Multithreading Programming Topics

2006-04-04

Contents

Tables and Listings

Introduction to Multithreading Programming Topics

Multithreading is a technique generally used to improve the responsiveness of applications. This responsiveness can be either the result of real performance improvements, or simply the perception of improved performance. Using multiple threads of execution in your code, you can separate out data processing and I/O operations from the management of your program's user interface. In so doing, you can prevent any long data processing operations from reducing the responsiveness of your user interface.

Of course, the performance advantages of multithreading come at the cost of increased complexity in the design and maintenance of your code. Because threads of an application share the same memory space, you must synchronize access to shared data structures to prevent the application from entering into an invalid state or crashing. Overprotection of your data structures can cause problems too, however. An overuse of locks may protect your data but could lead to performance that is worse than that of a single threaded application. Striking a balance between performance and protection requires careful consideration of your data structures and the intended usage pattern for your extra threads.

Mac OS X contains several technologies for implementing threading and basic data synchronization. This document introduces you to those technologies and also provides some basic threading concepts and guidelines for developers who may not be familiar with creating multithreaded applications.

Important: The threading and locking classes described in this document are only available for applications that use C-based languages. If you are interested in multithreading for Java applications, see the documentation for the `java.util.concurrent` package on Sun's website (http://java.sun.com/j2se/1.5.0/docs/api/).

Organization of This Document

This document contains the following articles:

- "Threads" (page 13) describes what threads are and why they are useful.
- "Thread Packages" (page 17) describes the advantages and disadvantages of the different threads packages in Mac OS X.
- "Synchronization and Locking" (page 21) describes the technologies available for synchronizing and protecting critical regions of code.
- "Thread Communication" (page 27) discusses the technologies available for communicating between threads.

- "Thread Safety Guidelines" (page 31) offers guidance on how to avoid many of the problems that arise with multiple threads of execution.
- "Cocoa Thread Safety" (page 37) describes safety issues to be aware of when using Cocoa objects in threads.
- "Core Foundation Thread Safety" (page 43) describes safety issues to be aware of when using Core Foundation objects in threads.
- "Creating Threads in Cocoa" (page 45) demonstrates how to create a new thread in a Cocoa application.
- "Creating Threads in Carbon" (page 49) shows how to create a new thread in a Carbon application.
- "Using Locks in Cocoa" (page 53) demonstrates the use of Cocoa locking techniques.
- "Using POSIX Thread Locks" (page 59) demonstrates the use of pthread mutexes and conditions.
- "Communicating With Mach Ports" (page 63) demonstrates the use of Mach ports to communicate between Carbon and Cocoa threads.
- "Communicating With Distributed Objects" (page 71) demonstrates the use of Distributed Objects to communicate between Cocoa threads.

Thread Terminology Issues in Mac OS X

If you are familiar with Carbon's Multiprocessor Services API, then you have probably seen the term "task" used to describe a separate thread of execution in a process. At the time, the term "task" was used to distinguish between threads created using the Multiprocessor Services and Carbon Thread Manager APIs. Unfortunately, on UNIX systems, the term "task" is used to refer to a running process. This discrepancy in terminology usage can lead to potential confusion when reading either *Multiprocessing Services Reference* or *Multiprocessing Services Programming Guide*.

In practical terms, a Multiprocessor Services task is equivalent to a preemptively scheduled thread in Mac OS X. In fact, with one exception, all threads in Mac OS X are scheduled preemptively. The exception is the Carbon Thread Manager, which uses a cooperative scheme for scheduling threads. This scheme was maintained to ensure that applications written for earlier versions of Mac OS would not break when ported to Mac OS X.

To clarify matters when discussing different threading technologies, this document adopts the following conventions:

- The term **thread** is used to refer to a separate thread of execution.
- 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 the job being performed by a thread.

See Also

The threading and locking classes described in this document are only available for applications that use C-based languages. If you are interested in multithreading for Java applications, see the Java documentation on Sun's website.

For basic information about creating POSIX threads, see the `pthread` man page. For a more detailed overview and examples showing you how to use POSIX threads and locks, see *Programming With Posix Threads* by David R. Butenhof.

For more information on Java threading, see *Concurrent Programming in Java(TM): Design Principles and Pattern (2nd Edition)* by Doug Lea.

Threads

In Mac OS X, each process (application) is made up of one or more threads. Each thread represents a single stream of execution for the application's code. Every application starts with a single thread, which is then used to run the application's `main` function. Applications can spawn additional threads, each of which executes the code of a specific function.

Threads let your program perform multiple tasks in parallel. For example, you can use threads to perform several, lengthy calculations while your user interface continues to respond to user commands. You can also use threads to divide a large job into several smaller jobs.

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, though, all threads in a single application share the same virtual memory space and have the same access rights as the process itself.

Mac OS X provides several APIs for creating and managing threads in your program. The underlying technology for each API is the same, but each one offers a different way to create and manage your threads. The Cocoa environment provides an object-oriented threading model. The Carbon environment provides a set of interfaces familiar to developers for previous versions of Mac OS. The Darwin layer provides the POSIX interfaces that offer the greatest control over your threads.

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 States

During its lifetime, a thread can be in one of four primary states: running, blocked, ready, or terminated. These states are described in Table 1. (page 13)

Table 1 Thread states

State	Description
Running	The thread is currently executing code.

State	Description
Blocked	The thread is waiting for work to do or is waiting to acquire a lock.
Ready	The thread is not running but is able to run as soon as processing time is available for it.
Terminated	The thread has finished processing and is waiting for its resources to be reclaimed.

Thread Lifetime

When you spawn a new thread from your code, you specify the function you want to execute on that thread. The function you specify represents the main entry point for the thread. It also represents the exit point because execution of the thread ends when that function returns, although threads can be terminated in other ways.

When designing your program, you should carefully consider what tasks you want to perform on separate threads. Creating and managing a thread imposes some overhead on your program in terms of memory use and time. If your task is relatively short, it may take more time to create the thread than perform the task. In general, a task should take at least a few hundred microseconds to complete before you consider running it on a separate thread.

Depending on your needs, you can overcome the thread creation costs for multiple short tasks by reusing your threads. Rather than create a separate thread for each task, you could create a single worker thread capable of handling multiple smaller tasks on demand. When you need to perform a particular task, you post a request to the worker thread. When the worker thread is not performing tasks, it sits idle and waits for another request.

Other alternatives for short tasks include timers and run-loop observers, which defer the execution of code until a time when your main thread might not be so busy. Both of these techniques let you avoid the thread creation costs as well as avoid the need to synchronize access to your program data. For more information about thread safety and synchronization, see "Thread Safety Guidelines." (page 31)

Threads and Run Loops

When you create long-lived threads, unless you are simply performing one very long task, it is generally a good idea to put the thread into a loop. Inside this loop, the thread processes incoming requests, performs any needed work, communicates its status to any interested listeners, and returns to an idle state until the next request arrives. To avoid the need to poll repeatedly for incoming requests, and thus waste CPU time, threads use a run loop to block their own execution until there is work to do.

Both Cocoa and Core Foundation provide facilities to manage a thread's run loop. Every thread has exactly one run loop to monitor incoming events and dispatch those requests to the appropriate handlers in the thread. When no events are pending, the run loop blocks.

Run loops require some configuration before you can use them. Before events can be posted to the run loop, you must install one or more **timers** or **input sources**. Timers and input sources are simply sources of events. Timers generate one-time or periodic events after a preset interval. Input sources are Mach ports or sockets that your thread monitors for incoming data. If you do not install any input sources on your thread's run loop (or you do not start the run loop), your thread's entry-point function simply does its prescribed work and then exits, ending the life of the thread.

Once a run loop is properly configured, you tell it to start running. Explicitly running a run loop is required only for the secondary threads you create. The run loop for your application's main thread is usually started as a result of your application's user event handling code. For example, Cocoa automatically starts the run loop for the application's main thread. Carbon applications start the main run loop by calling the `RunApplicationEventLoop` function, which you would call anyway as part of your event handling setup.

To terminate a thread, you should design your thread code to exit gracefully from its run loop. The simplest way to exit a run loop is to put your run loop call inside a while loop and put the run loop into "one-shot" mode. In this mode, the run loop processes only one event before returning control to your code. Your while loop can then check a status flag to determine whether it should exit. The more forceful way to terminate a thread is to kill it outright. This not only exits the run loop but also prevents any further code from being executed on the thread.

Warning: It is recommended that you design your threads to exit gracefully. Killing a thread outright prevents the thread from freeing up any resources it is currently using. This can lead to memory leaks or even hang your application if the thread currently holds a lock.

For information about working with run loops in Core Foundation, see *Run Loops* in Core Foundation Events & Other Input Documentation. For information about working with run loops in Cocoa, see *Run Loops* in Cocoa Events & Other Input Documentation.

Thread Costs

Whenever you create a thread, there are certain costs you have to pay in your program. Creating a well-behaved multithreaded application is not an easy task. You must carefully consider your design decisions and protect all data structures that can be accessed by multiple threads. Because threads run independently of each other, debugging a multithreaded application can be tricky. The same code run multiple times can execute in a different order each time. If your data structures are not designed properly, this can lead to data corruption. Even if your data structures are well-designed, timing issues can make tracking down any problems you do encounter extremely difficult.

Another cost to threading your application is the memory overhead. Whenever you create a secondary thread in your program, the system must dedicate some memory to the thread. In addition to the basic thread data structure, each thread can have a set of attributes associated with it. The combination of the thread data structures and attributes can occupy close to 1K of memory. Of greater importance is the amount of stack space allocated for the thread. By default, Mac OS X assigns 512K of memory as stack space for each thread. Some threading APIs let you change this value, but the minimum stack size allowed is 16K. Fortunately, stack space is allocated through the virtual memory system, so memory pages are loaded only as they are touched.

Note: The 512K default stack size applies only to secondary threads you create. Mac OS X allocates 8Mb of stack space for your application's main thread by default.

All of the data structures you create to support threading also incur memory costs. Any time your thread blocks on a condition or mutex, the system must create a semaphore, which is a kernel primitive that handles the blocking and unblocking of your thread. If you avoid the need for locks by creating copies of your data set for each thread, each of those data sets incurs its own memory cost.

Thread Packages

Mac OS X includes several APIs for creating application-level threads. Behaviorally, threads created with one API are essentially identical to threads created with other APIs. Choosing an API for your application is often determined by the type of application (Carbon, Cocoa, Darwin) that you are creating, but it can also involve trade-offs between complexity and performance. For example, Cocoa threads are simple to use but offer less flexibility and performance than POSIX threads.

The following sections introduce you to the available threading APIs, along with the advantages and disadvantages of using each one. All threads (except where noted) are preemptively scheduled for execution based on their current run state and priority.

Note: The following sections are ordered from the lowest-level thread packages to the highest-level ones, showing how each successive threads package builds on the preceding technologies. The order should not be construed as a recommendation of the packages you should use in your applications. Your choice should always be driven by the features and performance offered by a particular package.

Mach Threads

Mach threads (sometimes referred to as kernel threads) provide the basic implementation for threads on the system. All other threading APIs are derived in some way from Mach threads. Application developers rarely, if ever, use Mach threads directly. Mach threads are intended for use in kernel-level programs, such as kernel extensions. Applications should use POSIX threads or any of the higher-level APIs for their threading needs.

Note: Although I/O Kit drivers run in the kernel, they do not use Mach threads directly. Instead, the I/O Kit defines its own classes and functions for implementing threads. For more information, see *Kernel Framework Reference*.

POSIX Threads

A POSIX thread (commonly referred to as a "pthread") is a lightweight wrapper around a Mach thread that enables it to be used by user-level processes. POSIX threads are the basis for all of the application-level threads packages, including Cocoa threads, Carbon Thread Manager threads, and Multiprocessing Services threads.

The API for POSIX threads is C-based and provides comprehensive control for creating and accessing the thread information. This expanded control means that your program must set up and maintain the threads more than is necessary for other threading APIs. (For example, unlike other threading packages, a POSIX thread does not automatically associate itself with your application's run loop.) Because they offer more control over the thread behavior, though, POSIX threads offer the best direct performance of any of the threading packages.

Another advantage of POSIX threads is that the API for accessing them is the same on many different platforms. Therefore, code you write using POSIX threads then can also be used on most UNIX and Windows systems.

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

Cocoa Threads

Cocoa threads (as embodied by the `NSThread` class) provide a high-level abstraction for creating and managing threads. Cocoa threads post appropriate Cocoa notifications to your application to inform you of its threaded state. Internally, the `NSThread` class uses POSIX threads to implement its threading behavior, although the class interface provides fewer options for managing the thread than POSIX does. The features that are exposed are sufficient for implementing most advanced multithreaded programs.

For more information about creating Cocoa threads and using the `NSThread` class, see "Creating Threads in Cocoa." (page 45)

Carbon Multiprocessing Services

Carbon Multiprocessing Services is an API for managing preemptive threads in a Carbon application. This model is based on the MP Task API of Mac OS 9, which was created as an alternative to the cooperative thread model offered by the Carbon Thread Manager. With the Multiprocessing Services API, an application can create multiple, preemptive threads of execution, capable of running in parallel on multiprocessor machines.

For Carbon applications, Multiprocessing Services is the preferred way to create threads. The API creates preemptive threads just like those provided by the POSIX thread and `NSThread` packages. In addition, this API provides additional support for thread synchronization, notification, timers, remote procedure calls, exception handling, semaphores, and critical region marking.

For information about creating threads using the Multiprocessing Services API, see "Creating Threads in Carbon." (page 49) For general information about using the Multiprocessing Services API, see *Multiprocessing Services Programming Guide*. For reference information, see *Multiprocessing Services Reference*.

Carbon Thread Manager

The Carbon Thread Manager is a legacy API for managing a collection of cooperative threads in a Carbon application. Unlike preemptively scheduled threads, where each thread runs independently of the other threads, only one of an application's cooperative threads executes at any given time. Threads are chosen in a round-robin manner, where each thread can either run or yield time to the next thread in line. Because of this behavior, you can effectively think of cooperative threads as a single group when it comes to receiving time from the kernel. Only the active thread runs when told to by the kernel; all other threads stay asleep until the active thread yields to them.

Important: Active development with the Carbon Thread Manager is not recommended. The API is intended only for developers who are porting their applications to Mac OS X and whose code relies on the cooperative threading model. If you are writing a new Carbon application, you should use POSIX threads or the Multiprocessing Services API instead.

Synchronization and Locking

When you write code for a multithreaded application, you have to be careful about how you access shared data structures and code. If one thread begins modifying a data structure and is preempted, another thread could come along and modify the same data structure in a way that invalidates what the first thread did. For example, suppose thread one gets the number of objects in an array and is preempted by thread two, which removes objects from the same array. When it resumes execution, thread one now thinks the array has more objects than it actually does.

To prevent these types of conflict in your code, you use synchronization and locking. Synchronization is a general term used to coordinate access to your application data structures and state information. Locking is a specific technique used to protect those data structures.

Synchronization Tools

Synchronizing the execution of threads prevents them from interfering with each other at runtime. The following sections list some ways to synchronize the execution of code within your threads.

Locks

Locks limit access to shared resources by identifying each resource as being available or unavailable. Successful acquisition of a lock means that your code now has the right to use the protected resource. Relinquishing the lock gives up this right so that other threads can attempt to use the resource. As long as each thread acquires the lock before using the resource, the resource is protected. You can use locks to protect data structures or to mark the beginnings of critical code sections.

To keep threads from spinning needlessly, the kernel puts threads to sleep if a requested lock is currently unavailable. When the lock is released by its current owner, the system notifies any sleeping threads that are waiting for the lock. At that point, each thread attempts to acquire the lock again.

Mac OS X defines many different types of locks to meet the needs of your code. For more information, see "Lock Types." (page 22)

Semaphores

A semaphore is a flag that signals the presence of a specific condition in your program. Semaphores offer a way to synchronize the execution of code. They can also be used in conjunction with locks to implement conditional locking, whereby the lock is acquired only when a certain condition is signaled by another thread.

You might use a semaphore to manage access to a pool of indistinct resources. Threads wanting to use the resources in this pool wait on a given semaphore. The object that manages the pool then sends out a signal for each available resource in the pool. The same number of waiting threads would then be able to acquire a resource and do some work with it. As each thread finished with the resource, it would put it back in the pool, at which point the manager would send out another signal.

Carbon provides support for semaphores through the Multiprocessing Services API, which is documented in the *Multiprocessing Services Reference*. Although Cocoa does not provide direct support for semaphores, you can use the Multiprocessing Services semaphores in your Cocoa applications. For information about using semaphores at the kernel level, see *Kernel Programming Guide*.

Critical Regions

A critical region is a section of code that can be executed by only one thread at a time. You might designate a block of code as a critical region to prevent modification of a common data structure or to prevent race conditions as two threads execute the same code. Enforcement of critical regions is generally implemented by locks, although other techniques exist. The Multiprocessing Services API provides critical region support and the Objective-C compiler defines the `@synchronized` language primitive for marking a critical region of code.

For information on how to designate critical regions in a Carbon application, see *Multiprocessing Services Programming Guide*. For information about the `@synchronized` directive, see "Using the @synchronized Directive." (page 56)

Atomic Operations

In some cases, you can use atomic operations to avoid the need for a lock altogether. The Darwin layer provides atomic support for mathematical, compare and swap, and test and set operations. Most of these operations involve the use of memory barriers to protect the data being modified. They are more lightweight than mutexes and other types of system locks, although the types of operation you can perform are more limited.

For a list of supported atomic operations, see the `/usr/include/libkern/OSAtomic.h` header file.

Lock Types

Mac OS X provides several different types of locks. How much automatic support exists for the type of lock you choose depends on the threading API you use, although in some cases you can cross technologies to use locks defined in other frameworks. In many cases, the lock implementations are also simple enough to implement yourself if needed.

An application can have multiple locks, each protecting different resources or sections of code. You need to create locks before the resource they protect becomes available to any threads that might need it. You might do this in a critical region of code or when you initialize the resource (or your application).

Mutex Locks

A mutex lock is the simplest form of lock, providing mutually exclusive access (hence the term "mutex") to a shared resource. Threads wanting to access a resource protected by a mutex lock must first acquire the lock. Only one thread at a time may acquire the lock. Any other threads attempting to acquire the lock are blocked until the owner relinquishes it.

Important: Many mutex locking mechanisms are designed to return an error if a thread tries to acquire a mutex it already owns. Such an action could deadlock the thread and should be avoided. In the case of `NSLock` in Cocoa, attempting to acquire the lock twice on the same thread causes the thread to lock up permanently. If your thread needs to acquire the same lock multiple times, you should use a recursive lock, as described in "Recursive Locks." (page 23)

Many mutex locks also give you the option of specifying a timeout interval; when that interval expires the thread unblocks regardless of whether the lock was acquired. This feature prevents your thread from deadlocking in situations where it has to contend for a popular resource.

Cocoa, Carbon, and Darwin all provide implementations for a basic mutex lock. Carbon applications can use critical regions, which are described in *Multiprocessing Services Programming Guide*. Cocoa applications can use the `NSLock` class, the use of which is described in "Using an NSLock." (page 54) All applications (even Cocoa and Carbon applications) can use the `pthread_mutex_t` structure and associated functions defined in the Darwin layer to implement mutex locks. For more information about POSIX mutex locks, see "Using POSIX Thread Locks." (page 59)

Recursive Locks

A recursive lock is a type of mutex lock that can be acquired multiple times by the thread that currently owns it. Recursive locks are useful in situations where your code might need to acquire the same lock multiple times, such as in a recursive function. As with mutex locks, other threads are blocked until the current thread releases the lock. Of course, a recursive lock is not released until each lock call is balanced with an unlock call.

Cocoa and Carbon both provide recursive lock implementations. Carbon applications should use critical regions, which are described in *Multiprocessing Services Programming Guide*. Cocoa applications should use the `NSRecursiveLock` class, the use of which is described in "Using an NSRecursiveLock." (page 55)

Condition Locks

A condition lock (sometimes referred to as a "monitor") combines a semaphore with a mutex lock to implement a special type of lock. Rather than acquire a condition lock, threads block and wait for the condition to be signalled by another thread. As soon as the condition is signalled, one of the waiting threads is unblocked and allowed to continue execution. Instead of signalling just one thread, you can also signal multiple threads simultaneously, causing them all to unblock and continue execution.

Condition locks are often used as a tool for synchronizing threads. For example, you could use a condition lock to notify one or more worker threads that they should each begin processing some data.

Cocoa and Darwin both provide condition lock implementations. Cocoa applications should use the `NSConditionLock` class, the use of which is described in "Using an NSConditionLock." (page 54) For information about the functions used to create and manage condition locks, see the `pthread` man page.

Read-Write Locks

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.

Because of the added complexity, read-write locks require slightly more overhead than other types of locks. Carbon, Cocoa, and Darwin applications can use the read-write lock implementation provided by Darwin. For more information, see the `pthread` man page. For information on using read-write locks at the kernel level, see *Kernel Programming Guide*.

Spin Locks

A spin lock is a special type of lock that repeatedly polls the state of a lock until it becomes available. Spin locks are most often used on multiprocessor systems where the expected wait time for a lock is expected to be very 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.

Application developers should generally avoid the use of spin locks altogether and use mutexes and other types of locks instead. Kernel-level developers may use spin locks in isolated situations but should avoid their use whenever possible. For information on using spin locks in kernel-level programs, see *Kernel Programming Guide*.

Performance Issues

Although locks are important for protecting your program's data structures, the misuse of locks can lead to significant performance issues. When a thread acquires a lock, it prevents other threads from operating on the same data structure or section of code. The other threads "block" until the lock is released, during which time the thread sits idle.

If you must acquire a lock, you should try to release it as soon as it is practical. If you find yourself locking large sections of code, you should be sure that doing so is necessary to protect the integrity of your data. For example, you might need to hold a lock for a longer period of time if it delineates the scope of a single, long transaction. On the other hand, if you are simply trying to group together several disparate transactions, you might be better off reorganizing or rewriting your code to eliminate the need to hold the lock for such a long time. Alternatively, you could investigate using multiple locks to protect each distinct transaction.

Overall, you should try to minimize the number of simultaneous locks a thread needs to finish its work. The more locks a thread needs, the higher the risk of the thread being deadlocked while trying to compete for resources with other threads. More locks also means a greater potential for your thread to be put to sleep as it waits for resources to become available.

Thread Communication

Creating a new thread is relatively easy, but communicating between multiple threads requires some work to set up and maintain the communication channels. There are several ways to communicate between threads, with some being more efficient than others. The following sections provide an overview of several common techniques, including their advantages and disadvantages.

Shared Memory

One of the simplest ways to communicate state information between threads is to use a shared object or shared block of memory. A shared object requires very little setup—all you have to do is make sure each thread has a pointer to the object. The object contains whatever custom information you need to communicate between your threads, so it should be very efficient. To prevent the object from being corrupted, though, you should also include one or more mutex or read-write locks to prevent threads from reading or writing the shared data at the same time.

If you need a fast and simple communication model for threads, and you do not expect to reuse that model later, shared objects can save you implementation time. The disadvantage of using a shared object is that all of the involved threads must have intimate knowledge about the composition of the shared object. If you plan to expand the behavior of your threads later, you may find it cumbersome to update a shared object to handle the new behavior.

Port-Based Communication

Ports offer a fast and reliable way to communicate between threads and processes on the same or different computers. Ports are also a fairly standard form of communication on many different platforms and their use is well established. In Mac OS X, a port implementation is provided by the Mach kernel. These Mach ports can be used to pass data between processes on the same computer.

Ports do not care what data you send or how you organize it; you send as much or as little data as you want. The ports simply transfer the bytes you provide to the receiver on the other end with very little overhead. This has tremendous performance advantages over some other techniques, such as distributed objects, which often transmit additional information behind the scenes to support ease-of-use. If your threads send a large number of messages back and forth, the performance gains offered by port-based communication can be significant.

Mac OS X offers abstractions for Mach ports at several different levels of the system. Cocoa provides the `NSMachPort` and `NSMessagePort` classes. Core Foundation provides the `CFMachPort` and `CFMessagePort` opaque types. Darwin also includes an implementation of UNIX domain sockets, which provide local communication for processes much like Mach ports do.

For information on how to set up a port-based connection, see "Communicating With Mach Ports." (page 63) For information about the Cocoa port classes, see the Foundation Reference for Objective-C. For information on the Core Foundation port objects, see the Core Foundation Reference.

Sockets

Sockets are semantically very similar to ports but are generally optimized for network-based communication.

The socket implementations in Mac OS X are based on BSD sockets. Mac OS X offers abstractions for sockets at several different levels of the system. Cocoa provides the `NSSocketPort` class. Core Foundation provides the `CFSocket` opaque type. You can also use BSD sockets directly from the Darwin layer of the system.

Just like ports, sockets do not care what data you send or how you organize it; you send as much or as little data as you want. The socket simply transfers the bytes you give it to the receiver on the other end with very little overhead.

Although the examples in "Communicating With Mach Ports" (page 63) are implemented using Mach ports, you can replace the relevant port-based classes with sockets. For information about the `CFSocket` opaque type or `NSSocketPort` class, see the corresponding reference documentation.

Message Queues

If you are using the Multiprocessing Services API to implement your threads, message queues offer an easy-to-use abstraction for thread communication. A message queue is a first-in, first-out (FIFO) queue that manages incoming and outgoing data for the thread. A thread can have both an input and an output queue. The input queue contains work the thread needs to perform, while the output queue contains the results of that work.

Although message queues offer a simple and convenient abstraction, they do so at a cost. Message queues typically offer slower performance when compared to other types of communication methods. If you are using them to transfer a few messages a second, the performance cost should be negligible; however, if you are transferring hundreds or thousands of messages a second, the penalty would be much more noticeable.

For more information about how to use message queues, see *Multiprocessing Services Programming Guide*.

Distributed Objects

Cocoa distributed objects are an abstraction for port-based communication that gives a thread the illusion of acting directly on an object in another thread. In this scheme, a thread vends its interface through an object. Other threads acquire a proxy to this vended object and use the proxy to communicate with the thread. The threads actually communicate using port messages, but to your code it appears as if you are simply calling methods of a local object. Although intended primarily for interprocess messaging, distributed objects can also be used for messaging between threads.

The main advantage of distributed objects is its abstraction model, which significantly simplifies the code you have to write. The cost of this simpler model is performance. If you send a few dozen messages a second, you might not notice any performance hit. If you send thousands of messages, though, the additional overhead generated by distributed objects starts to add up. At that point, you might be better off using port-based communications.

For an example of how to implement thread communication with distributed objects, see "Communicating With Distributed Objects." (page 71) For additional information on distributed objects, see *Distributed Objects*.

Thread Safety Guidelines

Thread safety is an important concern for designers of multithreaded applications. When multiple execution threads exist, it is possible for more than one of those threads to operate on the same data structures at the same time. If this happens and the structures are not protected, the data in them can become corrupted and possibly lead to the program crashing.

There are many situations that can create thread-safety problems in your code. The following sections list some of those situations and offer guidance on how to solve the problem.

Appropriate Situations for Using Threads

Before you start using threads in your program, you should look at the problem you are trying to solve and decide if threads are the best solution. In some cases, there may be alternatives to threading that are simpler. The following list shows some of the situations where threads might benefit your program:

- You want to avoid blocking your user interface and have work that can be done in the background.
- You want to handle multiple I/O sources in parallel.
- You want to employ multiple CPU cores to increase the amount of work done by your application in a given amount of time.

Even if one of these situations seems appropriate to you, you should still consider alternatives to threading. Other solutions may exist that provide the same benefits without requiring the use of threads. The following list provides some situations in which threading may not be the best solution:

- An asynchronous API exists to perform the same behavior. For example, Bonjour performs network service lookups asynchronously.
- The duration of work to perform is short (less than a few milliseconds).
- The work you have to do is serialized or does not lend itself easily to parallelization. Note that you might still create one background thread (rather than multiple threads) to process data serially without blocking your user interface.
- The underlying subsystems you are using are not thread safe.

Avoid Sharing Data Structures Across Threads

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.

Lock Data at the Right Level

As you design your code, be careful of where you put locks and how long you leave data structures locked. Locks not only introduce a performance bottleneck but if you use them at the wrong level, they may not offer the protection you were expecting.

The following series of examples illustrate locking at the right level. Suppose you have an array of objects that is shared by multiple threads. It might make sense to wrap access calls to that array with a lock, as in the following code:

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

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

[anObject doSomething];
```

However, there is a problem with the preceding code. It’s possible that after the array is unlocked, the current thread could block before the `doSomething` method is called. If another thread were to come along and delete `anObject` from the array (and cause its memory to be released), then when the first thread resumed it would be operating on an invalid object. So in this case, the lock needs to protect not just the array but also its contents. You could try to avoid this problem by retaining the object before releasing the lock, as shown here:

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

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

[anObject doSomething];
```

This modified code is better. Retaining `anObject` should ensure that the object remains valid, but this still might not be sufficient. If the implementation of `doSomething` expected the object to be in the array, calling the method at this point might still cause problems or be entirely unnecessary.

Of course, the other problem involves locking too much code. Doing the following might be a potential performance bottleneck if the `doSomething` method takes a long time to complete:

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

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

It is up to you to decide where the trade-off point is between the safety of locking and the performance advantages of running your code without locks. There is no single formula, but you can try organizing your data structures in a way that lets you use multiple locks, and thus lets more than one thread operate on the structure. You can also use read-write locks if you know that most accesses will be threads reading the contents. You might even want to rework your algorithm to avoid some of the trade-offs in the first place.

For more information about how to use locks, see "Using Locks in Cocoa." (page 53) For additional information and examples of how to make your program thread safe, see Technical Note TN2059: "Using Collection Classes Safely in Multithreaded Applications."

Avoid Volatile Variables in Protected Code

If you are already using mutexes to protect a section of code, you should not use the `volatile` keyword to protect variables declared in those sections. The compiler uses the `volatile` keyword as a hint that it may need to load the variable from memory in situations when its contents could change. The compiler may still optimize the access to that variable but in general, using the keyword does increase the number of fetch instructions in the associated code. In general, you should not use volatile variables to avoid the use of mutexes; mutexes are generally a better way to protect your variables over a section of code.

Catch Local Exceptions in Your Threads

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. The `@synchronized` directive in Objective-C contains an implicit exception handler.

Threads and Your User Interface

If your application has a graphical user interface, it is recommended that you do most of the work of maintaining that interface 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; however, it also simplifies the logic for managing your user interface.

With user interface behavior handled by your main thread, your secondary threads should be used mostly for managing your application's data model. You can use secondary threads to perform lengthy calculations, communicate across the network, and generally update or manage your data structures. All other work should be done on the main thread.

Note: In Cocoa, because they interact with the user interface, an application's controller objects would mostly operate in the main thread.

There are a few notable cases where it is advantageous to perform graphical operations from other threads. 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. Using secondary threads for these operations can greatly increase performance. Similarly, in Carbon and Cocoa you can draw from secondary threads as long as you set up your graphics context correctly. There are likely other exceptions, but if you're not sure about a particular graphical operation, 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 "Cocoa Thread Safety." (page 37) For more information about drawing in Cocoa, see *Cocoa Drawing Guide*.

Thread Safety in System Frameworks

Before you begin designing your program, you should be familiar with the thread safety of the frameworks and libraries you plan to use. Frameworks such as Cocoa and Core Foundation support multithreading but usually with some caveats. For example, Cocoa immutable objects are generally thread safe but Cocoa mutable objects must be protected by locks.

For information on Cocoa-specific thread-safety issues, see "Cocoa Thread Safety." (page 37) For information on Core Foundation thread-safety issues, see "Core Foundation Thread Safety." (page 43) For information regarding other frameworks and libraries, see their documentation.

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.

Important: 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.

Cocoa Thread Safety

Some general 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 39)
- The main thread of the application is responsible for handling events. While 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 With Cocoa." (page 48)

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

`NSArray`	`NSNotification`
`NSAssertionHandler`	`NSNotificationCenter`
`NSAttributedString`	`NSNumber`
`NSCalendarDate`	`NSObject`

`NSCharacterSet`	`NSPortCoder`
`NSConditionLock`	`NSPortMessage`
`NSConnection`	`NSPortNameServer`
`NSData`	`NSProtocolChecker`
`NSDate`	`NSProxy`
`NSDecimal` functions	`NSRecursiveLock`
`NSDecimalNumber`	`NSSet`
`NSDecimalNumberHandler`	`NSString`
`NSDeserializer`	`NSThread`
`NSDictionary`	`NSTimeZone`
`NSDistantObject`	`NSTimer`
`NSDistributedLock`	`NSUserDefaults`
`NSDistributedNotificationCenter`	`NSValue`
`NSException`	Object allocation and retain count functions
`NSLock`	Zone and memory functions
`NSLog`/`NSLogv`	
`NSMethodSignature`	

Thread-Unsafe Classes

The classes and functions in the following table are generally not thread-safe. Some of these items may be made thread-safe in the future but for now you should use a lock or the `@synchronized` directive if there is a potential for access by multiple threads. For some objects, like `NSAppleScript`, you should use the object only from your application's main thread. Check the class documentation to see if any additional guidelines are available.

`NSAppleScript`	`NSMutableAttributedString`
`NSArchiver`	`NSMutableCharacterSet`
`NSAutoreleasePool`	`NSMutableData`
`NSBundle`	`NSMutableDictionary`
`NSCoder`	`NSMutableSet`
`NSCountedSet`	`NSMutableString`

`NSDateFormatter`	`NSNotificationQueue`
`NSEnumerator`	`NSNumberFormatter`
`NSFileHandle`	`NSPipe`
`NSFileManager`	`NSPort`
`NSFormatter`	`NSProcessInfo`
`NSHashTable` functions	`NSRunLoop`
`NSHost`	`NSScanner`
`NSInvocation`	`NSSerializer`
`NSJavaSetup` functions	`NSTask`
`NSMapTable` functions	`NSUnarchiver`
`NSMutableArray`	`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.

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 "Using Locks in Cocoa" (page 53)). 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	`NSNotificationCenter`
`NSConditionLock`	`NSRecursiveLock`
`NSDistributedLock`	`NSRunLoop`
`NSLock`	`NSUserDefaults`
`NSLog/NSLogv`	

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.

Windows

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.

Events

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

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 below. Additional information about drawing and threads is available in *Cocoa Drawing Guide*.

NSView

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:` 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

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

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 Foundation Thread Safety

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.

Creating Threads in Cocoa

If you want to implement threads in a Cocoa application, the `NSThread` class provides a simple interface for detaching your threads and convenient methods for getting information about your threads at runtime. If you want more control over your threads than `NSThread` offers, you can use POSIX threads directly.

Creating the Thread

Creating a new thread in Cocoa is very simple. Use the `detachNewThreadSelector:toTarget:withObject:` class method of `NSThread` to detach a thread. The target object must contain an implementation of the specified method. Although class methods are more commonly used, you can specify either a class or instance method for the selector. Your method should take one parameter (of type `id`) and have a return type of `void`. Listing 1 shows a simple example that launches a thread using the class method named `MyThreadRoutine:`.

Listing 1 Creating a new thread in Cocoa

```
// Signature of thread entry point routine.
// + (void)MyThreadRoutine:(id)param;

// Detach the new thread.
[NSThread detachNewThreadSelector:@selector(MyThreadRoutine:)
toTarget:[MyCustomClass class] withObject:nil];
```

The `detatchNewThreadSelector:toTarget:withObject:` method lets you pass in an object containing any initial data your thread needs. For example, you could use this parameter to pass in custom data for the thread to process. A more likely use for this parameter, though, is to pass in connection information in the form of one or more ports. The detached thread can then use the ports to communicate with the rest of your application.

The first time you call `detachNewThreadSelector:toTarget:withObject:`, your application becomes multithreaded. When this occurs, the `NSThread` object posts `NSWillBecomeMultiThreadedNotification` from the application's main thread, executing any registered observer methods in that thread.

If you want to know if your application is multithreaded, call the `isMultiThreaded` method of `NSThread`. Once it becomes multithreaded, your application stays multithreaded. Thus, calling `isMultiThreaded` returns `YES` even if all detached threads have finished executing.

When a thread exits, it posts `NSThreadWillExitNotification` to the application. The handler methods for registered observers execute in the context of the exiting thread.

Implementing Your Thread Routine

The method you use for your thread entry point can be a class or instance method that takes a single parameter (of type `id`) and returns no value. Thus, either of the following methods could be used as a thread entry point:

```
@implementation MyClass
+ (void) MyClassThreadMethod:(id)anObject;
- (void) MyInstanceThreadMethod:(id)anObject;
@end
```

Inside the body of your method, you set up the necessary data structures, perform the work you want to do, and exit. If you want the thread to perform additional work later on, you must configure and execute the thread's run loop.

Creating an Autorelease Pool

A detached thread needs to create its own `NSAutoreleasePool` object before making any other Cocoa calls. Cocoa depends on there being an autorelease pool available in the current thread. If one is unavailable, your application will leak any autoreleased objects. To create this pool, simply create a new instance of `NSAutoreleasePool` at the beginning of your thread code and release it when your code is finished, as shown in Listing 2.

Listing 2 Creating an autorelease pool for a thread

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

    // Do thread work here

    [pool release];
}
```

If your thread does not use Cocoa beyond the detachment of the thread, the autorelease pool is not needed.

Configuring Your Run Loop

If you want to keep your thread alive and communicate with it later, you need to attach at least one timer or input source to the thread's run loop. Timers and input sources provide a means for the system to put your thread to sleep and wake it when there is something to do. You can use a timer to perform a task periodically or simply delay the execution of the current task. Input sources, such as sockets and Mach ports, are a way for you to communicate with other threads and processes.

To process the data coming from timers and input sources for a run loop, you must get the current run loop and start it. You can get the run loop using the `currentRunLoop` class method of `NSRunLoop`. To start it, you use one of the methods listed in Table 1. (page 47)

Table 1 Starting a run loop in Cocoa

Method	Description
`run`	Puts the run loop into an effectively permanent loop. Once started, the thread continues processing input sources until the thread is forcibly terminated using the `exit` class method of `NSThread`. (Note, this method returns immediately if no input sources are attached to the run loop.)
`runUntilDate:`	Behaves in a similar manner to the `run` method except that it also exits the run loop once the specified date is reached.
`runMode:beforeDate:`	Starts the run loop but returns after input is received for the specified mode or the specified date is reached. You can use this method in a while loop to process data repeatedly until you want the run loop to end.

Warning: You should never pass run loop objects back and forth between your threads or store them globally where other threads could modify them. Run loop objects may only be modified from the thread on which they are installed. Modifying a run loop object from a different thread is likely to cause unexpected behavior or a crash.

Listing 3 shows a basic example of how to install a communications channel on a secondary thread's run loop. A Mach port is created and `myThreadObject` (an object that does the main work for the thread) is made its delegate. Incoming messages are delivered to the `handlePortMessage:` method of `myThreadObject`, which then processes them. Because the `runMode:beforeDate:` method returns after processing a single message, it is called repeatedly in a do-while loop, with the `shouldExit` method of `myThreadObject` acting as a gatekeeper for when to exit the run loop.

Listing 3 Adding an input source to a Cocoa run loop

```
// Create a port for intra-thread communication
NSMachPort* portObj = [NSMachPort port];
[portObj setDelegate:myThreadObject];

[[NSRunLoop currentRunLoop] addPort:portObj forMode:NSDefaultRunLoopMode];

// Start the run loop.
NSDate*    endDate = [NSDate distantFuture];
do
{
    [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode beforeDate:endDate];
}
while (![myThreadObject shouldExit]);
```

For detailed examples of how to set up a run loop to process events and communicate results back to the main thread, see either "Communicating With Mach Ports" (page 63) or "Communicating With Distributed Objects." (page 71) For additional information on run loop see *Run Loops*.

Exiting a Thread's Run Loop

If you want a thread to exit before your application does, the preferred way to start your run loop is using the `runMode:beforeDate:` method. By placing this method in a `while` loop, your code controls the exit point of the run loop. When it is time to exit, you can simply set a flag and wake up your run loop.

When you start a run loop using the `run` or `runUntilDate:` methods of `NSRunLoop`, the thread enters a permanent loop, during which it processes events associated with the attached input sources and timers. To exit the run loop, you have to wait until the specified time (in the case of `runUntilDate:`) or terminate the thread using the `exit` class method of `NSThread`. Exiting a thread in this manner may make it difficult to clean up all of your thread's data structures, however, and should be avoided if possible.

For more information about run loops and how they interact with your threads, see *Run Loops*.

Using POSIX Threads With Cocoa

The underlying implementation of `NSThread` is based on POSIX threads. If you need the improved performance and flexibility offered by POSIX threads, you can use them instead of `NSThread` in your application if you choose. If you still intend to use Cocoa, though, you first put your application into "multithreaded mode" before creating any POSIX threads.

When you detach an `NSThread` for the first time in your application, Cocoa puts your application into multithreaded mode and enables a number of safety measures required to protect data structures for the framework itself. If you create POSIX threads directly without first using `NSThread`, Cocoa does not receive the notification that your application has become multithreaded and thus does not enact any of its safety measures.

If you intend to use Cocoa calls, you must force Cocoa into its multithreaded mode before detaching any POSIX threads. To do this, simply detach an `NSThread` and have it promptly exit. This is enough to ensure that the locks needed by the Cocoa frameworks are put in place. To verify that Cocoa is in multithreaded mode, you can use the `isMultiThreaded` class method of `NSThread`.

Creating Threads in Carbon

If you are writing a new Carbon application, or porting an existing single-threaded application, you can create new threads using Multiprocessing Services. This API creates preemptively scheduled threads and is the preferred technology to use.

Creating the Thread

Creating a new thread in Carbon is very simple. Use `MPCreateTask` to detach your thread and begin running the specified function. Listing 1 shows a simple example that runs the function named `MyThreadFunction` in a separate thread.

Listing 1 Creating a new thread in Carbon

```
OSStatus returnErr = MPCreateTask(&MyThreadFunction, myParam,
kCarbonThreadStackSize, NULL, NULL, NULL, 0, &taskID);
```

The Carbon libraries do not track the threaded state of your program. When you create threads in Carbon, you are responsible for handling any issues that involve your application becoming multithreaded. For example, before spawning your first thread, you might want to create the locks required to protect your application's data structures.

Implementing Your Thread Routine

The function you use for your thread entry point takes a single parameter (of type `void *`) and returns an `OSStatus` error code, as shown in the following declaration:

```
OSStatus MyThreadFunction(void* parameter);
```

Inside the body of your method, you set up the necessary data structures, perform the work you want to do, and exit. If you want the thread to perform additional work later on, you must configure and execute the thread's run loop. The run loop monitors the events you specify and calls your registered event handler methods. When no events are waiting, the run loop puts your thread to sleep.

The following sections list a few of the basic tasks you might perform inside your thread function, along with some of the associated restrictions. For a more thorough discussion of how to use Carbon threads and the associated restrictions, see *Multiprocessing Services Programming Guide*.

Allocating Memory

If you need to allocate memory for local data in your thread, you can use `malloc`, the Core Foundation allocator functions, or the `MPAllocateAligned` function from Multiprocessing Services. All of these functions allocate memory along 16-byte memory boundaries, which is appropriate for most operations. In addition, `MPAllocateAligned` lets you allocate memory along a greater or smaller byte boundary, which you specify when you call the function.

For information about `malloc`, see the `malloc` man page. For information about Core Foundation allocator functions, see *CFAllocator Reference*. For information about `MPAllocateAligned` and related functions, see *Multiprocessing Services Reference*.

Configuring Your Run Loop

If you want to keep your thread alive and communicate with it later, you need to attach at least one timer or input source to your run loop. Timers and input sources provide a means for the system to put your thread to sleep and wake it when there is something to do. You can use a timer to perform a task periodically or simply delay the execution of the current task. Input sources, such as sockets and Mach ports, become a way of communicating with other threads and processes.

In a Carbon application, you use the Core Foundation framework to configure and start the run loop for your thread. You obtain the run loop object for the current thread using the `CFRunLoopGetCurrent` function. After adding timers or input sources to the run loop, call `CFRunLoopRun` or `CFRunLoopRunInMode` to begin processing events from those sources.

Warning: You should never pass run loop objects back and forth between your threads or store them globally where other threads could modify them. Run loop objects may only be modified from the thread on which they are installed. Modifying a run loop object from a different thread is likely to cause unexpected behavior or a crash.

Listing 2 shows how to install a communications channel for the current thread's run loop. The `CFMessagePortRef` type uses a mach port to handle communications within the process. Once you have a port, you can use the `CFMessagePortCreateRunLoopSource` function to obtain an input source for that port and the `CFRunLoopAddSource` function to install it. Incoming messages are delivered to the `MyEventHandlerFunc` callback function.

Listing 2 Adding an input source to a Carbon run loop

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

myPort = CFMessagePortCreateLocal(NULL,
               CFSTR("PortName"),
               &MyEventHandlerFunc,
               &context,
               &shouldFreeInfo);

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

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

For a detailed example of how to set up a run loop to process events and communicate results back to the main thread, see "Communicating With Mach Ports." (page 63) For additional information on run loop see *Run Loops*.

Exiting a Thread's Run Loop

When you execute a run loop using the `CFRunLoopRun` function, the thread enters a permanent loop, during which it processes events associated with the attached input sources and timers. To exit the run loop, you must call the `CFRunLoopStop` function. If you use the `CFRunLoopRunInMode` function, you can have this function return after processing one input source. By placing a call to this function in a do-while loop, you can set up your own exit conditions and exit the run loop without calling `CFRunLoopStop`.

If you want to terminate the thread immediately, without first exiting the run loop, you can call the `MPExit` function from within the thread. This function is a much more abrupt way to terminate the thread and should be used carefully. If you use it, be sure to clean up the thread's data structures first.

Warning: If you need to terminate a completely different thread, you can use the `MPTerminateTask` function. However, this function is recommended only in extreme cases because it does not give the thread a chance to free any currently held resources, like memory or locks.

For more information about run loops and how they interact with your threads, see *Run Loops*.

Using Locks in Cocoa

The Foundation framework defines several types of locks. Classes that conform to the `NSLocking` protocol—`NSLock`, `NSConditionLock`, and `NSRecursiveLock`—coordinate the actions of multiple threads in a single application. Another class—`NSDistributedLock`—does not conform to the protocol and coordinates the actions of multiple applications accessing a shared, external resource, such as a file. The `@synchronized` directive in Objective-C lets you synchronize access without using a lock class.

The Foundation framework's locking classes are designed to work in a well-behaved, multithreaded environment; however, the protection they offer can be subverted by the use of signal handlers. A signal handler can interrupt a thread, execute code that affects shared data, and then let the thread resume without notifying the thread of what happened. This behavior could negate the normal protection offered by locks and lead to data corruption. For this reason, you should avoid the use of signal handlers in multithreaded applications.

Note: Regardless of the locking mechanism you use, remember that locks are not a panacea for thread-safety. You must still lock the right objects at the appropriate level to ensure safe access. For more information and guidance, see "Locking at the Right Level." (page 32)

NSLocking Protocol

The `NSLocking` protocol declares the elementary methods adopted by most lock classes. To acquire a lock, call the `lock` method of the lock object, To relinquish the lock, call the `unlock` method. Some lock classes may define additional methods for acquiring or releasing a lock. (For a complete list of locking methods, see the class documentation.)

The basic `lock` method, as defined by the `NSLocking` protocol, blocks the thread execution until the lock can be acquired. The locking mechanism causes the thread to sleep, rather than force it to poll the system constantly. Thus, lock objects can be used without causing system performance to degrade. Classes that extend the basic locking behavior usually do so to add methods that offer nonblocking alternatives.

Table 1 lists the Foundation classes that conform to the `NSLocking` protocol, with the additional features each class adds to support locking.

Table 1 Classes that support the NSLocking protocol

Class	Adds these features to the basic protocol
`NSLock`	A nonblocking lock method; the ability to limit the duration of a locking attempt.
`NSConditionLock`	The ability to postpone entry to a critical section until a condition is met.
`NSRecursiveLock`	The ability for a single thread to acquire a lock more than once without deadlocking.

Using an NSLock

An `NSLock` object is used to coordinate the operation of multiple threads of execution within the same application. This object can be used to mediate access to an application's global data or to protect a critical section of code, allowing it to run atomically.

The basic interface to `NSLock` is defined by the `NSLocking` protocol, which provides the `lock` and `unlock` methods. 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 if the lock is not acquired within a specific 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 an NSConditionLock

The `NSConditionLock` class defines an object whose lock can be associated with specific, user-defined conditions. A lock of this type acquires its lock only when the currently specified condition is signalled by another thread. Once it has acquired the lock and executed the critical section of code, the thread can relinquish the lock and set the associated condition to something new. (The conditions themselves are arbitrary; you define them as needed for your application.)

Typically, you use an `NSConditionLock` object when threads in your application need to execute in a particular order, such as when one thread produces data that another consumes. While the producer is executing, the consumer sleeps, waiting to acquire a lock that is conditional upon the producer's completion of its operation.

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`.

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 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 blocks while the consumer is processing the queue data but then acquires the lock regardless of whether the queue has data or is empty.

In the following code, the consumer thread waits until the queue contains some data and is not locked by another thread. Upon processing the data, the thread updates the condition to indicate whether the queue still contains data or is empty. If the queue is empty, this thread would block until the producer added more data to the queue.

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

Using an NSRecursiveLock

The `NSRecursiveLock` class defines a lock that can be acquired multiple times by the same thread without causing the thread to deadlock. The lock keeps track of how many times the lock 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 released and available to other threads.

As its name implies, this type of lock is most useful inside of a recursive function. Here’s an example of a simple recursive function that acquires the lock through each 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: Generally, you should avoid the use of recursive locks for performance reasons. Holding any lock for an extended period of time can cause other threads to block until the recursion completes. Consider rewriting your code to use a mutex lock instead.

Using an NSDistributedLock

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 a basic mutex lock that is implemented using a file-system entry, 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, `NSDistrubutedLock` 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 the @synchronized Directive

In addition to lock classes, the Objective-C language includes the `@synchronized` directive for locking a block of code. The directive takes a single parameter, which is the object you want to be used as the key for locking the code. The compiler then creates a mutex lock based on that object. Threads attempting to lock the same object block until the current synchronized block finishes executing. The following example shows what this directive looks like when used in your code.

```
- (void) myMethod:(id)anObject
{
    @synchronized(anObject)
    {
        // Perform any operations that depend on exclusive access.
        [anObject modify];
    }
    // The mutex is now released.
}
```

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. However, this means that you must enable Objective-C exception handling in your code to use this directive. 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 POSIX Thread Locks

The Darwin layer of Mac OS X includes implementations of mutex and condition locks that you can use in your Carbon, Cocoa, and Darwin programs. These locks form the basis of many other locking classes in Mac OS X, including Cocoa's locking classes and Carbon's critical regions.

Using a POSIX Thread Mutex Lock

POSIX mutex locks are extremely easy to use from any application in Mac OS X. 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 1 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 1 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 POSIX Thread Condition Locks

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. Mixing and matching can cause errors.

Due to the subtleties of implementing multiprocessor APIs, condition locks are permitted to return with spurious success even if they were not actually signalled by your code. To avoid problems caused by these spurious signals, it is recommended that you also use a predicate in conjunction with your condition lock. The predicate is a more concrete way of notifying your waiting thread that it should start doing some work. The condition simply keeps your thread asleep until the predicate can be set by the signaling thread.

Listing 2 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 2 Using a condition lock

```
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 signalling thread is responsible both for setting the predicate and for sending the signal to the condition lock. Listing 3 (page 61) shows the code for implementing this behavior. In this example, the condition is actually signalled outside of the mutex. This works for most situations where you are simply communicating between a couple of threads. If your protocol involves multiple threads acting on a condition, you could move the signal call inside of the lock.

Listing 3 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;
    pthread_mutex_unlock(&mutex);

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

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.

Communicating With Mach Ports

The following sections show you how to set up a connection between two threads using ports. In each example, the main thread sets up a port on which to receive messages and then spawns a worker thread. The worker thread sets up its own receive port and sends a checkin message back to the main thread.

Using Ports 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 1 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 `CFMessagePort` object 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 1 Spawning the 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 (!shouldFreeInfo)
    {
        // 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 checkin message arrives, it is dispatched to the main thread's `MainThreadResponseHandler` function, shown in Listing 2. This function extracts the port name for the worker thread and creates a conduit for future communication.

Listing 2 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);
        }

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

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 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 checkin message to the main thread that includes the local port name.

Listing 3 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 checkin 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.

Using Ports in Cocoa

In Cocoa, the technique used to set up a communications port depends on the port object you are using. If you are using an `NSMachPort` object, you can configure both the local and remote threads using the same object. If you are using an `NSMessagePort` object, you must configure the local and remote port objects differently.

Connecting With NSMachPort

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 4 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 1 (page 63)); 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 4 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 5 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 5 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 6 shows the worker thread 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 7 (page 69)) creates a local port for the worker thread and sends a checkin message back to the main thread.

Listing 6 Launching the worker thread using Mach Ports

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

    // Setup 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 7 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 7 Sending the check-in message using Mach ports

```
// Worker thread checkin 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 checkin 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]];
    }
}
```

Connecting With NSMessagePort

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 8 shows the port creation and registration process in cases where you want to use message ports.

Listing 8 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];
```

Communicating With Distributed Objects

Distributed objects provide a thread-safe way to communicate information between threads or processes in a Cocoa application. The preferred way to set up a distributed object connection is to create an `NSConnection` object in each thread. This object contains one port for each of the two threads. Only one thread needs to create the actual port objects. Once created, it passes them to the other thread, which uses them to create its own `NSConnection` object.

The example code that follows shows how to set up a distributed objects connection between the main thread and a newly created worker thread. In the main thread, the application's delegate object handles the task of setting up the initial connection and spawning the worker thread. Once it is up and running, the worker thread uses the connection information it receives from the main thread to complete the connection.

Listing 1 shows the process for creating a worker thread and distributed objects connection. The main thread of the application creates two new ports and a connection object to use when it communicates with the worker thread. It then passes the ports as a parameter to a new thread entry-point method, reversing their order so that the worker receives and sends messages on the correct ports. (Although it is not shown here, your own code should store a reference to `kitConnection` for later use.)

Listing 1 Initiating the connection in the main thread

```
- (void)applicationDidFinishLaunching:(NSNotification *)note
{
    NSPort *port1 = [NSPort port];
    NSPort *port2 = [NSPort port];
    NSArray *portArray = nil;
    NSConnection* kitConnection = nil;

    kitConnection = [[NSConnection alloc] initWithReceivePort:port1
                sendPort:port2];
    [kitConnection setRootObject:self];

    // Ports switched here.
    portArray = [NSArray arrayWithObjects:port2, port1, nil];

    [NSThread detachNewThreadSelector:@selector(connectWithPorts:)
                toTarget:[Calculator class] withObject:portArray];

    return;
}
```

Listing 2 shows the `setServer:` method of the application's delegate object. The delegate implements this method as a way for the worker thread to check in. The worker thread calls this method remotely as soon as its own `NSConnection` object is established. The parameter passed into this method is the worker's proxy object. The main thread can call the methods of this proxy object directly to initiate computation requests on the worker thread.

Listing 2 Application delegate method for storing the worker interface

```
- (void)setServer:(id)anObject
{
    [anObject setProtocolForProxy:@protocol(CalculatorMethods)];
    calculator = (id <CalculatorMethods>)[anObject retain];
    return;
}
```

In the worker thread itself, the `connectWithPorts:` method provides the initial entry point for the thread. This class method of the `Calculator` class sets up the thread's data structures and completes the connection back to the main thread.

Listing 3 shows the `connectWithPorts:` method. The parameter to this method is an array containing the `NSPort` objects to use in creating the connection. Once the `NSConnection` object is created, the worker creates a new instance of the `Calculator` class and passes it back to the main thread using its `setServer:` method. This gives the main thread a way of communicating with the worker thread.

Listing 3 Worker thread entry-point method

```
+ (void)connectWithPorts:(NSArray *)portArray
{
    NSAutoreleasePool *pool;
    NSConnection *serverConnection;
    Calculator *serverObject;

    pool = [[NSAutoreleasePool alloc] init];

    serverConnection = [NSConnection
            connectionWithReceivePort:[portArray objectAtIndex:0]
            sendPort:[portArray objectAtIndex:1]];

    serverObject = [[self alloc] init];
    [serverConnection setRootObject:serverObject];
    [(id)[serverConnection rootProxy] setServer:serverObject];

    do
    {
        [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode
                beforeDate:[NSDate distantFuture]];
    }
    while (![serverObject shouldExit]);

    [serverObject release];
    [pool release];

    return;
}
```

Although the worker thread passes the `Calculator` object as the parameter to `setServer:`, the `NSConnection` object creates a proxy to the `Calculator` object behind the scenes and sends that back to the main thread instead. The proxy object responds to messages exactly as if it were the original `Calculator` object, so behaviorally there is no difference. The only difference is that when a method of the proxy `Calculator` object is called from the main thread, the parameters and calling information to the method are packaged and sent to the worker thread over the connection's ports.

By default, messages sent to a proxy object are forwarded over the connection synchronously; that is, the sender waits for the message to be processed and a reply received from the remote object. This occurs even for a method with a `void` return type, since the remote object can raise an exception that is passed back to the sender. Thus when calling methods of a proxy object, the local thread or application blocks until the message finishes executing. To avoid this, you can declare the method type as `oneway void`, which delivers the message asynchronously. If you expect a result back from the worker, and still want to use asynchronous messaging, the worker thread could return the results by calling a method of the root proxy object, just as it did with the `setServer:` method.

For more information on using distributed objects, see *Distributed Objects*.

Document Revision History

This table describes the changes to *Multithreading Programming Topics*.

Date	Notes
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.
	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.

Index