# IOKit Techniques.

## Threading in the Darwin Kernel.

Godfrey van der Linden
IOKit Architect
gvdl@cse.unsw.edu.au

# Prelude

- In 2003 I was a kernel developer for Apple Computer, Inc
- What follows is a presentation I gave at the Apple World Wide Developers Conference (WWDC)
- The audience was a group of driver developers who where lost and confused by the basic concepts of threading, Classic MacOS didn’t support threads
- Although this lecture is Mac OS X centric, the design and concepts are general
- IOKit design has clean IP, so steal away :-)

# Introduction

- A running Mac OS X system has some 150 threads even on an otherwise idle system. Mostly these threads are ‘sleeping’ in the kernel
- This session will discuss the various types of threads in a running system and how they interact with each other
- The intended audience are those developers that are writing kernel extensions (kexts) or applications that tightly integrate with them

# What you’ll learn

- You should get a better understanding of how threads work on Mac OS X
- How IOKit does its synchronisation
- How does Mac OS X do hot unplug in a multi-threaded environment

# Darwin kernel

- Mostly this presentation will be about the threading service provided by the Mach part of the Darwin kernel
- I will also discuss IOKit’s synchronisation
- And IOKit hot-unplug as a design problem. It is very difficult to free resources in a dynamic environment

# Thread scheduling

Or how what you don’t know can hurt you.

- Over the next few slides I shall explain how threads on a Mac OS X system interact with each other
  - X’s thread priority bands
  - How the X dispatcher works
  - What does a scheduler do?
  - So what does high priority mean?
  - Priority inversions

# X’s thread priority bands

## “I am not a number”.

| |
|---|
| Primary Interrupt |
| Real - Time Band |
| thread_call timers |
| Page Out Thread |
| IOWorkLoops & kernel_threads |
| Top user band & Window Manager Events |
| Carbon Async Threads |
| Regular User Time Share Threads |
| Idle Threads (one per CPU) |

# How the X dispatcher works

## Do one thing well, then move on.

- The dispatcher is half of what is usually called a scheduler.  It puts a thread to sleep then chooses and executes the next thread
- Every time the dispatcher is invoked it selects the highest priority runnable thread and runs it
- When a thread is finished running the dispatcher adds it to a thread linked list to either wait for some event or its next turn at the CPU
- L4 uses basically the same model, though it doesn’t transit through the dispatcher as often.

# What does a scheduler do?

## Oversight committee.

- The scheduler’s job is to try to balance the load on a system and to try to identify naughty/greedy threads
- The scheduler manipulates the run-queue that the dispatcher uses
- It records statistics on CPU usage per thread and, depending in policy, it computes and changes relative priorities

# What does a scheduler do?

- Example 1: Time Share thread.  If a thread is running for sufficient time then its internal priority is dropped
- Example 2: Misbehaving real-time thread.  Say a real-time thread goes into an infinite loop.  The scheduler detects this and takes the thread out of the real-time band and drops it down to user base priority

# So what does high priority mean?

## You only have 100% CPU?  But I want more!

- It is easier to define what it doesn’t mean.  High priority does not mean your thread goes faster
- A high priority gives you is a reasonable chance of getting low latency when waking up.  Remember you compete with other threads in your priority band
- If you choose a very high-priority you may be starving tasks that you depend on.  This is called a priority inversion

# So what does high priority mean?

- You need to really understand your latency requirements
  - Are you reacting to user events, for instance a midi-keyboard or a mixing board?
  - Do you need to react to data off the internet?
  - I’m waiting for the disks to get around to me

# Priority inversion!

## I want highest priority except when I don’t!

- Although priority inversions can happen under any circumstances, the Mac OS X real-time model trips over them constantly
- Unlike most other OSes, X’s real-time band has a higher priority than the I/O system.  This gives X extraordinarily low ‘jitter’ even under extreme system loads
- But it comes at a cost of complexity.  You need to give the rest of the system time to do its job
- And the amount of time a system needs is very hard to determine in advance

# Priority inversion strategies

- Best: Lower your priority.  Ask yourself why you need to execute at such a high priority?
- Next best: In kernel you can use locking priority handoff
- Good: Redesign your threading model, run only non-I/O dependent code at real-time in a *consume*r thread.  Lower the priority of the *producer* below the I/O thread priority, the top of the *user* band is nice
- Bad: Enforce no more than about 90% usage. Issuing a 1ms usleep() every 10ms will do this

# IOKit’s synchronisation model

- IOKit has a very unusual synchronisation model. Mac OS X runs clients threads directly in the kernel context
- On a context switch, the thread changes stack, but that is it, specifically a thread executing in the kernel is still pre-emptable
- Writing drivers in a massively multi-threaded environment is difficult
- IOKit’s IOWorkLoop provides a very efficient single threading context for drivers

# IOWorkLoop

## The IOWorkLoop is not a thread!

- It is better to think of a work loop as a lock. IOWorkGate would have been a better name
- A work loop is a container for a gate (i.e. a recursive lock), a list of event sources and a thread that is **only** used for delivering interrupt events
- The work loop’s gate is closed around all event source Action routines
- The concept of a work loop actions was stolen from Java’s synchronous functions

# IOWorkLoops and driver stacking

## If you only have one lock you can’t deadlock! Cool.

- By far the majority of IOKit drivers do not create a work loop but rely on their provider’s work loop. This is a recursive statement
- In general only PCI and motherboard device drivers create a work loop and override the IOService:: getWorkLoop() function
- If you don’t expect a lot of interrupts and you don’t have tight timing it is perfectly reasonable to use the system’s default work loop

# IOEventSource

## Something happened!  Really?

- Every work loop has a linked list of event sources associated with it
- Remember that IOKit is an object oriented driver model, an event source is a super class for *events*
- An event is any time an *Action* routine must execute in a synchronous environment
- The next few slides will discuss the more common subclasses that driver developers will use

# IOFilterInterruptEventSource

## What, where, who, why?

- This event source is used to deliver hardware interrupts to a driver
- It works by taking the primary interrupt and waking up the IOWorkLoop’s thread.  This is the only mechanism that can wake the thread
- The Action routine is synchronous w.r.t. the work loop but the Filter is totally Async
- It is recommended that you always implement a working Filter routine for all hardware device

# IOTimerEventSource

## Is it time already?

- A timer can be used for many reasons but probably the most common is for hardware timeouts
- The IOKit timer is based upon the thread_call.h APIs.  Note timer threads have higher priority than interrupt threads
- Warning: there is currently no synchronous way of canceling a timer.  The safest way to delete a timer is to wait for it to execute
- The timer’s Action routine is synchronous w.r.t the IOWorkLoop

# IOCommandGate

## You want to do what?

- A command gate isn’t a lock.  It provides access to the work loop’s gate.  I.e. no matter how many command gates you have, there is only one gate per work loop
- Command gates allow you to run code synchronously w.r.t. the work loop without a thread switch.  A client thread can schedule I/O requests directly to hardware
- It wraps access to a recursive lock, which is why they use the clunky runAction() interface

# commandSleep/commandWakeup

## No work? It’s nap time

- The command gate’s sleep and wakeup routines allow you to block a client thread until some condition is true
- For example: You are writing a driver for a data collection device
  - Your application calls into your driver and you don’t have any data yet.  Call commandSleep to block
  - Some data arrives, use your interrupts Action to wakeup your client thread
- Used to emulate interrupt callouts

# Synchronous device tear down

## “Oh No!  My device has gone!”

- Tearing down a stack has been one of the hardest things to get right in IOKit.  In fact we went through two design iterations and they still haven’t converted all of the drivers to the new teardown model (five years later)
- It is very easy to delete a driver object while a client thread is calling down the stack.  This causes all sorts of obscure panics
- Obviously we can not destroy a device until the clients are ready for it

# Synchronous device tear down

- Our solution has been to use work loop’s to provide a synchronous contexts
- When a bus discovers that a device has gone; it calls IOService::terminate(), while on the work loop, on the device nub that has gone
- IOService::terminate() calls doTerminate()

```
doTerminate() {
    client->willTerminate();
    for each client
        client->doTerminate();
    client->didTerminate();
}
```

# Synchronous device tear down

- The responsibility of a driver depends on its position in the stack.  If it is an intermediate driver it will probably override the willTerminate() function
- willTerminate() completes all outstanding I/O requests with a kIOReturnOffline error
- The top of the driver stack should see that **all** outstanding asynchronous I/Os have been completed with an error

# Synchronous device tear down

- If the driver is on the top of the stack it overrides didTerminate()
- When a driver receives didTerminate(), it needs to :-
  - Stop all future calls to its providers
  - Track any currently active call in the driver stack
  - close() providers as soon as possible
- This can be tricky to implement.  Usually IOKit provides the TOS drivers, but remember an user client is on the top of a driver stack