autoreleasepool {

        // Run task, presumably making use of taskIndex
        ...

        // Update task count
        @synchronized(self) {

            --taskCount;
            if (taskCount == 0) {
                [self performSelectorOnMainThread:@selector(tasksDidFinish)
                    withObject:nil
                    waitUntilDone:NO];
            }
        }
    }
}
Concurrency

- Threads do not solve the general problem of executing multiple tasks in a scalable way.
  - Threads place the burden of creating a scalable solution on the developer. You have to decide how many threads to create and adjust that number dynamically as system conditions change.
  - Another problem is that your application assumes most of the costs associated with creating and maintaining any threads it uses.
- Instead of relying on threads, iOS takes an asynchronous design approach to solving the concurrency problem.
  - Asynchronous functions have been present in operating systems for many years and are often used to initiate tasks that might take a long time, such as reading data from the disk.
  - When called, an asynchronous function does some work behind the scenes to start a task running but returns before that task might actually be complete.
  - Typically, this work involves acquiring a background thread, starting the desired task on that thread, and then sending a notification to the caller (usually through a callback function) when the task is done.
  - In the past, if an asynchronous function did not exist for what you want to do, you would have to write your own asynchronous function and create your own threads.
  - Now iOS provides technologies to allow you to perform any task asynchronously without having to manage the threads yourself.
Grand Central Dispatch (GCD)

- One of the technologies for starting tasks asynchronously is Grand Central Dispatch (GCD).
  - This technology takes the thread management code you would normally write in your own applications and moves that code down to the system level.
  - All you have to do is define the tasks you want to execute and add them to an appropriate dispatch queue. GCD takes care of creating the needed threads and of scheduling your tasks to run on those threads.
- Because the thread management is now part of the system, GCD provides a holistic approach to task management and execution, providing better efficiency than traditional threads.

Dispatch Queues

- GCD dispatch queues let you execute arbitrary blocks of code either asynchronously or synchronously with respect to the caller.
- You can use dispatch queues to perform nearly all of the tasks that you used to perform on separate threads.
- Dispatch queues provide several advantages over threads:
  - With threads, you have to write code both for the work you want to perform and for the creation and management of the threads themselves. Dispatch queues let you focus on the work you actually want to perform without having to worry about the thread creation and management. Instead, the system handles all of the thread creation and management for you.
  - The system is able to manage threads much more efficiently than any single application ever could. The system can scale the number of threads dynamically based on the available resources and current system conditions.
  - The system is usually able to start running your task more quickly than you could if you created the thread yourself.
## Dispatch Queues

**Table 2: Dispatch Queues**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Serial</strong></td>
<td>Serial queues (also known as <em>private dispatch queues</em>) execute one task at a time in the order in which they are added to the queue. The currently executing task runs on a distinct thread (which can vary from task to task) that is managed by the dispatch queue. Serial queues are often used to synchronize access to a specific resource. You can create as many serial queues as you need, and each queue operates concurrently with respect to all other queues. In other words, if you create four serial queues, each queue executes only one task at a time but up to four tasks could still execute concurrently, one from each queue.</td>
</tr>
<tr>
<td><strong>Concurrent</strong></td>
<td>Concurrent queues (also known as a type of <em>global dispatch queue</em>) execute one or more tasks concurrently, but tasks are still started in the order in which they were added to the queue. The currently executing tasks run on distinct threads that are managed by the dispatch queue. The exact number of tasks executing at any given point is variable and depends on system conditions. In iOS 5 and later, you can create concurrent dispatch queues yourself by specifying <code>DISPATCH_QUEUE_CONCURRENT</code> as the queue type. In addition, there are four predefined global concurrent queues for your application to use, each with a different priority level.</td>
</tr>
<tr>
<td><strong>Main dispatch queue</strong></td>
<td>The main dispatch queue is a globally available serial queue that executes tasks on the application’s main thread. This queue works with the application’s run loop to interleave the execution of queued tasks with the execution of other event sources attached to the run loop. Because it runs on your application’s main thread, the main queue is often used as a key synchronization point for an application.</td>
</tr>
</tbody>
</table>
Grand Central Dispatch API

dispatch_queue_t dispatch_get_global_queue(long priority, unsigned long flags);

- Returns the global concurrent queue with the specified priority.
- **Valid values for priority are** DISPATCH_QUEUE_PRIORITY_HIGH, DISPATCH_QUEUE_PRIORITY_DEFAULT, DISPATCH_QUEUE_PRIORITY_LOW, or DISPATCH_QUEUE_PRIORITY_BACKGROUND.
- Currently, the only valid value for flags is 0.

dispatch_queue_t dispatch_get_main_queue(void);

- Returns the serial dispatch queue associated with the application’s main thread.

dispatch_queue_t dispatch_queue_create(const char *label, dispatch_queue_attr_t attr);

- Creates a new dispatch queue to which blocks can be submitted.
- label is a C string that can be used to identify the queue when debugging.
- attr should be either DISPATCH_QUEUE_SERIAL (or NULL) to create a serial queue or DISPATCH_QUEUE_CONCURRENT to create a concurrent queue.
- Blocks submitted to a serial queue are executed one at a time in FIFO order. Note, however, that blocks submitted to independent queues may be executed concurrently with respect to each other. Blocks submitted to a concurrent queue are dequeued in FIFO order but may run concurrently if resources are available to do so.
- When your application no longer needs the dispatch queue, it should release it with the dispatch_release function. Any pending blocks submitted to a queue hold a reference to that queue, so the queue is not deallocated until all pending blocks have completed.
**Grand Central Dispatch API**

```c
void dispatch_release(dispatch_object_t object);
```

- Decrements the reference (retain) count of a dispatch queue or dispatch group.
- This function has no effect on the main or global concurrent queues.

```c
dispatch_queue_t dispatch_get_current_queue(void);
```

- Returns the queue on which the currently executing block is running.

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

- Submits a block for asynchronous execution on a dispatch queue and returns immediately.
- `block` must have no parameters and return `void`.

```c
dispatch_group_t dispatch_group_create(void);
```

- Creates a new dispatch group with which block objects can be associated.
- The dispatch group maintains a count of its outstanding associated tasks, incrementing the count when a new task is associated and decrementing it when a task completes. Functions such as `dispatch_group_notify` and `dispatch_group_wait` use that count to allow your application to determine when all tasks associated with the group have completed. At that time, your application can take any appropriate action.
- When your application no longer needs the dispatch group, it should call `dispatch_release` to release its reference to the group object and ultimately free its memory.

```c
void dispatch_group_async(dispatch_group_t group, dispatch_queue_t queue, dispatch_block_t block);
```

- Submits a block to a dispatch queue and associates the block object with the given dispatch group. The dispatch group can be used to wait for the completion of the block objects it references.
**Grand Central Dispatch API**

- **dispatch_group_wait** – Waits synchronously for the previously submitted block objects to complete; returns if the blocks do not complete before the specified timeout period has elapsed.
- **dispatch_group_notify** – Schedules a block object to be submitted to a queue when a group of previously submitted block objects have completed.
- **dispatch_semaphore_create** – Creates new counting semaphore with an initial value.
- **dispatch_semaphore_signal** – Signals (increments) a semaphore.
- **dispatch_semaphore_wait** – Waits for (decrements) a semaphore.
- **dispatch_suspend** – Suspends the invocation of block objects on a dispatch object.
- **dispatch_resume** – Resume the invocation of block objects on a dispatch object.
Example:

// When a button is pressed, find the sum of a large array of integers
// Work will be split across numTasks concurrent tasks
-(IBAction)addNumbers:(id)sender {

    // get references to global concurrent queue and main queue
    dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_queue_t mainQueue = dispatch_get_main_queue();

    taskCount = numTasks; // instance variable
    finalSum = 0; // instance variable

    for (int taskNum = 0; taskNum < numTasks; ++taskNum) {

        // this block will be executed concurrently
        dispatch_async(queue, ^{

            // loop-splitting technique divides array elements into numTasks sub-sets
            for (int i = taskNum, partialSum = 0; i < arraySize; i += numTasks)
                partialSum += array[i];

            // this block will be executed synchronously on the main thread
            dispatch_async(mainQueue, ^{
                --taskCount; // decrement task count
                finalSum += partialSum; // add partial total to final total

                // if all tasks are complete, update UI with final sum
                if (taskCount == 0) {
                    self.totalLabel.text = [NSString stringWithFormat:@"%d", finalSum];
                }
            });
        });
    }
}
References


APPLE INC., 2012. Concurrency programming guide