Concurrency and Application Design

In the early days of computing, the maximum amount of work per unit of time that a computer could perform was determined by the clock speed of the CPU. But as technology advanced and processor designs became more compact, heat and other physical constraints started to limit the maximum clock speeds of processors. And so, chip manufacturers looked for other ways to increase the total performance of their chips. The solution they settled on was increasing the number of processor cores on each chip. By increasing the number of cores, a single chip could execute more instructions per second without increasing the CPU speed or changing the chip size or thermal characteristics. The only problem was how to take advantage of the extra cores.

In order to take advantage of multiple cores, a computer needs software that can do multiple things simultaneously. For a modern, multitasking operating system like OS X or iOS, there can be a hundred or more programs running at any given time, so scheduling each program on a different core should be possible. However, most of these programs are either system daemons or background applications that consume very little real processing time. Instead, what is really needed is a way for individual applications to make use of the extra cores more effectively.

The traditional way for an application to use multiple cores is to create multiple threads. However, as the number of cores increases, there are problems with threaded solutions. The biggest problem is that threaded code does not scale very well to arbitrary numbers of cores. You cannot create as many threads as there are cores and expect a program to run well. What you would need to know is the number of cores that can be used efficiently, which is a challenging thing for an application to compute on its own. Even if you manage to get the numbers correct, there is still the challenge of programming for so many threads, of making them run efficiently, and of keeping them from interfering with one another.

So, to summarize the problem, there needs to be a way for applications to take advantage of a variable number of computer cores. The amount of work performed by a single application also needs to be able to scale dynamically to accommodate changing system conditions. And the solution has to be simple enough so as to not increase the amount of work needed to take advantage of those cores. The good news is that Apple's operating systems provide the solution to all of these problems, and this chapter takes a look at the technologies that comprise this solution and the design tweaks you can make to your code to take advantage of them.

The Move Away from Threads

Although threads have been around for many years and continue to have their uses, they do not solve the general problem of executing multiple tasks in a scalable way. With threads, the burden of creating a scalable solution rests squarely on the shoulders of you, 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, OS X and iOS take 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. But now, OS X and iOS provide technologies to allow you to perform any task asynchronously without having to manage the threads yourself.

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.

Operation queues are Objective-C objects that act very much like dispatch queues. You define the tasks you want to execute and then add them to an operation queue, which handles the scheduling and execution of those tasks. Like GCD, operation queues handle all of the thread management for you, ensuring that tasks are executed as quickly and as efficiently as possible on the system.

The following sections provide more information about dispatch queues, operation queues, and some other related asynchronous technologies you can use in your applications.

Dispatch Queues

Dispatch queues are a C-based mechanism for executing custom tasks. A *dispatch queue* executes tasks either serially or concurrently but always in a first-in, first-out order. (In other words, a dispatch queue always dequeues and starts tasks in the same order in which they were added to the queue.) A serial dispatch queue runs only one task at a time, waiting until that task is complete before dequeuing and starting a new one. By contrast, a concurrent dispatch queue starts as many tasks as it can without waiting for already started tasks to finish.

Dispatch queues have other benefits:

- They provide a straightforward and simple programming interface.
- They offer automatic and holistic thread pool management.
- They provide the speed of tuned assembly.
- They are much more memory efficient (because thread stacks do not linger in application memory).
- They do not trap to the kernel under load.
- The asynchronous dispatching of tasks to a dispatch queue cannot deadlock the queue.
- They scale gracefully under contention.
- Serial dispatch queues offer a more efficient alternative to locks and other synchronization primitives.

The tasks you submit to a dispatch queue must be encapsulated inside either a function or a block object. *Block objects* are a C language feature introduced in OS X v10.6 and iOS 4.0 that are similar to function pointers conceptually, but have some additional benefits. Instead of defining blocks in their own lexical scope, you typically define blocks inside another function or method so that they can access other variables from that function or method. Blocks can also be moved out of their original scope and copied onto the heap, which is what happens when you submit them to a dispatch queue. All of these semantics make it possible to implement very dynamic tasks with relatively little code.

Dispatch queues are part of the Grand Central Dispatch technology and are part of the C runtime. For more information about using dispatch queues in your applications, see Dispatch Queues. For more information about blocks and their benefits, see *Blocks Programming Topics*.

Dispatch Sources

Dispatch sources are a C-based mechanism for processing specific types of system events asynchronously. A dispatch source encapsulates information about a particular type of system event and submits a specific block object or function to a dispatch queue whenever that event occurs. You can use dispatch sources to monitor the following types of system events:

- Timers
- Signal handlers
- Descriptor-related events
- Process-related events
- Mach port events
- Custom events that you trigger

Dispatch sources are part of the Grand Central Dispatch technology. For information about using dispatch sources to receive events in your application, see Dispatch Sources.

Operation Queues

An operation queue is the Cocoa equivalent of a concurrent dispatch queue and is implemented by the `NSOperationQueue` class. Whereas dispatch queues always execute tasks in first-in, first-out order, operation queues take other factors into account when determining the execution order of tasks. Primary among these factors is whether a given task depends on the completion of other tasks. You configure dependencies when defining your tasks and can use them to create complex execution-order graphs for your tasks.

The tasks you submit to an operation queue must be instances of the `NSOperation` class. An *operation object* is an Objective-C object that encapsulates the work you want to perform and any data needed to perform it. Because the `NSOperation` class is essentially an abstract base class, you typically define custom subclasses to perform your tasks. However, the Foundation framework does include some concrete subclasses that you can create and use as is to perform tasks.

Operation objects generate key-value observing (KVO) notifications, which can be a useful way of monitoring the progress of your task. Although operation queues always execute operations concurrently, you can use dependencies to ensure they are executed serially when needed.

For more information about how to use operation queues, and how to define custom operation objects, see Operation Queues.

Asynchronous Design Techniques

Before you even consider redesigning your code to support concurrency, you should ask yourself whether doing so is necessary. Concurrency can improve the responsiveness of your code by ensuring that your main thread is free to respond to user events. It can even improve the efficiency of your code by leveraging more cores to do more work in the same amount of time. However, it also adds overhead and increases the overall complexity of your code, making it harder to write and debug your code.

Because it adds complexity, concurrency is not a feature that you can graft onto an application at the end of your product cycle. Doing it right requires careful consideration of the tasks your application performs and the data structures used to perform those tasks. Done incorrectly, you might find your code runs slower than before and is less responsive to the user. Therefore, it is worthwhile to take some time at the beginning of your design cycle to set some goals and to think about the approach you need to take.

Every application has different requirements and a different set of tasks that it performs. It is impossible for a document to tell you exactly how to design your application and its associated tasks. However, the following sections try to provide some guidance to help you make good choices during the design process.

Define Your Application's Expected Behavior

Before you even think about adding concurrency to your application, you should always start by defining what you deem to be the correct behavior of your application. Understanding your application's expected behavior gives you a way to validate your design later. It should also give you some idea of the expected performance benefits you might receive by introducing concurrency.

The first thing you should do is enumerate the tasks your application performs and the objects or data structures associated with each task. Initially, you might want to start with tasks that are performed when the user selects a menu item or clicks a button. These tasks offer discrete behavior and have a well defined start and end point. You should also enumerate other types of tasks your application may perform without user interaction, such as timer-based tasks.

After you have your list of high-level tasks, start breaking each task down further into the set of steps that must be taken to complete the task successfully. At this level, you should be primarily concerned

with the modifications you need to make to any data structures and objects and how those modifications affect your application's overall state. You should also note any dependencies between objects and data structures as well. For example, if a task involves making the same change to an array of objects, it is worth noting whether the changes to one object affect any other objects. If the objects can be modified independently of each other, that might be a place where you could make those modifications concurrently.

Factor Out Executable Units of Work

From your understanding of your application's tasks, you should already be able to identify places where your code might benefit from concurrency. If changing the order of one or more steps in a task changes the results, you probably need to continue performing those steps serially. If changing the order has no effect on the output, though, you should consider performing those steps concurrently. In both cases, you define the executable unit of work that represents the step or steps to be performed. This unit of work then becomes what you encapsulate using either a block or an operation object and dispatch to the appropriate queue.

For each executable unit of work you identify, do not worry too much about the amount of work being performed, at least initially. Although there is always a cost to spinning up a thread, one of the advantages of dispatch queues and operation queues is that in many cases those costs are much smaller than they are for traditional threads. Thus, it is possible for you to execute smaller units of work more efficiently using queues than you could using threads. Of course, you should always measure your actual performance and adjust the size of your tasks as needed, but initially, no task should be considered too small.

Identify the Queues You Need

Now that your tasks are broken up into distinct units of work and encapsulated using block objects or operation objects, you need to define the queues you are going to use to execute that code. For a given task, examine the blocks or operation objects you created and the order in which they must be executed to perform the task correctly.

If you implemented your tasks using blocks, you can add your blocks to either a serial or concurrent dispatch queue. If a specific order is required, you would always add your blocks to a serial dispatch queue. If a specific order is not required, you can add the blocks to a concurrent dispatch queue or add them to several different dispatch queues, depending on your needs.

If you implemented your tasks using operation objects, the choice of queue is often less interesting than the configuration of your objects. To perform operation objects serially, you must configure dependencies between the related objects. Dependencies prevent one operation from executing until the objects on which it depends have finished their work.

Tips for Improving Efficiency

In addition to simply factoring your code into smaller tasks and adding them to a queue, there are other ways to improve the overall efficiency of your code using queues:

- **Consider computing values directly within your task if memory usage is a factor.** If your application is already memory bound, computing values directly now may be faster than loading cached values from main memory. Computing values directly uses the registers and caches of the given processor core, which are much faster than main memory. Of course, you should only do this if testing indicates this is a performance win.
- **Identify serial tasks early and do what you can to make them more concurrent.** If a task must be executed serially because it relies on some shared resource, consider changing your architecture to remove that shared resource. You might consider making copies of the resource for each client that needs one or eliminate the resource altogether.
- **Avoid using locks.** The support provided by dispatch queues and operation queues makes locks unnecessary in most situations. Instead of using locks to protect some shared resource, designate a serial queue (or use operation object dependencies) to execute tasks in the correct order.
- **Rely on the system frameworks whenever possible.** The best way to achieve concurrency is to take advantage of the built-in concurrency provided by the system frameworks. Many frameworks

use threads and other technologies internally to implement concurrent behaviors. When defining your tasks, look to see if an existing framework defines a function or method that does exactly what you want and does so concurrently. Using that API may save you effort and is more likely to give you the maximum concurrency possible.

Performance Implications

Operation queues, dispatch queues, and dispatch sources are provided to make it easier for you to execute more code concurrently. However, these technologies do not guarantee improvements to the efficiency or responsiveness in your application. It is still your responsibility to use queues in a manner that is both effective for your needs and does not impose an undue burden on your application's other resources. For example, although you could create 10,000 operation objects and submit them to an operation queue, doing so would cause your application to allocate a potentially nontrivial amount of memory, which could lead to paging and decreased performance.

Before introducing any amount of concurrency to your code—whether using queues or threads—you should always gather a set of baseline metrics that reflect your application's current performance. After introducing your changes, you should then gather additional metrics and compare them to your baseline to see if your application's overall efficiency has improved. If the introduction of concurrency makes your application less efficient or responsive, you should use the available performance tools to check for the potential causes.

For an introduction to performance and the available performance tools, and for links to more advanced performance-related topics, see *Performance Overview*.

Concurrency and Other Technologies

Factoring your code into modular tasks is the best way to try and improve the amount of concurrency in your application. However, this design approach may not satisfy the needs of every application in every case. Depending on your tasks, there might be other options that can offer additional improvements in your application's overall concurrency. This section outlines some of the other technologies to consider using as part of your design.

OpenCL and Concurrency

In OS X, the *Open Computing Language (OpenCL)* is a standards-based technology for performing general-purpose computations on a computer's graphics processor. OpenCL is a good technology to use if you have a well-defined set of computations that you want to apply to large data sets. For example, you might use OpenCL to perform filter computations on the pixels of an image or use it to perform complex math calculations on several values at once. In other words, OpenCL is geared more toward problem sets whose data can be operated on in parallel.

Although OpenCL is good for performing massively data-parallel operations, it is not suitable for more general-purpose calculations. There is a nontrivial amount of effort required to prepare and transfer both the data and the required work kernel to a graphics card so that it can be operated on by a GPU. Similarly, there is a nontrivial amount of effort required to retrieve any results generated by OpenCL. As a result, any tasks that interact with the system are generally not recommended for use with OpenCL. For example, you would not use OpenCL to process data from files or network streams. Instead, the work you perform using OpenCL must be much more self-contained so that it can be transferred to the graphics processor and computed independently.

For more information about OpenCL and how you use it, see *OpenCL Programming Guide for Mac*.

When to Use Threads

Although operation queues and dispatch queues are the preferred way to perform tasks concurrently, they are not a panacea. Depending on your application, there may still be times when you need to create custom threads. If you do create custom threads, you should strive to create as few threads as possible yourself and you should use those threads only for specific tasks that cannot be implemented any other way.

Threads are still a good way to implement code that must run in real time. Dispatch queues make every attempt to run their tasks as fast as possible but they do not address real time constraints. If you need more predictable behavior from code running in the background, threads may still offer a better alternative.

As with any threaded programming, you should always use threads judiciously and only when absolutely necessary. For more information about thread packages and how you use them, see *Threading Programming Guide*.
