Technical Note TN2028
Threading Architectures

CONTENTS

This technote describes the various threading APIs on Mac OS 9 and Mac OS X, and how those APIs interact with the core operating system on each platform. It is not a practical guide to implementing threading, but an architectural overview of threading on each OS (with a few hints and tips along the way).

This technote is directed at anyone who wants to understand how Mac OS threading works. It would be especially beneficial to anyone porting a threaded application from Mac OS 9 to Mac OS X, where the platform differences may require one to revisit previously sensible decisions.

Updated: [Aug 08 2001]

Introduction

It should come as no surprise that Mac OS 9 and Mac OS X have different threading characteristics. While Mac OS X does support the Mac OS 9 threading APIs (as part of Carbon), it does so using an entirely different core OS; this core OS brings with it a number of subtle changes in behavior of those threading APIs.

This technote explains the basics of the threading architecture on both platforms. By understanding the fundamental design you can understand the subtle differences in how each platform handles threads. The ultimate goal is to make it easier for you to tune your threaded application to match the behavior of each platform. This should yield better performance for both your application and other applications on the system.

> **IMPORTANT:**
> If you're unfamiliar with the overall architecture of Mac OS X (you don't know the distinction between Mach, BSD, and IOKit), it would be a good idea to read the Inside Mac OS X: System Overview before starting this technote.

This technote describes the threading architecture for Mac OS 9.1 and Mac OS X 10.0.x, specifically. Things are different on older versions of traditional Mac OS and may be different on future versions of Mac OS X. Mac OS X is a rapidly evolving system. You should use this document as a guide to thinking about threading on Mac OS X, not as the final word about threading on that platform.

Back to top

Mac OS 9 Threading

This section describes threading on Mac OS 9. Mac OS 9 has two threading APIs.

- Thread Manager provides cooperatively scheduled threads within a process.
- MP tasks are preemptively scheduled by the nanokernel.

In addition to these threading APIs, it's impossible to understand Mac OS 9 threading without also understanding Mac OS 9 process scheduling, as implemented by the Process Manager.

Links to programming documentation for each of these APIs are given in the References section at the end of this technote.

> **IMPORTANT:**

While the "MP" in "MP tasks" stands for "multiprocessor," MP tasks are available and scheduled preemptively even on single processor systems. You don't need a multiprocessor system to take advantage of preemptive threading with MP tasks.

IMPORTANT:
Threading terminology can get confusing, especially on a system like Mac OS X that inherits terminology from many different sources. One particularly confusing term is "task." The MP API uses the term "MP task" to describe a thread of execution with a process. On the other hand, Mach uses the term "Mach task" to describe a collection of resources such as threads, memory, and ports (an idea more commonly known as a process). This technote always uses either "MP task" or "Mach task" to differentiate between these two concepts, and never uses the term "task" unqualified.

Note:
The title of this section is "Mac OS 9 Threading," not "Traditional Mac OS Threading." This section does not describe any obsolete traditional Mac OS threading concepts, such as:

- custom threading libraries that were popular before the introduction of Thread Manager,
- preemptively scheduled Thread Manager threads that were only supported on 68K systems, or
- Multiprocessing Services prior to version 2.0 (Mac OS 8.6). These were implemented by a shared library outside of the nanokernel and were not generally useful because they only worked with VM off.

Mac OS 9 Without MP Tasks

If you temporarily ignore MP tasks, the Mac OS 9 threading architecture is shown in Figure 1.

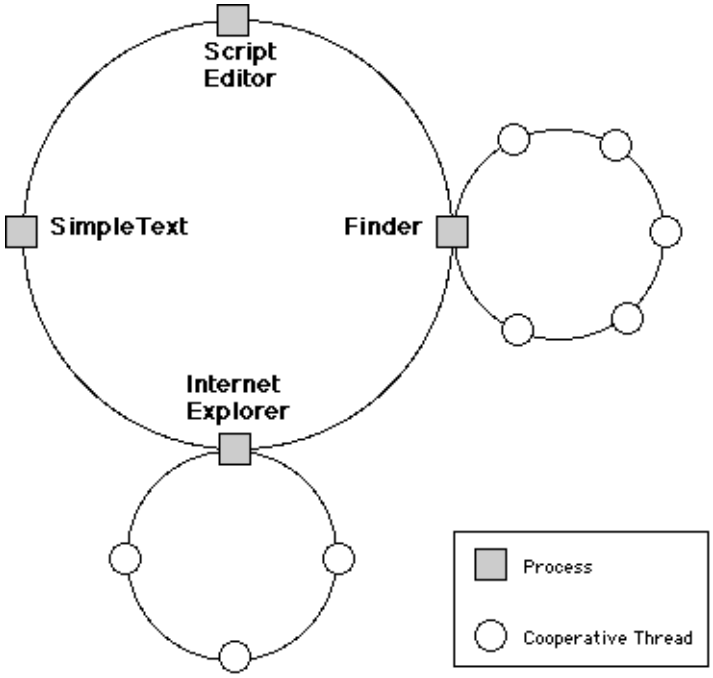


Figure 1. Mac OS 9 threading.

In this architecture, all scheduling is done cooperatively. Each process is round-robin scheduled by the Process Manager when the process calls a yielding function such as `WaitNextEvent`. Within each process there is a set of cooperatively scheduled threads, which are scheduled by the Thread Manager when the process calls `YieldToAnyThread` (or `YieldToThread`).

You can draw a number of conclusions from this diagram:

- Each process instantiates its own copy of the Thread Manager, which is responsible for scheduling threads solely within that process's context.
- Every thread belongs to a one specific process context.
- Because Thread Manager threads are always attached to a process and there is no system process, there is no way to create a system Thread Manager thread.
- When you yield to a thread, you are yielding to a thread in your process. There is no way for your process to directly yield to a thread in another process. Your threads must yield to your main thread, which in turns calls `WaitNextEvent`, which causes the Process Manager to schedule the other process's main thread, which can then yield to its threads.
- There is no way to yield to another process without handling user interface events. See DTS Q&A PS 06 Yielding Time

Without Getting Events for more on this topic.

- Because cooperative threads are local to your process, and thus don't involve any "kernel" resources, they are very lightweight and context switches between threads are cheap.

> **Note:**
> Some of the blanket statements above have noteworthy exceptions. For example, Apple software, such as NSL, does create system-wide Thread Manager threads. However, the mechanism that it uses is both private and not particularly pleasant. In addition, it is legal to call `WaitNextEvent` from a thread other than your main thread, but doing so tends to cause confusion. The confusion arises because you can't call `WaitNextEvent` without also accepting user events, which means you have to handle user events on your non-main thread.

One important feature of Mac OS 9 threading is that Process Manager processes are very expensive. They consume a lot of resources (processes have fixed-size memory partitions) and, for historical reasons, the context switch time between processes is very expensive.

Mac OS 9 With MP Tasks

If you consider MP tasks, Mac OS 9's threading architecture is only slightly more complex, as shown in Figure 2.

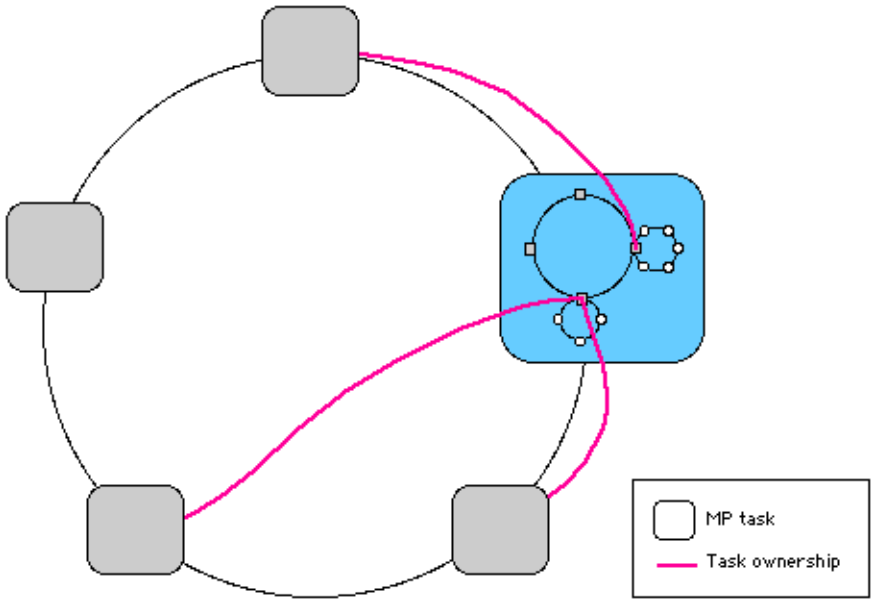


Figure 2. Mac OS 9 threading showing MP tasks.

The entire cooperative environment runs within a single MP task. This task is known as the **blue task**. All Process Manager processes and all Thread Manager threads are executed by the blue task. Other MP tasks, created either by the system or by an application, are executed as separate entities. The nanokernel schedules tasks to run on the processor (or processors) in a preemptive fashion.

> **IMPORTANT:**
> Figure 2 is a highly simplified representation of the nanokernel task scheduler. The scheduler has to deal with numerous complications that aren't represented here:
>
> - It must never run a task that's blocked (for example, waiting for queue).
> - On MP systems it must schedule tasks for each processor.
> - It must account for the various tasks' scheduling weights and latency requirements.

You should note the following consequences of this architecture:

- The nanokernel preemptively schedules all MP tasks, including the blue task, using its own internal scheduling policies. While you can affect these policies (for example, by calling `MPSetTaskWeight`) you are no longer in direct control of your scheduling points as you were with the Process Manager and Thread Manager. In fact, if the blue task is running the 68K emulator it can be preempted by another MP task midway through a single 68K instruction!
- The system tracks the MP tasks created by a process and ensures that these MP tasks are terminated when the process terminates. This relationship is shown by the magenta lines in Figure 2. However, as far as the nanokernel's task scheduler is concerned, all MP tasks are considered equal. The nanokernel can preempt process A's MP task and immediately switch to one of process B's MP tasks.
- However, the nanokernel has a number of special cases for the blue task. For example, when an MP task takes a page fault the nanokernel blocks that task then sends a software interrupt to the blue task to resolve the page fault. However, from the perspective of an external developer, all MP tasks are created equally.

- Only software running in the blue task has access to the 68K emulator. This means that MP tasks can directly use only a limited set of system services. This set of services is documented in DTS Technote 2006 MP-Safe Routines. Non-MP safe routines can be called via `MPRemoteCall`.

- The system providers both high- and low-latency communications between MP tasks and the blue task.
  - An MP task can use `MPRemoteCall` (passing either `kMPAnyRemoteContext` or `kMPOwningProcessRemoteContext` to the `context` parameter) to request that the blue task call a function at system task time. Because this function runs at system task time, there is a significant delay between when it is scheduled and when it is executed. The system must wait until the blue task calls one of the standard Process Manager descheduling routines (`WaitNextEvent`, `EventAvail`, and so on) before it can execute the function. However, the function is run at system task time, and thus has access to virtually all Mac OS system calls. As an example of how this service is used, `MPAllocateAligned` uses `MPRemoteCall` when it needs to grow the process's MP memory pool because the underlying Mac OS Memory Manager must be called at system task time.

  - Mac OS 9.0 and above allow an MP task to send a software interrupt to the blue task. This software interrupt is called with very little delay because it doesn't have to wait for the blue task to call a Process Manager descheduling routines. However, the software interrupt runs at interrupt time, and can only call system services that are interrupt-safe (see DTS Technote 1104 Interrupt-Safe Routines). Your code can schedule a software interrupt using either `MPRemoteCall` (under Mac OS 9.1 and above, passing `kMPInterruptRemoteContext` to the `context` parameter) or `DTInstall` (Mac OS 9.0 and above).

    Note:
    This software interrupt mechanism enables MP tasks to call synchronous File Manager routines on Mac OS 9.0 and later. When an MP task calls the File Manager, the File Manager puts the request on a queue and schedules a software interrupt to do the actual work at interrupt time within the blue task.

    IMPORTANT:
    Do not confuse the software interrupt mechanism described above with the software interrupts described in Designing PCI Cards and Drivers for Power Macintosh Computers. While conceptually similar (hence the names), the mechanisms are completely different. In fact, the latter is a Copland construct and was never actually implemented properly on traditional Mac OS.

Back to top

Mac OS X Threading

This section describes threading on Mac OS X, which provides five different threading APIs:

1. Mach threads represent the lowest level threading on the system.
2. POSIX threads (pthreads) are layered on top of Mach threads.
3. Cocoa threads (NSThreads) are layered directly on top of pthreads.
4. Carbon MP tasks, which are API compatible with the MP tasks in Mac OS 9, are layered on top of pthreads.
5. Carbon Thread Manager cooperative threads, which are also API compatible with their Mac OS 9 equivalent, are also layered on top of pthreads.

The Carbon Specification describes some of the issues you might encounter when porting MP task or Thread Manager code to Carbon on Mac OS X.

One useful way to think about Mac OS X's various threading APIs is to arrange them in an layered hierarchy. For example, each MP task is layered on top of a pthread, and each pthread is layered on top of a Mach thread. This hierarchy is shown in Figure 3.

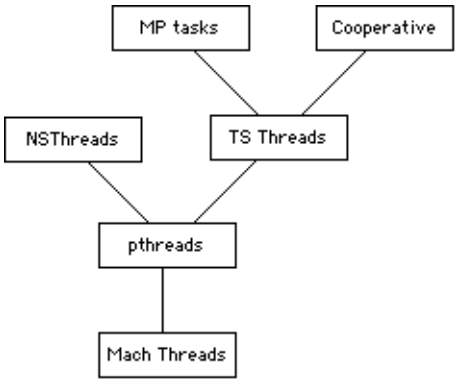


Figure 3. Mac OS X thread layering hierarchy.

Links to programming documentation for each of these APIs are given in the References section at the end of this technote.

> **Note:**
> Carbon implements a private threading abstraction layer known as Thread Support (TS) which is layered on top of pthreads. Both MP tasks and Thread Manager threads are currently layered on top of TS threads. Thread Support (TS) is not exposed as a public API and may well be removed in a future release of Mac OS X.

Thread API Choice

With five public threading APIs to choose from, it can be hard to decide which threading API to use. Here are some general guidelines:

- User-space processes should not create Mach threads directly.
- If you're working within the Cocoa environment, NSThreads are the obvious choice, although pthreads are also available and may make sense in some cases. For example, your Cocoa user interface might be acting as a front end to some existing code that already uses pthreads.
- If you're working in the Carbon environment the first decision you have to make is whether you can use preemptive threads or whether you must use cooperative threads. Many Carbon calls are not available to preemptive threads, even on Mac OS X. However, if you can use preemptive threads then you should; it's difficult to integrate cooperative threads into a well-behaved Mac OS X application.
- If your Carbon application has to use cooperative threads, your only choice is Thread Manager.
- If your Carbon application can use preemptive threads, you can choose either pthreads or MP tasks. However, you should take note of the following:
  - pthreads are not available on Mac OS 9.
  - Mac OS X does not provide a CFM binding for pthreads. If you want to call pthreads from a CFM application, you must create your own CFM-to-Mach-O glue. The CarbonLib SDK includes a number of samples that show how to do this.
- In all cases it's acceptable to mix different threads within the same process. For example, your Carbon application can create cooperative threads and MP tasks. Or your Cocoa application might use NSThreads and pthreads.
- You must be careful when mixing thread APIs on the same thread. This problem is described in a later section.

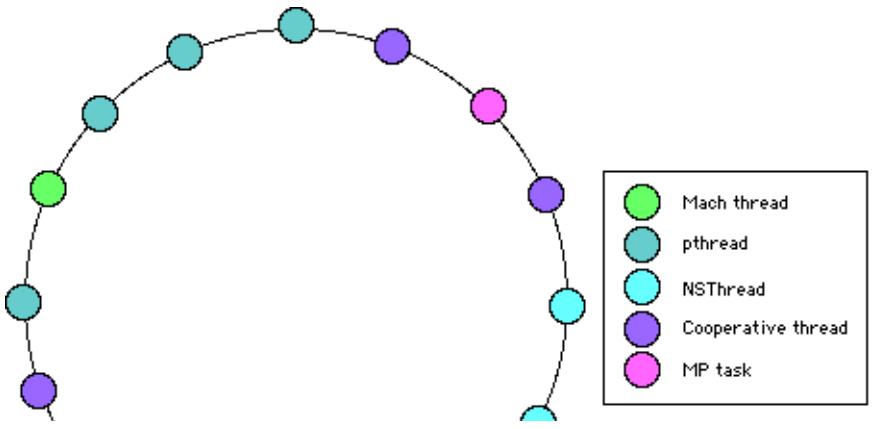
In contrast to Mac OS 9, all threads have roughly equal cost on Mac OS X. The overhead of wrapping a pthread up as an NSThread, MP task, or cooperative thread is small compared to the overhead of the base Mach thread. This also means that cooperative threads are more expensive (relative to other types of threads) than they are on Mac OS 9. For more details on the cost of threads in Mac OS X, see Inside Mac OS X: Performance.

> **Note:**
> Apple recognizes that using a Mach thread per Thread Manager thread is expensive and is looking at ways to reduce this cost in the future. However, this change is not in Mac OS X 10.0.x and will not be in Mac OS X 10.1.

Thread context switch time can vary depending on the threading API you choose. The overhead for an involuntary context switch and to block for I/O in the kernel is the same regardless of how you created the thread. The overhead for other context switches (for example, waiting on a semaphore) depends on the API you choose, with higher-level APIs generally being slower.

Mac OS X Thread Scheduling

One consequence of the thread hierarchy described earlier is that ultimately all threads are represented by Mach threads. As far as Mach is concerned, all threads are equal. When choosing a thread to run, the Mach scheduler always follows its scheduling policies to choose the next thread to run. It does not care whether the Mach thread was created as a pthread, an MP task, a cooperative thread, and so on. Thus you can represent the Mach scheduling as a single flat ring, as shown in Figure 4.



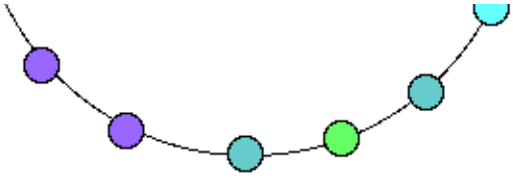


Figure 4. The Mach thread scheduler (highly simplified).

IMPORTANT:
Figure 4 is a highly simplified representation of the Mach thread scheduler. The scheduler has to deal with numerous complications that aren't represented here:

- It must never run a thread that's blocked (for example, waiting for I/O or a mutex).
- On MP systems it must schedule threads for each processor.
- It must follow the various threads' scheduling policies and priorities.

In addition, each Mach thread is owned by a particular Mach task. The relationship between threads and tasks is not represented in Figure 4, although in some scheduling policies this relationship is important.

To understand how cooperative threads are implemented, you need to turn this diagram on its side and look at how abstractions are layered on top of the fundamental Mach threads. Figure 5 shows this. At the bottom layer of each abstraction stack is the Mach thread that is actually scheduled by the kernel. Some Mach threads exist in their raw form (typically these threads are created to execute within the kernel). All user space Mach threads have a pthread layered on top of them. Threads created by Cocoa applications have an NSThread layered on top of that. Carbon can created threads both for itself and to implement higher-level threading APIs. If the thread is created for internal use by Carbon, it is implemented a very simple wrapper around a pthread. If the thread is created by Carbon at the behest of a Carbon application, it is also a pthread but with either a Thread Manager thread or an MP task layered on top of it.

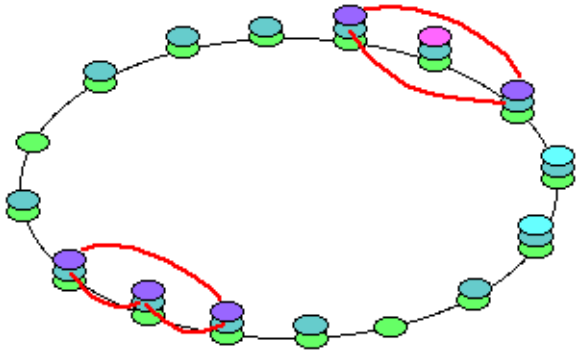


Figure 5. Mac OS X thread abstractions with cooperative thread token rings.

Carbon ensures that all Thread Manager threads created within a single process are scheduled cooperatively. Each process has a special synchronization token (the token is implemented as a Mach message) that is passed between the cooperative threads within the process. If the cooperative thread has the token it is allowed to run. If it does not have the token, it must block waiting for it. When the cooperative thread that has the token calls to `YieldToAnyThread`, Carbon chooses the next cooperative thread to run and passes the token to it. The new thread starts running and the original thread blocks waiting for the token.

This token passing arrangement is represented by the red lines in Figure 5.

Mach Scheduling Policies and Priorities

A Mach thread's **policy** controls the algorithm used to schedule the thread. The recommended thread policies on Mac OS X are:

- the standard policy (`THREAD_STANDARD_POLICY`), under which threads are scheduled by a system-defined fair algorithm
- the time constraint policy (`THREAD_TIME_CONSTRAINT_POLICY`), under which threads are scheduled according to real-time constraints, and
- the precedence policy (`THREAD_PRECEDENCE_POLICY`), which allows a task to specify the importance of a thread relative to the task's other threads.

For each policy there are a number of **policy parameters** that control how the thread acts within that policy. These parameters are roughly equivalent to the common idea of thread priority. For example, threads using the precedence policy have an "importance" parameter that controls the thread's priority with respect to other threads within that task. However, this simplistic equivalence breaks down for more the more complex time constraint policy, where the policy parameters are actually a number of real-time values.

Each threading API has its own idea of thread priorities. For example, MP tasks have the concept of task weight (as defined

by MPSetTaskWeight), while pthreads have thread scheduling parameters (set by pthread_setschedparam). Each threading API maps from its concept of priority to the underlying Mach policy and policy parameters. In general these algorithms are private and subject to change; however, you can learn more about how pthreads does this mapping by reading the Darwin source.

To learn more about Mac OS's recommended thread policies, see <mach/thread_policy.h>. Mac OS X also supports some older, deprecated thread policies, which are defined in <mach/policy.h>.

Mac OS X Thread Miscellanea

This section covers a number of miscellaneous topics related to threading on Mac OS X.

Main Threads

The main thread is a valid thread for all threading APIs. For example, you can call MPTaskIsPreemptive (an MP API) and ThreadCurrentStackSpace (a Thread Manager API) on the main thread. While this breaks a strict interpretation of the hierarchy shown in Figure 3, it is generally considered useful.

Carbon and Interrupts

Traditional Mac OS makes heavy use of asynchronous I/O with I/O completion routines that are delivered at interrupt time. However, the Mac OS X core OS does not support a callback-based asynchronous I/O model, and user space code never runs at interrupt time. Carbon simulates asynchronous I/O completion routines using preemptive threads.

For example, when you make an asynchronous File Manager request, Carbon simply puts the request on an internal queue and wakes up its asynchronous file I/O thread. That thread pulls the first item off the queue, executes it synchronously, and then calls its completion routine. If there are no more items on the queue the thread goes to sleep. The upshot of this is that the operations appear to operate asynchronously even though the underlying core OS does not have an asynchronous file system API.

Carbon uses techniques like this for many different interrupt time callback-based APIs, including File Manager, Time Manager, Deferred Task Manager, and Open Transport. Each subsystem creates its own pthread for making callbacks.

This design has a number of interesting consequences.

- As with traditional Mac OS, Carbon I/O completion routines are serialized within a given subsystem. For example, there is only one File Manager asynchronous I/O thread;, therefore, it can only be calling one completion routine at a time. This is generally considered a good thing.
- The fact that the file system subsystem within Carbon has only one thread for all asynchronous file I/O within a given process means that asynchronous file I/O is serialized. This is much like the File Manager on traditional Mac OS. However, the Mac OS X core OS does support multiple outstanding I/O requests. If you want to exploit this facility you should create multiple threads and have each thread do its own synchronous I/O.
- Carbon's I/O completion model differs from traditional Mac OS in that I/O completion routines do not "run to completion." Your I/O completion routine is executed by a preemptive thread. While this thread has a higher priority than the process's main thread, there is no guarantee that the I/O completion thread won't be preempted by that main thread. For example, if the I/O completion thread takes a page fault, Mach could decide to run the process's main thread. Moreover, if you're running on a multiprocessor system, the I/O completion thread and your main thread could be running simultaneously. If your application's synchronization model assumes that I/O completion routines run to completion, this change in system behavior can create bugs that are hard to reproduce, hard to debug, and hard to fix.
- Because all Carbon "interrupts" are actually executed by a preemptive thread, it stands to reason than any interrupt-safe routine is also preemptive-safe (on Mac OS X).

Mix 'n' match

With all of these threading APIs available, it's inevitable that at some point you will want to mix and match threads and their APIs. For example, you might want to call a Mach thread API on an MP task, or maybe an MP call on a pthread. If one thread is layered on top of another thread (assuming the layering hierarchy from Figure 3), you can, in general, do low-level operations on the high-level thread. For example, most pthread calls are safe to make on MP tasks. There are, however, a number of issues to consider.

The first issue is how to get a reference to the low-level thread. Only the pthread API provides an explicit mechanism to get a reference to the underlying thread (pthread_mach_thread_np returns the Mach thread for a given pthread). All of the other thread APIs provide no explicit support for this. The only way to get a reference to the low-level thread is to call the low-level "self" (or "current thread") API while in the context of the high-level thread. For example, to find the pthread for an MP task you can call pthread_self from within the MP task.

Another question is whether low-level operations actually work on a high-level thread. In general, most pthread APIs are safe on pthreads created by the higher-level thread APIs. On the other hand, you should be very careful when making Mach thread calls on pthreads (and hence on threads created by any high-level API). In general, it is not appropriate to make Mach thread calls on pthreads. Some exceptions are the Mach APIs for getting thread information, setting up thread exception handlers, and death notification, which should work for any Mach thread regardless of how it was created.

Finally, given that most high-level threads are implemented in terms of pthreads, is it safe to use high-level thread APIs on an arbitrary pthread? For example, is it safe to call MP API routines on a pthread? This approach definitely breaks the thread layering hierarchy and is not recommended. One notable, and useful, exception is that it is safe to make MP API synchronization calls (MPNotify/WaitOnQueue, MPSignal/WaitOnSemaphore, MPEnter/ExitCriticalRegion, and MPSet/WaitForEvent) on a pthread. This is useful because it allows you to use MP API routines to synchronize between your main thread and Carbon "interrupts."

Back to top

Mac OS X Kernel Threading

On traditional BSD-based systems the kernel is divided into upper and lower halves. The upper half is executed by the user thread that made a particular kernel call. It can block (sleep) while waiting for I/O to complete or for access to shared kernel resources. The lower half is executed as the result of hardware interrupts. Interrupts must not block. The upper half synchronizes with the lower half by disabling interrupts. McKusick, et al., provide a more in depth explanation of this design.

While Mac OS X uses a large part of the BSD kernel source code, it is not a traditional BSD system. Its Mach underpinnings require a rework of the BSD kernel synchronization mechanism. For example, Mach supports multiprocessor systems, so disabling interrupts is no longer a sufficient synchronization guarantee.

The upper half of the Mac OS X BSD kernel still runs in the context of the user thread that made the kernel call. However, the lower half is no longer executed in a hardware interrupt context. Hardware interrupts have a tiny scope within Mac OS X. When the hardware interrupt occurs it simply wakes up an IOKit workloop thread. These workloop threads are **kernel threads**; they are owned by the kernel, not by any BSD process. The workloop thread is the entity that executes the lower half of the BSD kernel. This means that the BSD kernel synchronization approach (using `splx` calls to disable interrupts) no longer works on Mac OS X; the upper and lower halves of the BSD kernel could be running on different threads on different processors!

The solution to this problem is the kernel funnel. The **kernel funnel** is a mutex that prevents more than one thread from running inside the BSD portions of the kernel at the same time. Each thread acquires the kernel funnel when it enters the BSD portion of the kernel, and releases it when it leaves. In addition, if a thread blocks (sleeps) while holding a funnel, it automatically drops the funnel, and thus allows other threads to enter the kernel.

Mac OS X 10.0.x implements **split funnels**. There is one funnel for the networking part of the kernel, and one funnel for the other BSD parts of the kernel (file system, process management, device management, and so on). This split results in a significant performance improvement for tasks that use both the disk and the network. Future versions of Mac OS X may use even more funnels, allowing even greater kernel re-entrancy, and even greater performance.

As far as the Mach scheduler is concerned, threads running within the BSD kernel are like any other Mach thread. One particularly interesting consequence of this is **kernel preemption**. When a high priority thread wakes up it will preempt a thread running within the BSD kernel. This happens regardless of whether the high priority thread is a kernel thread or a user thread. As long as the high priority thread does not attempt to acquire a kernel funnel (that is, it makes no BSD system calls), it can do its job despite the limited re-entrancy of the BSD parts of the kernel. This design allows Mac OS X to meet the real-time goals required by its real-time components, such as the highly responsive audio playback engine.

Back to top

Summary

With five different threading APIs, Mac OS X threading looks complicated, but it's actually simpler than Mac OS 9 threading. All you have to do is understand the fundamental concepts:

- Ultimately all threads are layered on top of Mach threads.
- The layers can be arranged into an hierarchy.
- You have to be careful when mixing thread APIs on threads from different layers.
- When scheduling threads, Mach ignores the threading API that created the thread.
- Carbon "interrupt" routines are simulated using preemptive threads.

References

Inside Mac OS X: System Overview

DTS Q&A PS 06 Yielding Time Without Getting Events -- This describes why it is impossible to yield to other processes without handling user interface events on traditional Mac OS.

DTS Q&A 1061 RunApplicationEventLoop and Thread Manager -- This discusses the problems with integrating cooperative threads into `RunApplicationEventLoop`-based Carbon application.

DTS Technote 1104 Interrupt-Safe Routines

DTS Technote 2006 MP-Safe Routines

Carbon Specification -- Describes some of the issues you might encounter when porting MP task or Thread Manager code to Carbon on Mac OS X.

Mach 3 Kernel Principles and Interfaces -- While these documents are somewhat out of date, they're still a valuable introduction to the Mach 3 kernel and its programming APIs.

Darwin -- The source code for Mac OS X's kernel (CVS module "xnu") and pthreads library (CVS module "Libc") is available as part of Apple's open source efforts.

The Open Group's Single UNIX Specification -- This includes reference documentation for the pthreads API.

David R Butenhof , *Programming with POSIX Threads*, Addison-Wesley, 1997, ISBN: 0201633922 -- A good introduction to pthreads programming for UNIX systems.

NSThread Class -- This is the reference documentation for the NSThread class.

Inside Macintosh: Processes -- This book describes the Process Manager and its APIs.

Inside Macintosh: Macintosh Toolbox Essentials -- This book includes a chapter on the Event Manager, which describes the all-important `WaitNextEvent` call.

Inside Macintosh: Thread Manager -- This book describes the Mac OS 9 Thread Manager.

Inside Carbon: Thread Manager -- This describes the Carbon Thread Manager.

Adding Multitasking Capability to Applications Using Multiprocessing Services -- This describes the Mac OS 9 MP task API.

Adding Multitasking Capability to Applications Using Multiprocessing Services -- This describes the Carbon MP task API.

Inside Mac OS X: Performance -- This book contains information about the memory cost of threads on Mac OS X.

Marshall Kirk McKusick, et al., *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley, 1996, ISBN: 0201549794

Back to top

Downloadables

Acrobat version of this Note (248K) Download

Back to top
