Concurrency Programming Guide

Developer

Contents

Tables and Listings

Objective-CSwift

Introduction

Concurrency is the notion of multiple things happening at the same time. With the proliferation of multicore CPUs and the realization that the number of cores in each processor will only increase, software developers need new ways to take advantage of them. Although operating systems like OS X and iOS are capable of running multiple programs in parallel, most of those programs run in the background and perform tasks that require little continuous processor time. It is the current foreground application that both captures the user's attention and keeps the computer busy. If an application has a lot of work to do but keeps only a fraction of the available cores occupied, those extra processing resources are wasted.

In the past, introducing concurrency to an application required the creation of one or more additional threads. Unfortunately, writing threaded code is challenging. Threads are a low-level tool that must be managed manually. Given that the optimal number of threads for an application can change dynamically based on the current system load and the underlying hardware, implementing a correct threading solution becomes extremely difficult, if not impossible to achieve. In addition, the synchronization mechanisms typically used with threads add complexity and risk to software designs without any guarantees of improved performance.

Both OS X and iOS adopt a more asynchronous approach to the execution of concurrent tasks than is traditionally found in thread-based systems and applications. Rather than creating threads directly, applications need only define specific tasks and then let the system perform them. By letting the system manage the threads, applications gain a level of scalability not possible with raw threads. Application developers also gain a simpler and more efficient programming model.

This document describes the technique and technologies you should be using to implement concurrency in your applications. The technologies described in this document are available in both OS X and iOS.

Organization of This Document

This document contains the following chapters:

- Concurrency and Application Design (page 10) introduces the basics of asynchronous application design and the technologies for performing your custom tasks asynchronously.
- Operation Queues (page 18) shows you how to encapsulate and perform tasks using Objective-C objects.
- Dispatch Queues (page 39) shows you how to execute tasks concurrently in C-based applications.
- Dispatch Sources (page 57) shows you how to handle system events asynchronously.

- Migrating Away from Threads (page 75) provides tips and techniques for migrating your existing thread-based code over to use newer technologies.

This document also includes a glossary that defines relevant terms.

A Note About Terminology

Before entering into a discussion about concurrency, it is necessary to define some relevant terminology to prevent confusion. Developers who are more familiar with UNIX systems or older OS X technologies may find the terms "task", "process", and "thread" used somewhat differently in this document. This document uses these terms in the following way:

- The term *thread* is used to refer to a separate path of execution for code. The underlying implementation for threads in OS X is based on the POSIX threads API.
- The term *process* is used to refer to a running executable, which can encompass multiple threads.
- The term *task* is used to refer to the abstract concept of work that needs to be performed.

For complete definitions of these and other key terms used by this document, see Glossary (page 85).

See Also

This document focuses on the preferred technologies for implementing concurrency in your applications and does not cover the use of threads. If you need information about using threads and other thread-related technologies, see *Threading Programming Guide*.

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 (page 39). 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 (page 57).

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 (page 18).

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*.

Operation Queues

SwiftObjective-C

Cocoa operations are an object-oriented way to encapsulate work that you want to perform asynchronously. Operations are designed to be used either in conjunction with an operation queue or by themselves. Because they are Objective-C based, operations are most commonly used in Cocoa-based applications in OS X and iOS.

This chapter shows you how to define and use operations.

About Operation Objects

An *operation object* is an instance of the `NSOperation` class (in the Foundation framework) that you use to encapsulate work you want your application to perform. The `NSOperation` class itself is an abstract base class that must be subclassed in order to do any useful work. Despite being abstract, this class does provide a significant amount of infrastructure to minimize the amount of work you have to do in your own subclasses. In addition, the Foundation framework provides two concrete subclasses that you can use as-is with your existing code. Table 2-1 lists these classes, along with a summary of how you use each one.

Table 2-1 Operation classes of the Foundation framework

Class	Description
`NSInvocation-Operation`	A class you use as-is to create an operation object based on an object and selector from your application. You can use this class in cases where you have an existing method that already performs the needed task. Because it does not require subclassing, you can also use this class to create operation objects in a more dynamic fashion. For information about how to use this class, see Creating an NSInvocationOperation Object (page 20).
`NSBlockOperation`	A class you use as-is to execute one or more block objects concurrently. Because it can execute more than one block, a block operation object operates using a group semantic; only when all of the associated blocks have finished executing is the operation itself considered finished. For information about how to use this class, see Creating an NSBlockOperation Object (page 21). This class is available in OS X v10.6 and later. For more information about blocks, see *Blocks Programming Topics*.

Class	Description
`NSOperation`	The base class for defining custom operation objects. Subclassing `NSOperation` gives you complete control over the implementation of your own operations, including the ability to alter the default way in which your operation executes and reports its status. For information about how to define custom operation objects, see Defining a Custom Operation Object (page 22).

All operation objects support the following key features:

- Support for the establishment of graph-based dependencies between operation objects. These dependencies prevent a given operation from running until all of the operations on which it depends have finished running. For information about how to configure dependencies, see Configuring Interoperation Dependencies (page 29).
- Support for an optional completion block, which is executed after the operation's main task finishes. (OS X v10.6 and later only.) For information about how to set a completion block, see Setting Up a Completion Block (page 31).
- Support for monitoring changes to the execution state of your operations using KVO notifications. For information about how to observe KVO notifications, see *Key-Value Observing Programming Guide*.
- Support for prioritizing operations and thereby affecting their relative execution order. For more information, see Changing an Operation's Execution Priority (page 30).
- Support for canceling semantics that allow you to halt an operation while it is executing. For information about how to cancel operations, see Canceling Operations (page 37). For information about how to support cancellation in your own operations, see Responding to Cancellation Events (page 23).

Operations are designed to help you improve the level of concurrency in your application. Operations are also a good way to organize and encapsulate your application's behavior into simple discrete chunks. Instead of running some bit of code on your application's main thread, you can submit one or more operation objects to a queue and let the corresponding work be performed asynchronously on one or more separate threads.

Concurrent Versus Non-concurrent Operations

Although you typically execute operations by adding them to an operation queue, doing so is not required. It is also possible to execute an operation object manually by calling its `start` method, but doing so does not guarantee that the operation runs concurrently with the rest of your code. The `isConcurrent` method of the

`NSOperation` class tells you whether an operation runs synchronously or asynchronously with respect to the thread in which its `start` method was called. By default, this method returns `NO`, which means the operation runs synchronously in the calling thread.

If you want to implement a *concurrent operation*—that is, one that runs asynchronously with respect to the calling thread—you must write additional code to start the operation asynchronously. For example, you might spawn a separate thread, call an asynchronous system function, or do anything else to ensure that the `start` method starts the task and returns immediately and, in all likelihood, before the task is finished.

Most developers should never need to implement concurrent operation objects. If you always add your operations to an operation queue, you do not need to implement concurrent operations. When you submit a nonconcurrent operation to an operation queue, the queue itself creates a thread on which to run your operation. Thus, adding a nonconcurrent operation to an operation queue still results in the asynchronous execution of your operation object code. The ability to define concurrent operations is only necessary in cases where you need to execute the operation asynchronously without adding it to an operation queue.

For information about how to create a concurrent operation, see Configuring Operations for Concurrent Execution (page 25) and *NSOperation Class Reference*.

Creating an NSInvocationOperation Object

The `NSInvocationOperation` class is a concrete subclass of `NSOperation` that, when run, invokes the selector you specify on the object you specify. Use this class to avoid defining large numbers of custom operation objects for each task in your application; especially if you are modifying an existing application and already have the objects and methods needed to perform the necessary tasks. You can also use it when the method you want to call can change depending on the circumstances. For example, you could use an invocation operation to perform a selector that is chosen dynamically based on user input.

The process for creating an invocation operation is straightforward. You create and initialize a new instance of the class, passing the desired object and selector to execute to the initialization method. Listing 2-1 shows two methods from a custom class that demonstrate the creation process. The `taskWithData:` method creates a new invocation object and supplies it with the name of another method, which contains the task implementation.

Listing 2-1 Creating an `NSInvocationOperation` object

```
@implementation MyCustomClass
- (NSOperation*)taskWithData:(id)data {
    NSInvocationOperation* theOp = [[NSInvocationOperation alloc] initWithTarget:self
```

```
                    selector:@selector(myTaskMethod:) object:data];

    return theOp;
}

// This is the method that does the actual work of the task.
- (void)myTaskMethod:(id)data {
    // Perform the task.
}
@end
```

Creating an NSBlockOperation Object

The `NSBlockOperation` class is a concrete subclass of `NSOperation` that acts as a wrapper for one or more block objects. This class provides an object-oriented wrapper for applications that are already using operation queues and do not want to create dispatch queues as well. You can also use block operations to take advantage of operation dependencies, KVO notifications, and other features that might not be available with dispatch queues.

When you create a block operation, you typically add at least one block at initialization time; you can add more blocks as needed later. When it comes time to execute an `NSBlockOperation` object, the object submits all of its blocks to the default-priority, concurrent dispatch queue. The object then waits until all of the blocks finish executing. When the last block finishes executing, the operation object marks itself as finished. Thus, you can use a block operation to track a group of executing blocks, much like you would use a thread join to merge the results from multiple threads. The difference is that because the block operation itself runs on a separate thread, your application's other threads can continue doing work while waiting for the block operation to complete.

Listing 2-2 shows a simple example of how to create an `NSBlockOperation` object. The block itself has no parameters and no significant return result.

Listing 2-2 Creating an `NSBlockOperation` object

```
NSBlockOperation* theOp = [NSBlockOperation blockOperationWithBlock: ^{
      NSLog(@"Beginning operation.\n");
      // Do some work.
   }];
```

After creating a block operation object, you can add more blocks to it using the `addExecutionBlock:` method. If you need to execute blocks serially, you must submit them directly to the desired dispatch queue.

Defining a Custom Operation Object

If the block operation and invocation operation objects do not quite meet the needs of your application, you can subclass `NSOperation` directly and add whatever behavior you need. The `NSOperation` class provides a general subclassing point for all operation objects. The class also provides a significant amount of infrastructure to handle most of the work needed for dependencies and KVO notifications. However, there may still be times when you need to supplement the existing infrastructure to ensure that your operations behave correctly. The amount of extra work you have to do depends on whether you are implementing a nonconcurrent or a concurrent operation.

Defining a nonconcurrent operation is much simpler than defining a concurrent operation. For a nonconcurrent operation, all you have to do is perform your main task and respond appropriately to cancellation events; the existing class infrastructure does all of the other work for you. For a concurrent operation, you must replace some of the existing infrastructure with your custom code. The following sections show you how to implement both types of object.

Performing the Main Task

At a minimum, every operation object should implement at least the following methods:

- A custom initialization method
- `main`

You need a custom initialization method to put your operation object into a known state and a custom `main` method to perform your task. You can implement additional methods as needed, of course, such as the following:

- Custom methods that you plan to call from the implementation of your `main` method
- Accessor methods for setting data values and accessing the results of the operation
- Methods of the `NSCoding` protocol to allow you to archive and unarchive the operation object

Listing 2-3 shows a starting template for a custom `NSOperation` subclass. (This listing does not show how to handle cancellation but does show the methods you would typically have. For information about handling cancellation, see Responding to Cancellation Events (page 23).) The initialization method for this class takes a single object as a data parameter and stores a reference to it inside the operation object. The `main` method would ostensibly work on that data object before returning the results back to your application.

Listing 2-3 Defining a simple operation object

```
@interface MyNonConcurrentOperation : NSOperation
@property id (strong) myData;
-(id)initWithData:(id)data;
@end

@implementation MyNonConcurrentOperation
- (id)initWithData:(id)data {
   if (self = [super init])
      myData = data;
   return self;
}

-(void)main {
   @try {
      // Do some work on myData and report the results.
   }
   @catch(...) {
      // Do not rethrow exceptions.
   }
}
@end
```

For a detailed example of how to implement an `NSOperation` subclass, see *NSOperationSample*.

Responding to Cancellation Events

After an operation begins executing, it continues performing its task until it is finished or until your code explicitly cancels the operation. Cancellation can occur at any time, even before an operation begins executing. Although the `NSOperation` class provides a way for clients to cancel an operation, recognizing the cancellation event is voluntary by necessity. If an operation were terminated outright, there might not be a way to reclaim resources that had been allocated. As a result, operation objects are expected to check for cancellation events and to exit gracefully when they occur in the middle of the operation.

To support cancellation in an operation object, all you have to do is call the object's `isCancelled` method periodically from your custom code and return immediately if it ever returns `YES`. Supporting cancellation is important regardless of the duration of your operation or whether you subclass `NSOperation` directly or use one of its concrete subclasses. The `isCancelled` method itself is very lightweight and can be called frequently without any significant performance penalty. When designing your operation objects, you should consider calling the `isCancelled` method at the following places in your code:

- Immediately before you perform any actual work
- At least once during each iteration of a loop, or more frequently if each iteration is relatively long
- At any points in your code where it would be relatively easy to abort the operation

Listing 2-4 provides a very simple example of how to respond to cancellation events in the `main` method of an operation object. In this case, the `isCancelled` method is called each time through a `while` loop, allowing for a quick exit before work begins and again at regular intervals.

Listing 2-4 Responding to a cancellation request

```
- (void)main {
   @try {
      BOOL isDone = NO;

      while (![self isCancelled] && !isDone) {
          // Do some work and set isDone to YES when finished
      }
   }
   @catch(...) {
      // Do not rethrow exceptions.
   }
}
```

Although the preceding example contains no cleanup code, your own code should be sure to free up any resources that were allocated by your custom code.

Configuring Operations for Concurrent Execution

Operation objects execute in a synchronous manner by default—that is, they perform their task in the thread that calls their `start` method. Because operation queues provide threads for nonconcurrent operations, though, most operations still run asynchronously. However, if you plan to execute operations manually and still want them to run asynchronously, you must take the appropriate actions to ensure that they do. You do this by defining your operation object as a concurrent operation.

Table 2-2 lists the methods you typically override to implement a concurrent operation.

Table 2-2 Methods to override for concurrent operations

Method	Description
`start`	(Required) All concurrent operations must override this method and replace the default behavior with their own custom implementation. To execute an operation manually, you call its `start` method. Therefore, your implementation of this method is the starting point for your operation and is where you set up the thread or other execution environment in which to execute your task. Your implementation must not call `super` at any time.
`main`	(Optional) This method is typically used to implement the task associated with the operation object. Although you could perform the task in the `start` method, implementing the task using this method can result in a cleaner separation of your setup and task code.
`isExecuting` `isFinished`	(Required) Concurrent operations are responsible for setting up their execution environment and reporting the status of that environment to outside clients. Therefore, a concurrent operation must maintain some state information to know when it is executing its task and when it has finished that task. It must then report that state using these methods. Your implementations of these methods must be safe to call from other threads simultaneously. You must also generate the appropriate KVO notifications for the expected key paths when changing the values reported by these methods.
`isConcurrent`	(Required) To identify an operation as a concurrent operation, override this method and return `YES`.

The rest of this section shows a sample implementation of the `MyOperation` class, which demonstrates the fundamental code needed to implement a concurrent operation. The `MyOperation` class simply executes its own `main` method on a separate thread that it creates. The actual work that the `main` method performs is irrelevant. The point of the sample is to demonstrate the infrastructure you need to provide when defining a concurrent operation.

Listing 2-5 shows the interface and part of the implementation of the `MyOperation` class. The implementations of the `isConcurrent`, `isExecuting`, and `isFinished` methods for the `MyOperation` class are relatively straightforward. The `isConcurrent` method should simply return `YES` to indicate that this is a concurrent operation. The `isExecuting` and `isFinished` methods simply return values stored in instance variables of the class itself.

Listing 2-5 Defining a concurrent operation

```
@interface MyOperation : NSOperation {
    BOOL        executing;
    BOOL        finished;
}
- (void)completeOperation;
@end

@implementation MyOperation
- (id)init {
    self = [super init];
    if (self) {
        executing = NO;
        finished = NO;
    }
    return self;
}

- (BOOL)isConcurrent {
    return YES;
}

- (BOOL)isExecuting {
    return executing;
}

- (BOOL)isFinished {
    return finished;
}
```

```
@end
```

Listing 2-6 shows the `start` method of `MyOperation`. The implementation of this method is minimal so as to demonstrate the tasks you absolutely must perform. In this case, the method simply starts up a new thread and configures it to call the `main` method. The method also updates the `executing` member variable and generates KVO notifications for the `isExecuting` key path to reflect the change in that value. With its work done, this method then simply returns, leaving the newly detached thread to perform the actual task.

Listing 2-6 The start method

```
- (void)start {
   // Always check for cancellation before launching the task.
   if ([self isCancelled])
   {
      // Must move the operation to the finished state if it is canceled.
      [self willChangeValueForKey:@"isFinished"];
      finished = YES;
      [self didChangeValueForKey:@"isFinished"];
      return;
   }

   // If the operation is not canceled, begin executing the task.
   [self willChangeValueForKey:@"isExecuting"];
   [NSThread detachNewThreadSelector:@selector(main) toTarget:self withObject:nil];
   executing = YES;
   [self didChangeValueForKey:@"isExecuting"];
}
```

Listing 2-7 shows the remaining implementation for the `MyOperation` class. As was seen in Listing 2-6 (page 27), the `main` method is the entry point for a new thread. It performs the work associated with the operation object and calls the custom `completeOperation` method when that work is finally done. The `completeOperation` method then generates the needed KVO notifications for both the `isExecuting` and `isFinished` key paths to reflect the change in state of the operation.

Listing 2-7 Updating an operation at completion time

```
- (void)main {
```

```
    @try {

        // Do the main work of the operation here.

        [self completeOperation];
    }
    @catch(...) {
       // Do not rethrow exceptions.
    }
}

- (void)completeOperation {
    [self willChangeValueForKey:@"isFinished"];
    [self willChangeValueForKey:@"isExecuting"];

    executing = NO;
    finished = YES;

    [self didChangeValueForKey:@"isExecuting"];
    [self didChangeValueForKey:@"isFinished"];
}
```

Even if an operation is canceled, you should always notify KVO observers that your operation is now finished with its work. When an operation object is dependent on the completion of other operation objects, it monitors the `isFinished` key path for those objects. Only when all objects report that they are finished does the dependent operation signal that it is ready to run. Failing to generate a finish notification can therefore prevent the execution of other operations in your application.

Maintaining KVO Compliance

The `NSOperation` class is key-value observing (KVO) compliant for the following key paths:

- `isCancelled`
- `isConcurrent`
- `isExecuting`
- `isFinished`

- `isReady`
- `dependencies`
- `queuePriority`
- `completionBlock`

If you override the `start` method or do any significant customization of an `NSOperation` object other than override `main`, you must ensure that your custom object remains KVO compliant for these key paths. When overriding the `start` method, the key paths you should be most concerned with are `isExecuting` and `isFinished`. These are the key paths most commonly affected by reimplementing that method.

If you want to implement support for dependencies on something besides other operation objects, you can also override the `isReady` method and force it to return `NO` until your custom dependencies were satisfied. (If you implement custom dependencies, be sure to call `super` from your `isReady` method if you still support the default dependency management system provided by the `NSOperation` class.) When the readiness status of your operation object changes, generate KVO notifications for the `isReady` key path to report those changes. Unless you override the `addDependency:` or `removeDependency:` methods, you should not need to worry about generating KVO notifications for the `dependencies` key path.

Although you could generate KVO notifications for other key paths of `NSOperation`, it is unlikely you would ever need to do so. If you need to cancel an operation, you can simply call the existing `cancel` method to do so. Similarly, there should be little need for you to modify the queue priority information in an operation object. Finally, unless your operation is capable of changing its concurrency status dynamically, you do not need to provide KVO notifications for the `isConcurrent` key path.

For more information on key-value observing and how to support it in your custom objects, see *Key-Value Observing Programming Guide*.

Customizing the Execution Behavior of an Operation Object

The configuration of operation objects occurs after you have created them but before you add them to a queue. The types of configurations described in this section can be applied to all operation objects, regardless of whether you subclassed `NSOperation` yourself or used an existing subclass.

Configuring Interoperation Dependencies

Dependencies are a way for you to serialize the execution of distinct operation objects. An operation that is dependent on other operations cannot begin executing until all of the operations on which it depends have finished executing. Thus, you can use dependencies to create simple one-to-one dependencies between two operation objects or to build complex object dependency graphs.

To establish dependencies between two operation objects, you use the `addDependency:` method of `NSOperation`. This method creates a one-way dependency from the current operation object to the target operation you specify as a parameter. This dependency means that the current object cannot begin executing until the target object finishes executing. Dependencies are also not limited to operations in the same queue. Operation objects manage their own dependencies and so it is perfectly acceptable to create dependencies between operations and add them all to different queues. One thing that is not acceptable, however, is to create circular dependencies between operations. Doing so is a programmer error that will prevent the affected operations from ever running.

When all of an operation's dependencies have themselves finished executing, an operation object normally becomes ready to execute. (If you customize the behavior of the `isReady` method, the readiness of the operation is determined by the criteria you set.) If the operation object is in a queue, the queue may start executing that operation at any time. If you plan to execute the operation manually, it is up to you to call the operation's `start` method.

> **Important:** You should always configure dependencies before running your operations or adding them to an operation queue. Dependencies added afterward may not prevent a given operation object from running.

Dependencies rely on each operation object sending out appropriate KVO notifications whenever the status of the object changes. If you customize the behavior of your operation objects, you may need to generate appropriate KVO notifications from your custom code in order to avoid causing issues with dependencies. For more information on KVO notifications and operation objects, see Maintaining KVO Compliance (page 28). For additional information on configuring dependencies, see *NSOperation Class Reference*.

Changing an Operation's Execution Priority

For operations added to a queue, execution order is determined first by the readiness of the queued operations and then by their relative priority. Readiness is determined by an operation's dependencies on other operations, but the priority level is an attribute of the operation object itself. By default, all new operation objects have a "normal" priority, but you can increase or decrease that priority as needed by calling the object's `setQueuePriority:` method.

Priority levels apply only to operations in the same operation queue. If your application has multiple operation queues, each prioritizes its own operations independently of any other queues. Thus, it is still possible for low-priority operations to execute before high-priority operations in a different queue.

Priority levels are not a substitute for dependencies. Priorities determine the order in which an operation queue starts executing only those operations that are currently ready. For example, if a queue contains both a high-priority and low-priority operation and both operations are ready, the queue executes the high-priority operation first. However, if the high-priority operation is not ready but the low-priority operation is, the queue

executes the low-priority operation first. If you want to prevent one operation from starting until another operation has finished, you must use dependencies (as described in Configuring Interoperation Dependencies (page 29)) instead.

Changing the Underlying Thread Priority

In OS X v10.6 and later, it is possible to configure the execution priority of an operation's underlying thread. Thread policies in the system are themselves managed by the kernel, but in general higher-priority threads are given more opportunities to run than lower-priority threads. In an operation object, you specify the thread priority as a floating-point value in the range 0.0 to 1.0, with 0.0 being the lowest priority and 1.0 being the highest priority. If you do not specify an explicit thread priority, the operation runs with the default thread priority of 0.5.

To set an operation's thread priority, you must call the `setThreadPriority:` method of your operation object before adding it to a queue (or executing it manually). When it comes time to execute the operation, the default `start` method uses the value you specified to modify the priority of the current thread. This new priority remains in effect for the duration of your operation's `main` method only. All other code (including your operation's completion block) is run with the default thread priority. If you create a concurrent operation, and therefore override the `start` method, you must configure the thread priority yourself.

Setting Up a Completion Block

In OS X v10.6 and later, an operation can execute a completion block when its main task finishes executing. You can use a completion block to perform any work that you do not consider part of the main task. For example, you might use this block to notify interested clients that the operation itself has completed. A concurrent operation object might use this block to generate its final KVO notifications.

To set a completion block, use the `setCompletionBlock:` method of `NSOperation`. The block you pass to this method should have no arguments and no return value.

Tips for Implementing Operation Objects

Although operation objects are fairly easy to implement, there are several things you should be aware of as you are writing your code. The following sections describe factors that you should take into account when writing the code for your operation objects.

Managing Memory in Operation Objects

The following sections describe key elements of good memory management in an operation object. For general information about memory management in Objective-C programs, see *Advanced Memory Management Programming Guide*.

Avoid Per-Thread Storage

Although most operations execute on a thread, in the case of nonconcurrent operations, that thread is usually provided by an operation queue. If an operation queue provides a thread for you, you should consider that thread to be owned by the queue and not to be touched by your operation. Specifically, you should never associate any data with a thread that you do not create yourself or manage. The threads managed by an operation queue come and go depending on the needs of the system and your application. Therefore, passing data between operations using per-thread storage is unreliable and likely to fail.

In the case of operation objects, there should be no reason for you to use per-thread storage in any case. When you initialize an operation object, you should provide the object with everything it needs to do its job. Therefore, the operation object itself provides the contextual storage you need. All incoming and outgoing data should be stored there until it can be integrated back into your application or is no longer required.

Keep References to Your Operation Object As Needed

Just because operation objects run asynchronously, you should not assume that you can create them and forget about them. They are still just objects and it is up to you to manage any references to them that your code needs. This is especially important if you need to retrieve result data from an operation after it is finished.

The reason you should always keep your own references to operations is that you may not get the chance to ask a queue for the object later. Queues make every effort to dispatch and execute operations as quickly as possible. In many cases, queues start executing operations almost immediately after they are added. By the time your own code goes back to the queue to get a reference to the operation, that operation could already be finished and removed from the queue.

Handling Errors and Exceptions

Because operations are essentially discrete entities inside your application, they are responsible for handling any errors or exceptions that arise. In OS X v10.6 and later, the default `start` method provided by the `NSOperation` class does not catch exceptions. (In OS X v10.5, the start method does catch and suppress exceptions.) Your own code should always catch and suppress exceptions directly. It should also check error codes and notify the appropriate parts of your application as needed. And if you replace the `start` method, you must similarly catch any exceptions in your custom implementation to prevent them from leaving the scope of the underlying thread.

Among the types of error situations you should be prepared to handle are the following:

- Check and handle UNIX `errno`-style error codes.
- Check explicit error codes returned by methods and functions.
- Catch exceptions thrown by your own code or by other system frameworks.
- Catch exceptions thrown by the `NSOperation` class itself, which throws exceptions in the following situations:
    - When the operation is not ready to execute but its `start` method is called
    - When the operation is executing or finished (possibly because it was canceled) and its `start` method is called again
    - When you try to add a completion block to an operation that is already executing or finished
    - When you try to retrieve the result of an `NSInvocationOperation` object that was canceled

If your custom code does encounter an exception or error, you should take whatever steps are needed to propagate that error to the rest of your application. The `NSOperation` class does not provide explicit methods for passing along error result codes or exceptions to other parts of your application. Therefore, if such information is important to your application, you must provide the necessary code.

Determining an Appropriate Scope for Operation Objects

Although it is possible to add an arbitrarily large number of operations to an operation queue, doing so is often impractical. Like any object, instances of the `NSOperation` class consume memory and have real costs associated with their execution. If each of your operation objects does only a small amount of work, and you create tens of thousands of them, you may find that you are spending more time dispatching operations than doing real work. And if your application is already memory-constrained, you might find that just having tens of thousands of operation objects in memory might degrade performance even further.

The key to using operations efficiently is to find an appropriate balance between the amount of work you need to do and to keep the computer busy. Try to make sure that your operations do a reasonable amount of work. For example, if your application creates 100 operation objects to perform the same task on 100 different values, consider creating 10 operation objects to process 10 values each instead.

You should also avoid adding large numbers of operations to a queue all at once, or avoid continuously adding operation objects to a queue faster than they can be processed. Rather than flood a queue with operation objects, create those objects in batches. As one batch finishes executing, use a completion block to tell your application to create a new batch. When you have a lot of work to do, you want to keep the queues filled with enough operations so that the computer stays busy, but you do not want to create so many operations at once that your application runs out of memory.

Of course, the number of operation objects you create, and the amount of work you perform in each, is variable and entirely dependent on your application. You should always use tools such as Instruments to help you find an appropriate balance between efficiency and speed. For an overview of Instruments and the other performance tools you can use to gather metrics for your code, see *Performance Overview*.

Executing Operations

Ultimately, your application needs to execute operations in order to do the associated work. In this section, you learn several ways to execute operations as well as how you can manipulate the execution of your operations at runtime.

Adding Operations to an Operation Queue

By far, the easiest way to execute operations is to use an operation queue, which is an instance of the `NSOperationQueue` class. Your application is responsible for creating and maintaining any operation queues it intends to use. An application can have any number of queues, but there are practical limits to how many operations may be executing at a given point in time. Operation queues work with the system to restrict the number of concurrent operations to a value that is appropriate for the available cores and system load. Therefore, creating additional queues does not mean that you can execute additional operations.

To create a queue, you allocate it in your application as you would any other object:

```
NSOperationQueue* aQueue = [[NSOperationQueue alloc] init];
```

To add operations to a queue, you use the `addOperation:` method. In OS X v10.6 and later, you can add groups of operations using the `addOperations:waitUntilFinished:` method, or you can add block objects directly to a queue (without a corresponding operation object) using the `addOperationWithBlock:` method. Each of these methods queues up an operation (or operations) and notifies the queue that it should begin processing them. In most cases, operations are executed shortly after being added to a queue, but the operation queue may delay execution of queued operations for any of several reasons. Specifically, execution may be delayed if queued operations are dependent on other operations that have not yet completed. Execution may also be delayed if the operation queue itself is suspended or is already executing its maximum number of concurrent operations. The following examples show the basic syntax for adding operations to a queue.

```
[aQueue addOperation:anOp]; // Add a single operation
[aQueue addOperations:anArrayOfOps waitUntilFinished:NO]; // Add multiple operations
[aQueue addOperationWithBlock:^{
   /* Do something. */
```

```
}];
```

Important: Never modify an operation object after it has been added to a queue. While waiting in a queue, the operation could start executing at any time, so changing its dependencies or the data it contains could have adverse effects. If you want to know the status of an operation, you can use the methods of the `NSOperation` class to determine if the operation is running, waiting to run, or already finished.

Although the `NSOperationQueue` class is designed for the concurrent execution of operations, it is possible to force a single queue to run only one operation at a time. The `setMaxConcurrentOperationCount:` method lets you configure the maximum number of concurrent operations for an operation queue object. Passing a value of 1 to this method causes the queue to execute only one operation at a time. Although only one operation at a time may execute, the order of execution is still based on other factors, such as the readiness of each operation and its assigned priority. Thus, a serialized operation queue does not offer quite the same behavior as a serial dispatch queue in Grand Central Dispatch does. If the execution order of your operation objects is important to you, you should use dependencies to establish that order before adding your operations to a queue. For information about configuring dependencies, see Configuring Interoperation Dependencies (page 29).

For information about using operation queues, see *NSOperationQueue Class Reference*. For more information about serial dispatch queues, see Creating Serial Dispatch Queues (page 45).

Executing Operations Manually

Although operation queues are the most convenient way to run operation objects, it is also possible to execute operations without a queue. If you choose to execute operations manually, however, there are some precautions you should take in your code. In particular, the operation must be ready to run and you must always start it using its `start` method.

An operation is not considered able to run until its `isReady` method returns `YES`. The `isReady` method is integrated into the dependency management system of the `NSOperation` class to provide the status of the operation's dependencies. Only when its dependencies are cleared is an operation free to begin executing.

When executing an operation manually, you should always use the `start` method to begin execution. You use this method, instead of `main` or some other method, because the `start` method performs several safety checks before it actually runs your custom code. In particular, the default `start` method generates the KVO notifications that operations require to process their dependencies correctly. This method also correctly avoids executing your operation if it has already been canceled and throws an exception if your operation is not actually ready to run.

If your application defines concurrent operation objects, you should also consider calling the `isConcurrent` method of operations prior to launching them. In cases where this method returns `NO`, your local code can decide whether to execute the operation synchronously in the current thread or create a separate thread first. However, implementing this kind of checking is entirely up to you.

Listing 2-8 shows a simple example of the kind of checks you should perform before executing operations manually. If the method returns `NO`, you could schedule a timer and call the method again later. You would then keep rescheduling the timer until the method returns `YES`, which could occur because the operation was canceled.

Listing 2-8 Executing an operation object manually

```
- (BOOL)performOperation:(NSOperation*)anOp
{
   BOOL        ranIt = NO;

   if ([anOp isReady] && ![anOp isCancelled])
   {
      if (![anOp isConcurrent])
         [anOp start];
      else
         [NSThread detachNewThreadSelector:@selector(start)
                   toTarget:anOp withObject:nil];
      ranIt = YES;
   }
   else if ([anOp isCancelled])
   {
      // If it was canceled before it was started,
      //  move the operation to the finished state.
      [self willChangeValueForKey:@"isFinished"];
      [self willChangeValueForKey:@"isExecuting"];
      executing = NO;
      finished = YES;
      [self didChangeValueForKey:@"isExecuting"];
      [self didChangeValueForKey:@"isFinished"];

      // Set ranIt to YES to prevent the operation from
```

```
        // being passed to this method again in the future.
        ranIt = YES;
    }
    return ranIt;
}
```

Canceling Operations

Once added to an operation queue, an operation object is effectively owned by the queue and cannot be removed. The only way to dequeue an operation is to cancel it. You can cancel a single individual operation object by calling its `cancel` method or you can cancel all of the operation objects in a queue by calling the `cancelAllOperations` method of the queue object.

You should cancel operations only when you are sure you no longer need them. Issuing a cancel command puts the operation object into the "canceled" state, which prevents it from ever being run. Because a canceled operation is still considered to be "finished", objects that are dependent on it receive the appropriate KVO notifications to clear that dependency. Thus, it is more common to cancel all queued operations in response to some significant event, like the application quitting or the user specifically requesting the cancellation, rather than cancel operations selectively.

Waiting for Operations to Finish

For the best performance, you should design your operations to be as asynchronous as possible, leaving your application free to do additional work while the operation executes. If the code that creates an operation also processes the results of that operation, you can use the `waitUntilFinished` method of `NSOperation` to block that code until the operation finishes. In general, though, it is best to avoid calling this method if you can help it. Blocking the current thread may be a convenient solution, but it does introduce more serialization into your code and limits the overall amount of concurrency.

> **Important:** You should never wait for an operation from your application's main thread. You should only do so from a secondary thread or from another operation. Blocking your main thread prevents your application from responding to user events and could make your application appear unresponsive.

In addition to waiting for a single operation to finish, you can also wait on all of the operations in a queue by calling the `waitUntilAllOperationsAreFinished` method of `NSOperationQueue`. When waiting for an entire queue to finish, be aware that your application's other threads can still add operations to the queue, thus prolonging the wait.

Suspending and Resuming Queues

If you want to issue a temporary halt to the execution of operations, you can suspend the corresponding operation queue using the `setSuspended:` method. Suspending a queue does not cause already executing operations to pause in the middle of their tasks. It simply prevents new operations from being scheduled for execution. You might suspend a queue in response to a user request to pause any ongoing work, because the expectation is that the user might eventually want to resume that work.

Dispatch Queues

Grand Central Dispatch (GCD) dispatch queues are a powerful tool for performing tasks. 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. The advantage of dispatch queues is that they are simpler to use and much more efficient at executing those tasks than the corresponding threaded code.

This chapter provides an introduction to dispatch queues, along with information about how to use them to execute general tasks in your application. If you want to replace existing threaded code with dispatch queues, you can find some additional tips for how to do that in Migrating Away from Threads (page 75).

About Dispatch Queues

Dispatch queues are an easy way to perform tasks asynchronously and concurrently in your application. A *task* is simply some work that your application needs to perform. For example, you could define a task to perform some calculations, create or modify a data structure, process some data read from a file, or any number of things. You define tasks by placing the corresponding code inside either a function or a block object and adding it to a dispatch queue.

A dispatch queue is an object-like structure that manages the tasks you submit to it. All dispatch queues are first-in, first-out data structures. Thus, the tasks you add to a queue are always started in the same order that they were added. GCD provides some dispatch queues for you automatically, but others you can create for specific purposes. Table 3-1 lists the types of dispatch queues available to your application and how you use them.

Table 3-1 Types of dispatch queues

Type	Description
Serial	Serial queues (also known as *private dispatch queues*) 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. For information on how to create serial queues, see Creating Serial Dispatch Queues (page 45).
Concurrent	Concurrent queues (also known as a type of *global dispatch queue*) 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 `DISPATCH_QUEUE_CONCURRENT` as the queue type. In addition, there are four predefined global concurrent queues for your application to use. For more information on how to get the global concurrent queues, see Getting the Global Concurrent Dispatch Queues (page 44).
Main dispatch queue	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 (if one is present) 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. Although you do not need to create the main dispatch queue, you do need to make sure your application drains it appropriately. For more information on how this queue is managed, see Performing Tasks on the Main Thread (page 52).

When it comes to adding concurrency to an application, dispatch queues provide several advantages over threads. The most direct advantage is the simplicity of the work-queue programming model. 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 advantage is that 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. In addition, the system is usually able to start running your task more quickly than you could if you created the thread yourself.

Although you might think rewriting your code for dispatch queues would be difficult, it is often easier to write code for dispatch queues than it is to write code for threads. The key to writing your code is to design tasks that are self-contained and able to run asynchronously. (This is actually true for both threads and dispatch queues.) However, where dispatch queues have an advantage is in predictability. If you have two tasks that access the same shared resource but run on different threads, either thread could modify the resource first and you would need to use a lock to ensure that both tasks did not modify that resource at the same time. With dispatch queues, you could add both tasks to a serial dispatch queue to ensure that only one task modified the resource at any given time. This type of queue-based synchronization is more efficient than locks because locks always require an expensive kernel trap in both the contested and uncontested cases, whereas a dispatch queue works primarily in your application's process space and only calls down to the kernel when absolutely necessary.

Although you would be right to point out that two tasks running in a serial queue do not run concurrently, you have to remember that if two threads take a lock at the same time, any concurrency offered by the threads is lost or significantly reduced. More importantly, the threaded model requires the creation of two threads, which take up both kernel and user-space memory. Dispatch queues do not pay the same memory penalty for their threads, and the threads they do use are kept busy and not blocked.

Some other key points to remember about dispatch queues include the following:

- Dispatch queues execute their tasks concurrently with respect to other dispatch queues. The serialization of tasks is limited to the tasks in a single dispatch queue.
- The system determines the total number of tasks executing at any one time. Thus, an application with 100 tasks in 100 different queues may not execute all of those tasks concurrently (unless it has 100 or more effective cores).
- The system takes queue priority levels into account when choosing which new tasks to start. For information about how to set the priority of a serial queue, see Providing a Clean Up Function For a Queue (page 47).
- Tasks in a queue must be ready to execute at the time they are added to the queue. (If you have used Cocoa operation objects before, notice that this behavior differs from the model operations use.)
- Private dispatch queues are reference-counted objects. In addition to retaining the queue in your own code, be aware that dispatch sources can also be attached to a queue and also increment its retain count. Thus, you must make sure that all dispatch sources are canceled and all retain calls are balanced with an appropriate release call. For more information about retaining and releasing queues, see Memory Management for Dispatch Queues (page 46). For more information about dispatch sources, see About Dispatch Sources (page 57).

For more information about interfaces you use to manipulate dispatch queues, see *Grand Central Dispatch (GCD) Reference*.

Queue-Related Technologies

In addition to dispatch queues, Grand Central Dispatch provides several technologies that use queues to help manage your code. Table 3-2 lists these technologies and provides links to where you can find out more information about them.

Table 3-2 Technologies that use dispatch queues

Technology	Description
Dispatch groups	A dispatch group is a way to monitor a set of block objects for completion. (You can monitor the blocks synchronously or asynchronously depending on your needs.) Groups provide a useful synchronization mechanism for code that depends on the completion of other tasks. For more information about using groups, see Waiting on Groups of Queued Tasks (page 54).
Dispatch semaphores	A dispatch semaphore is similar to a traditional semaphore but is generally more efficient. Dispatch semaphores call down to the kernel only when the calling thread needs to be blocked because the semaphore is unavailable. If the semaphore is available, no kernel call is made. For an example of how to use dispatch semaphores, see Using Dispatch Semaphores to Regulate the Use of Finite Resources (page 53).
Dispatch sources	A dispatch source generates notifications in response to specific types of system events. You can use dispatch sources to monitor events such as process notifications, signals, and descriptor events among others. When an event occurs, the dispatch source submits your task code asynchronously to the specified dispatch queue for processing. For more information about creating and using dispatch sources, see Dispatch Sources (page 57).

Implementing Tasks Using Blocks

Block objects are a C-based language feature that you can use in your C, Objective-C, and C++ code. Blocks make it easy to define a self-contained unit of work. Although they might seem akin to function pointers, a block is actually represented by an underlying data structure that resembles an object and is created and managed for you by the compiler. The compiler packages up the code you provide (along with any related data) and encapsulates it in a form that can live in the heap and be passed around your application.

One of the key advantages of blocks is their ability to use variables from outside their own lexical scope. When you define a block inside a function or method, the block acts as a traditional code block would in some ways. For example, a block can read the values of variables defined in the parent scope. Variables accessed by the block are copied to the block data structure on the heap so that the block can access them later. When blocks

are added to a dispatch queue, these values must typically be left in a read-only format. However, blocks that are executed synchronously can also use variables that have the `__block` keyword prepended to return data back to the parent's calling scope.

You declare blocks inline with your code using a syntax that is similar to the syntax used for function pointers. The main difference between a block and a function pointer is that the block name is preceded with a caret (^) instead of an asterisk (*). Like a function pointer, you can pass arguments to a block and receive a return value from it. Listing 3-1 shows you how to declare and execute blocks synchronously in your code. The variable `aBlock` is declared to be a block that takes a single integer parameter and returns no value. An actual block matching that prototype is then assigned to `aBlock` and declared inline. The last line executes the block immediately, printing the specified integers to standard out.

Listing 3-1 A simple block example

```
int x = 123;
int y = 456;

// Block declaration and assignment
void (^aBlock)(int) = ^(int z) {
    printf("%d %d %d\n", x, y, z);
};

// Execute the block
aBlock(789);   // prints: 123 456 789
```

The following is a summary of some of the key guidelines you should consider when designing your blocks:

- For blocks that you plan to perform asynchronously using a dispatch queue, it is safe to capture scalar variables from the parent function or method and use them in the block. However, you should not try to capture large structures or other pointer-based variables that are allocated and deleted by the calling context. By the time your block is executed, the memory referenced by that pointer may be gone. Of course, it is safe to allocate memory (or an object) yourself and explicitly hand off ownership of that memory to the block.
- Dispatch queues copy blocks that are added to them, and they release blocks when they finish executing. In other words, you do not need to explicitly copy blocks before adding them to a queue.
- Although queues are more efficient than raw threads at executing small tasks, there is still overhead to creating blocks and executing them on a queue. If a block does too little work, it may be cheaper to execute it inline than dispatch it to a queue. The way to tell if a block is doing too little work is to gather metrics for each path using the performance tools and compare them.

- Do not cache data relative to the underlying thread and expect that data to be accessible from a different block. If tasks in the same queue need to share data, use the context pointer of the dispatch queue to store the data instead. For more information on how to access the context data of a dispatch queue, see Storing Custom Context Information with a Queue (page 47).
- If your block creates more than a few Objective-C objects, you might want to enclose parts of your block's code in an @autorelease block to handle the memory management for those objects. Although GCD dispatch queues have their own autorelease pools, they make no guarantees as to when those pools are drained. If your application is memory constrained, creating your own autorelease pool allows you to free up the memory for autoreleased objects at more regular intervals.

For more information about blocks, including how to declare and use them, see *Blocks Programming Topics*. For information about how you add blocks to a dispatch queue, see Adding Tasks to a Queue (page 48).

Creating and Managing Dispatch Queues

Before you add your tasks to a queue, you have to decide what type of queue to use and how you intend to use it. Dispatch queues can execute tasks either serially or concurrently. In addition, if you have a specific use for the queue in mind, you can configure the queue attributes accordingly. The following sections show you how to create dispatch queues and configure them for use.

Getting the Global Concurrent Dispatch Queues

A concurrent dispatch queue is useful when you have multiple tasks that can run in parallel. A concurrent queue is still a queue in that it dequeues tasks in a first-in, first-out order; however, a concurrent queue may dequeue additional tasks before any previous tasks finish. The actual number of tasks executed by a concurrent queue at any given moment is variable and can change dynamically as conditions in your application change. Many factors affect the number of tasks executed by the concurrent queues, including the number of available cores, the amount of work being done by other processes, and the number and priority of tasks in other serial dispatch queues.

The system provides each application with four concurrent dispatch queues. These queues are global to the application and are differentiated only by their priority level. Because they are global, you do not create them explicitly. Instead, you ask for one of the queues using the `dispatch_get_global_queue` function, as shown in the following example:

```
dispatch_queue_t aQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0);
```

In addition to getting the default concurrent queue, you can also get queues with high- and low-priority levels by passing in the `DISPATCH_QUEUE_PRIORITY_HIGH` and `DISPATCH_QUEUE_PRIORITY_LOW` constants to the function instead, or get a background queue by passing the `DISPATCH_QUEUE_PRIORITY_BACKGROUND` constant. As you might expect, tasks in the high-priority concurrent queue execute before those in the default and low-priority queues. Similarly, tasks in the default queue execute before those in the low-priority queue.

> **Note:** The second argument to the `dispatch_get_global_queue` function is reserved for future expansion. For now, you should always pass `0` for this argument.

Although dispatch queues are reference-counted objects, you do not need to retain and release the global concurrent queues. Because they are global to your application, retain and release calls for these queues are ignored. Therefore, you do not need to store references to these queues. You can just call the `dispatch_get_global_queue` function whenever you need a reference to one of them.

Creating Serial Dispatch Queues

Serial queues are useful when you want your tasks to execute in a specific order. A serial queue executes only one task at a time and always pulls tasks from the head of the queue. You might use a serial queue instead of a lock to protect a shared resource or mutable data structure. Unlike a lock, a serial queue ensures that tasks are executed in a predictable order. And as long as you submit your tasks to a serial queue asynchronously, the queue can never deadlock.

Unlike concurrent queues, which are created for you, you must explicitly create and manage any serial queues you want to use. You can create any number of serial queues for your application but should avoid creating large numbers of serial queues solely as a means to execute as many tasks simultaneously as you can. If you want to execute large numbers of tasks concurrently, submit them to one of the global concurrent queues. When creating serial queues, try to identify a purpose for each queue, such as protecting a resource or synchronizing some key behavior of your application.

Listing 3-2 shows the steps required to create a custom serial queue. The `dispatch_queue_create` function takes two parameters: the queue name and a set of queue attributes. The debugger and performance tools display the queue name to help you track how your tasks are being executed. The queue attributes are reserved for future use and should be `NULL`.

Listing 3-2 Creating a new serial queue

```
dispatch_queue_t queue;
queue = dispatch_queue_create("com.example.MyQueue", NULL);
```

In addition to any custom queues you create, the system automatically creates a serial queue and binds it to your application's main thread. For more information about getting the queue for the main thread, see Getting Common Queues at Runtime (page 46).

Getting Common Queues at Runtime

Grand Central Dispatch provides functions to let you access several common dispatch queues from your application:

- Use the `dispatch_get_current_queue` function for debugging purposes or to test the identity of the current queue. Calling this function from inside a block object returns the queue to which the block was submitted (and on which it is now presumably running). Calling this function from outside of a block returns the default concurrent queue for your application.
- Use the `dispatch_get_main_queue` function to get the serial dispatch queue associated with your application's main thread. This queue is created automatically for Cocoa applications and for applications that either call the `dispatch_main` function or configure a run loop (using either the `CFRunLoopRef` type or an `NSRunLoop` object) on the main thread.
- Use the `dispatch_get_global_queue` function to get any of the shared concurrent queues. For more information, see Getting the Global Concurrent Dispatch Queues (page 44).

Memory Management for Dispatch Queues

Dispatch queues and other dispatch objects are reference-counted data types. When you create a serial dispatch queue, it has an initial reference count of 1. You can use the `dispatch_retain` and `dispatch_release` functions to increment and decrement that reference count as needed. When the reference count of a queue reaches zero, the system asynchronously deallocates the queue.

It is important to retain and release dispatch objects, such as queues, to ensure that they remain in memory while they are being used. As with memory-managed Cocoa objects, the general rule is that if you plan to use a queue that was passed to your code, you should retain the queue before you use it and release it when you no longer need it. This basic pattern ensures that the queue remains in memory for as long as you are using it.

> **Note:** You do not need to retain or release any of the global dispatch queues, including the concurrent dispatch queues or the main dispatch queue. Any attempts to retain or release the queues are ignored.

Even if you implement a garbage-collected application, you must still retain and release your dispatch queues and other dispatch objects. Grand Central Dispatch does not support the garbage collection model for reclaiming memory.

Storing Custom Context Information with a Queue

All dispatch objects (including dispatch queues) allow you to associate custom context data with the object. To set and get this data on a given object, you use the `dispatch_set_context` and `dispatch_get_context` functions. The system does not use your custom data in any way, and it is up to you to both allocate and deallocate the data at the appropriate times.

For queues, you can use context data to store a pointer to an Objective-C object or other data structure that helps identify the queue or its intended usage to your code. You can use the queue's finalizer function to deallocate (or disassociate) your context data from the queue before it is deallocated. An example of how to write a finalizer function that clears a queue's context data is shown in Listing 3-3 (page 47).

Providing a Clean Up Function For a Queue

After you create a serial dispatch queue, you can attach a finalizer function to perform any custom clean up when the queue is deallocated. Dispatch queues are reference counted objects and you can use the `dispatch_set_finalizer_f` function to specify a function to be executed when the reference count of your queue reaches zero. You use this function to clean up the context data associated with a queue and the function is called only if the context pointer is not `NULL`.

Listing 3-3 shows a custom finalizer function and a function that creates a queue and installs that finalizer. The queue uses the finalizer function to release the data stored in the queue's context pointer. (The `myInitializeDataContextFunction` and `myCleanUpDataContextFunction` functions referenced from the code are custom functions that you would provide to initialize and clean up the contents of the data structure itself.) The context pointer passed to the finalizer function contains the data object associated with the queue.

Listing 3-3 Installing a queue clean up function

```
void myFinalizerFunction(void *context)
{
    MyDataContext* theData = (MyDataContext*)context;
```

```
    // Clean up the contents of the structure
    myCleanUpDataContextFunction(theData);

    // Now release the structure itself.
    free(theData);
}

dispatch_queue_t createMyQueue()
{
    MyDataContext*  data = (MyDataContext*) malloc(sizeof(MyDataContext));
    myInitializeDataContextFunction(data);

    // Create the queue and set the context data.
    dispatch_queue_t serialQueue =
dispatch_queue_create("com.example.CriticalTaskQueue", NULL);
    if (serialQueue)
    {
        dispatch_set_context(serialQueue, data);
        dispatch_set_finalizer_f(serialQueue, &myFinalizerFunction);
    }

    return serialQueue;
}
```

Adding Tasks to a Queue

To execute a task, you must dispatch it to an appropriate dispatch queue. You can dispatch tasks synchronously or asynchronously, and you can dispatch them singly or in groups. Once in a queue, the queue becomes responsible for executing your tasks as soon as possible, given its constraints and the existing tasks already in the queue. This section shows you some of the techniques for dispatching tasks to a queue and describes the advantages of each.

Adding a Single Task to a Queue

There are two ways to add a task to a queue: asynchronously or synchronously. When possible, asynchronous execution using the `dispatch_async` and `dispatch_async_f` functions is preferred over the synchronous alternative. When you add a block object or function to a queue, there is no way to know when that code will execute. As a result, adding blocks or functions asynchronously lets you schedule the execution of the code and continue to do other work from the calling thread. This is especially important if you are scheduling the task from your application's main thread—perhaps in response to some user event.

Although you should add tasks asynchronously whenever possible, there may still be times when you need to add a task synchronously to prevent race conditions or other synchronization errors. In these instances, you can use the `dispatch_sync` and `dispatch_sync_f` functions to add the task to the queue. These functions block the current thread of execution until the specified task finishes executing.

Important: You should never call the `dispatch_sync` or `dispatch_sync_f` function from a task that is executing in the same queue that you are planning to pass to the function. This is particularly important for serial queues, which are guaranteed to deadlock, but should also be avoided for concurrent queues.

The following example shows how to use the block-based variants for dispatching tasks asynchronously and synchronously:

```
dispatch_queue_t myCustomQueue;
myCustomQueue = dispatch_queue_create("com.example.MyCustomQueue", NULL);

dispatch_async(myCustomQueue, ^{
    printf("Do some work here.\n");
});

printf("The first block may or may not have run.\n");

dispatch_sync(myCustomQueue, ^{
    printf("Do some more work here.\n");
});
printf("Both blocks have completed.\n");
```

Performing a Completion Block When a Task Is Done

By their nature, tasks dispatched to a queue run independently of the code that created them. However, when the task is done, your application might still want to be notified of that fact so that it can incorporate the results. With traditional asynchronous programming, you might do this using a callback mechanism, but with dispatch queues you can use a completion block.

A completion block is just another piece of code that you dispatch to a queue at the end of your original task. The calling code typically provides the completion block as a parameter when it starts the task. All the task code has to do is submit the specified block or function to the specified queue when it finishes its work.

Listing 3-4 shows an averaging function implemented using blocks. The last two parameters to the averaging function allow the caller to specify a queue and block to use when reporting the results. After the averaging function computes its value, it passes the results to the specified block and dispatches it to the queue. To prevent the queue from being released prematurely, it is critical to retain that queue initially and release it once the completion block has been dispatched.

Listing 3-4 Executing a completion callback after a task

```
void average_async(int *data, size_t len,
   dispatch_queue_t queue, void (^block)(int))
{
   // Retain the queue provided by the user to make
   // sure it does not disappear before the completion
   // block can be called.
   dispatch_retain(queue);

   // Do the work on the default concurrent queue and then
   // call the user-provided block with the results.
   dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
^{
      int avg = average(data, len);
      dispatch_async(queue, ^{ block(avg);});

      // Release the user-provided queue when done
      dispatch_release(queue);
   });
}
```

Performing Loop Iterations Concurrently

One place where concurrent dispatch queues might improve performance is in places where you have a loop that performs a fixed number of iterations. For example, suppose you have a `for` loop that does some work through each loop iteration:

```
for (i = 0; i < count; i++) {
   printf("%u\n",i);
}
```

If the work performed during each iteration is distinct from the work performed during all other iterations, and the order in which each successive loop finishes is unimportant, you can replace the loop with a call to the `dispatch_apply` or `dispatch_apply_f` function. These functions submit the specified block or function to a queue once for each loop iteration. When dispatched to a concurrent queue, it is therefore possible to perform multiple loop iterations at the same time.

You can specify either a serial queue or a concurrent queue when calling `dispatch_apply` or `dispatch_apply_f`. Passing in a concurrent queue allows you to perform multiple loop iterations simultaneously and is the most common way to use these functions. Although using a serial queue is permissible and does the right thing for your code, using such a queue has no real performance advantages over leaving the loop in place.

Important: Like a regular `for` loop, the `dispatch_apply` and `dispatch_apply_f` functions do not return until all loop iterations are complete. You should therefore be careful when calling them from code that is already executing from the context of a queue. If the queue you pass as a parameter to the function is a serial queue and is the same one executing the current code, calling these functions will deadlock the queue.

Because they effectively block the current thread, you should also be careful when calling these functions from your main thread, where they could prevent your event handling loop from responding to events in a timely manner. If your loop code requires a noticeable amount of processing time, you might want to call these functions from a different thread.

Listing 3-5 shows how to replace the preceding `for` loop with the `dispatch_apply` syntax. The block you pass in to the `dispatch_apply` function must contain a single parameter that identifies the current loop iteration. When the block is executed, the value of this parameter is `0` for the first iteration, `1` for the second, and so on. The value of the parameter for the last iteration is `count - 1`, where `count` is the total number of iterations.

Listing 3-5 Performing the iterations of a `for` loop concurrently

```
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0);


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

You should make sure that your task code does a reasonable amount of work through each iteration. As with any block or function you dispatch to a queue, there is overhead to scheduling that code for execution. If each iteration of your loop performs only a small amount of work, the overhead of scheduling the code may outweigh the performance benefits you might achieve from dispatching it to a queue. If you find this is true during your testing, you can use striding to increase the amount of work performed during each loop iteration. With striding, you group together multiple iterations of your original loop into a single block and reduce the iteration count proportionately. For example, if you perform 100 iterations initially but decide to use a stride of 4, you now perform 4 loop iterations from each block and your iteration count is 25. For an example of how to implement striding, see Improving on Loop Code (page 78).

Performing Tasks on the Main Thread

Grand Central Dispatch provides a special dispatch queue that you can use to execute tasks on your application's main thread. This queue is provided automatically for all applications and is drained automatically by any application that sets up a run loop (managed by either a `CFRunLoopRef` type or `NSRunLoop` object) on its main thread. If you are not creating a Cocoa application and do not want to set up a run loop explicitly, you must call the `dispatch_main` function to drain the main dispatch queue explicitly. You can still add tasks to the queue, but if you do not call this function those tasks are never executed.

You can get the dispatch queue for your application's main thread by calling the `dispatch_get_main_queue` function. Tasks added to this queue are performed serially on the main thread itself. Therefore, you can use this queue as a synchronization point for work being done in other parts of your application.

Using Objective-C Objects in Your Tasks

GCD provides built-in support for Cocoa memory management techniques so you may freely use Objective-C objects in the blocks you submit to dispatch queues. Each dispatch queue maintains its own autorelease pool to ensure that autoreleased objects are released at some point; queues make no guarantee about when they actually release those objects.

If your application is memory constrained and your block creates more than a few autoreleased objects, creating your own autorelease pool is the only way to ensure that your objects are released in a timely manner. If your block creates hundreds of objects, you might want to create more than one autorelease pool or drain your pool at regular intervals.

For more information about autorelease pools and Objective-C memory management, see *Advanced Memory Management Programming Guide*.

Suspending and Resuming Queues

You can prevent a queue from executing block objects temporarily by suspending it. You suspend a dispatch queue using the `dispatch_suspend` function and resume it using the `dispatch_resume` function. Calling `dispatch_suspend` increments the queue's suspension reference count, and calling `dispatch_resume` decrements the reference count. While the reference count is greater than zero, the queue remains suspended. Therefore, you must balance all suspend calls with a matching resume call in order to resume processing blocks.

Important: Suspend and resume calls are asynchronous and take effect only between the execution of blocks. Suspending a queue does not cause an already executing block to stop.

Using Dispatch Semaphores to Regulate the Use of Finite Resources

If the tasks you are submitting to dispatch queues access some finite resource, you may want to use a dispatch semaphore to regulate the number of tasks simultaneously accessing that resource. A dispatch semaphore works like a regular semaphore with one exception. When resources are available, it takes less time to acquire a dispatch semaphore than it does to acquire a traditional system semaphore. This is because Grand Central Dispatch does not call down into the kernel for this particular case. The only time it calls down into the kernel is when the resource is not available and the system needs to park your thread until the semaphore is signaled.

The semantics for using a dispatch semaphore are as follows:

1. When you create the semaphore (using the `dispatch_semaphore_create` function), you can specify a positive integer indicating the number of resources available.
2. In each task, call `dispatch_semaphore_wait` to wait on the semaphore.
3. When the wait call returns, acquire the resource and do your work.
4. When you are done with the resource, release it and signal the semaphore by calling the `dispatch_semaphore_signal` function.

For an example of how these steps work, consider the use of file descriptors on the system. Each application is given a limited number of file descriptors to use. If you have a task that processes large numbers of files, you do not want to open so many files at one time that you run out of file descriptors. Instead, you can use a semaphore to limit the number of file descriptors in use at any one time by your file-processing code. The basic pieces of code you would incorporate into your tasks is as follows:

```
// Create the semaphore, specifying the initial pool size
dispatch_semaphore_t fd_sema = dispatch_semaphore_create(getdtablesize() / 2);

// Wait for a free file descriptor
dispatch_semaphore_wait(fd_sema, DISPATCH_TIME_FOREVER);
fd = open("/etc/services", O_RDONLY);

// Release the file descriptor when done
close(fd);
dispatch_semaphore_signal(fd_sema);
```

When you create the semaphore, you specify the number of available resources. This value becomes the initial count variable for the semaphore. Each time you wait on the semaphore, the `dispatch_semaphore_wait` function decrements that count variable by 1. If the resulting value is negative, the function tells the kernel to block your thread. On the other end, the `dispatch_semaphore_signal` function increments the count variable by 1 to indicate that a resource has been freed up. If there are tasks blocked and waiting for a resource, one of them is subsequently unblocked and allowed to do its work.

Waiting on Groups of Queued Tasks

Dispatch groups are a way to block a thread until one or more tasks finish executing. You can use this behavior in places where you cannot make progress until all of the specified tasks are complete. For example, after dispatching several tasks to compute some data, you might use a group to wait on those tasks and then process the results when they are done. Another way to use dispatch groups is as an alternative to thread joins. Instead of starting several child threads and then joining with each of them, you could add the corresponding tasks to a dispatch group and wait on the entire group.

Listing 3-6 shows the basic process for setting up a group, dispatching tasks to it, and waiting on the results. Instead of dispatching tasks to a queue using the `dispatch_async` function, you use the `dispatch_group_async` function instead. This function associates the task with the group and queues it for execution. To wait on a group of tasks to finish, you then use the `dispatch_group_wait` function, passing in the appropriate group.

Listing 3-6 Waiting on asynchronous tasks

```
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0);
dispatch_group_t group = dispatch_group_create();

// Add a task to the group
dispatch_group_async(group, queue, ^{
   // Some asynchronous work
});

// Do some other work while the tasks execute.

// When you cannot make any more forward progress,
// wait on the group to block the current thread.
dispatch_group_wait(group, DISPATCH_TIME_FOREVER);

// Release the group when it is no longer needed.
dispatch_release(group);
```

Dispatch Queues and Thread Safety

It might seem odd to talk about thread safety in the context of dispatch queues, but thread safety is still a relevant topic. Any time you are implementing concurrency in your application, there are a few things you should know:

- Dispatch queues themselves are thread safe. In other words, you can submit tasks to a dispatch queue from any thread on the system without first taking a lock or synchronizing access to the queue.
- Do not call the `dispatch_sync` function from a task that is executing on the same queue that you pass to your function call. Doing so will deadlock the queue. If you need to dispatch to the current queue, do so asynchronously using the `dispatch_async` function.

- Avoid taking locks from the tasks you submit to a dispatch queue. Although it is safe to use locks from your tasks, when you acquire the lock, you risk blocking a serial queue entirely if that lock is unavailable. Similarly, for concurrent queues, waiting on a lock might prevent other tasks from executing instead. If you need to synchronize parts of your code, use a serial dispatch queue instead of a lock.
- Although you can obtain information about the underlying thread running a task, it is better to avoid doing so. For more information about the compatibility of dispatch queues with threads, see Compatibility with POSIX Threads (page 83).

For additional tips on how to change your existing threaded code to use dispatch queues, see Migrating Away from Threads (page 75).

Dispatch Sources

Whenever you interact with the underlying system, you must be prepared for that task to take a nontrivial amount of time. Calling down to the kernel or other system layers involves a change in context that is reasonably expensive compared to calls that occur within your own process. As a result, many system libraries provide asynchronous interfaces to allow your code to submit a request to the system and continue to do other work while that request is processed. Grand Central Dispatch builds on this general behavior by allowing you to submit your request and have the results reported back to your code using blocks and dispatch queues.

About Dispatch Sources

A *dispatch source* is a fundamental data type that coordinates the processing of specific low-level system events. Grand Central Dispatch supports the following types of dispatch sources:

- *Timer dispatch sources* generate periodic notifications.
- *Signal dispatch sources* notify you when a UNIX signal arrives.
- *Descriptor sources* notify you of various file- and socket-based operations, such as:
    - When data is available for reading
    - When it is possible to write data
    - When files are deleted, moved, or renamed in the file system
    - When file meta information changes
- *Process dispatch sources* notify you of process-related events, such as:
    - When a process exits
    - When a process issues a `fork` or `exec` type of call
    - When a signal is delivered to the process
- *Mach port dispatch sources* notify you of Mach-related events.
- *Custom dispatch sources* are ones you define and trigger yourself.

Dispatch sources replace the asynchronous callback functions that are typically used to process system-related events. When you configure a dispatch source, you specify the events you want to monitor and the dispatch queue and code to use to process those events. You can specify your code using block objects or functions. When an event of interest arrives, the dispatch source submits your block or function to the specified dispatch queue for execution.

Unlike tasks that you submit to a queue manually, dispatch sources provide a continuous source of events for your application. A dispatch source remains attached to its dispatch queue until you cancel it explicitly. While attached, it submits its associated task code to the dispatch queue whenever the corresponding event occurs. Some events, such as timer events, occur at regular intervals but most occur only sporadically as specific conditions arise. For this reason, dispatch sources retain their associated dispatch queue to prevent it from being released prematurely while events may still be pending.

To prevent events from becoming backlogged in a dispatch queue, dispatch sources implement an event coalescing scheme. If a new event arrives before the event handler for a previous event has been dequeued and executed, the dispatch source coalesces the data from the new event data with data from the old event. Depending on the type of event, coalescing may replace the old event or update the information it holds. For example, a signal-based dispatch source provides information about only the most recent signal but also reports how many total signals have been delivered since the last invocation of the event handler.

Creating Dispatch Sources

Creating a dispatch source involves creating both the source of the events and the dispatch source itself. The source of the events is whatever native data structures are required to process the events. For example, for a descriptor-based dispatch source you would need to open the descriptor and for a process-based source you would need to obtain the process ID of the target program. When you have your event source, you can then create the corresponding dispatch source as follows:

1. Create the dispatch source using the `dispatch_source_create` function.
2. Configure the dispatch source:
    - Assign an event handler to the dispatch source; see Writing and Installing an Event Handler (page 59).
    - For timer sources, set the timer information using the `dispatch_source_set_timer` function; see Creating a Timer (page 63).
3. Optionally assign a cancellation handler to the dispatch source; see Installing a Cancellation Handler (page 61).
4. Call the `dispatch_resume` function to start processing events; see Suspending and Resuming Dispatch Sources (page 74).

Because dispatch sources require some additional configuration before they can be used, the `dispatch_source_create` function returns dispatch sources in a suspended state. While suspended, a dispatch source receives events but does not process them. This gives you time to install an event handler and perform any additional configuration needed to process the actual events.

The following sections show you how to configure various aspects of a dispatch source. For detailed examples showing you how to configure specific types of dispatch sources, see Dispatch Source Examples (page 63). For additional information about the functions you use to create and configure dispatch sources, see *Grand Central Dispatch (GCD) Reference*.

Writing and Installing an Event Handler

To handle the events generated by a dispatch source, you must define an event handler to process those events. An event handler is a function or block object that you install on your dispatch source using the `dispatch_source_set_event_handler` or `dispatch_source_set_event_handler_f` function. When an event arrives, the dispatch source submits your event handler to the designated dispatch queue for processing.

The body of your event handler is responsible for processing any events that arrive. If your event handler is already queued and waiting to process an event when a new event arrives, the dispatch source coalesces the two events. An event handler generally sees information only for the most recent event, but depending on the type of the dispatch source it may also be able to get information about other events that occurred and were coalesced. If one or more new events arrive after the event handler has begun executing, the dispatch source holds onto those events until the current event handler has finished executing. At that point, it submits the event handler to the queue again with the new events.

Function-based event handlers take a single context pointer, containing the dispatch source object, and return no value. Block-based event handlers take no parameters and have no return value.

```
// Block-based event handler
void (^dispatch_block_t)(void)

// Function-based event handler
void (*dispatch_function_t)(void *)
```

Inside your event handler, you can get information about the given event from the dispatch source itself. Although function-based event handlers are passed a pointer to the dispatch source as a parameter, block-based event handlers must capture that pointer themselves. You can do this for your blocks by referencing the variable containing the dispatch source normally. For example, the following code snippet captures the `source` variable, which is declared outside the scope of the block.

```
dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,
                                  myDescriptor, 0, myQueue);
dispatch_source_set_event_handler(source, ^{
   // Get some data from the source variable, which is captured
   // from the parent context.
   size_t estimated = dispatch_source_get_data(source);

   // Continue reading the descriptor...
});
dispatch_resume(source);
```

Capturing variables inside of a block is commonly done to allow for greater flexibility and dynamism. Of course, captured variables are read-only within the block by default. Although the blocks feature provides support for modifying captured variables under specific circumstances, you should not attempt to do so in the event handlers associated with a dispatch source. Dispatch sources always execute their event handlers asynchronously, so the defining scope of any variables you captured is likely gone by the time your event handler executes. For more information about how to capture and use variables inside of blocks, see *Blocks Programming Topics* .

Table 4-1 lists the functions you can call from your event handler code to obtain information about an event.

Table 4-1 Getting data from a dispatch source

Function	Description
`dispatch_source_get_handle`	This function returns the underlying system data type that the dispatch source manages. For a descriptor dispatch source, this function returns an `int` type containing the descriptor associated with the dispatch source. For a signal dispatch source, this function returns an `int` type containing the signal number for the most recent event. For a process dispatch source, this function returns a `pid_t` data structure for the process being monitored. For a Mach port dispatch source, this function returns a `mach_port_t` data structure. For other dispatch sources, the value returned by this function is undefined.

Function	Description
`dispatch_source_get_data`	This function returns any pending data associated with the event. For a descriptor dispatch source that reads data from a file, this function returns the number of bytes available for reading. For a descriptor dispatch source that writes data to a file, this function returns a positive integer if space is available for writing. For a descriptor dispatch source that monitors file system activity, this function returns a constant indicating the type of event that occurred. For a list of constants, see the `dispatch_source_vnode_flags_t` enumerated type. For a process dispatch source, this function returns a constant indicating the type of event that occurred. For a list of constants, see the `dispatch_source_proc_flags_t` enumerated type. For a Mach port dispatch source, this function returns a constant indicating the type of event that occurred. For a list of constants, see the `dispatch_source_machport_flags_t` enumerated type. For a custom dispatch source, this function returns the new data value created from the existing data and the new data passed to the `dispatch_source_merge_data` function.
`dispatch_source_get_mask`	This function returns the event flags that were used to create the dispatch source. For a process dispatch source, this function returns a mask of the events that the dispatch source receives. For a list of constants, see the `dispatch_source_proc_flags_t` enumerated type. For a Mach port dispatch source with send rights, this function returns a mask of the desired events. For a list of constants, see the `dispatch_source_mach_send_flags_t` enumerated type. For a custom OR dispatch source, this function returns the mask used to merge the data values.

For some examples of how to write and install event handlers for specific types of dispatch sources, see Dispatch Source Examples (page 63).

Installing a Cancellation Handler

Cancellation handlers are used to clean up a dispatch source before it is released. For most types of dispatch sources, cancellation handlers are optional and only necessary if you have some custom behaviors tied to the dispatch source that also need to be updated. For dispatch sources that use a descriptor or Mach port, however,

you must provide a cancellation handler to close the descriptor or release the Mach port. Failure to do so can lead to subtle bugs in your code resulting from those structures being reused unintentionally by your code or other parts of the system.

You can install a cancellation handler at any time but usually you would do so when creating the dispatch source. You install a cancellation handler using the `dispatch_source_set_cancel_handler` or `dispatch_source_set_cancel_handler_f` function, depending on whether you want to use a block object or a function in your implementation. The following example shows a simple cancellation handler that closes a descriptor that was opened for a dispatch source. The `fd` variable is a captured variable containing the descriptor.

```
dispatch_source_set_cancel_handler(mySource, ^{
    close(fd); // Close a file descriptor opened earlier.
});
```

To see a complete code example for a dispatch source that uses a cancellation handler, see Reading Data from a Descriptor (page 65).

Changing the Target Queue

Although you specify the queue on which to run your event and cancellation handlers when you create a dispatch source, you can change that queue at any time using the `dispatch_set_target_queue` function. You might do this to change the priority at which the dispatch source's events are processed.

Changing a dispatch source's queue is an asynchronous operation and the dispatch source does its best to make the change as quickly as possible. If an event handler is already queued and waiting to be processed, it executes on the previous queue. However, other events arriving around the time you make the change could be processed on either queue.

Associating Custom Data with a Dispatch Source

Like many other data types in Grand Central Dispatch, you can use the `dispatch_set_context` function to associate custom data with a dispatch source. You can use the context pointer to store any data your event handler needs to process events. If you do store any custom data in the context pointer, you should also install a cancellation handler (as described in Installing a Cancellation Handler (page 61)) to release that data when the dispatch source is no longer needed.

If you implement your event handler using blocks, you can also capture local variables and use them within your block-based code. Although this might alleviate the need to store data in the context pointer of the dispatch source, you should always use this feature judiciously. Because dispatch sources may be long-lived

in your application, you should be careful when capturing variables containing pointers. If the data pointed to by a pointer could be deallocated at any time, you should either copy the data or retain it to prevent that from happening. In either case, you would then need to provide a cancellation handler to release the data later.

Memory Management for Dispatch Sources

Like other dispatch objects, dispatch sources are reference counted data types. A dispatch source has an initial reference count of 1 and can be retained and released using the `dispatch_retain` and `dispatch_release` functions. When the reference count of a queue reaches zero, the system automatically deallocates the dispatch source data structures.

Because of the way they are used, the ownership of dispatch sources can be managed either internally or externally to the dispatch source itself. With external ownership, another object or piece of code takes ownership of the dispatch source and is responsible for releasing it when it is no longer needed. With internal ownership, the dispatch source owns itself and is responsible for releasing itself at the appropriate time. Although external ownership is very common, you might use internal ownership in cases where you want to create an autonomous dispatch source and let it manage some behavior of your code without any further interactions. For example, if a dispatch source is designed to respond to a single global event, you might have it handle that event and then exit immediately.

Dispatch Source Examples

The following sections show you how to create and configure some of the more commonly used dispatch sources. For more information about configuring specific types of dispatch sources, see *Grand Central Dispatch (GCD) Reference*.

Creating a Timer

Timer dispatch sources generate events at regular, time-based intervals. You can use timers to initiate specific tasks that need to be performed regularly. For example, games and other graphics-intensive applications might use timers to initiate screen or animation updates. You could also set up a timer and use the resulting events to check for new information on a frequently updated server.

All timer dispatch sources are interval timers—that is, once created, they deliver regular events at the interval you specify. When you create a timer dispatch source, one of the values you must specify is a leeway value to give the system some idea of the desired accuracy for timer events. Leeway values give the system some flexibility in how it manages power and wakes up cores. For example, the system might use the leeway value to advance or delay the fire time and align it better with other system events. You should therefore specify a leeway value whenever possible for your own timers.

> **Note:** Even if you specify a leeway value of 0, you should never expect a timer to fire at the exact nanosecond you requested. The system does its best to accommodate your needs but cannot guarantee exact firing times.

When a computer goes to sleep, all timer dispatch sources are suspended. When the computer wakes up, those timer dispatch sources are automatically woken up as well. Depending on the configuration of the timer, pauses of this nature may affect when your timer fires next. If you set up your timer dispatch source using the `dispatch_time` function or the `DISPATCH_TIME_NOW` constant, the timer dispatch source uses the default system clock to determine when to fire. However, the default clock does not advance while the computer is asleep. By contrast, when you set up your timer dispatch source using the `dispatch_walltime` function, the timer dispatch source tracks its firing time to the wall clock time. This latter option is typically appropriate for timers whose firing interval is relatively large because it prevents there from being too much drift between event times.

Listing 4-1 shows an example of a timer that fires once every 30 seconds and has a leeway value of 1 second. Because the timer interval is relatively large, the dispatch source is created using the `dispatch_walltime` function. The first firing of the timer occurs immediately and subsequent events arrive every 30 seconds. The `MyPeriodicTask` and `MyStoreTimer` symbols represent custom functions that you would write to implement the timer behavior and to store the timer somewhere in your application's data structures.

Listing 4-1 Creating a timer dispatch source

```
dispatch_source_t CreateDispatchTimer(uint64_t interval,
              uint64_t leeway,
              dispatch_queue_t queue,
              dispatch_block_t block)
{
   dispatch_source_t timer = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER,
                                                    0, 0, queue);
   if (timer)
   {
      dispatch_source_set_timer(timer, dispatch_walltime(NULL, 0), interval,
leeway);
      dispatch_source_set_event_handler(timer, block);
      dispatch_resume(timer);
   }
   return timer;
```

```
}

void MyCreateTimer()
{
   dispatch_source_t aTimer = CreateDispatchTimer(30ull * NSEC_PER_SEC,
                                1ull * NSEC_PER_SEC,
                                dispatch_get_main_queue(),
                                ^{ MyPeriodicTask(); });

   // Store it somewhere for later use.
    if (aTimer)
    {
        MyStoreTimer(aTimer);
    }
}
```

Although creating a timer dispatch source is the main way to receive time-based events, there are other options available as well. If you want to perform a block once after a specified time interval, you can use the `dispatch_after` or `dispatch_after_f` function. This function behaves much like the `dispatch_async` function except that it allows you to specify a time value at which to submit the block to a queue. The time value can be specified as a relative or absolute time value depending on your needs.

Reading Data from a Descriptor

To read data from a file or socket, you must open the file or socket and create a dispatch source of type `DISPATCH_SOURCE_TYPE_READ`. The event handler you specify should be capable of reading and processing the contents of the file descriptor. In the case of a file, this amounts to reading the file data (or a subset of that data) and creating the appropriate data structures for your application. For a network socket, this involves processing newly received network data.

Whenever reading data, you should always configure your descriptor to use non-blocking operations. Although you can use the `dispatch_source_get_data` function to see how much data is available for reading, the number returned by that function could change between the time you make the call and the time you actually read the data. If the underlying file is truncated or a network error occurs, reading from a descriptor that blocks the current thread could stall your event handler in mid execution and prevent the dispatch queue from dispatching other tasks. For a serial queue, this could deadlock your queue, and even for a concurrent queue this reduces the number of new tasks that can be started.

Listing 4-2 shows an example that configures a dispatch source to read data from a file. In this example, the event handler reads the entire contents of the specified file into a buffer and calls a custom function (that you would define in your own code) to process the data. (The caller of this function would use the returned dispatch source to cancel it once the read operation was completed.) To ensure that the dispatch queue does not block unnecessarily when there is no data to read, this example uses the `fcntl` function to configure the file descriptor to perform nonblocking operations. The cancellation handler installed on the dispatch source ensures that the file descriptor is closed after the data is read.

Listing 4-2 Reading data from a file

```
dispatch_source_t ProcessContentsOfFile(const char* filename)
{
   // Prepare the file for reading.
   int fd = open(filename, O_RDONLY);
   if (fd == -1)
      return NULL;
   fcntl(fd, F_SETFL, O_NONBLOCK);  // Avoid blocking the read operation

   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
   dispatch_source_t readSource = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,
                                    fd, 0, queue);
   if (!readSource)
   {
      close(fd);
      return NULL;
   }

   // Install the event handler
   dispatch_source_set_event_handler(readSource, ^{
      size_t estimated = dispatch_source_get_data(readSource) + 1;
      // Read the data into a text buffer.
      char* buffer = (char*)malloc(estimated);
      if (buffer)
      {
         ssize_t actual = read(fd, buffer, (estimated));
         Boolean done = MyProcessFileData(buffer, actual);  // Process the data.
```

```
        // Release the buffer when done.
        free(buffer);

        // If there is no more data, cancel the source.
        if (done)
           dispatch_source_cancel(readSource);
      }
    });

   // Install the cancellation handler
   dispatch_source_set_cancel_handler(readSource, ^{close(fd);});

   // Start reading the file.
   dispatch_resume(readSource);
   return readSource;
}
```

In the preceding example, the custom `MyProcessFileData` function determines when enough file data has been read and the dispatch source can be canceled. By default, a dispatch source configured for reading from a descriptor schedules its event handler repeatedly while there is still data to read. If the socket connection closes or you reach the end of a file, the dispatch source automatically stops scheduling the event handler. If you know you do not need a dispatch source though, you can cancel it directly yourself.

Writing Data to a Descriptor

The process for writing data to a file or socket is very similar to the process for reading data. After configuring a descriptor for write operations, you create a dispatch source of type `DISPATCH_SOURCE_TYPE_WRITE`. Once that dispatch source is created, the system calls your event handler to give it a chance to start writing data to the file or socket. When you are finished writing data, use the `dispatch_source_cancel` function to cancel the dispatch source.

Whenever writing data, you should always configure your file descriptor to use non-blocking operations. Although you can use the `dispatch_source_get_data` function to see how much space is available for writing, the value returned by that function is advisory only and could change between the time you make the call and the time you actually write the data. If an error occurs, writing data to a blocking file descriptor

could stall your event handler in mid execution and prevent the dispatch queue from dispatching other tasks. For a serial queue, this could deadlock your queue, and even for a concurrent queue this reduces the number of new tasks that can be started.

Listing 4-3 shows the basic approach for writing data to a file using a dispatch source. After creating the new file, this function passes the resulting file descriptor to its event handler. The data being put into the file is provided by the `MyGetData` function, which you would replace with whatever code you needed to generate the data for the file. After writing the data to the file, the event handler cancels the dispatch source to prevent it from being called again. The owner of the dispatch source would then be responsible for releasing it.

Listing 4-3 Writing data to a file

```
dispatch_source_t WriteDataToFile(const char* filename)
{
    int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
                      (S_IRUSR | S_IWUSR | S_ISUID | S_ISGID));
    if (fd == -1)
        return NULL;
    fcntl(fd, F_SETFL); // Block during the write.

    dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_source_t writeSource =
dispatch_source_create(DISPATCH_SOURCE_TYPE_WRITE,
                           fd, 0, queue);
    if (!writeSource)
    {
        close(fd);
        return NULL;
    }

    dispatch_source_set_event_handler(writeSource, ^{
        size_t bufferSize = MyGetDataSize();
        void* buffer = malloc(bufferSize);

        size_t actual = MyGetData(buffer, bufferSize);
        write(fd, buffer, actual);
```

```
        free(buffer);

        // Cancel and release the dispatch source when done.
        dispatch_source_cancel(writeSource);
    });

    dispatch_source_set_cancel_handler(writeSource, ^{close(fd);});
    dispatch_resume(writeSource);
    return (writeSource);
}
```

Monitoring a File-System Object

If you want to monitor a file system object for changes, you can set up a dispatch source of type `DISPATCH_SOURCE_TYPE_VNODE`. You can use this type of dispatch source to receive notifications when a file is deleted, written to, or renamed. You can also use it to be alerted when specific types of meta information for a file (such as its size and link count) change.

> **Note:** The file descriptor you specify for your dispatch source must remain open while the source itself is processing events.

Listing 4-4 shows an example that monitors a file for name changes and performs some custom behavior when it does. (You would provide the actual behavior in place of the `MyUpdateFileName` function called in the example.) Because a descriptor is opened specifically for the dispatch source, the dispatch source includes a cancellation handler that closes the descriptor. Because the file descriptor created by the example is associated with the underlying file-system object, this same dispatch source can be used to detect any number of filename changes.

Listing 4-4 Watching for filename changes

```
dispatch_source_t MonitorNameChangesToFile(const char* filename)
{
   int fd = open(filename, O_EVTONLY);
   if (fd == -1)
      return NULL;
```

```
   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
   dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_VNODE,
               fd, DISPATCH_VNODE_RENAME, queue);
   if (source)
   {
      // Copy the filename for later use.
      int length = strlen(filename);
      char* newString = (char*)malloc(length + 1);
      newString = strcpy(newString, filename);
      dispatch_set_context(source, newString);

      // Install the event handler to process the name change
      dispatch_source_set_event_handler(source, ^{
            const char*  oldFilename = (char*)dispatch_get_context(source);
            MyUpdateFileName(oldFilename, fd);
      });

      // Install a cancellation handler to free the descriptor
      // and the stored string.
      dispatch_source_set_cancel_handler(source, ^{
          char* fileStr = (char*)dispatch_get_context(source);
          free(fileStr);
          close(fd);
      });

      // Start processing events.
      dispatch_resume(source);
   }
   else
      close(fd);

   return source;
}
```

Monitoring Signals

UNIX signals allow the manipulation of an application from outside of its domain. An application can receive many different types of signals ranging from unrecoverable errors (such as illegal instructions) to notifications about important information (such as when a child process exits). Traditionally, applications use the `sigaction` function to install a signal handler function, which processes signals synchronously as soon as they arrive. If you just want to be notified of a signal's arrival and do not actually want to handle the signal, you can use a signal dispatch source to process the signals asynchronously.

Signal dispatch sources are not a replacement for the synchronous signal handlers you install using the `sigaction` function. Synchronous signal handlers can actually catch a signal and prevent it from terminating your application. Signal dispatch sources allow you to monitor only the arrival of the signal. In addition, you cannot use signal dispatch sources to retrieve all types of signals. Specifically, you cannot use them to monitor the `SIGILL`, `SIGBUS`, and `SIGSEGV` signals.

Because signal dispatch sources are executed asynchronously on a dispatch queue, they do not suffer from some of the same limitations as synchronous signal handlers. For example, there are no restrictions on the functions you can call from your signal dispatch source's event handler. The tradeoff for this increased flexibility is the fact that there may be some increased latency between the time a signal arrives and the time your dispatch source's event handler is called.

Listing 4-5 shows how you configure a signal dispatch source to handle the `SIGHUP` signal. The event handler for the dispatch source calls the `MyProcessSIGHUP` function, which you would replace in your application with code to process the signal.

Listing 4-5 Installing a block to monitor signals

```
void InstallSignalHandler()
{
   // Make sure the signal does not terminate the application.
   signal(SIGHUP, SIG_IGN);

   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
   dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_SIGNAL,
 SIGHUP, 0, queue);

   if (source)
   {
      dispatch_source_set_event_handler(source, ^{
         MyProcessSIGHUP();
```

```
        });

        // Start processing signals
        dispatch_resume(source);
    }
}
```

If you are developing code for a custom framework, an advantage of using signal dispatch sources is that your code can monitor signals independent of any applications linked to it. Signal dispatch sources do not interfere with other dispatch sources or any synchronous signal handlers the application might have installed.

For more information about implementing synchronous signal handlers, and for a list of signal names, see `signal` man page.

Monitoring a Process

A process dispatch source lets you monitor the behavior of a specific process and respond appropriately. A parent process might use this type of dispatch source to monitor any child processes it creates. For example, the parent process could use it to watch for the death of a child process. Similarly, a child process could use it to monitor its parent process and exit if the parent process exits.

Listing 4-6 shows the steps for installing a dispatch source to monitor for the termination of a parent process. When the parent process dies, the dispatch source sets some internal state information to let the child process know it should exit. (Your own application would need to implement the `MySetAppExitFlag` function to set an appropriate flag for termination.) Because the dispatch source runs autonomously, and therefore owns itself, it also cancels and releases itself in anticipation of the program shutting down.

Listing 4-6 Monitoring the death of a parent process

```
void MonitorParentProcess()
{
    pid_t parentPID = getppid();

    dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_PROC,
                                                      parentPID, DISPATCH_PROC_EXIT,
 queue);
```

```
    if (source)
    {
        dispatch_source_set_event_handler(source, ^{
            MySetAppExitFlag();
            dispatch_source_cancel(source);
            dispatch_release(source);
        });
        dispatch_resume(source);
    }
}
```

Canceling a Dispatch Source

Dispatch sources remain active until you cancel them explicitly using the `dispatch_source_cancel` function. Canceling a dispatch source stops the delivery of new events and cannot be undone. Therefore, you typically cancel a dispatch source and then immediately release it, as shown here:

```
void RemoveDispatchSource(dispatch_source_t mySource)
{
    dispatch_source_cancel(mySource);
    dispatch_release(mySource);
}
```

Cancellation of a dispatch source is an asynchronous operation. Although no new events are processed after you call the `dispatch_source_cancel` function, events that are already being processed by the dispatch source continue to be processed. After it finishes processing any final events, the dispatch source executes its cancellation handler if one is present.

The cancellation handler is your chance to deallocate memory or clean up any resources that were acquired on behalf of the dispatch source. If your dispatch source uses a descriptor or mach port, you must provide a cancellation handler to close the descriptor or destroy the port when cancellation occurs. Other types of dispatch sources do not require cancellation handlers, although you still should provide one if you associate any memory or data with the dispatch source. For example, you should provide one if you store data in the dispatch source's context pointer. For more information about cancellation handlers, see Installing a Cancellation Handler (page 61).

Suspending and Resuming Dispatch Sources

You can suspend and resume the delivery of dispatch source events temporarily using the `dispatch_suspend` and `dispatch_resume` methods. These methods increment and decrement the suspend count for your dispatch object. As a result, you must balance each call to `dispatch_suspend` with a matching call to `dispatch_resume` before event delivery resumes.

When you suspend a dispatch source, any events that occur while that dispatch source is suspended are accumulated until the queue is resumed. When the queue resumes, rather than deliver all of the events, the events are coalesced into a single event before delivery. For example, if you were monitoring a file for name changes, the delivered event would include only the last name change. Coalescing events in this way prevents them from building up in the queue and overwhelming your application when work resumes.

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 (page 77).
- 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 (page 80).
- 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 (page 81).
- If you are using threads to read and write from descriptors, or monitor file operations, use the dispatch sources as described in Dispatch Sources (page 57).

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 (page 79). 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 (page 54). For information about setting up dependencies between operation objects, see Configuring Interoperation Dependencies (page 29).

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 (page 53).

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 (page 57).

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.

Glossary

application A specific style of program that displays a graphical interface to the user.

asynchronous design approach The principle of organizing an application around blocks of code that can be run concurrently with an application's main thread or other threads of execution. Asynchronous tasks are started by one thread but actually run on a different thread, taking advantage of additional processor resources to finish their work more quickly.

block object A C construct for encapsulating inline code and data so that it can be performed later. You use blocks to encapsulate tasks you want to perform, either inline in the current thread or on a separate thread using a dispatch queue. For more information, see *Blocks Programming Topics*.

concurrent operation An operation object that does not perform its task in the thread from which its `start` method was called. A concurrent operation typically sets up its own thread or calls an interface that sets up a separate thread on which to perform the work.

condition A construct used to synchronize access to a resource. A thread waiting on a condition is not allowed to proceed until another thread explicitly signals the condition.

critical section A portion of code that must be executed by only one thread at a time.

custom source A dispatch source used to process application-defined events. A custom source calls your custom event handler in response to events that your application generates.

descriptor An abstract identifier used to access a file, socket, or other system resource.

dispatch queue A Grand Central Dispatch (GCD) structure that you use to execute your application's tasks. GCD defines dispatch queues for executing tasks either serially or concurrently.

dispatch source A Grand Central Dispatch (GCD) data structure that you create to process system-related events.

descriptor dispatch source A dispatch source used to process file-related events. A file descriptor source calls your custom event handler either when file data is available for reading or writing or in response to file system changes.

dynamic shared library A binary executable that is loaded dynamically into an application's process space rather than linked statically as part of the application binary.

framework A type of bundle that packages a dynamic shared library with the resources and header files that support that library. For more information, see *Framework Programming Guide*.

global dispatch queue A dispatch queue provided to your application automatically by Grand Central Dispatch (GCD). You do not have to create global queues yourself or retain or release them. Instead, you retrieve them using the system-provided functions.

Grand Central Dispatch (GCD) A technology for executing asynchronous tasks concurrently. GCD is available in OS X v10.6 and later and iOS 4.0 and later.

input source A source of asynchronous events for a thread. Input sources can be port based or manually triggered and must be attached to the thread's run loop.

joinable thread A thread whose resources are not reclaimed immediately upon termination. Joinable threads must be explicitly detached or be joined by another thread before the resources can be reclaimed. Joinable threads provide a return value to the thread that joins with them.

library A UNIX feature for monitoring low-level system events. For more information see the `kqueue` man page.

Mach port dispatch source A dispatch source used to process events arriving on a Mach port.

main thread A special type of thread created when its owning process is created. When the main thread of a program exits, the process ends.

mutex A lock that provides mutually exclusive access to a shared resource. A mutex lock can be held by only one thread at a time. Attempting to acquire a mutex held by a different thread puts the current thread to sleep until the lock is finally acquired.

Open Computing Language (OpenCL) A standards-based technology for performing general-purpose computations on a computer's graphics processor. For more information, see *OpenCL Programming Guide for Mac* .

operation object An instance of the `NSOperation` class. Operation objects wrap the code and data associated with a task into an executable unit.

operation queue An instance of the `NSOperationQueue` class. Operation queues manage the execution of operation objects.

private dispatch queue A dispatch queue that you create, retain, and release explicitly.

process The runtime instance of an application or program. A process has its own virtual memory space and system resources (including port rights) that are independent of those assigned to other programs. A process always contains at least one thread (the main thread) and may contain any number of additional threads.

process dispatch source A dispatch source used to handle process-related events. A process source calls your custom event handler in response to changes to the process you specify.

program A combination of code and resources that can be run to perform some task. Programs need not have a graphical user interface, although graphical applications are also considered programs.

reentrant Code that can be started on a new thread safely while it is already running on another thread.

run loop An event-processing loop, during which events are received and dispatched to appropriate handlers.

run loop mode A collection of input sources, timer sources, and run loop observers associated with a particular name. When run in a specific "mode," a run loop monitors only the sources and observers associated with that mode.

run loop object An instance of the `NSRunLoop` class or `CFRunLoopRef` opaque type. These objects provide the interface for implementing an event-processing loop in a thread.

run loop observer A recipient of notifications during different phases of a run loop's execution.

semaphore A protected variable that restricts access to a shared resource. Mutexes and conditions are both different types of semaphore.

signal A UNIX mechanism for manipulating a process from outside its domain. The system uses signals to deliver important messages to an application, such as whether the application executed an illegal instruction. For more information see the `signal` man page.

signal dispatch source A dispatch source used to process UNIX signals. A signal source calls your custom event handler whenever the process receives a UNIX signal.

task A quantity of work to be performed. Although some technologies (most notably Carbon Multiprocessing Services) use this term differently, the preferred usage is as an abstract concept indicating some quantity of work to be performed.

thread A flow of execution in a process. Each thread has its own stack space but otherwise shares memory with other threads in the same process.

timer dispatch source A dispatch source used to process periodic events. A timer source calls your custom event handler at regular, time-based intervals.

Document Revision History

This table describes the changes to *Concurrency Programming Guide*.

Date	Notes
2012-12-13	Corrected explanation of dispatch queues.
2012-07-17	Removed obsolete information about autorelease pool usage with operations.
2011-01-19	Updated the code for manually executing operations to handle cancellation correctly. Added information about using Objective-C objects in conjunction with dispatch queues.
2010-04-13	Updated to reflect support for iOS.
2009-08-07	Corrected the start method for the nonconcurrent operation object example.
2009-05-22	New document that describes technologies for executing multiple code paths in a concurrent manner.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010