Migrating Away from Threads

There are many ways to adapt existing threaded code to take advantage of Grand Central Dispatch and operation objects. Although moving away from threads may not be possible in all cases, performance (and the simplicity of your code) can improve dramatically in places where you do make the switch. Specifically, using dispatch queues and operation queues instead of threads has several advantages:

- It reduces the memory penalty your application pays for storing thread stacks in the application's memory space.
- It eliminates the code needed to create and configure your threads.
- It eliminates the code needed to manage and schedule work on threads.
- It simplifies the code you have to write.

This chapter provides some tips and guidelines on how to replace your existing thread-based code and instead use dispatch queues and operation queues to achieve the same types of behaviors.

Replacing Threads with Dispatch Queues

To understand how you might replace threads with dispatch queues, first consider some of the ways you might be using threads in your application today:

- **Single task threads**. Create a thread to perform a single task and release the thread when the task is done.
- **Worker threads**. Create one or more worker threads with specific tasks in mind for each. Dispatch tasks to each thread periodically.
- **Thread pools**. Create a pool of generic threads and set up run loops for each one. When you have a task to perform, grab a thread from the pool and dispatch the task to it. If there are no free threads, queue the task and wait for a thread to become available.

Although these might seem like dramatically different techniques, they are really just variants on the same principle. In each case, a thread is being used to run some task that the application has to perform. The only difference between them is the code used to manage the threads and the queueing of tasks. With dispatch queues and operation queues, you can eliminate all of your thread and thread-communication code and instead focus on just the tasks you want to perform.

If you are using one of the above threading models, you should already have a pretty good idea of the type of tasks your application performs. Instead of submitting a task to one of your custom threads, try encapsulating that task in an operation object or a block object and dispatching it to the appropriate queue. For tasks that are not particularly contentious—that is, tasks that do not take locks—you should be able to make the following direct replacements:

- For a single task thread, encapsulate the task in a block or operation object and submit it to a concurrent queue.
- For worker threads, you need to decide whether to use a serial queue or a concurrent queue. If you use worker threads to synchronize the execution of specific sets of tasks, use a serial queue. If you do use worker threads to execute arbitrary tasks with no interdependencies, use a concurrent queue.
- For thread pools, encapsulate your tasks in a block or operation object and dispatch them to a concurrent queue for execution.

Of course, simple replacements like this may not work in all cases. If the tasks you are executing contend for shared resources, the ideal solution is to try to remove or minimize that contention first. If there are ways that you can refactor or rearchitect your code to eliminate mutual dependencies on shared resources, that is certainly preferable. However, if doing so is not possible or might be less

efficient, there are still ways to take advantage of queues. A big advantage of queues is that they offer a more predictable way to execute your code. This predictability means that there are still ways to synchronize the execution of your code without using locks or other heavyweight synchronization mechanisms. Instead of using locks, you can use queues to perform many of the same tasks:

- If you have tasks that must execute in a specific order, submit them to a serial dispatch queue. If you prefer to use operation queues, use operation object dependencies to ensure that those objects execute in a specific order.
- If you are currently using locks to protect a shared resource, create a serial queue to execute any tasks that modify that resource. The serial queue then replaces your existing locks as the synchronization mechanism. For more information techniques for getting rid of locks, see Eliminating Lock-Based Code.
- If you use thread joins to wait for background tasks to complete, consider using dispatch groups instead. You can also use an `NSBlockOperation` object or operation object dependencies to achieve similar group-completion behaviors. For more information on how to track groups of executing tasks, see Replacing Thread Joins.
- If you use a producer-consumer algorithm to manage a pool of finite resources, consider changing your implementation to the one shown in Changing Producer-Consumer Implementations.
- If you are using threads to read and write from descriptors, or monitor file operations, use the dispatch sources as described in Dispatch Sources.

It is important to remember that queues are not a panacea for replacing threads. The asynchronous programming model offered by queues is appropriate in situations where latency is not an issue. Even though queues offer ways to configure the execution priority of tasks in the queue, higher execution priorities do not guarantee the execution of tasks at specific times. Therefore, threads are still a more appropriate choice in cases where you need minimal latency, such as in audio and video playback.

Eliminating Lock-Based Code

For threaded code, locks are one of the traditional ways to synchronize access to resources that are shared between threads. However, the use of locks comes at a cost. Even in the non-contested case, there is always a performance penalty associated with taking a lock. And in the contested case, there is the potential for one or more threads to block for an indeterminate amount of time while waiting for the lock to be released.

Replacing your lock-based code with queues eliminates many of the penalties associated with locks and also simplifies your remaining code. Instead of using a lock to protect a shared resource, you can instead create a queue to serialize the tasks that access that resource. Queues do not impose the same penalties as locks. For example, queueing a task does not require trapping into the kernel to acquire a mutex.

When queueing tasks, the main decision you have to make is whether to do so synchronously or asynchronously. Submitting a task asynchronously lets the current thread continue to run while the task is performed. Submitting a task synchronously blocks the current thread until the task is completed. Both options have appropriate uses, although it is certainly advantageous to submit tasks asynchronously whenever you can.

The following sections show you how to replace your existing lock-based code with the equivalent queue-based code.

Implementing an Asynchronous Lock

An asynchronous lock is a way for you to protect a shared resource without blocking any code that modifies that resource. You might use an asynchronous lock when you need to modify a data structure as a side effect of some other work your code is doing. Using traditional threads, the way you would normally implement this code would be to take a lock for the shared resource, make the necessary changes, release the lock, and continue with the main part of your task. However, using dispatch queues, the calling code can make the modifications asynchronously without waiting for those changes to be completed.

Listing 5-1 shows an example of an asynchronous lock implementation. In this example, the protected resource defines its own serial dispatch queue. The calling code submits a block object to this queue that contains the modifications that need to be made to the resource. Because the queue itself executes blocks serially, changes to the resource are guaranteed to be made in the order in which they were received; however, because the task was executed asynchronously, the calling thread does not block.

Listing 5-1 Modifying protected resources asynchronously

```
dispatch_async(obj->serial_queue, ^{
    // Critical section
});
```

Executing Critical Sections Synchronously

If the current code cannot continue until a given task is complete, you can submit the task synchronously using the `dispatch_sync` function. This function adds the task to a dispatch queue and then blocks the current thread until the task finishes executing. The dispatch queue itself can be a serial or concurrent queue depending on your needs. Because this function blocks the current thread, though, you should use it only when necessary. Listing 5-2 shows the technique for wrapping a critical section of your code using `dispatch_sync`.

Listing 5-2 Running critical sections synchronously

```
dispatch_sync(my_queue, ^{
    // Critical section
});
```

If you are already using a serial queue to protect a shared resource, dispatching to that queue synchronously does not protect the shared resource any more than if you dispatched asynchronously. The only reason to dispatch synchronously is to prevent the current code from continuing until the critical section finishes. For example, if you wanted to get some value from the shared resource and use it right away, you would need to dispatch synchronously. If the current code does not need to wait for the critical section to complete, or if it can simply submit additional follow-up tasks to the same serial queue, submitting asynchronously is generally preferred.

Improving on Loop Code

If your code has loops, and the work being done each time through the loop is independent of the work being done in the other iterations, you might consider reimplementing that loop code using the `dispatch_apply` or `dispatch_apply_f` function. These functions submit each iteration of a loop separately to a dispatch queue for processing. When used in conjunction with a concurrent queue, this feature lets you perform multiple iterations of the loop concurrently.

The `dispatch_apply` and `dispatch_apply_f` functions are synchronous function calls that block the current thread of execution until all of the loop iterations are complete. When submitted to a concurrent queue, the execution order of the loop iterations is not guaranteed. The threads running each iteration could block and cause a given iteration to finish before or after the others around it. Therefore, the block object or function you use for each loop iteration must be reentrant.

Listing 5-3 shows how to replace a `for` loop with its dispatch-based equivalent. The block or function you pass to `dispatch_apply` or `dispatch_apply_f` must take an integer value indicating the current loop iteration. In this example, the code simply prints the current loop number to the console.

Listing 5-3 Replacing a `for` loop without striding

```
queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_apply(count, queue, ^(size_t i) {
   printf("%u\n", i);
});
```

Although the preceding example is a simplistic one, it demonstrates the basic techniques for replacing a loop using dispatch queues. And although this can be a good way to improve performance in loop-based code, you must still use this technique discerningly. Although dispatch queues have very low overhead, there are still costs to scheduling each loop iteration on a thread. Therefore, you should make sure your loop code does enough work to warrant the costs. Exactly how much work you need to do is something you have to measure using the performance tools.

A simple way to increase the amount of work in each loop iteration is to use striding. With striding, you rewrite your block code to perform more than one iteration of the original loop. You then reduce the count value you specify to the `dispatch_apply` function by a proportional amount. Listing 5-4 shows how you might implement striding for the loop code shown in Listing 5-3. In Listing 5-4, the block calls the `printf` statement the same number of times as the stride value, which in this case is 137. (The actual stride value is something you should configure based on the work being done by your code.) Because there is a remainder left over when dividing the total number of iterations by a stride value, any remaining iterations are performed inline.

Listing 5-4 Adding a stride to a dispatched for loop

```
int stride = 137;
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
0);

dispatch_apply(count / stride, queue, ^(size_t idx){
    size_t j = idx * stride;
    size_t j_stop = j + stride;
    do {
       printf("%u\n", (unsigned int)j++);
    }while (j < j_stop);
});

size_t i;
for (i = count - (count % stride); i < count; i++)
   printf("%u\n", (unsigned int)i);
```

There are some definite performance advantages to using strides. In particular, strides offer benefits when the original number of loop iterations is high, relative to the stride. Dispatching fewer blocks concurrently means that more time is spent executing the code of those blocks than dispatching them. As with any performance metric though, you may have to play with the striding value to find the most efficient value for your code.

Replacing Thread Joins

Thread joins allow you to spawn one or more threads and then have the current thread wait until those threads are finished. To implement a thread join, a parent creates a child thread as a *joinable thread*. When the parent can no longer make progress without the results from a child thread, it joins with the child. This process blocks the parent thread until the child finishes its task and exits, at which point the parent can gather the results from the child and continue with its own work. If the parent needs to join with multiple child threads, it does so one at a time.

Dispatch groups offer semantics that are similar to those of thread joins but that have some additional advantages. Like thread joins, dispatch groups are a way for a thread to block until one or more child tasks finishes executing. Unlike thread joins, a dispatch group waits on all of its child tasks simultaneously. And because dispatch groups use dispatch queues to perform the work, they are very efficient.

To use a dispatch group to perform the same work performed by joinable threads, you would do the following:

1. Create a new dispatch group using the `dispatch_group_create` function.
2. Add tasks to the group using the `dispatch_group_async` or `dispatch_group_async_f` function. Each task you submit to the group represents work you would normally perform on a joinable thread.
3. When the current thread cannot make any more forward progress, call the `dispatch_group_wait` function to wait on the group. This function blocks the current thread until all of the tasks in the group finish executing.

If you are using operation objects to implement your tasks, you can also implement thread joins using dependencies. Instead of having a parent thread wait for one or more tasks to complete, you would move the parent code to an operation object. You would then set up dependencies between the parent operation object and any number of child operation objects set up to do the work normally performed by the joinable threads. Having dependencies on other operation objects prevents the parent operation object from executing until all of the operations have finished.

For an example of how to use dispatch groups, see Waiting on Groups of Queued Tasks. For information about setting up dependencies between operation objects, see Configuring Interoperation Dependencies.

Changing Producer-Consumer Implementations

A producer-consumer model lets you manage a finite number of dynamically produced resources. While the producer creates new resources (or work), one or more consumers wait for those resources (or work) to be ready and consume them when they are. The typical mechanisms for implementing a producer-consumer model are conditions or semaphores.

Using conditions, the producer thread typically does the following:

1. Lock the mutex associated with the condition (using `pthread_mutex_lock`).
2. Produce the resource or work to be consumed.
3. Signal the condition variable that there is something to consume (using `pthread_cond_signal`)
4. Unlock the mutex (using `pthread_mutex_unlock`).

In turn, the corresponding consumer thread does the following:

1. Lock the mutex associated with the condition (using `pthread_mutex_lock`).
2. Set up a `while` loop to do the following:
    a. Check to see whether there is really work to be done.
    b. If there is no work to do (or no resource available), call `pthread_cond_wait` to block the current thread until a corresponding signal occurs.
3. Get the work (or resource) provided by the producer.
4. Unlock the mutex (using `pthread_mutex_unlock`).
5. Process the work.

With dispatch queues, you can simplify the producer and consumer implementations into a single call:

```
dispatch_async(queue, ^{
```

```
    // Process a work item.
});
```

When your producer has work to be done, all it has to do is add that work to a queue and let the queue process the item. The only part of the preceding code that changes is the queue type. If the tasks generated by the producer need to be performed in a specific order, you use a serial queue. If the tasks generated by the producer can be performed concurrently, you add them to a concurrent queue and let the system execute as many of them as possible simultaneously.

Replacing Semaphore Code

If you are currently using semaphores to restrict access to a shared resource, you should consider using dispatch semaphores instead. Traditional semaphores always require calling down to the kernel to test the semaphore. In contrast, dispatch semaphores test the semaphore state quickly in user space and trap into the kernel only when the test fails and the calling thread needs to be blocked. This behavior results in dispatch semaphores being much faster than traditional semaphores in the uncontested case. In all other aspects, though, dispatch semaphores offer the same behavior as traditional semaphores.

For an example of how to use dispatch semaphores, see Using Dispatch Semaphores to Regulate the Use of Finite Resources.

Replacing Run-Loop Code

If you are using run loops to manage the work being performed on one or more threads, you may find that queues are much simpler to implement and maintain going forward. Setting up a custom run loop involves setting up both the underlying thread and the run loop itself. The run-loop code consists of setting up one or more run loop sources and writing callbacks to handle events arriving on those sources. Instead of all that work, you can simply create a serial queue and dispatch tasks to it. Thus, you can replace all of your thread and run-loop creation code with one line of code:

```
dispatch_queue_t myNewRunLoop = dispatch_queue_create("com.apple.MyQueue", NULL);
```

Because the queue automatically executes any tasks added to it, there is no extra code required to manage the queue. You do not have to create or configure a thread, and you do not have to create or attach any run-loop sources. In addition, you can perform new types of work on the queue by simply adding the tasks to it. To do the same thing with a run loop, you would need to modify your existing run loop source or create a new one to handle the new data.

One common configuration for run loops is to process data arriving asynchronously on a network socket. Instead of configuring a run loop for this type of behavior, you can attach a dispatch source to the desired queue. Dispatch sources also offer more options for processing data than traditional run loop sources. In addition to processing timer and network port events, you can use dispatch sources to read and write to files, monitor file system objects, monitor processes, and monitor signals. You can even define custom dispatch sources and trigger them from other parts of your code asynchronously. For more information on setting up dispatch sources, see Dispatch Sources.

Compatibility with POSIX Threads

Because Grand Central Dispatch manages the relationship between the tasks you provide and the threads on which those tasks run, you should generally avoid calling POSIX thread routines from your task code. If you do need to call them for some reason, you should be very careful about which routines you call. This section provides you with an indication of which routines are safe to call and

which are not safe to call from your queued tasks. This list is not complete but should give you an indication of what is safe to call and what is not.

In general, your application must not delete or mutate objects or data structures that it did not create. Consequently, block objects that are executed using a dispatch queue must not call the following functions:

```
pthread_detach
pthread_cancel
pthread_join
pthread_kill
pthread_exit
```

Although it is alright to modify the state of a thread while your task is running, you must return the thread to its original state before your task returns. Therefore, it is safe to call the following functions as long as you return the thread to its original state:

```
pthread_setcancelstate
pthread_setcanceltype
pthread_setschedparam
pthread_sigmask
pthread_setspecific
```

The underlying thread used to execute a given block can change from invocation to invocation. As a result, your application should not rely on the following functions returning predictable results between invocations of your block:

```
pthread_self
pthread_getschedparam
pthread_get_stacksize_np
pthread_get_stackaddr_np
pthread_mach_thread_np
pthread_from_mach_thread_np
pthread_getspecific
```

Important: Blocks must catch and suppress any language-level exceptions thrown within them. Other errors that occur during the execution of your block should similarly be handled by the block or used to notify other parts of your application.

For more information about POSIX threads and the functions mentioned in this section, see the `pthread` man pages.