Threading Programming Guide

Performance

Contents

5

Figures, Tables, and Listings

Introduction

Threads are one of several technologies that make it possible to execute multiple code paths concurrently inside a single application. Although newer technologies such as operation objects and Grand Central Dispatch (GCD) provide a more modern and efficient infrastructure for implementing concurrency, Mac OS X and iOS also provide interfaces for creating and managing threads.

This document provides an introduction to the thread packages available in Mac OS X and shows you how to use them. This document also describes the relevant technologies provided to support threading and the synchronization of multithreaded code inside your application.

> **Important:** If you are developing a new application, you are encouraged to investigate the alternative Mac OS X technologies for implementing concurrency. This is especially true if you are not already familiar with the design techniques needed to implement a threaded application. These alternative technologies simplify the amount of work you have to do to implement concurrent paths of execution and offer much better performance than traditional threads. For information about these technologies, see *Concurrency Programming Guide*.

Organization of This Document

This document has the following chapters and appendixes:

- "About Threaded Programming" (page 11) introduces the concept of threads and their role in application design.
- "Thread Management" (page 21) provides information about the threading technologies in Mac OS X and how you use them.
- "Run Loops" (page 33) provides information about how to manage event-processing loops in secondary threads.
- "Synchronization" (page 57) describes synchronization issues and the tools you use to prevent multiple threads from corrupting data or crashing your program.
- "Thread Safety Summary" (page 73) provides a high-level summary of the inherent thread safety of Mac OS X and iOS and some of their key frameworks.

See Also

For information about the alternatives to threads, see *Concurrency Programming Guide*.

This document provides only a light coverage of the use of the POSIX threads API. For more information about the available POSIX thread routines, see the `pthread` man page. For a more in-depth explanation of POSIX threads and their usage, see *Programming with POSIX Threads* by David R. Butenhof.

About Threaded Programming

For many years, maximum computer performance was limited largely by the speed of a single microprocessor at the heart of the computer. As the speed of individual processors started reaching their practical limits, however, chip makers switched to multicore designs, giving the computer the opportunity to perform multiple tasks simultaneously. And although Mac OS X takes advantage of these cores whenever it can to perform system-related tasks, your own applications can also take advantage of them through threads.

What Are Threads?

Threads are a relatively lightweight way to implement multiple paths of execution inside of an application. At the system level, programs run side by side, with the system doling out execution time to each program based on its needs and the needs of other programs. Inside each program, however, exists one or more threads of execution, which can be used to perform different tasks simultaneously or in a nearly simultaneous manner. The system itself actually manages these threads of execution, scheduling them to run on the available cores and preemptively interrupting them as needed to allow other threads to run.

From a technical standpoint, a thread is a combination of the kernel-level and application-level data structures needed to manage the execution of code. The kernel-level structures coordinate the dispatching of events to the thread and the preemptive scheduling of the thread on one of the available cores. The application-level structures include the call stack for storing function calls and the structures the application needs to manage and manipulate the thread's attributes and state.

In a nonconcurrent application, there is only one thread of execution. That thread starts and ends with your application's `main` routine and branches one-by-one to different methods or functions to implement the application's overall behavior. By contrast, an application that supports concurrency starts with one thread and adds more as needed to create additional execution paths. Each new path has its own custom start routine that runs independently of the code in the application's `main` routine. Having multiple threads in an application provides two very important potential advantages:

■ Multiple threads can improve an application's perceived responsiveness.

■ Multiple threads can improve an application's real-time performance on multicore systems.

If your application has only one thread, that one thread must do everything. It must respond to events, update your application's windows, and perform all of the computations needed to implement your application's behavior. The problem with having just one thread is that it can only do one thing at a time. So what happens when one of your computations takes a long time to finish? While your code is busy computing the values it needs, your application stops responding to user events and updating its windows. If this behavior continues long enough, a user might think your application is hung and try to forcibly quit it. If you moved your custom computations onto a separate thread, however, your application's main thread would be free to respond to user interactions in a more timely manner.

With multicore computers common these days, threads provide a way to increase performance in some types of applications. Threads that perform different tasks can do so simultaneously on different processor cores, making it possible for an application to increase the amount of work it does in a given amount of time.

Of course, threads are not a panacea for fixing an application's performance problems. Along with the benefits offered by threads come the potential problems. Having multiple paths of execution in an application can add a considerable amount of complexity to your code. Each thread has to coordinate its actions with other threads to prevent it from corrupting the application's state information. Because threads in a single application share the same memory space, they have access to all of the same data structures. If two threads try to manipulate the same data structure at the same time, one thread might overwrite another's changes in a way that corrupts the resulting data structure. Even with proper protections in place, you still have to watch out for compiler optimizations that introduce subtle (and not so subtle) bugs into your code.

Threading Terminology

Before getting too far into discussions about threads and their supporting technologies, it is necessary to define some basic terminology.

If you are familiar with Carbon's Multiprocessor Services API or with UNIX systems, you may find that the term "task" is used differently by this document. In earlier versions of Mac OS, the term "task" was used to distinguish between threads created using the Multiprocessor Services and those created using the Carbon Thread Manager API. On UNIX systems, the term "task" is also used at times to refer to a running process. In practical terms, a Multiprocessor Services task is equivalent to a preemptively scheduled thread.

Given that both the Carbon Thread Manager and Multiprocessor Services APIs are legacy technologies in Mac OS X, this document adopts the following terminology:

- The term **thread** is used to refer to a separate path of execution for code.
- 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.

Alternatives to Threads

One problem with creating threads yourself is that they add uncertainty to your code. Threads are a relatively low-level and complicated way to support concurrency in your application. If you do not fully understand the implications of your design choices, you could easily encounter synchronization or timing issues, the severity of which can range from subtle behavioral changes to the crashing of your application and the corruption of the user's data.

Another factor to consider is whether you need threads or concurrency at all. Threads solve the specific problem of how to execute multiple code paths concurrently inside the same process. There may be cases, though, where the amount of work you are doing does not warrant concurrency. Threads introduce a tremendous amount of overhead to your process, both in terms of memory consumption and CPU time. You may discover that this overhead is too great for the intended task, or that other options are easier to implement.

Table 1-1 lists some of the alternatives to threads. This table includes both replacement technologies for threads (such as operation objects and GCD) and alternatives that are geared towards efficiently using the single thread you already have.

Table 1-1   Alternative technologies to threads

Technology	Description
Operation objects	Introduced in Mac OS X v10.5, an operation object is a wrapper for a task that would normally be executed on a secondary thread. This wrapper hides the thread management aspects of performing the task, leaving you free to focus on the task itself. You typically use these objects in conjunction with an operation queue object, which actually manages the execution of the operation objects on one more threads. For more information on how to use operation objects, see *Concurrency Programming Guide*.
Grand Central Dispatch (GCD)	Introduced in Mac OS x v10.6, Grand Central Dispatch is another alternative to threads that lets you focus on the tasks you need to perform rather than on thread management. With GCD, you define the task you want to perform and add it to a work queue, which handles the scheduling of your task on an appropriate thread. Work queues take into account the number of available cores and the current load to execute your tasks more efficiently than you could do yourself using threads. For information on how to use GCD and work queues, see *Concurrency Programming Guide*
Idle-time notifications	For tasks that are relatively short and very low priority, idle time notifications let you perform the task at a time when your application is not as busy. Cocoa provides support for idle-time notifications using the `NSNotificationQueue` object. To request an idle-time notification, post a notification to the default `NSNotificationQueue` object using the `NSPostWhenIdle` option. The queue delays the delivery of your notification object until the run loop becomes idle. For more information, see *Notification Programming Topics*.
Asynchronous functions	The system interfaces include many asynchronous functions that provide automatic concurrency for you. These APIs may use system daemons and processes or create custom threads to perform their task and return the results to you. (The actual implementation is irrelevant because it is separated from your code.) As you design your application, look for functions that offer asynchronous behavior and consider using them instead of using the equivalent synchronous function on a custom thread.
Timers	You can use timers on your application's main thread to perform periodic tasks that are too trivial to require a thread, but which still require servicing at regular intervals. For information on timers, see "Timer Sources" (page 37).
Separate processes	Although more heavyweight than threads, creating a separate process might be useful in cases where the task is only tangentially related to your application. You might use a process if a task requires a significant amount of memory or must be executed using root privileges. For example, you might use a 64-bit server process to compute a large data set while your 32-bit application displays the results to the user.

> ⚠️ **Warning:** When launching separate processes using the `fork` function, you must always follow a call to `fork` with a call to `exec` or a similar function. Applications that depend on the Core Foundation, Cocoa, or Core Data frameworks (either explicitly or implicitly) must make a subsequent call to an `exec` function or those frameworks may behave improperly.

Threading Support

If you have existing code that uses threads, Mac OS X and iOS provide several technologies for creating threads in your applications. In addition, both systems also provide support for managing and synchronizing the work that needs to be done on those threads. The following sections describe some of the key technologies that you need to be aware of when working with threads in Mac OS X and iOS.

Threading Packages

Although the underlying implementation mechanism for threads is Mach threads, you rarely (if ever) work with threads at the Mach level. Instead, you usually use the more convenient POSIX API or one of its derivatives. The Mach implementation does provide the basic features of all threads, however, including the preemptive execution model and the ability to schedule threads so they are independent of each other.

Listing 2-2 lists the threading technologies you can use in your applications.

Table 1-2   Thread technologies

Technology	Description
Cocoa threads	Cocoa implements threads using the `NSThread` class. Cocoa also provides methods on `NSObject` for spawning new threads and executing code on already-running threads. For more information, see "Using NSThread" (page 22) and "Using NSObject to Spawn a Thread" (page 25).
POSIX threads	POSIX threads provide a C-based interface for creating threads. If you are not writing a Cocoa application, this is the best choice for creating threads. The POSIX interface is relatively simple to use and offers ample flexibility for configuring your threads. For more information, see "Using POSIX Threads" (page 24)
Multiprocessing Services	Multiprocessing Services is a legacy C-based interface used by applications transitioning from older versions of Mac OS. This technology is available in Mac OS X only and should be avoided for any new development. Instead, you should use the `NSThread` class or POSIX threads. For more information about this technology, however, see *Multiprocessing Services Programming Guide*.

At the application level, all threads behave in essentially the same way as on other platforms. After starting a thread, the thread runs in one of three main states: running, ready, or blocked. If a thread is not currently running, it is either blocked and waiting for input or it is ready to run but not scheduled to do so yet. The thread continues moving back and forth among these states until it finally exits and moves to the terminated state.

When you create a new thread, you must specify an entry-point function (or an entry-point method in the case of Cocoa threads) for that thread. This entry-point function constitutes the code you want to run on the thread. When the function returns, or when you terminate the thread explicitly, the thread stops permanently and is reclaimed by the system. Because threads are relatively expensive to create in terms of memory and time, it is therefore recommended that your entry point function do a significant amount of work or set up a run loop to allow for recurring work to be performed.

For more information about the available threading technologies and how to use them, see "Thread Management" (page 21).

Run Loops

A run loop is a piece of infrastructure used to manage events arriving asynchronously on a thread. A run loop works by monitoring one or more event sources for the thread. As events arrive, the system wakes up the thread and dispatches the events to the run loop, which then dispatches them to the handlers you specify. If no events are present and ready to be handled, the run loop puts the thread to sleep.

You are not required to use a run loop with any threads you create but doing so can provide a better experience for the user. Run loops make it possible to create long-lived threads that use a minimal amount of resources. Because a run loop puts its thread to sleep when there is nothing to do, it eliminates the need for polling, which wastes CPU cycles and prevents the processor itself from sleeping and saving power.

To configure a run loop, all you have to do is launch your thread, get a reference to the run loop object, install your event handlers, and tell the run loop to run. The infrastructure provided by both Cocoa and Carbon handles the configuration of the main thread's run loop for you automatically. If you plan to create long-lived secondary threads, however, you must configure the run loop for those threads yourself.

Details about run loops and examples of how to use them are provided in "Run Loops" (page 33).

Synchronization Tools

One of the hazards of threaded programming is resource contention among multiple threads. If multiple threads try to use or modify the same resource at the same time, problems can occur. One way to alleviate the problem is to eliminate the shared resource altogether and make sure each thread has its own distinct set of resources on which to operate. When maintaining completely separate resources is not an option though, you may have to synchronize access to the resource using locks, conditions, atomic operations, and other techniques.

Locks provide a brute force form of protection for code that can be executed by only one thread at a time. The most common type of lock is mutual exclusion lock, also known as a **mutex**. When a thread tries to acquire a mutex that is currently held by another thread, it blocks until the lock is released by the other thread. Several system frameworks provide support for mutex locks, although they are all based on the same underlying technology. In addition, Cocoa provides several variants of the mutex lock to support different types of behavior, such as recursion. For more information about the available types of locks, see "Locks" (page 58).

In addition to locks, the system provides support for conditions, which ensure the proper sequencing of tasks within your application. A condition acts as a gatekeeper, blocking a given thread until the condition it represents becomes true. When that happens, the condition releases the thread and allows it to continue. The POSIX layer and Foundation framework both provide direct support for conditions. (If you use operation objects, you can configure dependencies among your operation objects to sequence the execution of tasks, which is very similar to the behavior offered by conditions.)

Although locks and conditions are very common in concurrent design, atomic operations are another way to protect and synchronize access to data. Atomic operations offer a lightweight alternative to locks in situations where you can perform mathematical or logical operations on scalar data types. Atomic operations use special hardware instructions to ensure that modifications to a variable are completed before other threads have a chance to access it.

For more information about the available synchronization tools, see "Synchronization Tools" (page 57).

Interthread Communication

Although a good design minimizes the amount of required communication, at some point, communication between threads becomes necessary. (A thread's job is to do work for your application, but if the results of that job are never used, what good is it?) Threads may need to process new job requests or report their progress to your application's main thread. In these situations, you need a way to get information from one thread to another. Fortunately, the fact that threads share the same process space means you have lots of options for communication.

There are many ways to communicate between threads, each with its own advantages and disadvantages. "Configuring Thread-Local Storage" lists the most common communication mechanisms you can use in Mac OS X. (With the exception of message queues and Cocoa distributed objects, these technologies are also available in iOS.) The techniques in this table are listed in order of increasing complexity.

Table 1-3   Communication mechanisms

Mechanism	Description
Direct messaging	Cocoa applications support the ability to perform selectors directly on other threads. This capability means that one thread can essentially execute a method on any other thread. Because they are executed in the context of the target thread, messages sent this way are automatically serialized on that thread. For information about input sources, see "Cocoa Perform Selector Sources" (page 36).
Global variables, shared memory, and objects	Another simple way to communicate information between two threads is to use a global variable, shared object, or shared block of memory. Although shared variables are fast and simple, they are also more fragile than direct messaging. Shared variables must be carefully protected with locks or other synchronization mechanisms to ensure the correctness of your code. Failure to do so could lead to race conditions, corrupted data, or crashes.
Conditions	Conditions are a synchronization tool that you can use to control when a thread executes a particular portion of code. You can think of conditions as gate keepers, letting a thread run only when the stated condition is met. For information on how to use conditions, see "Using Conditions" (page 70).
Run loop sources	A custom run loop source is one that you set up to receive application-specific messages on a thread. Because they are event driven, run loop sources put your thread to sleep automatically when there is nothing to do, which improves your thread's efficiency. For information about run loops and run loop sources, see "Run Loops" (page 33).
Ports and sockets	Port-based communication is a more elaborate way to communication between two threads, but it is also a very reliable technique. More importantly, ports and sockets can be used to communicate with external entities, such as other processes and services. For efficiency, ports are implemented using run loop sources, so your thread sleeps when there is no data waiting on the port. For information about run loops and about port-based input sources, see "Run Loops" (page 33).

Mechanism	Description
Message queues	Multiprocessing Services defines a first-in, first-out (FIFO) queue abstraction for managing incoming and outgoing data. Although message queues are simple and convenient, they are not as efficient as some other communications techniques. For more information about how to use message queues, see *Multiprocessing Services Programming Guide*.
Cocoa distributed objects	Distributed objects is a Cocoa technology that provides a high-level implementation of port-based communications. Although it is possible to use this technology for interthread communication, doing so is highly discouraged because of the amount of overhead it incurs. Distributed objects is much more suitable for communicating with other processes, where the overhead of going between processes is already high. For more information, see *Distributed Objects Programming Topics*.

Design Tips

The following sections offer guidelines to help you implement threads in a way that ensures the correctness of your code. Some of these guidelines also offer tips for achieving better performance with your own threaded code. As with any performance tips, you should always gather relevant performance statistics before, during, and after you make changes to your code.

Avoid Creating Threads Explicitly

Writing thread-creation code manually is tedious and potentially error-prone and you should avoid it whenever possible. Mac OS X and iOS provide implicit support for concurrency through other APIs. Rather than create a thread yourself, consider using asynchronous APIs, GCD, or operation objects to do the work. These technologies do the thread-related work behind the scenes for you and are guaranteed to do it correctly. In addition, technologies such as GCD and operation objects are designed to manage threads much more efficiently than your own code ever could by adjusting the number of active threads based on the current system load. For more information about GCD and operation objects, see *Concurrency Programming Guide*.

Keep Your Threads Reasonably Busy

If you decide to create and manage threads manually, remember that threads consume precious system resources. You should do your best to make sure that any tasks you assign to threads are reasonably long-lived and productive. At the same time, you should not be afraid to terminate threads that are spending most of their time idle. Threads use a nontrivial amount of memory, some of it wired, so releasing an idle thread not only helps reduce your application's memory footprint, it also frees up more physical memory for other system processes to use.

> **Important:** Before you start terminating idle threads, you should always record a set of baseline measurements of your applications current performance. After trying your changes, take additional measurements to verify that the changes are actually improving performance, rather than hurting it.

Avoid Shared Data Structures

The simplest and easiest way to avoid thread-related resource conflicts is to give each thread in your program its own copy of whatever data it needs. Parallel code works best when you minimize the communication and resource contention among your threads.

Creating a multithreaded application is hard. Even if you are very careful and lock shared data structures at all the right junctures in your code, your code may still be semantically unsafe. For example, your code could run into problems if it expected shared data structures to be modified in a specific order. Changing your code to a transaction-based model to compensate could subsequently negate the performance advantage of having multiple threads. Eliminating the resource contention in the first place often results in a simpler design with excellent performance.

Threads and Your User Interface

If your application has a graphical user interface, it is recommended that you receive user-related events and initiate interface updates from your application's main thread. This approach helps avoid synchronization issues associated with handling user events and drawing window content. Some frameworks, such as Cocoa, generally require this behavior, but even for those that do not, keeping this behavior on the main thread has the advantage of simplifying the logic for managing your user interface.

There are a few notable exceptions where it is advantageous to perform graphical operations from other threads. For example, the QuickTime API includes a number of operations that can be performed from secondary threads, including opening movie files, rendering movie files, compressing movie files, and importing and exporting images. Similarly, in Carbon and Cocoa you can use secondary threads to create and process images and perform other image-related calculations. Using secondary threads for these operations can greatly increase performance. If you are not sure about a particular graphical operation though, plan on doing it from your main thread.

For more information about QuickTime thread safety, see Technical Note TN2125: "Thread-Safe Programming in QuickTime." For more information about Cocoa thread safety, see "Thread Safety Summary" (page 73). For more information about drawing in Cocoa, see *Cocoa Drawing Guide*.

Be Aware of Thread Behaviors at Quit Time

A process runs until all nondetached threads have exited. By default, only the application's main thread is created as nondetached, but you can create other threads that way as well. When the user quits an application, it is usually considered appropriate behavior to terminate all detached threads immediately, because the work done by detached threads is considered optional. If your application is using background threads to save data to disk or do other critical work, however, you may want to create those threads as nondetached to prevent the loss of data when the application exits.

Creating threads as nondetached (also known as joinable) requires extra work on your part. Because most high-level thread technologies do not create joinable threads by default, you may have to use the POSIX API to create your thread. In addition, you must add code to your application's main thread to join with the nondetached threads when they do finally exit. For information on creating joinable threads, see "Setting the Detached State of a Thread" (page 27).

If you are writing a Cocoa application, you can also use the `applicationShouldTerminate:` delegate method to delay the termination of the application until a later time or cancel it altogether. When delaying termination, your application would need to wait until any critical threads have finished their tasks and then invoke the `replyToApplicationShouldTerminate:` method. For more information on these methods, see *NSApplication Class Reference*.

Handle Exceptions

Exception handling mechanisms rely on the current call stack to perform any necessary clean up when an exception is thrown. Because each thread has its own call stack, each thread is therefore responsible for catching its own exceptions. Failing to catch an exception in a secondary thread is the same as failing to catch an exception in your main thread: the owning process is terminated. You cannot throw an uncaught exception to a different thread for processing.

If you need to notify another thread (such as the main thread) of an exceptional situation in the current thread, you should catch the exception and simply send a message to the other thread indicating what happened. Depending on your model and what you are trying to do, the thread that caught the exception can then continue processing (if that is possible), wait for instructions, or simply exit.

> **Note:** In Cocoa, an `NSException` object is a self-contained object that can be passed from thread to thread once it has been caught.

In some cases, an exception handler may be created for you automatically. For example, the `@synchronized` directive in Objective-C contains an implicit exception handler.

Terminate Your Threads Cleanly

The best way for a thread to exit is naturally, by letting it reach the end of its main entry point routine. Although there are functions to terminate threads immediately, those functions should be used only as a last resort. Terminating a thread before it has reached its natural end point prevents the thread from cleaning up after itself. If the thread has allocated memory, opened a file, or acquired other types of resources, your code may be unable to reclaim those resources, resulting in memory leaks or other potential problems.

For more information on the proper way to exit a thread, see "Terminating a Thread" (page 30).

Thread Safety in Libraries

Although an application developer has control over whether an application executes with multiple threads, library developers do not. When developing libraries, you must assume that the calling application is multithreaded or could switch to being multithreaded at any time. As a result, you should always use locks for critical sections of code.

For library developers, it is unwise to create locks only when an application becomes multithreaded. If you need to lock your code at some point, create the lock object early in the use of your library, preferably in some sort of explicit call to initialize the library. Although you could also use a static library initialization function to create such locks, try to do so only when there is no other way. Execution of an initialization function adds to the time required to load your library and could adversely affect performance.

> **Note:** Always remember to balance calls to lock and unlock a mutex lock within your library. You should also remember to lock library data structures rather than rely on the calling code to provide a thread-safe environment.

If you are developing a Cocoa library, you can register as an observer for the `NSWillBecomeMultiThreadedNotification` if you want to be notified when the application becomes multithreaded. You should not rely on receiving this notification, though, as it might be dispatched before your library code is ever called.

Thread Management

Each process (application) in Mac OS X or iOS is made up of one or more threads, each of which represents a single path of execution through the application's code. Every application starts with a single thread, which runs the application's `main` function. Applications can spawn additional threads, each of which executes the code of a specific function.

When an application spawns a new thread, that thread becomes an independent entity inside of the application's process space. Each thread has its own execution stack and is scheduled for runtime separately by the kernel. A thread can communicate with other threads and other processes, perform I/O operations, and do anything else you might need it to do. Because they are inside the same process space, however, all threads in a single application share the same virtual memory space and have the same access rights as the process itself.

This chapter provides an overview of the thread technologies available in Mac OS X and iOS along with examples of how to use those technologies in your applications.

> **Note:** For a historical look at the threading architecture of Mac OS, and for additional background information on threads, see Technical Note TN2028, "Threading Architectures".

Thread Costs

Threading has a real cost to your program (and the system) in terms of memory use and performance. Each thread requires the allocation of memory in both the kernel memory space and your program's memory space. The core structures needed to manage your thread and coordinate its scheduling are stored in the kernel using wired memory. Your thread's stack space and per-thread data is stored in your program's memory space. Most of these structures are created and initialized when you first create the thread—a process that can be relatively expensive because of the required interactions with the kernel.

Table 2-1 quantifies the approximate costs associated with creating a new user-level thread in your application. Some of these costs are configurable, such as the amount of stack space allocated for secondary threads. The time cost for creating a thread is a rough approximation and should be used only for relative comparisons with each other. Thread creation times can vary greatly depending on processor load, the speed of the computer, and the amount of available system and program memory.

Table 2-1 Thread creation costs

Item	Approximate cost	Notes
Kernel data structures	Approximately 1 KB	This memory is used to store the thread data structures and attributes, much of which is allocated as wired memory and therefore cannot be paged to disk.

Item	Approximate cost	Notes
Stack space	512 KB (secondary threads) 8 MB (Mac OS X main thread) 1 MB (iOS main thread)	The minimum allowed stack size for secondary threads is 16 KB and the stack size must be a multiple of 4 KB. The space for this memory is set aside in your process space at thread creation time, but the actual pages associated with that memory are not created until they are needed.
Creation time	Approximately 90 microseconds	This value reflects the time between the initial call to create the thread and the time at which the thread's entry point routine began executing. The figures were determined by analyzing the mean and median values generated during thread creation on an Intel-based iMac with a 2 GHz Core Duo processor and 1 GB of RAM running Mac OS X v10.5.

> **Note:** Because of their underlying kernel support, operation objects can often create threads more quickly. Rather than creating threads from scratch every time, they use pools of threads already residing in the kernel to save on allocation time. For more information about using operation objects, see *Concurrency Programming Guide*.

Another cost to consider when writing threaded code is the production costs. Designing a threaded application can sometimes require fundamental changes to the way you organize your application's data structures. Making those changes might be necessary to avoid the use of synchronization, which can itself impose a tremendous performance penalty on poorly designed applications. Designing those data structures, and debugging problems in threaded code, can increase the time it takes to develop a threaded application. Avoiding those costs can create bigger problems at runtime, however, if your threads spend too much time waiting on locks or doing nothing.

Creating a Thread

Creating low-level threads is relatively simple. In all cases, you must have a function or method to act as your thread's main entry point and you must use one of the available thread routines to start your thread. The following sections show the basic creation process for the more commonly used thread technologies. Threads created using these techniques inherit a default set of attributes, determined by the technology you use. For information on how to configure your threads, see "Configuring Thread Attributes" (page 26).

Using NSThread

There are two ways to create a thread using the `NSThread` class:

- Use the `detachNewThreadSelector:toTarget:withObject:` class method to spawn the new thread.

- Create a new `NSThread` object and call its `start` method. (Supported only in iOS and Mac OS X v10.5 and later.)

Both techniques create a detached thread in your application. A detached thread means that the thread's resources are automatically reclaimed by the system when the thread exits. It also means that your code does not have to join explicitly with the thread later.

Because the `detachNewThreadSelector:toTarget:withObject:` method is supported in all versions of Mac OS X, it is often found in existing Cocoa applications that use threads. To detach a new thread, you simply provide the name of the method (specified as a selector) that you want to use as the thread's entry point, the object that defines that method, and any data you want to pass to the thread at startup. The following example shows a basic invocation of this method that spawns a thread using a custom method of the current object.

```
[NSThread detachNewThreadSelector:@selector(myThreadMainMethod:) toTarget:self
 withObject:nil];
```

Prior to Mac OS X v10.5, you used the `NSThread` class primarily to spawn threads. Although you could get an `NSThread` object and access some thread attributes, you could only do so from the thread itself after it was running. In Mac OS X v10.5, support was added for creating `NSThread` objects without immediately spawning the corresponding new thread. (This support is also available in iOS.) This support made it possible to get and set various thread attributes prior to starting the thread. It also made it possible to use that thread object to refer to the running thread later.

The simple way to initialize an `NSThread` object in Mac OS X v10.5 and later is to use the `initWithTarget:selector:object:` method. This method takes the exact same information as the `detachNewThreadSelector:toTarget:withObject:` method and uses it to initialize a new `NSThread` instance. It does not start the thread, however. To start the thread, you call the thread object's `start` method explicitly, as shown in the following example:

```
NSThread* myThread = [[NSThread alloc] initWithTarget:self
                                       selector:@selector(myThreadMainMethod:)
                                       object:nil];
[myThread start];  // Actually create the thread
```

> **Note:** An alternative to using the `initWithTarget:selector:object:` method is to subclass `NSThread` and override its `main` method. You would use the overridden version of this method to implement your thread's main entry point. For more information, see the subclassing notes in *NSThread Class Reference*.

If you have an `NSThread` object whose thread is currently running, one way you can send messages to that thread is to use the `performSelector:onThread:withObject:waitUntilDone:` method of almost any object in your application. Support for performing selectors on threads (other than the main thread) was introduced in Mac OS X v10.5 and is a convenient way to communicate between threads. (This support is also available in iOS.) The messages you send using this technique are executed directly by the other thread as part of its normal run-loop processing. (Of course, this does mean that the target thread has to be running in its run loop; see "Run Loops" (page 33).) You may still need some form of synchronization when you communicate this way, but it is simpler than setting up communications ports between the threads.

> **Note:** Although good for occasional communication between threads, you should not use the `performSelector:onThread:withObject:waitUntilDone:` method for time critical or frequent communication between threads.

For a list of other thread communication options, see "Setting the Detached State of a Thread" (page 27).

Using POSIX Threads

Mac OS X and iOS provide C-based support for creating threads using the POSIX thread API. This technology can actually be used in any type of application (including Cocoa and Cocoa Touch applications) and might be more convenient if you are writing your software for multiple platforms. The POSIX routine you use to create threads is called, appropriately enough, `pthread_create`.

Listing 2-1 shows two custom functions for creating a thread using POSIX calls. The `LaunchThread` function creates a new thread whose main routine is implemented in the `PosixThreadMainRoutine` function. Because POSIX creates threads as joinable by default, this example changes the thread's attributes to create a detached thread. Marking the thread as detached gives the system a chance to reclaim the resources for that thread immediately when it exits.

Listing 2-1   Creating a thread in C

```c
#include <assert.h>
#include <pthread.h>

void* PosixThreadMainRoutine(void* data)
{
    // Do some work here.

    return NULL;
}

void LaunchThread()
{
    // Create the thread using POSIX routines.
    pthread_attr_t  attr;
    pthread_t       posixThreadID;
    int             returnVal;

    returnVal = pthread_attr_init(&attr);
    assert(!returnVal);
    returnVal = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    assert(!returnVal);

    int     threadError = pthread_create(&posixThreadID, &attr,
&PosixThreadMainRoutine, NULL);

    returnVal = pthread_attr_destroy(&attr);
    assert(!returnVal);
    if (threadError != 0)
    {
        // Report an error.
    }
}
```

If you add the code from the preceding listing to one of your source files and call the `LaunchThread` function, it would create a new detached thread in your application. Of course, new threads created using this code would not do anything useful. The threads would launch and almost immediately exit. To make things more interesting, you would need to add code to the `PosixThreadMainRoutine` function to do some actual work. To ensure that a thread knows what work to do, you can pass it a pointer to some data at creation time. You pass this pointer as the last parameter of the `pthread_create` function.

To communicate information from your newly created thread back to your application's main thread, you need to establish a communications path between the target threads. For C-based applications, there are several ways to communicate between threads, including the use of ports, conditions, or shared memory. For long-lived threads, you should almost always set up some sort of interthread communications mechanism to give your application's main thread a way to check the status of the thread or shut it down cleanly when the application exits.

For more information about POSIX thread functions, see the `pthread` man page.

Using NSObject to Spawn a Thread

In iOS and Mac OS X v10.5 and later, all objects have the ability to spawn a new thread and use it to execute one of their methods. The `performSelectorInBackground:withObject:` method creates a new detached thread and uses the specified method as the entry point for the new thread. For example, if you have some object (represented by the variable `myObj`) and that object has a method called `doSomething` that you want to run in a background thread, you could could use the following code to do that:

```
[myObj performSelectorInBackground:@selector(doSomething) withObject:nil];
```

The effect of calling this method is the same as if you called the `detachNewThreadSelector:toTarget:withObject:` method of `NSThread` with the current object, selector, and parameter object as parameters. The new thread is spawned immediately using the default configuration and begins running. Inside the selector, you must configure the thread just as you would any thread. For example, you would need to set up an autorelease pool (if you were not using garbage collection) and configure the thread's run loop if you planned to use it. For information on how to configure new threads, see "Configuring Thread Attributes" (page 26).

Using Other Threading Technologies

Although the POSIX routines and `NSThread` class are the recommended technologies to use for creating low-level threads, other C-based technologies are available in Mac OS X. Of these, the only other one you might consider using is Multiprocessing Services, which is itself implemented on top of POSIX threads. Multiprocessing Services was developed originally for earlier versions of Mac OS and was later made available for Carbon applications in Mac OS X. If you have existing code that uses this technology, you can continue to use it, although you should also consider porting your thread-related code to POSIX. This technology is not available in iOS.

For information on how to use Multiprocessing Services, see *Multiprocessing Services Programming Guide*.

Using POSIX Threads in a Cocoa Application

Although the `NSThread` class is the main interface for creating threads in Cocoa applications, you are free to use POSIX threads instead if doing so is more convenient for you. For example, you might use POSIX threads if you already have code that uses them and you do not want to rewrite it. If you do plan to use the POSIX threads in a Cocoa application, you should still be aware of the interactions between Cocoa and threads and obey the guidelines in the following sections.

Protecting the Cocoa Frameworks

For multithreaded applications, Cocoa frameworks use locks and other forms of internal synchronization to ensure they behave correctly. To prevent these locks from degrading performance in the single-threaded case, however, Cocoa does not create them until the application spawns its first new thread using the `NSThread` class. If you spawn threads using only POSIX thread routines, Cocoa does not receive the notifications it needs to know that your application is now multithreaded. When that happens, operations involving the Cocoa frameworks may destabilize or crash your application.

To let Cocoa know that you intend to use multiple threads, all you have to do is spawn a single thread using the `NSThread` class and let that thread immediately exit. Your thread entry point need not do anything. Just the act of spawning a thread using `NSThread` is enough to ensure that the locks needed by the Cocoa frameworks are put in place.

If you are not sure if Cocoa thinks your application is multithreaded or not, you can use the `isMultiThreaded` method of `NSThread` to check.

Mixing POSIX and Cocoa Locks

It is safe to use a mixture of POSIX and Cocoa locks inside the same application. Cocoa lock and condition objects are essentially just wrappers for POSIX mutexes and conditions. For a given lock, however, you must always use the same interface to create and manipulate that lock. In other words, you cannot use a Cocoa `NSLock` object to manipulate a mutex you created using the `pthread_mutex_init` function, and vice versa.

Configuring Thread Attributes

After you create a thread, and sometimes before, you may want to configure different portions of the thread environment. The following sections describe some of the changes you can make and when you might make them.

Configuring the Stack Size of a Thread

For each new thread you create, the system allocates a specific amount of memory in your process space to act as the stack for that thread. The stack manages the stack frames and is also where any local variables for the thread are declared. The amount of memory allocated for threads is listed in "Thread Costs" (page 21).

If you want to change the stack size of a given thread, you must do so before you create the thread. All of the threading technologies provide some way of setting the stack size, although setting the stack size using `NSThread` is available only in iOS and Mac OS X v10.5 and later. Table 2-2 lists the different options for each technology.

Table 2-2   Setting the stack size of a thread

Technology	Option
Cocoa	In iOS and Mac OS X v10.5 and later, allocate and initialize an `NSThread` object (do not use the `detachNewThreadSelector:toTarget:withObject:` method). Before calling the `start` method of the thread object, use the `setStackSize:` method to specify the new stack size.
POSIX	Create a new `pthread_attr_t` structure and use the `pthread_attr_-setstacksize` function to change the default stack size. Pass the attributes to the `pthread_create` function when creating your thread.
Multiprocessing Services	Pass the appropriate stack size value to the `MPCreateTask` function when you create your thread.

Configuring Thread-Local Storage

Each thread maintains a dictionary of key-value pairs that can be accessed from anywhere in the thread. You can use this dictionary to store information that you want to persist throughout the execution of your thread. For example, you could use it to store state information that you want to persist through multiple iterations of your thread's run loop.

Cocoa and POSIX store the thread dictionary in different ways, so you cannot mix and match calls to the two technologies. As long as you stick with one technology inside your thread code, however, the end results should be similar. In Cocoa, you use the `threadDictionary` method of an `NSThread` object to retrieve an `NSMutableDictionary` object, to which you can add any keys required by your thread. In POSIX, you use the `pthread_setspecific` and `pthread_getspecific` functions to set and get the keys and values of your thread.

Setting the Detached State of a Thread

Most high-level thread technologies create detached threads by default. In most cases, detached threads are preferred because they allow the system to free up the thread's data structures immediately upon completion of the thread. Detached threads also do not require explicit interactions with your program. The means of retrieving results from the thread is left to your discretion. By comparison, the system does not reclaim the resources for joinable threads until another thread explicitly joins with that thread, a process which may block the thread that performs the join.

You can think of joinable threads as akin to child threads. Although they still run as independent threads, a joinable thread must be joined by another thread before its resources can be reclaimed by the system. Joinable threads also provide an explicit way to pass data from an exiting thread to another thread. Just before it exits, a joinable thread can pass a data pointer or other return value to the `pthread_exit` function. Another thread can then claim this data by calling the `pthread_join` function.

> **Important:** At application exit time, detached threads can be terminated immediately but joinable threads cannot. Each joinable thread must be joined before the process is allowed to exit. Joinable threads may therefore be preferable in cases where the thread is doing critical work that should not be interrupted, such as saving data to disk.

If you do want to create joinable threads, the only way to do so is using POSIX threads. POSIX creates threads as joinable by default. To mark a thread as detached or joinable, modify the thread attributes using the `pthread_attr_setdetachstate` function prior to creating the thread. After the thread begins, you can change a joinable thread to a detached thread by calling the `pthread_detach` function. For more information about these POSIX thread functions, see the `pthread` man page. For information on how to join with a thread, see the `pthread_join` man page.

Setting the Thread Priority

Any new thread you create has a default priority associated with it. The kernel's scheduling algorithm takes thread priorities into account when determining which threads to run, with higher priority threads being more likely to run than threads with lower priorities. Higher priorities do not guarantee a specific amount of execution time for your thread, just that it is more likely to be chosen by the scheduler when compared to lower-priority threads.

> **Important:** It is generally a good idea to leave the priorities of your threads at their default values. Increasing the priorities of some threads also increases the likelihood of starvation among lower-priority threads. If your application contains high-priority and low-priority threads that must interact with each other, the starvation of lower-priority threads may block other threads and create performance bottlenecks.

If you do want to modify thread priorities, both Cocoa and POSIX provide a way to do so. For Cocoa threads, you can use the `setThreadPriority:` class method of `NSThread` to set the priority of the currently running thread. For POSIX threads, you use the `pthread_setschedparam` function. For more information, see *NSThread Class Reference* or `pthread_setschedparam` man page.

Writing Your Thread Entry Routine

For the most part, the structure of your thread's entry point routines is the same in Mac OS X as it is on other platforms. You initialize your data structures, do some work or optionally set up a run loop, and clean up when your thread's code is done. Depending on your design, there may be some additional steps you need to take when writing your entry routine.

Creating an Autorelease Pool

Applications that link in Objective-C frameworks typically must create at least one autorelease pool in each of their threads. If an application uses the managed model—where the application handles the retaining and releasing of objects—the autorelease pool catches any objects that are autoreleased from that thread.

If an application uses garbage collection instead of the managed memory model, creation of an autorelease pool is not strictly necessary. The presence of an autorelease pool in a garbage-collected application is not harmful, and for the most part is simply ignored. It is allowed for cases where a code module must support both garbage collection and the managed memory model. In such a case, the autorelease pool must be present to support the managed memory model code and is simply ignored if the application is run with garbage collection enabled.

If your application uses the managed memory model, creating an autorelease pool should be the first thing you do in your thread entry routine. Similarly, destroying this autorelease pool should be the last thing you do in your thread. This pool ensures that autoreleased objects are caught, although it does not release them until the thread itself exits. Listing 2-2 shows the structure of a basic thread entry routine that uses an autorelease pool.

Listing 2-2   Defining your thread entry point routine

```
- (void)myThreadMainRoutine
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init]; // Top-level
pool

    // Do thread work here.

    [pool release];  // Release the objects in the pool.
}
```

Because the top-level autorelease pool does not release its objects until the thread exits, long-lived threads should create additional autorelease pools to free objects more frequently. For example, a thread that uses a run loop might create and release an autorelease pool each time through that run loop. Releasing objects more frequently prevents your application's memory footprint from growing too large, which can lead to performance problems. As with any performance-related behavior though, you should measure the actual performance of your code and tune your use of autorelease pools appropriately.

For more information on memory management and autorelease pools, see *Memory Management Programming Guide*.

Setting Up an Exception Handler

If your application catches and handles exceptions, your thread code should be prepared to catch any exceptions that might occur. Although it is best to handle exceptions at the point where they might occur, failure to catch a thrown exception in a thread causes your application to exit. Installing a final try/catch in your thread entry routine allows you to catch any unknown exceptions and provide an appropriate response.

You can use either the C++ or Objective-C exception handling style when building your project in Xcode. For information about setting how to raise and catch exceptions in Objective-C, see *Exception Programming Topics*.

Setting Up a Run Loop

When writing code you want to run on a separate thread, you have two options. The first option is to write the code for a thread as one long task to be performed with little or no interruption, and have the thread exit when it finishes. The second option is put your thread into a loop and have it process requests dynamically as they arrive. The first option requires no special setup for your code; you just start doing the work you want to do. The second option, however, involves setting up your thread's run loop.

Mac OS X and iOS provide built-in support for implementing run loops in every thread. Cocoa, Carbon, and UIKit start the run loop of your application's main thread automatically, but if you create any secondary threads, you must configure the run loop and start it manually.

For information on using and configuring run loops, see "Run Loops" (page 33).

Terminating a Thread

The recommended way to exit a thread is to let it exit its entry point routine normally. Although Cocoa, POSIX, and Multiprocessing Services offer routines for killing threads directly, the use of such routines is strongly discouraged. Killing a thread prevents that thread from cleaning up after itself. Memory allocated by the thread could potentially be leaked and any other resources currently in use by the thread might not be cleaned up properly, creating potential problems later.

If you anticipate the need to terminate a thread in the middle of an operation, you should design your threads from the outset to respond to a cancel or exit message. For long-running operations, this might mean stopping work periodically and checking to see if such a message arrived. If a message does come in asking the thread to exit, the thread would then have the opportunity to perform any needed cleanup and exit gracefully; otherwise, it could simply go back to work and process the next chunk of data.

One way to respond to cancel messages is to use a run loop input source to receive such messages. Listing 2-3 shows the structure of how this code might look in your thread's main entry routine. (The example shows the main loop portion only and does not include the steps for setting up an autorelease pool or configuring the actual work to do.) The example installs a custom input source on the run loop that presumably can be messaged from another one of your threads; for information on setting up input sources, see "Configuring Run Loop Sources" (page 43). After performing a portion of the total amount of work, the thread runs the run loop briefly to see if a message arrived on the input source. If not, the run loop exits immediately and the loop continues with the next chunk of work. Because the handler does not have direct access to the exitNow local variable, the exit condition is communicated through a key-value pair in the thread dictionary.

Listing 2-3   Checking for an exit condition during a long job

```
- (void)threadMainRoutine
{
    BOOL moreWorkToDo = YES;
    BOOL exitNow = NO;
    NSRunLoop* runLoop = [NSRunLoop currentRunLoop];

    // Add the exitNow BOOL to the thread dictionary.
    NSMutableDictionary* threadDict = [[NSThread currentThread] threadDictionary];
    [threadDict setValue:[NSNumber numberWithBool:exitNow]
forKey:@"ThreadShouldExitNow"];

    // Install an input source.
```

```
    [self myInstallCustomInputSource];

    while (moreWorkToDo && !exitNow)
    {
        // Do one chunk of a larger body of work here.
        // Change the value of the moreWorkToDo Boolean when done.

        // Run the run loop but timeout immediately if the input source isn't
waiting to fire.
        [runLoop runUntilDate:[NSDate date]];

        // Check to see if an input source handler changed the exitNow value.
        exitNow = [[threadDict valueForKey:@"ThreadShouldExitNow"] boolValue];
    }
}
```

Run Loops

Run loops are part of the fundamental infrastructure associated with threads. A **run loop** is an event processing loop that you use to schedule work and coordinate the receipt of incoming events. The purpose of a run loop is to keep your thread busy when there is work to do and put your thread to sleep when there is none.

Run loop management is not entirely automatic. You must still design your thread's code to start the run loop at appropriate times and respond to incoming events. Both Cocoa and Core Foundation provide **run loop objects** to help you configure and manage your thread's run loop. Your application does not need to create these objects explicitly; each thread, including the application's main thread, has an associated run loop object. Only secondary threads need to run their run loop explicitly, however. In both Carbon and Cocoa applications, the main thread automatically sets up and runs its run loop as part of the general application startup process.

The following sections provide more information about run loops and how you configure them for your application. For additional information about run loop objects, see *NSRunLoop Class Reference* and *CFRunLoop Reference*.

Anatomy of a Run Loop

A run loop is very much like its name sounds. It is a loop your thread enters and uses to run event handlers in response to incoming events. Your code provides the control statements used to implement the actual loop portion of the run loop—in other words, your code provides the `while` or `for` loop that drives the run loop. Within your loop, you use a run loop object to "run" the event-processing code that receives events and calls the installed handlers.

A run loop receives events from two different types of sources. **Input sources** deliver asynchronous events, usually messages from another thread or from a different application. **Timer sources** deliver synchronous events, occurring at a scheduled time or repeating interval. Both types of source use an application-specific handler routine to process the event when it arrives.

Figure 3-1 shows the conceptual structure of a run loop and a variety of sources. The input sources deliver asynchronous events to the corresponding handlers and cause the `runUntilDate:` method (called on the thread's associated `NSRunLoop` object) to exit. Timer sources deliver events to their handler routines but do not cause the run loop to exit.

Figure 3-1   Structure of a run loop and its sources
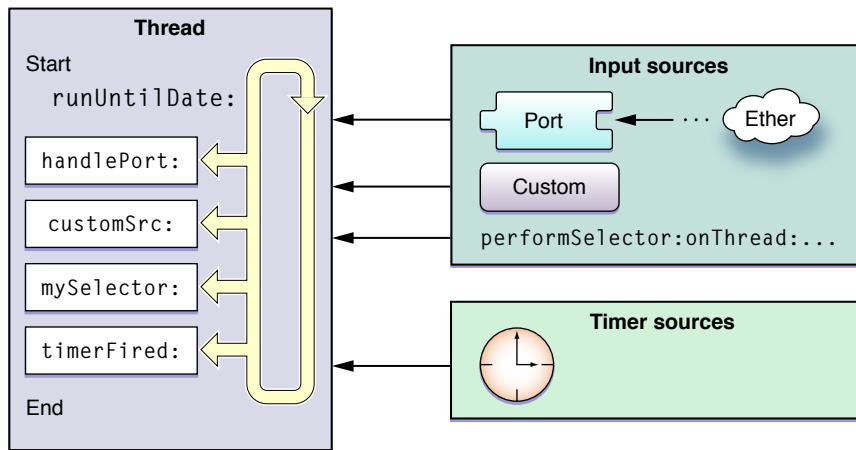
In addition to handling sources of input, run loops also generate notifications about the run loop's behavior. Registered **run-loop observers** can receive these notifications and use them to do additional processing on the thread. You use Core Foundation to install run-loop observers on your threads.

The following sections provide more information about the components of a run loop and the modes in which they operate. They also describe the notifications that are generated at different times during the handling of events.

Run Loop Modes

A **run loop mode** is a collection of input sources and timers to be monitored and a collection of run loop observers to be notified. Each time you run your run loop, you specify (either explicitly or implicitly) a particular "mode" in which to run. During that pass of the run loop, only sources associated with that mode are monitored and allowed to deliver their events. (Similarly, only observers associated with that mode are notified of the run loop's progress.) Sources associated with other modes hold on to any new events until subsequent passes through the loop in the appropriate mode.

In your code, you identify modes by name. Both Cocoa and Core Foundation define a default mode and several commonly used modes, along with strings for specifying those modes in your code. You can define custom modes by simply specifying a custom string for the mode name. Although the names you assign to custom modes are arbitrary, the contents of those modes are not. You must be sure to add one or more input sources, timers, or run-loop observers to any modes you create for them to be useful.

You use modes to filter out events from unwanted sources during a particular pass through your run loop. Most of the time, you will want to run your run loop in the system-defined "default" mode. A modal panel, however, might run in the "modal" mode. While in this mode, only sources relevant to the modal panel would deliver events to the thread. For secondary threads, you might use custom modes to prevent low-priority sources from delivering events during time-critical operations.

> **Note:** Modes discriminate based on the source of the event, not the type of the event. For example, you would not use modes to match only mouse-down events or only keyboard events. You could use modes to listen to a different set of ports, suspend timers temporarily, or otherwise change the sources and run loop observers currently being monitored.

Table 3-1 lists the standard modes defined by Cocoa and Core Foundation along with a description of when you use that mode. The name column lists the actual constants you use to specify the mode in your code.

Table 3-1   Predefined run loop modes

Mode	Name	Description
Default	`NSDefaultRunLoopMode` (Cocoa) `kCFRunLoopDefaultMode` (Core Foundation)	The default mode is the one used for most operations. Most of the time, you should use this mode to start your run loop and configure your input sources.
Connection	`NSConnectionReplyMode` (Cocoa)	Cocoa uses this mode in conjunction with `NSConnection` objects to monitor replies. You should rarely need to use this mode yourself.
Modal	`NSModalPanelRunLoopMode` (Cocoa)	Cocoa uses this mode to identify events intended for modal panels.
Event tracking	`NSEventTracking-RunLoopMode` (Cocoa)	Cocoa uses this mode to restrict incoming events during mouse-dragging loops and other sorts of user interface tracking loops.
Common modes	`NSRunLoopCommonModes` (Cocoa) `kCFRunLoopCommonModes` (Core Foundation)	This is a configurable group of commonly used modes. Associating an input source with this mode also associates it with each of the modes in the group. For Cocoa applications, this set includes the default, modal, and event tracking modes by default. Core Foundation includes just the default mode initially. You can add custom modes to the set using the `CFRunLoopAddCommonMode` function.

Input Sources

Input sources deliver events asynchronously to your threads. The source of the event depends on the type of the input source, which is generally one of two categories. Port-based input sources monitor your application's Mach ports. Custom input sources monitor custom sources of events. As far as your run loop is concerned, it should not matter whether an input source is port-based or custom. The system typically implements input sources of both types that you can use as is. The only difference between the two sources is how they are signaled. Port-based sources are signaled automatically by the kernel, and custom sources must be signaled manually from another thread.

When you create an input source, you assign it to one or more modes of your run loop. Modes affect which input sources are monitored at any given moment. Most of the time, you run the run loop in the default mode, but you can specify custom modes too. If an input source is not in the currently monitored mode, any events it generates are held until the run loop runs in the correct mode.

The following sections describe some of the input sources.

Port-Based Sources

Cocoa and Core Foundation provide built-in support for creating port-based input sources using port-related objects and functions. For example, in Cocoa, you never have to create an input source directly at all. You simply create a port object and use the methods of `NSPort` to add that port to the run loop. The port object handles the creation and configuration of the needed input source for you.

In Core Foundation, you must manually create both the port and its run loop source. In both cases, you use the functions associated with the port opaque type (`CFMachPortRef`, `CFMessagePortRef`, or `CFSocketRef`) to create the appropriate objects.

For examples of how to set up and configure custom port-based sources, see "Configuring a Port-Based Input Source" (page 49).

Custom Input Sources

To create a custom input source, you must use the functions associated with the `CFRunLoopSourceRef` opaque type in Core Foundation. You configure a custom input source using several callback functions. Core Foundation calls these functions at different points to configure the source, handle any incoming events, and tear down the source when it is removed from the run loop.

In addition to defining the behavior of the custom source when an event arrives, you must also define the event delivery mechanism. This part of the source runs on a separate thread and is responsible for providing the input source with its data and for signaling it when that data is ready for processing. The event delivery mechanism is up to you but need not be overly complex.

For an example of how to create a custom input source, see "Defining a Custom Input Source" (page 43). For reference information for custom input sources, see also *CFRunLoopSource Reference*.

Cocoa Perform Selector Sources

In addition to port-based sources, Cocoa defines a custom input source that allows you to perform a selector on any thread. Like a port-based source, perform selector requests are serialized on the target thread, alleviating many of the synchronization problems that might occur with multiple methods being run on one thread. Unlike a port-based source, a perform selector source removes itself from the run loop after it performs its selector.

> **Note:** Prior to Mac OS X v10.5, perform selector sources were used mostly to send messages to the main thread, but in Mac OS X v10.5 and later and in iOS, you can use them to send messages to any thread.

When performing a selector on another thread, the target thread must have an active run loop. For threads you create, this means waiting until your code explicitly starts the run loop. Because the main thread starts its own run loop, however, you can begin issuing calls on that thread as soon as the application calls the `applicationDidFinishLaunching:` method of the application delegate. The run loop processes all queued perform selector calls each time through the loop, rather than processing one during each loop iteration.

Table 3-2 lists the methods defined on `NSObject` that can be used to perform selectors on other threads. Because these methods are declared on `NSObject`, you can use them from any threads where you have access to Objective-C objects, including POSIX threads. These methods do not actually create a new thread to perform the selector.

Table 3-2   Performing selectors on other threads

Methods	Description
`performSelectorOnMainThread: withObject: waitUntilDone:` `performSelectorOnMainThread: withObject: waitUntilDone:modes:`	Performs the specified selector on the application's main thread during that thread's next run loop cycle. These methods give you the option of blocking the current thread until the selector is performed.
`performSelector: onThread:withObject: waitUntilDone:` `performSelector: onThread:withObject: waitUntilDone:modes:`	Performs the specified selector on any thread for which you have an `NSThread` object. These methods give you the option of blocking the current thread until the selector is performed.
`performSelector: withObject: afterDelay:` `performSelector: withObject: afterDelay:inModes:`	Performs the specified selector on the current thread during the next run loop cycle and after an optional delay period. Because it waits until the next run loop cycle to perform the selector, these methods provide an automatic mini delay from the currently executing code. Multiple queued selectors are performed one after another in the order they were queued.
`cancelPreviousPerformRequestsWithTarget:` `cancelPreviousPerformRequestsWithTarget: selector:object:`	Lets you cancel a message sent to the current thread using the `performSelector: withObject: afterDelay:` or `performSelector: withObject: afterDelay:inModes:` method.

For detailed information about each of these methods, see *NSObject Class Reference*.

Timer Sources

Timer sources deliver events synchronously to your threads at a preset time in the future. Timers are a way for a thread to notify itself to do something. For example, a search field could use a timer to initiate an automatic search once a certain amount of time has passed between successive key strokes from the user. The use of this delay time gives the user a chance to type as much of the desired search string as possible before beginning the search.

Although it generates time-based notifications, a timer is not a real-time mechanism. Like input sources, timers are associated with specific modes of your run loop. If a timer is not in the mode currently being monitored by the run loop, it does not fire until you run the run loop in one of the timer's supported modes. Similarly, if a timer fires when the run loop is in the middle of executing a handler routine, the timer waits until the next time through the run loop to invoke its handler routine. If the run loop is not running at all, the timer never fires.

You can configure timers to generate events only once or repeatedly. A repeating timer reschedules itself automatically based on the scheduled firing time, not the actual firing time. For example, if a timer is scheduled to fire at a particular time and every 5 seconds after that, the scheduled firing time will always fall on the original 5 second time intervals, even if the actual firing time gets delayed. If the firing time is delayed so much that it misses one or more of the scheduled firing times, the timer is fired only once for the missed time period. After firing for the missed period, the timer is rescheduled for the next scheduled firing time.

For more information on configuring timer sources, see "Configuring Timer Sources" (page 48). For reference information, see *NSTimer Class Reference* or *CFRunLoopTimer Reference*.

Run Loop Observers

In contrast to sources, which fire when an appropriate asynchronous or synchronous event occurs, run loop observers fire at special locations during the execution of the run loop itself. You might use run loop observers to prepare your thread to process a given event or to prepare the thread before it goes to sleep. You can associate run loop observers with the following events in your run loop:

■   The entrance to the run loop.

■   When the run loop is about to process a timer.

■   When the run loop is about to process an input source.

■   When the run loop is about to go to sleep.

■   When the run loop has woken up, but before it has processed the event that woke it up.

■   The exit from the run loop.

You can add run loop observers to both Cocoa and Carbon applications, but to define one and add it to your run loop, you must use Core Foundation. To create a run loop observer, you create a new instance of the `CFRunLoopObserverRef` opaque type. This type keeps track of your custom callback function and the activities in which it is interested.

Similar to timers, run-loop observers can be used once or repeatedly. A one-shot observer removes itself from the run loop after it fires, while a repeating observer remains attached. You specify whether an observer runs once or repeatedly when you create it.

For an example of how to create a run-loop observer, see "Configuring the Run Loop" (page 40). For reference information, see *CFRunLoopObserver Reference*.

The Run Loop Sequence of Events

Each time you run it, your thread's run loop processes pending events and generates notifications for any attached observers. The order in which it does this is very specific and is as follows:

1.   Notify observers that the run loop has been entered.

2.   Notify observers that any ready timers are about to fire.

3.   Notify observers that any input sources that are not port based are about to fire.

4.   Fire any non-port-based input sources that are ready to fire.

5. If a port-based input source is ready and waiting to fire, process the event immediately. Go to step 9.

6. Notify observers that the thread is about to sleep.

7. Put the thread to sleep until one of the following events occurs:

    ■ An event arrives for a port-based input source.

    ■ A timer fires.

    ■ The timeout value set for the run loop expires.

    ■ The run loop is explicitly woken up.

8. Notify observers that the thread just woke up.

9. Process the pending event.

    ■ If a user-defined timer fired, process the timer event and restart the loop. Go to step 2.

    ■ If an input source fired, deliver the event.

    ■ If the run loop was explicitly woken up but has not yet timed out, restart the loop. Go to step 2.

10. Notify observers that the run loop has exited.

Because observer notifications for timer and input sources are delivered before those events actually occur, there may be a gap between the time of the notifications and the time of the actual events. If the timing between these events is critical, you can use the sleep and awake-from-sleep notifications to help you correlate the timing between the actual events.

Because timers and other periodic events are delivered when you run the run loop, circumventing that loop disrupts the delivery of those events. The typical example of this behavior occurs whenever you implement a mouse-tracking routine by entering a loop and repeatedly requesting events from the application. Because your code is grabbing events directly, rather than letting the application dispatch those events normally, active timers would be unable to fire until after your mouse-tracking routine exited and returned control to the application.

A run loop can be explicitly woken up using the run loop object. Other events may also cause the run loop to be woken up. For example, adding another non-port-based input source wakes up the run loop so that the input source can be processed immediately, rather than waiting until some other event occurs.

When Would You Use a Run Loop?

The only time you need to run a run loop explicitly is when you create secondary threads for your application. The run loop for your application's main thread is a crucial piece of infrastructure. As a result, both Cocoa and Carbon provide the code for running the main application loop and start that loop automatically. The `run` method of `UIApplication` in iOS (or `NSApplication` in Mac OS X) starts an application's main loop as part of the normal startup sequence. Similarly, the `RunApplicationEventLoop` function starts the main loop for Carbon applications. If you use the Xcode template projects to create your application, you should never have to call these routines explicitly.

For secondary threads, you need to decide whether a run loop is necessary, and if it is, configure and start it yourself. You do not need to start a thread's run loop in all cases. For example, if you use a thread to perform some long-running and predetermined task, you can probably avoid starting the run loop. Run loops are intended for situations where you want more interactivity with the thread. For example, you need to start a run loop if you plan to do any of the following:

- Use ports or custom input sources to communicate with other threads.
- Use timers on the thread.
- Use any of the `performSelector…` methods in a Cocoa application.
- Keep the thread around to perform periodic tasks.

If you do choose to use a run loop, the configuration and setup is straightforward. As with all threaded programming though, you should have a plan for exiting your secondary threads in appropriate situations. It is always better to end a thread cleanly by letting it exit than to force it to terminate. Information on how to configure and exit a run loop is described in "Using Run Loop Objects" (page 40).

Using Run Loop Objects

A run loop object provides the main interface for adding input sources, timers, and run-loop observers to your run loop and then running it. Every thread has a single run loop object associated with it. In Cocoa, this object is an instance of the `NSRunLoop` class. In a Carbon or BSD application, it is a pointer to a `CFRunLoopRef` opaque type.

Getting a Run Loop Object

To get the run loop for the current thread, you use one of the following:

- In a Cocoa application, use the `currentRunLoop` class method of `NSRunLoop` to retrieve an `NSRunLoop` object.
- Use the `CFRunLoopGetCurrent` function.

Although they are not toll-free bridged types, you can get a `CFRunLoopRef` opaque type from an `NSRunLoop` object when needed. The `NSRunLoop` class defines a `getCFRunLoop` method that returns a `CFRunLoopRef` type that you can pass to Core Foundation routines. Because both objects refer to the same run loop, you can intermix calls to the `NSRunLoop` object and `CFRunLoopRef` opaque type as needed.

Configuring the Run Loop

Before you run a run loop on a secondary thread, you must add at least one input source or timer to it. If a run loop does not have any sources to monitor, it exits immediately when you try to run it. For examples of how to add sources to a run loop, see "Configuring Run Loop Sources" (page 43).

In addition to installing sources, you can also install run loop observers and use them to detect different execution stages of the run loop. To install a run loop observer, you create a `CFRunLoopObserverRef` opaque type and use the `CFRunLoopAddObserver` function to add it to your run loop. Run loop observers must be created using Core Foundation, even for Cocoa applications.

Listing 3-1 shows the main routine for a thread that attaches a run loop observer to its run loop. The purpose of the example is to show you how to create a run loop observer, so the code simply sets up a run loop observer to monitor all run loop activities. The basic handler routine (not shown) simply logs the run loop activity as it processes the timer requests.

Listing 3-1   Creating a run loop observer

```
- (void)threadMain
{
    // The application uses garbage collection, so no autorelease pool is needed.
    NSRunLoop* myRunLoop = [NSRunLoop currentRunLoop];

    // Create a run loop observer and attach it to the run loop.
    CFRunLoopObserverContext  context = {0, self, NULL, NULL, NULL};
    CFRunLoopObserverRef    observer =
CFRunLoopObserverCreate(kCFAllocatorDefault,
            kCFRunLoopAllActivities, YES, 0, &myRunLoopObserver, &context);

    if (observer)
    {
        CFRunLoopRef    cfLoop = [myRunLoop getCFRunLoop];
        CFRunLoopAddObserver(cfLoop, observer, kCFRunLoopDefaultMode);
    }

    // Create and schedule the timer.
    [NSTimer scheduledTimerWithTimeInterval:0.1 target:self
                selector:@selector(doFireTimer:) userInfo:nil repeats:YES];

    NSInteger    loopCount = 10;
    do
    {
        // Run the run loop 10 times to let the timer fire.
        [myRunLoop runUntilDate:[NSDate dateWithTimeIntervalSinceNow:1]];
        loopCount--;
    }
    while (loopCount);
}
```

When configuring the run loop for a long-lived thread, it is better to add at least one input source to receive messages. Although you can enter the run loop with only a timer attached, once the timer fires, it is typically invalidated, which would then cause the run loop to exit. Attaching a repeating timer could keep the run loop running over a longer period of time, but would involve firing the timer periodically to wake your thread, which is effectively another form of polling. By contrast, an input source waits for an event to happen, keeping your thread asleep until it does.

Starting the Run Loop

Starting the run loop is necessary only for the secondary threads in your application. A run loop must have at least one input source or timer to monitor. If one is not attached, the run loop exits immediately.

There are several ways to start the run loop, including the following:

- Unconditionally

- With a set time limit

- In a particular mode

Entering your run loop unconditionally is the simplest option, but it is also the least desirable. Running your run loop unconditionally puts the thread into a permanent loop, which gives you very little control over the run loop itself. You can add and remove input sources and timers, but the only way to stop the run loop is to kill it. There is also no way to run the run loop in a custom mode.

Instead of running a run loop unconditionally, it is better to run the run loop with a timeout value. When you use a timeout value, the run loop runs until an event arrives or the allotted time expires. If an event arrives, that event is dispatched to a handler for processing and then the run loop exits. Your code can then restart the run loop to handle the next event. If the allotted time expires instead, you can simply restart the run loop or use the time to do any needed housekeeping.

In addition to a timeout value, you can also run your run loop using a specific mode. Modes and timeout values are not mutually exclusive and can both be used when starting a run loop. Modes limit the types of sources that deliver events to the run loop and are described in more detail in "Run Loop Modes" (page 34).

Listing 3-2 shows a skeleton version of a thread's main entry routine. The key portion of this example shows the basic structure of a run loop. In essence, you add your input sources and timers to the run loop and then repeatedly call one of the routines to start the run loop. Each time the the run loop routine returns, you check to see if any conditions have arisen that might warrant exiting the thread. The example uses the Core Foundation run loop routines so that it can check the return result and determine why the run loop exited. You could also use the methods of the NSRunLoop class to run the run loop in a similar manner if you are using Cocoa and do not need to check the return value. (For an example of a run loop that calls methods of the NSRunLoop class, see Listing 3-14 (page 51).)

Listing 3-2   Running a run loop

```
- (void)skeletonThreadMain
{
    // Set up an autorelease pool here if not using garbage collection.
    BOOL done = NO;

    // Add your sources or timers to the run loop and do any other setup.

    do
    {
        // Start the run loop but return after each source is handled.
        SInt32   result = CFRunLoopRunInMode(kCFRunLoopDefaultMode, 10, YES);

        // If a source explicitly stopped the run loop, or if there are no
        // sources or timers, go ahead and exit.
        if ((result == kCFRunLoopRunStopped) || (result == kCFRunLoopRunFinished))
            done = YES;

        // Check for any other exit conditions here and set the
        // done variable as needed.
    }
    while (!done);

    // Clean up code here. Be sure to release any allocated autorelease pools.
```

```
}
```

It is possible to run a run loop recursively. In other words, you can call `CFRunLoopRun`, `CFRunLoopRunInMode`, or any of the `NSRunLoop` methods for starting the run loop from within the handler routine of an input source or timer. When doing so, you can use any mode you want to run the nested run loop, including the mode in use by the outer run loop.

Exiting the Run Loop

There are two ways to make a run loop exit before it has processed an event:

- Configure the run loop to run with a timeout value.

- Tell the run loop to stop.

Using a timeout value is certainly preferred, if you can manage it. Specifying a timeout value lets the run loop finish all of its normal processing, including delivering notifications to run loop observers, before exiting.

Stopping the run loop explicitly with the `CFRunLoopStop` function produces a result similar to a timeout. The run loop sends out any remaining run-loop notifications and then exits. The difference is that you can use this technique on run loops you started unconditionally.

Although removing a run loop's input sources and timers may also cause the run loop to exit, this is not a reliable way to stop a run loop. Some system routines add input sources to a run loop to handle needed events. Because your code might not be aware of these input sources, it would be unable to remove them, which would prevent the run loop from exiting.

Thread Safety and Run Loop Objects

Thread safety varies depending on which API you are using to manipulate your run loop. The functions in Core Foundation are generally thread-safe and can be called from any thread. If you are performing operations that alter the configuration of the run loop, however, it is still good practice to do so from the thread that owns the run loop whenever possible.

The Cocoa `NSRunLoop` class is not as inherently thread safe as its Core Foundation counterpart. If you are using the `NSRunLoop` class to modify your run loop, you should do so only from the same thread that owns that run loop. Adding an input source or timer to a run loop belonging to a different thread could cause your code to crash or behave in an unexpected way.

Configuring Run Loop Sources

The following sections show examples of how to set up different types of input sources in both Cocoa and Core Foundation.

Defining a Custom Input Source

Creating a custom input source involves defining the following:

■   The information you want your input source to process.

■   A scheduler routine to let interested clients know how to contact your input source.

■   A handler routine to perform requests sent by any clients.

■   A cancellation routine to invalidate your input source.

Because you create a custom input source to process custom information, the actual configuration is designed to be flexible. The scheduler, handler, and cancellation routines are the key routines you almost always need for your custom input source. Most of the rest of the input source behavior, however, happens outside of those handler routines. For example, it is up to you to define the mechanism for passing data to your input source and for communicating the presence of your input source to other threads.

Figure 3-2 shows a sample configuration of a custom input source. In this example, the application's main thread maintains references to the input source, the custom command buffer for that input source, and the run loop on which the input source is installed. When the main thread has a task it wants to hand off to the worker thread, it posts a command to the command buffer along with any information needed by the worker thread to start the task. (Because both the main thread and the input source of the worker thread have access to the command buffer, that access must be synchronized.) Once the command is posted, the main thread signals the input source and wakes up the worker thread's run loop. Upon receiving the wake up command, the run loop calls the handler for the input source, which processes the commands found in the command buffer.

Figure 3-2   Operating a custom input source
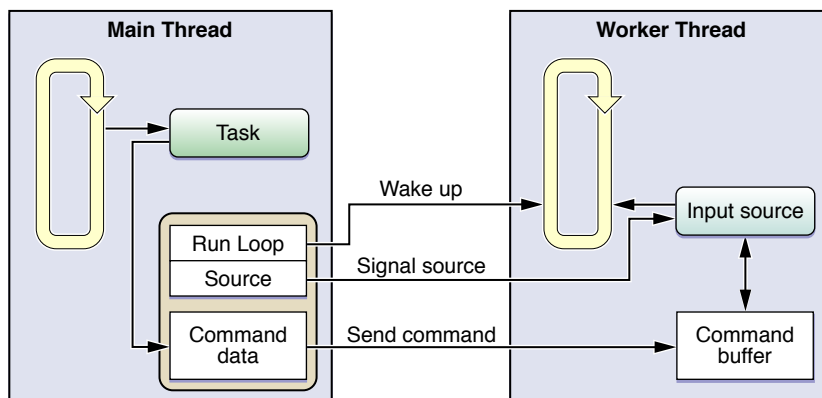
The following sections explain the implementation of the custom input source from the preceding figure and show the key code you would need to implement.

Defining the Input Source

Defining a custom input source requires the use of Core Foundation routines to configure your run loop source and attach it to a run loop. Although the basic handlers are C-based functions, that does not preclude you from writing wrappers for those functions and using Objective-C or C++ to implement the body of your code.

The input source introduced in Figure 3-2 (page 44) uses an Objective-C object to manage a command buffer and coordinate with the run loop. Listing 3-3 shows the definition of this object. The `RunLoopSource` object manages a command buffer and uses that buffer to receive messages from other threads. This listing also shows the definition of the `RunLoopContext` object, which is really just a container object used to pass a `RunLoopSource` object and a run loop reference to the application's main thread.

Listing 3-3   The custom input source object definition

```
@interface RunLoopSource : NSObject
{
    CFRunLoopSourceRef runLoopSource;
    NSMutableArray* commands;
}

- (id)init;
- (void)addToCurrentRunLoop;
- (void)invalidate;

// Handler method
- (void)sourceFired;

// Client interface for registering commands to process
- (void)addCommand:(NSInteger)command withData:(id)data;
- (void)fireAllCommandsOnRunLoop:(CFRunLoopRef)runloop;

@end


// These are the CFRunLoopSourceRef callback functions.
void RunLoopSourceScheduleRoutine (void *info, CFRunLoopRef rl, CFStringRef
mode);
void RunLoopSourcePerformRoutine (void *info);
void RunLoopSourceCancelRoutine (void *info, CFRunLoopRef rl, CFStringRef mode);

// RunLoopContext is a container object used during registration of the input
source.
@interface RunLoopContext : NSObject
{
    CFRunLoopRef        runLoop;
    RunLoopSource*       source;
}
@property (readonly) CFRunLoopRef runLoop;
@property (readonly) RunLoopSource* source;

- (id)initWithSource:(RunLoopSource*)src andLoop:(CFRunLoopRef)loop;
@end
```

Although the Objective-C code manages the custom data of the input source, attaching the input source to a run loop requires C-based callback functions. The first of these functions is called when you actually attach the run loop source to your run loop, and is shown in Listing 3-4. Because this input source has only one client (the main thread), it uses the scheduler function to send a message to register itself with the application delegate on that thread. When the delegate wants to communicate with the input source, it uses the information in `RunLoopContext` object to do so.

Listing 3-4   Scheduling a run loop source

```
void RunLoopSourceScheduleRoutine (void *info, CFRunLoopRef rl, CFStringRef
mode)
```

```
{
    RunLoopSource* obj = (RunLoopSource*)info;
    AppDelegate*   del = [AppDelegate sharedAppDelegate];
    RunLoopContext* theContext = [[RunLoopContext alloc] initWithSource:obj
andLoop:rl];

    [del performSelectorOnMainThread:@selector(registerSource:)
                          withObject:theContext waitUntilDone:NO];
}
```

One of the most important callback routines is the one used to process custom data when your input source is signaled. Listing 3-5 shows the perform callback routine associated with the `RunLoopSource` object. This function simply forwards the request to do the work to the `sourceFired` method, which then processes any commands present in the command buffer.

Listing 3-5   Performing work in the input source

```
void RunLoopSourcePerformRoutine (void *info)
{
    RunLoopSource*  obj = (RunLoopSource*)info;
    [obj sourceFired];
}
```

If you ever remove your input source from its run loop using the `CFRunLoopSourceInvalidate` function, the system calls your input source's cancellation routine. You can use this routine to notify clients that your input source is no longer valid and that they should remove any references to it. Listing 3-6 shows the cancellation callback routine registered with the `RunLoopSource` object. This function sends another `RunLoopContext` object to the application delegate, but this time asks the delegate to remove references to the run loop source.

Listing 3-6   Invalidating an input source

```
void RunLoopSourceCancelRoutine (void *info, CFRunLoopRef rl, CFStringRef mode)
{
    RunLoopSource* obj = (RunLoopSource*)info;
    AppDelegate* del = [AppDelegate sharedAppDelegate];
    RunLoopContext* theContext = [[RunLoopContext alloc] initWithSource:obj
andLoop:rl];

    [del performSelectorOnMainThread:@selector(removeSource:)
                          withObject:theContext waitUntilDone:YES];
}
```

> **Note:** The code for the application delegate's `registerSource:` and `removeSource:` methods is shown in "Coordinating with Clients of the Input Source" (page 47).

Installing the Input Source on the Run Loop

Listing 3-7 shows the `init` and `addToCurrentRunLoop` methods of the `RunLoopSource` class. The `init` method creates the `CFRunLoopSourceRef` opaque type that must actually be attached to the run loop. It passes the `RunLoopSource` object itself as the contextual information so that the callback routines have a pointer to the object. Installation of the input source does not occur until the worker thread invokes the `addToCurrentRunLoop` method, at which point the `RunLoopSourceScheduleRoutine` callback function is called. Once the input source is added to the run loop, the thread can run its run loop to wait on it.

Listing 3-7   Installing the run loop source

```
- (id)init
{
    CFRunLoopSourceContext    context = {0, self, NULL, NULL, NULL, NULL, NULL,
                                         &RunLoopSourceScheduleRoutine,
                                         RunLoopSourceCancelRoutine,
                                         RunLoopSourcePerformRoutine};

    runLoopSource = CFRunLoopSourceCreate(NULL, 0, &context);
    commands = [[NSMutableArray alloc] init];

    return self;
}

- (void)addToCurrentRunLoop
{
    CFRunLoopRef runLoop = CFRunLoopGetCurrent();
    CFRunLoopAddSource(runLoop, runLoopSource, kCFRunLoopDefaultMode);
}
```

Coordinating with Clients of the Input Source

For your input source to be useful, you need to manipulate it and signal it from another thread. The whole point of an input source is to put its associated thread to sleep until there is something to do. That fact necessitates having other threads in your application know about the input source and have a way to communicate with it.

One way to notify clients about your input source is to send out registration requests when your input source is first installed on its run loop. You can register your input source with as many clients as you want, or you can simply register it with some central agency that then vends your input source to interested clients. Listing 3-8 shows the registration method defined by the application delegate and invoked when the `RunLoopSource` object's scheduler function is called. This method receives the `RunLoopContext` object provided by the `RunLoopSource` object and adds it to its list of sources. This listing also shows the routine used to unregister the input source when it is removed from its run loop.

Listing 3-8   Registering and removing an input source with the application delegate

```
- (void)registerSource:(RunLoopContext*)sourceInfo;
{
    [sourcesToPing addObject:sourceInfo];
}

- (void)removeSource:(RunLoopContext*)sourceInfo
{
    id    objToRemove = nil;

    for (RunLoopContext* context in sourcesToPing)
    {
        if ([context isEqual:sourceInfo])
        {
            objToRemove = context;
            break;
        }
    }
```

```
    if (objToRemove)
        [sourcesToPing removeObject:objToRemove];
}
```

> **Note:** The callback functions that call the methods in the preceding listing are shown in Listing 3-4 (page 45) and Listing 3-6 (page 46).

Signaling the Input Source

After it hands off its data to the input source, a client must signal the source and wake up its run loop. Signaling the source lets the run loop know that the source is ready to be processed. And because the thread might be asleep when the signal occurs, you should always wake up the run loop explicitly. Failing to do so might result in a delay in processing the input source.

Listing 3-9 shows the `fireCommandsOnRunLoop` method of the `RunLoopSource` object. Clients invoke this method when they are ready for the source to process the commands they added to the buffer.

Listing 3-9   Waking up the run loop

```
- (void)fireCommandsOnRunLoop:(CFRunLoopRef)runloop
{
    CFRunLoopSourceSignal(runLoopSource);
    CFRunLoopWakeUp(runloop);
}
```

> **Note:** You should never try to handle a `SIGHUP` or other type of process-level signal by messaging a custom input source. The Core Foundation functions for waking up the run loop are not signal safe and should not be used inside your application's signal handler routines. For more information about signal handler routines, see the `sigaction` man page.

Configuring Timer Sources

To create a timer source, all you have to do is create a timer object and schedule it on your run loop. In Cocoa, you use the `NSTimer` class to create new timer objects, and in Core Foundation you use the `CFRunLoopTimerRef` opaque type. Internally, the `NSTimer` class is simply an extension of Core Foundation that provides some convenience features, like the ability to create and schedule a timer using the same method.

In Cocoa, you can create and schedule a timer all at once using either of these class methods:

■   `scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:`

■   `scheduledTimerWithTimeInterval:invocation:repeats:`

These methods create the timer and add it to the current thread's run loop in the default mode (`NSDefaultRunLoopMode`). You can also schedule a timer manually if you want by creating your `NSTimer` object and then adding it to the run loop using the `addTimer:forMode:` method of `NSRunLoop`. Both techniques do basically the same thing but give you different levels of control over the timer's configuration. For example, if you create the timer and add it to the run loop manually, you can do so using a mode other

than the default mode. Listing 3-10 shows how to create timers using both techniques. The first timer has an initial delay of 1 second but then fires regularly every 0.1 seconds after that. The second timer begins firing after an initial 0.2 second delay and then fires every 0.2 seconds after that.

Listing 3-10   Creating and scheduling timers using NSTimer

```
NSRunLoop* myRunLoop = [NSRunLoop currentRunLoop];

// Create and schedule the first timer.
NSDate* futureDate = [NSDate dateWithTimeIntervalSinceNow:1.0];
NSTimer* myTimer = [[NSTimer alloc] initWithFireDate:futureDate
                        interval:0.1
                        target:self
                        selector:@selector(myDoFireTimer1:)
                        userInfo:nil
                        repeats:YES];
[myRunLoop addTimer:myTimer forMode:NSDefaultRunLoopMode];

// Create and schedule the second timer.
[NSTimer scheduledTimerWithTimeInterval:0.2
                        target:self
                        selector:@selector(myDoFireTimer2:)
                        userInfo:nil
                        repeats:YES];
```

Listing 3-11 shows the code needed to configure a timer using Core Foundation functions. Although this example does not pass any user-defined information in the context structure, you could use this structure to pass around any custom data you needed for your timer. For more information about the contents of this structure, see its description in *CFRunLoopTimer Reference*.

Listing 3-11   Creating and scheduling a timer using Core Foundation

```
CFRunLoopRef runLoop = CFRunLoopGetCurrent();
CFRunLoopTimerContext context = {0, NULL, NULL, NULL, NULL};
CFRunLoopTimerRef timer = CFRunLoopTimerCreate(kCFAllocatorDefault, 0.1, 0.3,
0, 0,
                                    &myCFTimerCallback, &context);

CFRunLoopAddTimer(runLoop, timer, kCFRunLoopCommonModes);
```

Configuring a Port-Based Input Source

Both Cocoa and Core Foundation provide port-based objects for communicating between threads or between processes. The following sections show you how to set up port communication using several different types of ports.

Configuring an NSMachPort Object

To establish a local connection with an `NSMachPort` object, you create the port object and add it to your primary thread's run loop. When launching your secondary thread, you pass the same object to your thread's entry-point function. The secondary thread can use the same object to send messages back to your primary thread.

Implementing the Main Thread Code

Listing 3-12 shows the primary thread code for launching a secondary worker thread. Because the Cocoa framework performs many of the intervening steps for configuring the port and run loop, the `launchThread` method is noticeably shorter than its Core Foundation equivalent (Listing 3-17 (page 52)); however, the behavior of the two is nearly identical. One difference is that instead of sending the name of the local port to the worker thread, this method sends the `NSPort` object directly.

Listing 3-12   Main thread launch method

```
- (void)launchThread
{
    NSPort* myPort = [NSMachPort port];
    if (myPort)
    {
        // This class handles incoming port messages.
        [myPort setDelegate:self];

        // Install the port as an input source on the current run loop.
        [[NSRunLoop currentRunLoop] addPort:myPort forMode:NSDefaultRunLoopMode];

        // Detach the thread. Let the worker release the port.
        [NSThread detachNewThreadSelector:@selector(LaunchThreadWithPort:)
                toTarget:[MyWorkerClass class] withObject:myPort];
    }
}
```

In order to set up a two-way communications channel between your threads, you might want to have the worker thread send its own local port to your main thread in a check-in message. Receiving the check-in message lets your main thread know that all went well in launching the second thread and also gives you a way to send further messages to that thread.

Listing 3-13 shows the `handlePortMessage:` method for the primary thread. This method is called when data arrives on the thread's own local port. When a check-in message arrives, the method retrieves the port for the secondary thread directly from the port message and saves it for later use.

Listing 3-13   Handling Mach port messages

```
#define kCheckinMessage 100

// Handle responses from the worker thread.
- (void)handlePortMessage:(NSPortMessage *)portMessage
{
    unsigned int message = [portMessage msgid];
    NSPort* distantPort = nil;

    if (message == kCheckinMessage)
    {
        // Get the worker thread's communications port.
        distantPort = [portMessage sendPort];

        // Retain and save the worker port for later use.
        [self storeDistantPort:distantPort];
    }
    else
    {
        // Handle other messages.
```

```
    }
}
```

Implementing the Secondary Thread Code

For the secondary worker thread, you must configure the thread and use the specified port to communicate information back to the primary thread.

Listing 3-14 shows the code for setting up the worker thread. After creating an autorelease pool for the thread, the method creates a worker object to drive the thread execution. The worker object's `sendCheckinMessage:` method (shown in Listing 3-15 (page 51)) creates a local port for the worker thread and sends a check-in message back to the main thread.

Listing 3-14   Launching the worker thread using Mach ports

```
+(void)LaunchThreadWithPort:(id)inData
{
    NSAutoreleasePool*  pool = [[NSAutoreleasePool alloc] init];

    // Set up the connection between this thread and the main thread.
    NSPort* distantPort = (NSPort*)inData;

    MyWorkerClass*  workerObj = [[self alloc] init];
    [workerObj sendCheckinMessage:distantPort];
    [distantPort release];

    // Let the run loop process things.
    do
    {
        [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode
                            beforeDate:[NSDate distantFuture]];
    }
    while (![workerObj shouldExit]);

    [workerObj release];
    [pool release];
}
```

When using `NSMachPort`, local and remote threads can use the same port object for one-way communication between the threads. In other words, the local port object created by one thread becomes the remote port object for the other thread.

Listing 3-15 shows the check-in routine of the secondary thread. This method sets up its own local port for future communication and then sends a check-in message back to the main thread. The method uses the port object received in the `LaunchThreadWithPort:` method as the target of the message.

Listing 3-15   Sending the check-in message using Mach ports

```
// Worker thread check-in method
- (void)sendCheckinMessage:(NSPort*)outPort
{
    // Retain and save the remote port for future use.
    [self setRemotePort:outPort];

    // Create and configure the worker thread port.
    NSPort* myPort = [NSMachPort port];
```

```
    [myPort setDelegate:self];
    [[NSRunLoop currentRunLoop] addPort:myPort forMode:NSDefaultRunLoopMode];

    // Create the check-in message.
    NSPortMessage* messageObj = [[NSPortMessage alloc] initWithSendPort:outPort
                                      receivePort:myPort components:nil];

    if (messageObj)
    {
        // Finish configuring the message and send it immediately.
        [messageObj setMsgid:kCheckinMessage];
        [messageObj sendBeforeDate:[NSDate date]];
    }
}
```

Configuring an NSMessagePort Object

To establish a local connection with an `NSMessagePort` object, you cannot simply pass port objects between threads. Remote message ports must be acquired by name. Making this possible in Cocoa requires registering your local port with a specific name and then passing that name to the remote thread so that it can obtain an appropriate port object for communication. Listing 3-16 shows the port creation and registration process in cases where you want to use message ports.

Listing 3-16   Registering a message port

```
NSPort* localPort = [[[NSMessagePort alloc] init] retain];

// Configure the object and add it to the current run loop.
[localPort setDelegate:self];
[[NSRunLoop currentRunLoop] addPort:localPort forMode:NSDefaultRunLoopMode];

// Register the port using a specific name. The name must be unique.
NSString* localPortName = [NSString stringWithFormat:@"MyPortName"];
[[NSMessagePortNameServer sharedInstance] registerPort:localPort
                name:localPortName];
```

Configuring a Port-Based Input Source in Core Foundation

This section shows how to set up a two-way communications channel between your application's main thread and a worker thread using Core Foundation.

Listing 3-17 shows the code called by the application's main thread to launch the worker thread. The first thing the code does is set up a `CFMessagePortRef` opaque type to listen for messages from worker threads. The worker thread needs the name of the port to make the connection, so that string value is delivered to the entry point function of the worker thread. Port names should generally be unique within the current user context; otherwise, you might run into conflicts.

Listing 3-17   Attaching a Core Foundation message port to a new thread

```
#define kThreadStackSize        (8 *4096)

OSStatus MySpawnThread()
{
    // Create a local port for receiving responses.
    CFStringRef myPortName;
```

```
    CFMessagePortRef myPort;
    CFRunLoopSourceRef rlSource;
    CFMessagePortContext context = {0, NULL, NULL, NULL, NULL};
    Boolean shouldFreeInfo;

    // Create a string with the port name.
    myPortName = CFStringCreateWithFormat(NULL, NULL,
CFSTR("com.myapp.MainThread"));

    // Create the port.
    myPort = CFMessagePortCreateLocal(NULL,
                myPortName,
                &MainThreadResponseHandler,
                &context,
                &shouldFreeInfo);

    if (myPort != NULL)
    {
        // The port was successfully created.
        // Now create a run loop source for it.
        rlSource = CFMessagePortCreateRunLoopSource(NULL, myPort, 0);

        if (rlSource)
        {
            // Add the source to the current run loop.
            CFRunLoopAddSource(CFRunLoopGetCurrent(), rlSource,
kCFRunLoopDefaultMode);

            // Once installed, these can be freed.
            CFRelease(myPort);
            CFRelease(rlSource);
        }
    }

    // Create the thread and continue processing.
    MPTaskID        taskID;
    return(MPCreateTask(&ServerThreadEntryPoint,
                (void*)myPortName,
                kThreadStackSize,
                NULL,
                NULL,
                NULL,
                0,
                &taskID));
}
```

With the port installed and the thread launched, the main thread can continue its regular execution while it waits for the thread to check in. When the check-in message arrives, it is dispatched to the main thread's `MainThreadResponseHandler` function, shown in Listing 3-18. This function extracts the port name for the worker thread and creates a conduit for future communication.

Listing 3-18   Receiving the checkin message

```
#define kCheckinMessage 100

// Main thread port message handler
CFDataRef MainThreadResponseHandler(CFMessagePortRef local,
                SInt32 msgid,
```

```
                    CFDataRef data,
                    void* info)
{
    if (msgid == kCheckinMessage)
    {
        CFMessagePortRef messagePort;
        CFStringRef threadPortName;
        CFIndex bufferLength = CFDataGetLength(data);
        UInt8* buffer = CFAllocatorAllocate(NULL, bufferLength, 0);

        CFDataGetBytes(data, CFRangeMake(0, bufferLength), buffer);
        threadPortName = CFStringCreateWithBytes (NULL, buffer, bufferLength,
kCFStringEncodingASCII, FALSE);

        // You must obtain a remote message port by name.
        messagePort = CFMessagePortCreateRemote(NULL,
(CFStringRef)threadPortName);

        if (messagePort)
        {
            // Retain and save the thread's comm port for future reference.
            AddPortToListOfActiveThreads(messagePort);

            // Since the port is retained by the previous function, release
            // it here.
            CFRelease(messagePort);
        }

        // Clean up.
        CFRelease(threadPortName);
        CFAllocatorDeallocate(NULL, buffer);
    }
    else
    {
        // Process other messages.
    }

    return NULL;
}
```

With the main thread configured, the only thing remaining is for the newly created worker thread to create its own port and check in. Listing 3-19 shows the entry point function for the worker thread. The function extracts the main thread's port name and uses it to create a remote connection back to the main thread. The function then creates a local port for itself, installs the port on the thread's run loop, and sends a check-in message to the main thread that includes the local port name.

Listing 3-19   Setting up the thread structures

```
OSStatus ServerThreadEntryPoint(void* param)
{
    // Create the remote port to the main thread.
    CFMessagePortRef mainThreadPort;
    CFStringRef portName = (CFStringRef)param;

    mainThreadPort = CFMessagePortCreateRemote(NULL, portName);

    // Free the string that was passed in param.
    CFRelease(portName);
```

```
    // Create a port for the worker thread.
    CFStringRef myPortName = CFStringCreateWithFormat(NULL, NULL,
CFSTR("com.MyApp.Thread-%d"), MPCurrentTaskID());

    // Store the port in this thread's context info for later reference.
    CFMessagePortContext context = {0, mainThreadPort, NULL, NULL, NULL};
    Boolean shouldFreeInfo;
    Boolean shouldAbort = TRUE;

    CFMessagePortRef myPort = CFMessagePortCreateLocal(NULL,
                myPortName,
                &ProcessClientRequest,
                &context,
                &shouldFreeInfo);

    if (shouldFreeInfo)
    {
        // Couldn't create a local port, so kill the thread.
        MPExit(0);
    }

    CFRunLoopSourceRef rlSource = CFMessagePortCreateRunLoopSource(NULL, myPort,
0);
    if (!rlSource)
    {
        // Couldn't create a local port, so kill the thread.
        MPExit(0);
    }

    // Add the source to the current run loop.
    CFRunLoopAddSource(CFRunLoopGetCurrent(), rlSource, kCFRunLoopDefaultMode);

    // Once installed, these can be freed.
    CFRelease(myPort);
    CFRelease(rlSource);

    // Package up the port name and send the check-in message.
    CFDataRef returnData = nil;
    CFDataRef outData;
    CFIndex stringLength = CFStringGetLength(myPortName);
    UInt8* buffer = CFAllocatorAllocate(NULL, stringLength, 0);

    CFStringGetBytes(myPortName,
                CFRangeMake(0,stringLength),
                kCFStringEncodingASCII,
                0,
                FALSE,
                buffer,
                stringLength,
                NULL);

    outData = CFDataCreate(NULL, buffer, stringLength);

    CFMessagePortSendRequest(mainThreadPort, kCheckinMessage, outData, 0.1, 0.0,
NULL, NULL);

    // Clean up thread data structures.
```

```
    CFRelease(outData);
    CFAllocatorDeallocate(NULL, buffer);

    // Enter the run loop.
    CFRunLoopRun();
}
```

Once it enters its run loop, all future events sent to the thread's port are handled by the `ProcessClientRequest` function. The implementation of that function depends on the type of work the thread does and is not shown here.

Synchronization

The presence of multiple threads in an application opens up potential issues regarding safe access to resources from multiple threads of execution. Two threads modifying the same resource might interfere with each other in unintended ways. For example, one thread might overwrite another's changes or put the application into an unknown and potentially invalid state. If you are lucky, the corrupted resource might cause obvious performance problems or crashes that are relatively easy to track down and fix. If you are unlucky, however, the corruption may cause subtle errors that do not manifest themselves until much later, or the errors might require a significant overhaul of your underlying coding assumptions.

When it comes to thread safety, a good design is the best protection you have. Avoiding shared resources and minimizing the interactions between your threads makes it less likely for those threads to interfere with each other. A completely interference-free design is not always possible, however. In cases where your threads must interact, you need to use synchronization tools to ensure that when they interact, they do so safely.

Mac OS X and iOS provide numerous synchronization tools for you to use, ranging from tools that provide mutually exclusive access to those that sequence events correctly in your application. The following sections describe these tools and how you use them in your code to affect safe access to your program's resources.

Synchronization Tools

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

Atomic Operations

Atomic operations are a simple form of synchronization that work on simple data types. The advantage of atomic operations is that they do not block competing threads. For simple operations, such as incrementing a counter variable, this can lead to much better performance than taking a lock.

Mac OS X and iOS include numerous operations to perform basic mathematical and logical operations on 32-bit and 64-bit values. Among these operations are atomic versions of the compare-and-swap, test-and-set, and test-and-clear operations. For a list of supported atomic operations, see the `/usr/include/libkern/OSAtomic.h` header file or see the `atomic` man page.

Memory Barriers and Volatile Variables

In order to achieve optimal performance, compilers often reorder assembly-level instructions to keep the instruction pipeline for the processor as full as possible. As part of this optimization, the compiler may reorder instructions that access main memory when it thinks doing so would not generate incorrect data.

Unfortunately, it is not always possible for the compiler to detect all memory-dependent operations. If seemingly separate variables actually influence each other, the compiler optimizations could update those variables in the wrong order, generating potentially incorrect results.

A memory barrier is a type of nonblocking synchronization tool used to ensure that memory operations occur in the correct order. A memory barrier acts like a fence, forcing the processor to complete any load and store operations positioned in front of the barrier before it is allowed to perform load and store operations positioned after the barrier. Memory barriers are typically used to ensure that memory operations by one thread (but visible to another) always occur in an expected order. The lack of a memory barrier in such a situation might allow other threads to see seemingly impossible results. (For an example, see the Wikipedia entry for memory barriers.) To employ a memory barrier, you simply call the `OSMemoryBarrier` function at the appropriate point in your code.

Volatile variables apply another type of memory constraint to individual variables. The compiler often optimizes code by loading the values for variables into registers. For local variables, this is usually not a problem. If the variable is visible from another thread however, such an optimization might prevent the other thread from noticing any changes to it. Applying the `volatile` keyword to a variable forces the compiler to load that variable from memory each time it is used. You might declare a variable as `volatile` if its value could be changed at any time by an external source that the compiler may not be able to detect.

Because both memory barriers and volatile variables decrease the number of optimizations the compiler can perform, they should be used sparingly and only where needed to ensure correctness. For information about using memory barriers, see the `OSMemoryBarrier` man page.

Locks

Locks are one of the most commonly used synchronization tools. You can use locks to protect a **critical section** of your code, which is a segment of code that only one thread at a time is allowed access. For example, a critical section might manipulate a particular data structure or use some resource that supports at most one client at a time. By placing a lock around this section, you exclude other threads from making changes that might affect the correctness of your code.

Table 4-1 lists some of the locks that are commonly used by programmers. Mac OS X and iOS provide implementations for most of these lock types, but not all of them. For unsupported lock types, the description column explains the reasons why those locks are not implemented directly on the platform.

Table 4-1   Lock types

Lock	Description
Mutex	A mutually exclusive (or **mutex**) lock acts as a protective barrier around a resource. A mutex is a type of semaphore that grants access to only one thread at a time. If a mutex is in use and another thread tries to acquire it, that thread blocks until the mutex is released by its original holder. If multiple threads compete for the same mutex, only one at a time is allowed access to it.
Recursive lock	A recursive lock is a variant on the mutex lock. A recursive lock allows a single thread to acquire the lock multiple times before releasing it. Other threads remain blocked until the owner of the lock releases the lock the same number of times it acquired it. Recursive locks are used during recursive iterations primarily but may also be used in cases where multiple methods each need to acquire the lock separately.

Lock	Description
Read-write lock	A read-write lock is also referred to as a shared-exclusive lock. This type of lock is typically used in larger-scale operations and can significantly improve performance if the protected data structure is read frequently and modified only occasionally. During normal operation, multiple readers can access the data structure simultaneously. When a thread wants to write to the structure, though, it blocks until all readers release the lock, at which point it acquires the lock and can update the structure. While a writing thread is waiting for the lock, new reader threads block until the writing thread is finished. The system supports read-write locks using POSIX threads only. For more information on how to use these locks, see the `pthread` man page.
Distributed lock	A distributed lock provides mutually exclusive access at the process level. Unlike a true mutex, a distributed lock does not block a process or prevent it from running. It simply reports when the lock is busy and lets the process decide how to proceed.
Spin lock	A spin lock polls its lock condition repeatedly until that condition becomes true. Spin locks are most often used on multiprocessor systems where the expected wait time for a lock is small. In these situations, it is often more efficient to poll than to block the thread, which involves a context switch and the updating of thread data structures. The system does not provide any implementations of spin locks because of their polling nature, but you can easily implement them in specific situations. For information on implementing spin locks in the kernel, see *Kernel Programming Guide*.
Double-checked lock	A double-checked lock is an attempt to reduce the overhead of taking a lock by testing the locking criteria prior to taking the lock. Because double-checked locks are potentially unsafe, the system does not provide explicit support for them and their use is discouraged.

> **Note:** Most types of locks also incorporate a memory barrier to ensure that any preceding load and store instructions are completed before entering the critical section.

For information on how to use locks, see "Using Locks" (page 66).

Conditions

A condition is another type of semaphore that allows threads to signal each other when a certain condition is true. Conditions are typically used to indicate the availability of a resource or to ensure that tasks are performed in a specific order. When a thread tests a condition, it blocks unless that condition is already true. It remains blocked until some other thread explicitly changes and signals the condition. The difference between a condition and a mutex lock is that multiple threads may be permitted access to the condition at the same time. The condition is more of a gatekeeper that lets different threads through the gate depending on some specified criteria.

One way you might use a condition is to manage a pool of pending events. The event queue would use a condition variable to signal waiting threads when there were events in the queue. If one event arrives, the queue would signal the condition appropriately. If a thread were already waiting, it would be woken up whereupon it would pull the event from the queue and process it. If two events came in to the queue at roughly the same time, the queue would signal the condition twice to wake up two threads.

The system provides support for conditions in several different technologies. The correct implementation of conditions requires careful coding, however, so you should look at the examples in "Using Conditions" (page 70) before using them in your own code.

Perform Selector Routines

Cocoa applications have a convenient way of delivering messages in a synchronized manner to a single thread. The `NSObject` class declares methods for performing a selector on one of the application's active threads. These methods let your threads deliver messages asynchronously with the guarantee that they will be performed synchronously by the target thread. For example, you might use perform selector messages to deliver results from a distributed computation to your application's main thread or to a designated coordinator thread. Each request to perform a selector is queued on the target thread's run loop and the requests are then processed sequentially in the order in which they were received.

For a summary of perform selector routines and more information about how to use them, see "Cocoa Perform Selector Sources" (page 36).

Synchronization Costs and Performance

Synchronization helps ensure the correctness of your code, but does so at the expense of performance. The use of synchronization tools introduces delays, even in uncontested cases. Locks and atomic operations generally involve the use of memory barriers and kernel-level synchronization to ensure code is properly protected. And if there is contention for a lock, your threads could block and experience even greater delays.

Table 4-2 lists some of the approximate costs associated with mutexes and atomic operations in the uncontested case. These measurements represented average times taken over several thousand samples. As with thread creation times though, mutex acquisition times (even in the uncontested case) can vary greatly depending on processor load, the speed of the computer, and the amount of available system and program memory.

Table 4-2   Mutex and atomic operation costs

Item	Approximate cost	Notes
Mutex acquisition time	Approximately 0.2 microseconds	This is the lock acquisition time in an uncontested case. If the lock is held by another thread, the acquisition time can be much greater. The figures were determined by analyzing the mean and median values generated during mutex acquisition on an Intel-based iMac with a 2 GHz Core Duo processor and 1 GB of RAM running Mac OS X v10.5.
Atomic compare-and-swap	Approximately 0.05 microseconds	This is the compare-and-swap time in an uncontested case. The figures were determined by analyzing the mean and median values for the operation and were generated on an Intel-based iMac with a 2 GHz Core Duo processor and 1 GB of RAM running Mac OS X v10.5.

When designing your concurrent tasks, correctness is always the most important factor, but you should also consider performance factors as well. Code that executes correctly under multiple threads, but slower than the same code running on a single thread, is hardly an improvement.

If you are retrofitting an existing single-threaded application, you should always take a set of baseline measurements of the performance of key tasks. Upon adding additional threads, you should then take new measurements for those same tasks and compare the performance of the multithreaded case to the single-threaded case. If after tuning your code, threading does not improve performance, you may want to reconsider your specific implementation or the use of threads altogether.

For information about performance and the tools for gathering metrics, see *Performance Overview*. For specific information about the cost of locks and atomic operations, see "Thread Costs" (page 21).

Thread Safety and Signals

When it comes to threaded applications, nothing causes more fear or confusion than the issue of handling signals. Signals are a low-level BSD mechanism that can be used to deliver information to a process or manipulate it in some way. Some programs use signals to detect certain events, such as the death of a child process. The system uses signals to terminate runaway processes and communicate other types of information.

The problem with signals is not what they do, but their behavior when your application has multiple threads. In a single-threaded application, all signal handlers run on the main thread. In a multithreaded application, signals that are not tied to a specific hardware error (such as an illegal instruction) are delivered to whichever thread happens to be running at the time. If multiple threads are running simultaneously, the signal is delivered to whichever one the system happens to pick. In other words, signals can be delivered to any thread of your application.

The first rule for implementing signal handlers in applications is to avoid assumptions about which thread is handling the signal. If a specific thread wants to handle a given signal, you need to work out some way of notifying that thread when the signal arrives. You cannot just assume that installation of a signal handler from that thread will result in the signal being delivered to the same thread.

For more information about signals and installing signal handlers, see `signal` and `sigaction` man pages.

Tips for Thread-Safe Designs

Synchronization tools are a useful way to make your code thread-safe, but they are not a panacea. Used too much, locks and other types of synchronization primitives can actually decrease your application's threaded performance compared to its nonthreaded performance. Finding the right balance between safety and performance is an art that takes experience. The following sections provide tips to help you choose the appropriate level of synchronization for your application.

Avoid Synchronization Altogether

For any new projects you work on, and even for existing projects, designing your code and data structures to avoid the need for synchronization is the best possible solution. Although locks and other synchronization tools are useful, they do impact the performance of any application. And if the overall design causes high contention among specific resources, your threads could be waiting even longer.

The best way to implement concurrency is to reduce the interactions and inter-dependencies between your concurrent tasks. If each task operates on its own private data set, it does not need to protect that data using locks. Even in situations where two tasks do share a common data set, you can look at ways of partitioning that set or providing each task with its own copy. Of course, copying data sets has its costs too, so you have to weigh those costs against the costs of synchronization before making your decision.

Understand the Limits of Synchronization

Synchronization tools are effective only when they are used consistently by all threads in an application. If you create a mutex to restrict access to a specific resource, all of your threads must acquire the same mutex before trying to manipulate the resource. Failure to do so defeats the protection offered by the mutex and is a programmer error.

Be Aware of Threats to Code Correctness

When using locks and memory barriers, you should always give careful thought to their placement in your code. Even locks that seem well placed can actually lull you into a false sense of security. The following series of examples attempt to illustrate this problem by pointing out the flaws in seemingly innocuous code. The basic premise is that you have a mutable array containing a set of immutable objects. Suppose you want to invoke a method of the first object in the array. You might do so using the following code:

```
NSLock* arrayLock = GetArrayLock();
NSMutableArray* myArray = GetSharedArray();
id anObject;

[arrayLock lock];
anObject = [myArray objectAtIndex:0];
[arrayLock unlock];

[anObject doSomething];
```

Because the array is mutable, the lock around the array prevents other threads from modifying the array until you get the desired object. And because the object you retrieve is itself immutable, a lock is not needed around the call to the `doSomething` method.

There is a problem with the preceding example, though. What happens if you release the lock and another thread comes in and removes all objects from the array before you have a chance to execute the `doSomething` method? In an application without garbage collection, the object your code is holding could be released, leaving `anObject` pointing to an invalid memory address. To fix the problem, you might decide to simply rearrange your existing code and release the lock after your call to `doSomething`, as shown here:

```
NSLock* arrayLock = GetArrayLock();
NSMutableArray* myArray = GetSharedArray();
id anObject;
```

```
[arrayLock lock];
anObject = [myArray objectAtIndex:0];
[anObject doSomething];
[arrayLock unlock];
```

By moving the `doSomething` call inside the lock, your code guarantees that the object is still valid when the method is called. Unfortunately, if the `doSomething` method takes a long time to execute, this could cause your code to hold the lock for a long time, which could create a performance bottleneck.

The problem with the code is not that the critical region was poorly defined, but that the actual problem was not understood. The real problem is a memory management issue that is triggered only by the presence of other threads. Because it can be released by another thread, a better solution would be to retain `anObject` before releasing the lock. This solution addresses the real problem of the object being released and does so without introducing a potential performance penalty.

```
NSLock* arrayLock = GetArrayLock();
NSMutableArray* myArray = GetSharedArray();
id anObject;

[arrayLock lock];
anObject = [myArray objectAtIndex:0];
[anObject retain];
[arrayLock unlock];

[anObject doSomething];
```

Although the preceding examples are very simple in nature, they do illustrate a very important point. When it comes to correctness, you have to think beyond the obvious problems. Memory management and other aspects of your design may also be affected by the presence of multiple threads, so you have to think about those problems up front. In addition, you should always assume that the compiler will do the worst possible thing when it comes to safety. This kind of awareness and vigilance should help you avoid potential problems and ensure that your code behaves correctly.

For additional examples of how to make your program thread-safe, see Technical Note TN2059: "Using Collection Classes Safely in Multithreaded Applications."

Watch Out for Deadlocks and Livelocks

Any time a thread tries to take more than one lock at the same time, there is a potential for a deadlock to occur. A deadlock occurs when two different threads hold a lock that the other one needs and then try to acquire the lock held by the other thread. The result is that each thread blocks permanently because it can never acquire the other lock.

A livelock is similar to a deadlock and occurs when two threads compete for the same set of resources. In a livelock situation, a thread gives up its first lock in an attempt to acquire its second lock. Once it acquires the second lock, it goes back and tries to acquire the first lock again. It locks up because it spends all its time releasing one lock and trying to acquire the other lock rather than doing any real work.

The best way to avoid both deadlock and livelock situations is to take only one lock at a time. If you must acquire more than one lock at a time, you should make sure that other threads do not try to do something similar.

Use Volatile Variables Correctly

If you are already using a mutex to protect a section of code, do not automatically assume you need to use the `volatile` keyword to protect important variables inside that section. A mutex includes a memory barrier to ensure the proper ordering of load and store operations. Adding the `volatile` keyword to a variable within a critical section forces the value to be loaded from memory each time it is accessed. The combination of the two synchronization techniques may be necessary in specific cases but also leads to a significant performance penalty. If the mutex alone is enough to protect the variable, omit the `volatile` keyword.

It is also important that you do not use volatile variables in an attempt to avoid the use of mutexes. In general, mutexes and other synchronization mechanisms are a better way to protect the integrity of your data structures than volatile variables. The `volatile` keyword only ensures that a variable is loaded from memory rather than stored in a register. It does not ensure that the variable is accessed correctly by your code.

Using Atomic Operations

Nonblocking synchronization is a way to perform some types of operations and avoid the expense of locks. Although locks are an effective way to synchronize two threads, acquiring a lock is a relatively expensive operation, even in the uncontested case. By contrast, many atomic operations take a fraction of the time to complete and can be just as effective as a lock.

Atomic operations let you perform simple mathematical and logical operations on 32-bit or 64-bit values. These operations rely on special hardware instructions (and an optional memory barrier) to ensure that the given operation completes before the affected memory is accessed again. In the multithreaded case, you should always use the atomic operations that incorporate a memory barrier to ensure that the memory is synchronized correctly between threads.

Table 4-3 lists the available atomic mathematical and logical operations and the corresponding function names. These functions are all declared in the `/usr/include/libkern/OSAtomic.h` header file, where you can also find the complete syntax. The 64-bit versions of these functions are available only in 64-bit processes.

Table 4-3   Atomic math and logic operations

Operation	Function name	Description

Add	`OSAtomicAdd32`
	`OSAtomicAdd32Barrier`
	`OSAtomicAdd64`
	`OSAtomicAdd64Barrier`
		Adds two integer values together and stores the result in one of the specified variables.

Increment	`OSAtomicIncrement32`
	`OSAtomicIncrement32Barrier`
	`OSAtomicIncrement64`
	`OSAtomicIncrement64Barrier`
		Increments the specified integer value by 1.

Operation	Function name	Description

Decrement	`OSAtomicDecrement32`
	`OSAtomicDecrement32Barrier`
	`OSAtomicDecrement64`
	`OSAtomicDecrement64Barrier`
		Decrements the specified integer value by 1.

Logical OR	`OSAtomicOr32` `OSAtomicOr32Barrier`	Performs a logical OR between the specified 32-bit value and a 32-bit mask.
Logical AND	`OSAtomicAnd32` `OSAtomicAnd32Barrier`	Performs a logical AND between the specified 32-bit value and a 32-bit mask.
Logical XOR	`OSAtomicXor32` `OSAtomicXor32Barrier`	Performs a logical XOR between the specified 32-bit value and a 32-bit mask.

Compare and swap	`OSAtomicCompareAndSwap32`
	`OSAtomicCompareAnd-Swap32Barrier`
	`OSAtomicCompareAndSwap64`
	`OSAtomicCompareAnd-Swap64Barrier`
	`OSAtomicCompareAndSwapPtr`
	`OSAtomicCompareAnd-SwapPtrBarrier`
	`OSAtomicCompareAndSwapInt`
	`OSAtomicCompareAnd-SwapIntBarrier`
	`OSAtomicCompareAndSwapLong`
	`OSAtomicCompareAnd-SwapLongBarrier`
		Compares a variable against the specified old value. If the two values are equal, this function assigns the specified new value to the variable; otherwise, it does nothing. The comparison and assignment are done as one atomic operation and the function returns a Boolean value indicating whether the swap actually occurred.

Test and set	`OSAtomicTestAndSet` `OSAtomicTestAndSetBarrier`	Tests a bit in the specified variable, sets that bit to 1, and returns the value of the old bit as a Boolean value. Bits are tested according to the formula `(0x80>>(n&-7))` of byte `((char*)address + (n >> 3))` where `n` is the bit number and `address` is a pointer to the variable. This formula effectively breaks up the variable into 8-bit sized chunks and orders the bits in each chunk in reverse. For example, to test the lowest-order bit (bit 0) of a 32-bit integer, you would actually specify 7 for the bit number; similarly, to test the highest order bit (bit 32), you would specify 24 for the bit number.

Operation	Function name	Description
Test and clear	`OSAtomicTestAndClear` `OSAtomicTestAndClearBarrier`	Tests a bit in the specified variable, sets that bit to 0, and returns the value of the old bit as a Boolean value. Bits are tested according to the formula `(0x80>>(n&-7))` of byte `((char*)address + (n >> 3))` where `n` is the bit number and `address` is a pointer to the variable. This formula effectively breaks up the variable into 8-bit sized chunks and orders the bits in each chunk in reverse. For example, to test the lowest-order bit (bit 0) of a 32-bit integer, you would actually specify 7 for the bit number; similarly, to test the highest order bit (bit 32), you would specify 24 for the bit number.

The behavior of most atomic functions should be relatively straightforward and what you would expect. Listing 4-1, however, shows the behavior of atomic test-and-set and compare-and-swap operations, which are a little more complex. The first three calls to the `OSAtomicTestAndSet` function demonstrate how the bit manipulation formula being used on an integer value and its results might differ from what you would expect. The last two calls show the behavior of the `OSAtomicCompareAndSwap32` function. In all cases, these functions are being called in the uncontested case when no other threads are manipulating the values.

Listing 4-1   Performing atomic operations

```
int32_t  theValue = 0;
OSAtomicTestAndSet(0, &theValue);
// theValue is now 128.

theValue = 0;
OSAtomicTestAndSet(7, &theValue);
// theValue is now 1.

theValue = 0;
OSAtomicTestAndSet(15, &theValue)
// theValue is now 256.

OSAtomicCompareAndSwap32(256, 512, &theValue);
// theValue is now 512.

OSAtomicCompareAndSwap32(256, 1024, &theValue);
// theValue is still 512.
```

For information about atomic operations, see the `atomic` man page and the `/usr/include/libkern/OSAtomic.h` header file.

Using Locks

Locks are a fundamental synchronization tool for threaded programming. Locks enable you to protect large sections of code easily so that you can ensure the correctness of that code. Mac OS X and iOS provide basic mutex locks for all application types and the Foundation framework defines some additional variants of the mutex lock for special situations. The following sections show you how to use several of these lock types.

Using a POSIX Mutex Lock

POSIX mutex locks are extremely easy to use from any application. To create the mutex lock, you declare and initialize a `pthread_mutex_t` structure. To lock and unlock the mutex lock, you use the `pthread_mutex_lock` and `pthread_mutex_unlock` functions. Listing 4-2 shows the basic code required to initialize and use a POSIX thread mutex lock. When you are done with the lock, simply call `pthread_mutex_destroy` to free up the lock data structures.

Listing 4-2   Using a mutex lock

```
pthread_mutex_t mutex;
void MyInitFunction()
{
    pthread_mutex_init(&mutex, NULL);
}

void MyLockingFunction()
{
    pthread_mutex_lock(&mutex);
    // Do work.
    pthread_mutex_unlock(&mutex);
}
```

> **Note:** The preceding code is a simplified example intended to show the basic usage of the POSIX thread mutex functions. Your own code should check the error codes returned by these functions and handle them appropriately.

Using the NSLock Class

An `NSLock` object implements a basic mutex for Cocoa applications. The interface for all locks (including `NSLock`) is actually defined by the `NSLocking` protocol, which defines the `lock` and `unlock` methods. You use these methods to acquire and release the lock just as you would any mutex.

In addition to the standard locking behavior, the `NSLock` class adds the `tryLock` and `lockBeforeDate:` methods. The `tryLock` method attempts to acquire the lock but does not block if the lock is unavailable; instead, the method simply returns `NO`. The `lockBeforeDate:` method attempts to acquire the lock but unblocks the thread (and returns `NO`) if the lock is not acquired within the specified time limit.

The following example shows how you could use an `NSLock` object to coordinate the updating of a visual display, whose data is being calculated by several threads. If the thread cannot acquire the lock immediately, it simply continues its calculations until it can acquire the lock and update the display.

```
BOOL moreToDo = YES;
NSLock *theLock = [[NSLock alloc] init];
...
while (moreToDo) {
    /* Do another increment of calculation */
    /* until there's no more to do. */
    if ([theLock tryLock]) {
        /* Update display used by all threads. */
        [theLock unlock];
    }
```

```
}
```

Using the @synchronized Directive

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

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

The object passed to the `@synchronized` directive is a unique identifier used to distinguish the protected block. If you execute the preceding method in two different threads, passing a different object for the `anObj` parameter on each thread, each would take its lock and continue processing without being blocked by the other. If you pass the same object in both cases, however, one of the threads would acquire the lock first and the other would block until the first thread completed the critical section.

As a precautionary measure, the `@synchronized` block implicitly adds an exception handler to the protected code. This handler automatically releases the mutex in the event that an exception is thrown. This means that in order to use the `@synchronized` directive, you must also enable Objective-C exception handling in your code. If you do not want the additional overhead caused by the implicit exception handler, you should consider using the lock classes.

For more information about the `@synchronized` directive, see *The Objective-C Programming Language*.

Using Other Cocoa Locks

The following sections describe the process for using several other types of Cocoa locks.

Using an NSRecursiveLock Object

The `NSRecursiveLock` class defines a lock that can be acquired multiple times by the same thread without causing the thread to deadlock. A recursive lock keeps track of how many times it was successfully acquired. Each successful acquisition of the lock must be balanced by a corresponding call to unlock the lock. Only when all of the lock and unlock calls are balanced is the lock actually released so that other threads can acquire it.

As its name implies, this type of lock is commonly used inside a recursive function to prevent the recursion from blocking the thread. You could similarly use it in the nonrecursive case to call functions whose semantics demand that they also take the lock. Here's an example of a simple recursive function that acquires the lock through recursion. If you did not use an `NSRecursiveLock` object for this code, the thread would deadlock when the function was called again.

```
NSRecursiveLock *theLock = [[NSRecursiveLock alloc] init];

void MyRecursiveFunction(int value)
{
    [theLock lock];
    if (value != 0)
    {
        --value;
        MyRecursiveFunction(value);
    }
    [theLock unlock];
}

MyRecursiveFunction(5);
```

> **Note:** Because a recursive lock is not released until all lock calls are balanced with unlock calls, you should carefully weigh the decision to use a performance lock against the potential performance implications. Holding any lock for an extended period of time can cause other threads to block until the recursion completes. If you can rewrite your code to eliminate the recursion or eliminate the need to use a recursive lock, you may achieve better performance.

Using an NSConditionLock Object

An `NSConditionLock` object defines a mutex lock that can be locked and unlocked with specific values. You should not confuse this type of lock with a condition (see "Conditions" (page 59)). The behavior is somewhat similar to conditions, but is implemented very differently.

Typically, you use an `NSConditionLock` object when threads need to perform tasks in a specific order, such as when one thread produces data that another consumes. While the producer is executing, the consumer acquires the lock using a condition that is specific to your program. (The condition itself is just an integer value that you define.) When the producer finishes, it unlocks the lock and sets the lock condition to the appropriate integer value to wake the consumer thread, which then proceeds to process the data.

The locking and unlocking methods that `NSConditionLock` objects respond to can be used in any combination. For example, you can pair a `lock` message with `unlockWithCondition:`, or a `lockWhenCondition:` message with `unlock`. Of course, this latter combination unlocks the lock but might not release any threads waiting on a specific condition value.

The following example shows how the producer-consumer problem might be handled using condition locks. Imagine that an application contains a queue of data. A producer thread adds data to the queue, and consumer threads extract data from the queue. The producer need not wait for a specific condition, but it must wait for the lock to be available so it can safely add data to the queue.

```
id condLock = [[NSConditionLock alloc] initWithCondition:NO_DATA];

while(true)
{
    [condLock lock];
    /* Add data to the queue. */
    [condLock unlockWithCondition:HAS_DATA];
}
```

Because the initial condition of the lock is set to `NO_DATA`, the producer thread should have no trouble acquiring the lock initially. It fills the queue with data and sets the condition to `HAS_DATA`. During subsequent iterations, the producer thread can add new data as it arrives, regardless of whether the queue is empty or still has some data. The only time it blocks is when a consumer thread is extracting data from the queue.

Because the consumer thread must have data to process, it waits on the queue using a specific condition. When the producer puts data on the queue, the consumer thread wakes up and acquires its lock. It can then extract some data from the queue and update the queue status. The following example shows the basic structure of the consumer thread's processing loop.

```
while (true)
{
    [condLock lockWhenCondition:HAS_DATA];
    /* Remove data from the queue. */
    [condLock unlockWithCondition:(isEmpty ? NO_DATA : HAS_DATA)];

    // Process the data locally.
}
```

Using an NSDistributedLock Object

The `NSDistributedLock` class can be used by multiple applications on multiple hosts to restrict access to some shared resource, such as a file. The lock itself is effectively a mutex lock that is implemented using a file-system item, such as a file or directory. For an `NSDistributedLock` object to be usable, the lock must be writable by all applications that use it. This usually means putting it on a file system that is accessible to all of the computers that are running the application.

Unlike other types of lock, `NSDistributedLock` does not conform to the `NSLocking` protocol and thus does not have a `lock` method. A `lock` method would block the execution of the thread and require the system to poll the lock at a predetermined rate. Rather than impose this penalty on your code, `NSDistributedLock` provides a `tryLock` method and lets you decide whether or not to poll.

Because it is implemented using the file system, an `NSDistributedLock` object is not released unless the owner explicitly releases it. If your application crashes while holding a distributed lock, other clients will be unable to access the protected resource. In this situation, you can use the `breakLock` method to break the existing lock so that you can acquire it. Breaking locks should generally be avoided, though, unless you are certain the owning process died and cannot release the lock.

As with other types of locks, when you are done using an `NSDistributedLock` object, you release it by calling the `unlock` method.

Using Conditions

Conditions are a special type of lock that you can use to synchronize the order in which operations must proceed. They differ from mutex locks in a subtle way. A thread waiting on a condition remains blocked until that condition is signaled explicitly by another thread.

Due to the subtleties involved in implementing operating systems, condition locks are permitted to return with spurious success even if they were not actually signaled by your code. To avoid problems caused by these spurious signals, you should always use a predicate in conjunction with your condition lock. The predicate is a more concrete way of determining whether it is safe for your thread to proceed. The condition simply keeps your thread asleep until the predicate can be set by the signaling thread.

The following sections show you how to use conditions in your code.

Using the NSCondition Class

The `NSCondition` class provides the same semantics as POSIX conditions, but wraps both the required lock and condition data structures in a single object. The result is an object that you can lock like a mutex and then wait on like a condition.

Listing 4-3 shows a code snippet demonstrating the sequence of events for waiting on an `NSCondition` object. The `cocoaCondition` variable contains an `NSCondition` object and the `timeToDoWork` variable is an integer that is incremented from another thread immediately prior to signaling the condition.

Listing 4-3   Using a Cocoa condition

```
[cocoaCondition lock];
while (timeToDoWork <= 0)
    [cocoaCondition wait];

timeToDoWork--;

// Do real work here.

[cocoaCondition unlock];
```

Listing 4-4 shows the code used to signal the Cocoa condition and increment the predicate variable. You should always lock the condition before signaling it.

Listing 4-4   Signaling a Cocoa condition

```
[cocoaCondition lock];
timeToDoWork++;
[cocoaCondition signal];
[cocoaCondition unlock];
```

Using POSIX Conditions

POSIX thread condition locks require the use of both a condition data structure and a mutex. Although the two lock structures are separate, the mutex lock is intimately tied to the condition structure at runtime. Threads waiting on a signal should always use the same mutex lock and condition structures together. Changing the pairing can cause errors.

Listing 4-5 shows the basic initialization and usage of a condition and predicate. After initializing both the condition and the mutex lock, the waiting thread enters a while loop using the `ready_to_go` variable as its predicate. Only when the predicate is set and the condition subsequently signaled does the waiting thread wake up and start doing its work.

Listing 4-5   Using a POSIX condition

```
pthread_mutex_t mutex;
pthread_cond_t condition;
Boolean     ready_to_go = true;

void MyCondInitFunction()
{
    pthread_mutex_init(&mutex);
    pthread_cond_init(&condition, NULL);
}

void MyWaitOnConditionFunction()
{
    // Lock the mutex.
    pthread_mutex_lock(&mutex);

    // If the predicate is already set, then the while loop is bypassed;
    // otherwise, the thread sleeps until the predicate is set.
    while(ready_to_go == false)
    {
        pthread_cond_wait(&condition, &mutex);
    }

    // Do work. (The mutex should stay locked.)

    // Reset the predicate and release the mutex.
    ready_to_go = false;
    pthread_mutex_unlock(&mutex);
}
```

The signaling thread is responsible both for setting the predicate and for sending the signal to the condition lock. Listing 4-6 shows the code for implementing this behavior. In this example, the condition is signaled inside of the mutex to prevent race conditions from occurring between the threads waiting on the condition.

Listing 4-6   Signaling a condition lock

```
void SignalThreadUsingCondition()
{
    // At this point, there should be work for the other thread to do.
    pthread_mutex_lock(&mutex);
    ready_to_go = true;

    // Signal the other thread to begin work.
    pthread_cond_signal(&condition);

    pthread_mutex_unlock(&mutex);
}
```

> **Note:** The preceding code is a simplified example intended to show the basic usage of the POSIX thread condition functions. Your own code should check the error codes returned by these functions and handle them appropriately.

Thread Safety Summary

This appendix describes the high-level thread safety of some key frameworks in Mac OS X and iOS. The information in this appendix is subject to change.

Cocoa

Guidelines for using Cocoa from multiple threads include the following:

- Immutable objects are generally thread-safe. Once you create them, you can safely pass these objects to and from threads. On the other hand, mutable objects are generally not thread-safe. To use mutable objects in a threaded application, the application must synchronize appropriately. For more information, see "Mutable Versus Immutable" (page 76).

- Many objects deemed "thread-unsafe" are only unsafe to use from multiple threads. Many of these objects can be used from any thread as long as it is only one thread at a time. Objects that are specifically restricted to the main thread of an application are called out as such.

- The main thread of the application is responsible for handling events. Although the Application Kit continues to work if other threads are involved in the event path, operations can occur out of sequence.

- If you want to use a thread to draw to a view, bracket all drawing code between the `lockFocusIfCanDraw` and `unlockFocus` methods of `NSView`.

- To use POSIX threads with Cocoa, you must first put Cocoa into multithreaded mode. For more information, see "Using POSIX Threads in a Cocoa Application" (page 25).

Foundation Framework Thread Safety

There is a misconception that the Foundation framework is thread-safe and the Application Kit framework is not. Unfortunately, this is a gross generalization and somewhat misleading. Each framework has areas that are thread-safe and areas that are not thread-safe. The following sections describe the general thread safety of the Foundation framework.

Thread-Safe Classes

The following classes and functions are generally considered to be thread-safe. You can use the same instance from multiple threads without first acquiring a lock.

```
NSArray
NSAssertionHandler
NSAttributedString
NSCalendarDate
NSCharacterSet
```

```
NSConditionLock
NSConnection
NSData
NSDate
```
NSDecimal **functions**
```
NSDecimalNumber
NSDecimalNumberHandler
NSDeserializer
NSDictionary
NSDistantObject
NSDistributedLock
NSDistributedNotificationCenter
NSException
```
NSFileManager **(in Mac OS X v10.5 and later)**
```
NSHost
NSLock
```
NSLog/NSLogv
```
NSMethodSignature
NSNotification
NSNotificationCenter
NSNumber
NSObject
NSPortCoder
NSPortMessage
NSPortNameServer
NSProtocolChecker
NSProxy
NSRecursiveLock
NSSet
NSString
NSThread
NSTimer
NSTimeZone
NSUserDefaults
NSValue
```
Object allocation and retain count functions

Zone and memory functions

Thread-Unsafe Classes

The following classes and functions are generally not thread-safe. In most cases, you can use these classes from any thread as long as you use them from only one thread at a time. Check the class documentation for additional details.

```
NSArchiver
NSAutoreleasePool
```

```
NSBundle
NSCalendar
NSCoder
NSCountedSet
NSDateFormatter
NSEnumerator
NSFileHandle
NSFormatter
```
NSHashTable **functions**
```
NSInvocation
```
NSJavaSetup **functions**

NSMapTable **functions**
```
NSMutableArray
NSMutableAttributedString
NSMutableCharacterSet
NSMutableData
NSMutableDictionary
NSMutableSet
NSMutableString
NSNotificationQueue
NSNumberFormatter
NSPipe
NSPort
NSProcessInfo
NSRunLoop
NSScanner
NSSerializer
NSTask
NSUnarchiver
NSUndoManager
```
User name and home directory functions

Note that although `NSSerializer`, `NSArchiver`, `NSCoder`, and `NSEnumerator` objects are themselves thread-safe, they are listed here because it is not safe to change the data objects wrapped by them while they are in use. For example, in the case of an archiver, it is not safe to change the object graph being archived. For an enumerator, it is not safe for any thread to change the enumerated collection.

Main Thread Only Classes

The following class must be used only from the main thread of an application.

```
NSAppleScript
```

Mutable Versus Immutable

Immutable objects are generally thread-safe; once you create them, you can safely pass these objects to and from threads. Of course, when using immutable objects, you still need to remember to use reference counts correctly. If you inappropriately release an object you did not retain, you could cause an exception later.

Mutable objects are generally not thread-safe. To use mutable objects in a threaded application, the application must synchronize access to them using locks. (For more information, see "Atomic Operations" (page 57)). In general, the collection classes (for example, `NSMutableArray`, `NSMutableDictionary`) are not thread-safe when mutations are concerned. That is, if one or more threads are changing the same array, problems can occur. You must lock around spots where reads and writes occur to assure thread safety.

Even if a method claims to return an immutable object, you should never simply assume the returned object is immutable. Depending on the method implementation, the returned object might be mutable or immutable. For example, a method with the return type of `NSString` might actually return an `NSMutableString` due to its implementation. If you want to guarantee that the object you have is immutable, you should make an immutable copy.

Reentrancy

Reentrancy is only possible where operations "call out" to other operations in the same object or on different objects. Retaining and releasing objects is one such "call out" that is sometimes overlooked.

The following table lists the portions of the Foundation framework that are explicitly reentrant. All other classes may or may not be reentrant, or they may be made reentrant in the future. A complete analysis for reentrancy has never been done and this list may not be exhaustive.

Distributed Objects
`NSConditionLock`
`NSDistributedLock`
`NSLock`
`NSLog`/`NSLogv`
`NSNotificationCenter`
`NSRecursiveLock`
`NSRunLoop`
`NSUserDefaults`

Class Initialization

The Objective-C runtime system sends an `initialize` message to every class object before the class receives any other messages. This gives the class a chance to set up its runtime environment before it's used. In a multithreaded application, the runtime guarantees that only one thread—the thread that happens to send the first message to the class—executes the `initialize` method. If a second thread tries to send messages to the class while the first thread is still in the `initialize` method, the second thread blocks until the `initialize` method finishes executing. Meanwhile, the first thread can continue to call other methods on the class. The `initialize` method should not rely on a second thread calling methods of the class; if it does, the two threads become deadlocked.

Due to a bug in Mac OS X version 10.1.x and earlier, a thread could send messages to a class before another thread finished executing that class's `initialize` method. The thread could then access values that have not been fully initialized, perhaps crashing the application. If you encounter this problem, you need to either introduce locks to prevent access to the values until after they are initialized or force the class to initialize itself before becoming multithreaded.

Autorelease Pools

Each thread maintains its own stack of `NSAutoreleasePool` objects. Cocoa expects there to be an autorelease pool always available on the current thread's stack. If a pool is not available, objects do not get released and you leak memory. An `NSAutoreleasePool` object is automatically created and destroyed in the main thread of applications based on the Application Kit, but secondary threads (and Foundation-only applications) must create their own before using Cocoa. If your thread is long-lived and potentially generates a lot of autoreleased objects, you should periodically destroy and create autorelease pools (like the Application Kit does on the main thread); otherwise, autoreleased objects accumulate and your memory footprint grows. If your detached thread does not use Cocoa, you do not need to create an autorelease pool.

Run Loops

Every thread has one and only one run loop. Each run loop, and hence each thread, however, has its own set of input modes that determine which input sources are listened to when the run loop is run. The input modes defined in one run loop do not affect the input modes defined in another run loop, even though they may have the same name.

The run loop for the main thread is automatically run if your application is based on the Application Kit, but secondary threads (and Foundation-only applications) must run the run loop themselves. If a detached thread does not enter the run loop, the thread exits as soon as the detached method finishes executing.

Despite some outward appearances, the `NSRunLoop` class is not thread safe. You should call the instance methods of this class only from the thread that owns it.

Application Kit Framework Thread Safety

The following sections describe the general thread safety of the Application Kit framework.

Thread-Unsafe Classes

The following classes and functions are generally not thread-safe. In most cases, you can use these classes from any thread as long as you use them from only one thread at a time. Check the class documentation for additional details.

- `NSGraphicsContext`. For more information, see "NSGraphicsContext Restrictions" (page 79).
- `NSImage`. For more information, see "NSImage Restrictions" (page 79).
- `NSResponder`
- `NSWindow` and all of its descendants. For more information, see "Window Restrictions" (page 78).

Main Thread Only Classes

The following classes must be used only from the main thread of an application.

- `NSCell` and all of its descendants
- `NSView` and all of its descendants. For more information, see .

Window Restrictions

You can create a window on a secondary thread. The Application Kit ensures that the data structures associated with a window are deallocated on the main thread to avoid race conditions. There is some possibility that window objects may leak in an application that deals with a lot of windows concurrently.

You can create a modal window on a secondary thread. The Application Kit blocks the calling secondary thread while the main thread runs the modal loop.

Event Handling Restrictions

The main thread of the application is responsible for handling events. The main thread is the one blocked in the `run` method of `NSApplication`, usually invoked in an application's `main` function. While the Application Kit continues to work if other threads are involved in the event path, operations can occur out of sequence. For example, if two different threads are responding to key events, the keys could be received out of order. By letting the main thread process events, you achieve a more consistent user experience. Once received, events can be dispatched to secondary threads for further processing if desired.

You can call the `postEvent:atStart:` method of `NSApplication` from a secondary thread to post an event to the main thread's event queue. Order is not guaranteed with respect to user input events, however. The main thread of the application is still responsible for handling events in the event queue.

Drawing Restrictions

The Application Kit is generally thread-safe when drawing with its graphics functions and classes, including the `NSBezierPath` and `NSString` classes. Details for using particular classes are described in the following sections. Additional information about drawing and threads is available in *Cocoa Drawing Guide*.

NSView Restrictions

The `NSView` class is generally thread-safe, with a few exceptions. You should create, destroy, resize, move, and perform other operations on `NSView` objects only from the main thread of an application. Drawing from secondary threads is thread-safe as long as you bracket drawing calls with calls to `lockFocusIfCanDraw` and `unlockFocus`.

If a secondary thread of an application wants to cause portions of the view to be redrawn on the main thread, it must not do so using methods like `display`, `setNeedsDisplay:`, `setNeedsDisplayInRect:`, or `setViewsNeedDisplay:`. Instead, it should send a message to the main thread or call those methods using the `performSelectorOnMainThread:withObject:waitUntilDone:` method instead.

The view system's graphics states (gstates) are per-thread. Using graphics states used to be a way to achieve better drawing performance over a single-threaded application but that is no longer true. Incorrect use of graphics states can actually lead to drawing code that is less efficient than drawing in the main thread.

NSGraphicsContext Restrictions

The `NSGraphicsContext` class represents the drawing context provided by the underlying graphics system. Each `NSGraphicsContext` instance holds its own independent graphics state: coordinate system, clipping, current font, and so on. An instance of the class is automatically created on the main thread for each `NSWindow` instance. If you do any drawing from a secondary thread, a new instance of `NSGraphicsContext` is created specifically for that thread.

If you do any drawing from a secondary thread, you must flush your drawing calls manually. Cocoa does not automatically update views with content drawn from secondary threads, so you need to call the `flushGraphics` method of `NSGraphicsContext` when you finish your drawing. If your application draws content from the main thread only, you do not need to flush your drawing calls.

NSImage Restrictions

One thread can create an `NSImage` object, draw to the image buffer, and pass it off to the main thread for drawing. The underlying image cache is shared among all threads. For more information about images and how caching works, see *Cocoa Drawing Guide*.

Core Data Framework

The Core Data framework generally supports threading, although there are some usage caveats that apply. For information on these caveats, see Multi-Threading with Core Data in *Core Data Programming Guide*.

Core Foundation

Core Foundation is sufficiently thread-safe that, if you program with care, you should not run into any problems related to competing threads. It is thread-safe in the common cases, such as when you query, retain, release, and pass around immutable objects. Even central shared objects that might be queried from more than one thread are reliably thread-safe.

Like Cocoa, Core Foundation is not thread-safe when it comes to mutations to objects or their contents. For example, modifying a mutable data or mutable array object is not thread-safe, as you might expect, but neither is modifying an object inside of an immutable array. One reason for this is performance, which is critical in these situations. Moreover, it is usually not possible to achieve absolute thread safety at this level. You cannot rule out, for example, indeterminate behavior resulting from retaining an object obtained from a collection. The collection itself might be freed before the call to retain the contained object is made.

In those cases where Core Foundation objects are to be accessed from multiple threads and mutated, your code should protect against simultaneous access by using locks at the access points. For instance, the code that enumerates the objects of a Core Foundation array should use the appropriate locking calls around the enumerating block to protect against someone else mutating the array.

Glossary

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

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.

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.

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.

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.

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.

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.

recursive lock A lock that can be locked multiple times by the same 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.

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 source A source of synchronous events for a thread. Timers generate one-shot or repeated events at a scheduled future time.

Document Revision History

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

Date	Notes
2010-04-28	Corrected typos.
2009-05-22	Moved information about operation objects to the Concurrency Programming Guide. Refocused this book solely on threads.
2008-10-15	Updated sample code pertaining to operation objects and operation queues.
2008-03-21	Updated for iOS.
2008-02-08	Performed a major rewrite and update of thread-related concepts and tasks.
	Added more information about configuring threads.
	Reorganized the synchronization tools sections into a chapter and added information about atomic operations, memory barriers, and volatile variables.
	Added more details regarding the use and configuration of run loops.
	Changed document title from *Multithreading Programming Topics*.
2007-10-31	Added information about the NSOperation and NSOperationQueue objects.
2006-04-04	Added some new guidelines and updated the information about run loops. Verified the accuracy of the distributed object code examples and updated the code examples in several other articles.
2005-03-03	Updated port examples to use NSPort instead of NSMessagePort.
2005-01-11	Reorganized articles and expanded document to cover more than just Cocoa threading techniques.
	Updated thread conceptual information and added information covering the different threading packages in Mac OS X.
	Incorporated material from Core Foundation multithreading document.
	Added information on performing socket-based communication between threads.
	Added sample code and information on creating and using Carbon threads.
	Added thread safety guidelines.
	Added information about POSIX threads and locks.

Date	Notes
	Added sample code demonstrating port-based communications.
	This document replaces information about threading that was published previously in *Multithreading*.
2003-07-28	Updated the advice for using locks in libraries in third-party libraries.
2003-04-08	Restated information on lock/unlock balancing in third-party libraries.
2002-11-12	Revision history was added to existing topic.