USENIX Association

Proceedings of the BSDCon 2002 Conference

San Francisco, California, USA
February 11-14, 2002

Advanced Synchronization in Mac OS X: Extending Unix to SMP and Real-Time

Louis G. Gerbarg
Apple Computer, Inc.
louis@apple.com

Abstract

Throughout the years, as Unix has grown and evolved so has computer hardware. The 4.4BSD-Lite2 distribution had no support for two features that are becoming more and more important: SMP and real-time processing.

With the release Mac OS X Apple has made extensive alterations to our kernel in order to support both SMP and real-time processing. These alterations affected both the BSD and Mach portions of our kernel, as well as shaping our driver system, IOKit.

These changes range from scheduling policies, enabling support for kernel preemption, altering locking hierarchies, and defining new serialization primitives, as well as designing a driver architecture that allows developers to easily make their drivers SMP and preemption safe.

1 Introduction

Traditional BSD kernels do some things very well. SMP is not one of them. The 4.4BSD-Lite2 source, on which NetBSD, FreeBSD, OpenBSD, and Mac OS X are based did not have support for SMP. Its locking mechanisms were not set up for multiple processors, the kernel was not reentrant, and bottom half (interrupt time) drivers always work directly within an interrupt context. As FreeBSD, Mac OS X, and NetBSD have moved to support SMP they have had to overcome these shortcomings. Some aspects of their solutions are similar, some are wildly divergent. Both xnu and FreeBSD have decided to adopt interrupt thread contexts, as well as a number of similar new locking primitives.

2 An Introduction to xnu

Before delving into the intricacies of Mac OS X's advanced features a brief overview of the kernel architecture and its history is necessary. Mac OS X is based around a BSD distribution known as Darwin. At the heart of Darwin is its kernel, xnu. xnu is a monolithic kernel based on sources from the OSF/mk Mach Kernel, the BSD-Lite2 kernel source, as well as source that was developed at NeXT. All of this has been significantly modified by Apple.

xnu is not a traditional microkernel as its Mach heritage might imply. Over the years various people have tried methods of speeding up microkernels, including collocation (MkLinux), and optimized messaging mechanisms (L4)[microperf]. Since Mac OS X was not intended to work as a multi-server, and a crash of a BSD server was equivalent to a system crash from a user perspective the advantages of protecting Mach from BSD were negligible. Rather than simple collocation, message passing was short circuited by having BSD directly call Mach functions. While the abstractions are maintained within

the kernel at source level, the kernel is in fact monolithic. xnu exports both Mach 3.0 and BSD interfaces for userland applications to use. Use of the Mach interface is discouraged except for IPC, and if it is necessary to use a Mach API it should most likely be used indirectly through a system provided wrapper API.

3 Basic Synchronization

Operating systems use a number of structures and algorithms to ensure proper synchronization between various parts of the kernel. xnu uses several different locking structures, including the BSD lockmanager, Mach mutexes, simple locks, read-write locks, and funnels. Additionally thread control is complicated by the use of Mach continuations, and kernel preemption.

3.1 Simple Locks

Simple locks in Mach are standard spin locks. When a thread attempts to access a simple lock that is in use it loops until the lock becomes free. This is useful when allowing the thread to sleep could cause a deadlock, or when one of the threads could be running in an interrupt context.

Simple locks are the safest general synchronization primitive to use when in doubt, but their CPU cost is very high. In general is is better to use a mutex if at all possible. If a piece of code attempts to acquire a simple lock it already holds it will result in a kernel panic.

```
void
simple_lock_init(
    usimple_lock_t,etap_event_t);

void
simple_lock(usimple_lock_t);

void
simple_unlock(usimple_lock_t);

unsigned
int simple_lock_try(usimple_lock_t);
```

3.2 Mutexes

Mach mutexes are very primitive. Since they are sleep locks, and do not have the rich semantics that FreeBSDs mutexes have. They are sleep locks, when a thread attempts to access an inuse mutex it will sleep until that mutex is available. Mutexes can be used from a thread context (though it is not always the best performance decision for things like drivers). If a piece of code attempts to acquire a mutex it already holds it will result in a kernel panic.

```
//The etap_event parameter is
//deprecated, just pass a value of 0.
mutex_t *
mutex_alloc  (etap_event_t);

void
mutex_free (mutex_t*);

void
mutex_lock (mutex_t*);

void
mutex_unlock (mutex_t*);

boolean_t
mutex_try (mutex_t*);
```

3.3 Read-Write Locks

Many variables within the kernel are safe to be read, so long as they are not being written. If a lock is highly contended, generally it is primarily being protected for readers. Read-write locks solve this problem by allowing either multiple reads, or a single writer to possess the lock. While there are API's for promoting and demoting locks between the read and write states, their usage is discouraged and subject to change.

```
void
lock_write (lock_t*);

void
lock_read (lock_t*);

void
lock_done (lock_t*);

#define lock_read_done(l) \
    lock_done(l);
#define lock_write_done(l) \
    lock_done(l);
```

3.4 Continuations

One of the costs typically associated with context switches is saving and restoring thread stacks. This uses both CPU time and wired memory. In order to avoid this cost, Mac OS X uses Mach continuations whenever possible. A continuation allows the kernel to avoid saving or restoring a kernel stack across schedulings of the thread.

Continuations work within a non-preemptible context. Since the thread is not going to be preempted, its entry and exit points are well-defined. The thread begins executing through a call to a function pointer. It ends execution by making a call that tells the scheduler to schedule a new thread, and leaves a pointer to a function that should be executed the next time the thread is scheduled. It is the thread's responsibility to save and restore its own variables.

While it is useful to be aware of continuations, it is not generally necessary to directly interact with them. They may be useful for doing extremely low overhead threading, but in general it is best to use them indirectly through other kernel mechanisms such as IOWorkLoops.

```
void thread_set_cont_arg(int);
int thread_get_cont_arg(void);
```

4 Funnels: Serializing access to BSD

Funnels are quite possibly one of the most confusing elements of xnu for people familiar with other BSD kernels. They are not a lock in the traditional sense of the word (though they are sometimes referred to as "flock" within the kernel). Funnels are used to serialize access to the BSD segment of the kernel. This is necessary because that portion on the codebase does not have fine-grained locking, and is not fully reentrant. There are currently two funnels within the kernel, the kernel funnel (it might be more appropriate to call it the filesystem funnel, though it does protect a few calls besids the file systems), and the network funnel.

4.1 Funnels

Funnels first appeared into Digital UNIX[dgux], though their implementation in Mac OS X is entirely different, and significantly improved. Funnels are actually built on top of Mach mutexes. Each funnel backs into a mutex, and once a thread gains a funnel it is holding that funnel while it is executing. The difference between a funnel and a mutex is that a mutex is held across rescheduling. The scheduler drops a thread's funnel when it is rescheduled, and reacquires the funnel when it is rescheduled. That means that holding a funnel does not guarantee that another thread will not enter a critical section before a thread drops the funnel. What it does mean is that on a multiprocessor system it is guaranteed that no other thread will access the section concurrently from another CPU.

Originally there was a single funnel protecting the entirety of the BSD kernel. It was in many ways analoguous to FreeBSD-current's Giant mutex (more on that later). Since networking and other kernel functions are generally seperate, splitting the funnel into two is a major win for dual processor machines. Unfortunately, since holding

both funnels can result in nasty deadlocks and other problems, holding both at the same time causes a panic. This can cause significant problems for entities that need to access items that are protected by each funnel. The primary entities this effects are network file systems. The funnel API has a call for swapping funnels, but in some cases this has proven to be too complicated to orchestrate (such as NFS serving). The API also provides a merge call which will combine the two funnels into a single funnel, backed by a single mutex. Unfortunately, the funnels cannot be unmerged, which causes a net performance loss.

The primary difference between Digital UNIX funnels and Mac OS X funnels are that on Digital UNIX there can only be one funnel, and it always will be on the primary CPU. On Mac OS X there can be multiple funnels, and funnels can run on any CPU (although a particular funnel may only be on one CPU at any given time).

There are primitives for creating funnels, but in general nobody should be creating new funnels. All control of the funnels is done through the thread_funnel_set call().

```
boolean_t
thread_funnel_set
    (funnel_t * fnl, boolean_t funneled);
```

4.2 So long spl...

In BSD the various spl priority levels formed a locking hierarchy that could be used to guarantee synchronization between the interrupt and non-interrupt segments of a driver. Unfortunately the spl's definitions got less and less fine-grained over the years, and they were never particularly well suited for SMP. For these reasons Mac OS X no longer uses them. Instead it manages its synchronization through mutexes, and the BSD funnel serializations.

If this sounds familiar to FreeBSD users, that is probably because FreeBSD-current actually has a funnel (or rather a magic mutex), Giant. FreeBSD plays scheduler games with Giant that are almost identical to what xnu does with funnels, although Mac OS X deals with them explicitly, through a different API than its mutexes. Like FreeBSD, xnu has replaced the functionality of the spl's with these more flexible syncronization primitives. Unlike FreeBSD, the spl calls are still sprinkled through the kernel. Through the development of xnu they have been no-ops, wrappers to getting the funnels, and most recently they act as asserts to make sure the funnels are in the correct state when they are called.

5 Real-Time

There are two important aspects to real-time scheduling. One is the scheduling algorithm, the other is guaranteeing latencies within the kernel are not excessive. While both will be discussed, this section focuses mostly on the latencies related issues.

5.1 Interrupt Handling

True interrupt handlers cannot be preempted, and cannot sleep. Therefore, if there is a long path in an interrupt handler it will lead to high latency. In order to handle this, xnu generally uses a simple interrupt handler that processes the interrupt by triggering a handler in a regular kernel thread context that a driver has registered for the interrupt handler. This "pseudo interrupt" handler is run in a normal kernel thread context, where it can access the full kernel API. If true interrupt handling is necessary the correct mechanism is generally an IOFilterInterruptEventSource (see below).

5.2 Scheduling Bands

xnu internally has 128 priority levels, ranging from 0 (lowest priority) to 127 (highest priority). They are divided into several major bands. 0 through 51 correspond to what is available through the traditional BSD interface. The default priority is 31. 52 through 63 correspond to elevated priorities. 64-79 are the highest priority regular threads, and are used by things like WindowServer. 80 through 95 are for kernel mode threads. Finally 96 through 127 correspond to real-time threads, which are treated differently than other threads by the scheduler.

5.3 Fixed and Degrading priorities

By default the scheduler creates threads with degradable priorities. These threads will have lower and lower effective priorities as they use (and abuse) their time allocations. This is particularly significant for real-time threads, since if they are truly abusive they will eventually degrade into non-real-time threads. This mechanism means that it is possible to allow non-superusers to create real-time threads.

There are also mechanisms to create fixed priority threads which will not degrade. Their creation is much more restrictive than degradable threads, since they can be used very effectively to perform a denial of service against a system.

5.4 Kernel Preemption

Kernel preemption is the main tool xnu uses to achieve low latencies. The kernel is preemptible, though in standard usage kernel preemption is turned off. Kernel preemption begins when a real-time thread is scheduled. Since the real-time thread has a higher priority than a kernel thread it should be scheduled in favor of the kernel thread, and that is the point at which kernel preemption is activated.

Preemption changes the runtime characteristics of the kernel dramatically. Continuations are no longer nearly as useful, since the thread may be rescheduled at any point, which will require a stack. Additionally, all sorts of new deadlocks can arise. In order to cope with this the locking primitives have been modified to work with preemption. Simple locks disable preemption while they are spinning. Mutexes only disable preemption while the thread is trying to gain access to its interlock (a spin lock protecting the mutexes private data structures). Additionally the true interrupt handler is not preemptible

What this means is that well written code should not need to be at all aware of the fact that kernel preemption is enabled, and should just work if they properly use the locking primitives. It should be transparent to most kernel extensions and drivers. It may not be transparent if the driver uses an IOFilterInterruptEventSource, or does not make proper use of an IOWorkLoop, as described in the next section.

6 IOKit

IOKit is the driver subsystem of the Mac OS X kernel. IOKit provides a number of synchronization primitives, ranging from simple wrappers to the Mach primitives, all the way through complex new synchronization constructs that massively simplify writing drivers for devices that are SMP clean and preemptible. IOKit is implemented in eC++ [eC++], a subset of C++, and uses a custom runtime type system.

6.1 IOLocks

IOKit provides wrappers to the Mach locking primitives. These wrappers provide some convenience as well as a consistent interface to the locking primitives.

6.1.1 IORWLock

IORWLock provides a wrapper to the standard Mach read-write locks.

```
IORWLock *
IORWLockAlloc( void );

void
IORWLockFree( IORWLock * lock);

void
IORWLockRead( IORWLock * lock);

void
IORWLockWrite( IORWLock * lock);

void
IORWLockUnlock( IORWLock * lock);
```

6.1.2 IORecursiveLock

IORecursiveLock provides a wrapper to the standard Mach mutexes. Additionally, it has an internal reference counting mechanism that allows it to be locked recursively.

```
IORecursiveLock *
IORecursiveLockAlloc( void );

void
IORecursiveLockFree(
    IORecursiveLock * lock);

void
IORecursiveLockLock(
    IORecursiveLock * lock);

boolean_t
IORecursiveLockTryLock(
    IORecursiveLock * lock);

void
IORecursiveLockUnlock(
    IORecursiveLock * lock);
```

6.1.3 IOLock

IOLock provides a wrapper to the standard Mach mutexes. The semantics are the same. Recursive locking is not allowed.

```
IOLock *
IOLockAlloc( void );

void
IOLockFree( IOLock * lock);

void
IOLockLock( IOLock * lock);

boolean_t
IOLockTryLock( IOLock * lock);

void
IOLockUnlock( IOLock * lock);
```

6.1.4 IOSimpleLock

IOSimpleLock provides a wrapper to the standard Mach simple_locks. Additionally, it has an interface for enabling and disabling interrupts (drivers should probably be using a IOWorkLoop for synchronization, which will take care of interrupt related issues).

```
IOSimpleLock *
IOSimpleLockAlloc( void );

void
IOSimpleLockFree( IOSimpleLock * lock );

void
IOSimpleLockLock( IOSimpleLock * lock );

boolean_t
IOSimpleLockTryLock( IOSimpleLock * lock );

void
IOSimpleLockUnlock( IOSimpleLock * lock );

IOInterruptState
```

```
IOSimpleLockLockDisableInterrupt(
    IOSimpleLock * lock );

void
IOSimpleLockUnlockEnableInterrupt(
    IOSimpleLock * lock,
    IOInterruptState state );
```

6.2 IOWorkLoop

IOWorkLoops are constructs designed to simplify synchronization issues that arise when working with hardware in the multi-threaded, reentrant, preemptible environment present within xnu. Unlike the other locking primitives discussed earlier in this paper, the IOWorkLoop is a very complex entity that takes care of most of the more mundane synchronization issues for driver writers. Its interface is rather extensive, and somewhat complex.

The basic idea behind a work loop is that it forces anything attached to the work loop to run effectively single threaded. So while anything is holding the work loop none of the other event handlers or runActions associated can run. This effectively synchronizes the various items that are attached to the work loop. It also provides a convenient mechanism for servicing interrupts and timers while keeping them synchronized.

The work loop also takes care of a bunch of mundane issues such as turning on and off interrupts during certain locking procedures, meaning that driver writers can concentrate on getting their drivers working, not keeping their locking straight. Inherently there is some overhead in using work loops, and they do not serve every purpose, but they are quite flexible, and allow programmers to write correct drivers without intimate knowledge of xnu's internal synchronization mechanisms.

6.2.1 EventSources

IOEventSources are very flexible constructs for dealing with asynchronous events. While it is possible to implement new event sources, in general the provided IOInterruptEventSource and IOTimerEventSource are sufficient.

Event sources allow functions to be associated with asynchronous events, such as interrupts and timers. The full details and subtleties of how they work falls outside the scope of this paper, but the basic interfaces for creating new event sources are provided below.

Once an event source has been created it can then be added to a work loop. After that any time the event happens it will automatically be processed by the function that was specified when it was created, and in the work loop context.

```
IOTimerEventSource *
IOTimerEventSource::timerEventSource(
    OSObject *owner, Action action = 0);

IOInterruptEventSource *
IOInterruptEventSource::interruptEventSource(
    OSObject *owner, Action action,
    IOService *provider = 0,
    int intIndex = 0);

virtual IOReturn
IOWorkLoop::addEventSource(
    IOEventSource *newEvent);

virtual IOReturn
removeEventSource(
    IOEventSource *toRemove);
```

6.2.2 runActions

Event sources solve a significant amount of the synchronization issues drivers face dealing with the bottom half (interrupt time) of the driver, but they do not deal with the synchronizing the top half (non-interrupt)

and bottom half of the driver. This synchronization is achieved through the use of runActions.

runActions simply link a particular invocation of a function to the work loop. While the runAction is operating it is holding the work loop, thus forcing synchronization with everything else on the work loop, including the interrupt and timer event handlers.

```
typedef IOReturn (*Action)
    (OSObject *target,
    void *arg0, void *arg1,
    void *arg2, void *arg3);


virtual IOReturn
IOWorkLoop::runAction
    (Action action, OSObject *target,
    void *arg0 = 0, void *arg1 = 0,
    void *arg2 = 0, void *arg3 = 0);
```

6.3 IOFilterInterruptEventSource

IOFilterInterruptEventSource is a subclass of IOInterruptEventSource It is special because in addition to running within the work loop thread's context it runs directly on the primary interrupt context. This allows for much faster interrupt response time, but also means that an IOFilterInterruptEventSource cannot block, and must not use any kernel API that may block. In general IOFilterInterruptEventSoures should be used for cases where there are a lot of potential spurious interrupts, such as when a device shares an interrupt, or when processing only needs to be performed after several interrupts. The IOFilterInterruptSource can choose to ignore the interrupts that do not need processing, and pass the ones that do need processing onto an IOInterruptEventSource. A full description of limitations imposed on code running within the primary interrupt context is beyond the scope of this paper.

7 Conclusions

Darwin provides a number of synchronization primitives, both traditional and unique. They provide mechanisms for writing high performance drivers, without requiring driver writers to become intimately familiar with the OS. This both simplifies driver bring up, and encourages more people to write Mac OS X drivers for their devices.

Mac OS X is an evolving system, and many of these features are still in their infancy. Over time it will likely evolve into a more fine grained locking model, with certain compromises that are currently present will be phased out. The basic architecture needed to support SMP and real-time exists, and for most things the interfaces should remain stable for the foreseeable future.

References

[eC++] *eC++ Overview*,
`http://www.infoxpress.com/reviewtracker/reprints.asp?page_id=840`, (1997)

[dgux] *Digital UNIX Writing Device Drivers: Advanced Topics*,
`http://www.unix.digital.com/docs/dev_doc/ DOCUMENTATION/PDF/AA-Q7RPB.PDF`, (1996)

[iokit] *Mac OS X: I/O Kit Fundamentals*,
`http://developer.apple.com/techpubs/macosx/Darwin/IOKitFundamentals/index.html`, (2001)

[kernenv] *Mac OS X: Kernel Environment*,
`http://developer.apple.com/techpubs/macosx/Darwin/General/KernelEnvironment/ndex.html`, (2001)

[microperf] Herman Härtig, Michael Hohmuth, Jochen Liedtke, Sebastion Schoönberg, and Jean Wolter, *The Performance of μ-Kernel-Based Systems*, 16th ACM Symposium on Operating System Principles (1997)