Mac OS X IOAudio

Godfrey van der Linden

COMP9242 2009-10-09

Who am I

- I'm Godfrey van der Linden
- Past member of Apple's Mac OS X CoreOS kernel group
- One of the two architects for the object oriented input/output infrastructure group, IOKit
- I designed and implemented much of IOKit's synchronisation (threading model) and its user/kernel memory description objects
- Involved in many family designs, though I personally only owned the serial family

Audio case study

- This presentation is a case study of Mac OS X's IOAudio family
  - First hour will describe the design process
  - Second hour will discuss subtle threading issues with the design
  - Last hour will discuss the latest IOAudio's evolution
- Questions are welcome and in the last hour I'd prefer to lead a discussion rather than lecture
- Slides for the last hour will be posted soon after this lecture

Mac OS X

- Mach ‘micro’-kernel encourages light-weight thread use
- Also provides extensive, flexible and slow virtual-memory APIs
- Dispatching is priority sorted, with round robin within 127 priority levels
- ‘Time constrained’ - threads are considered soft real-time
  - There is no guaranteed scheduling
- High priority does not increase CPU allocation but rather speeds thread re-schedule
- Only primary interrupts have a higher priority than time-constrained threads

Thread priority bands

Primary interrupts
Time constrained band
thread_call timers
Page-out thread
IOWorkLoops & kernel_threads
Top user band & Window Manager events
Carbon Async threads
user time-share threads
Idle threads (one per CPU)

Why is audio interesting

- In 1998 Apple was dying
  - Still held onto its multi-media customers, just!
- They needed a modern operating system and must not loose their remaining loyal customers
- Audio is very hard to do
  - Especially on an established platform
- IOAudio is the IOKit solution to this problem
  - It required extensive ground up implementation from interrupt path through to user land libraries
    - Kernel: Interrupt latency, thread priority protocols
    - User-land: Source code compatibility with Classic

Goals

- Must be capable of delivering glitch free audio
- Minimise system load
- Mixing audio from multiple processes and threads
- Shortest possible input to output audio latency
- Highly reactive to audio controls
- Reasonably bounded ‘jitter’, i.e. variation from scheduled code start time
- Clients must live entirely in user-space
- Kernel has its own address space, typically 4/4KVM
  - This implies expensive user/kernel transitions

Classic audio drivers

- Audio drivers typically used the double buffer paradigm
  - After buffer completion, next buffer is scheduled...
  - ...and all clients are then called to fill the alternate
- Problems
  - Single address space
  - Low latency required very small buffers, which forced high interrupt rates. Usually not required
  - Totally dependant on other interrupt handlers
  - Clients often have an optimal buffer size for their algorithm, usually requires yet more double buffers and complexity

Classic duty cycle

1. Wait for a buffer done interrupt
2. On interrupt call out to sound mixing library to fill buffer
3. On return, wrap buffer in DMA request
4. Schedule DMA

- Step 2 can take an arbitrarily long time, if it doesn't return promptly the audio engine will idle
- If buffers are too big then audio isn't reactive to user input
- If buffers are too small then too many interrupts are delivered
- If sound clients live in a different address space then switching costs soon mount

Introducing karaoke

- It seems trivial use, but internet streaming background and mixing microphone input is one of the hardest applications to support
  - It requires microphone to speaker latency < ~4ms
  - Sound track is streamed across the internet, i.e. long delays
  - Video is expensive in CPU resource
  - May need to signal process input for echo removal, needs actually output audio
- Ideal implementation would be to combine a public address app and a streaming media player, that's karaoke

Why is karaoke hard

- The PA app requires very low latency but is otherwise lightweight
- The background stream needs large buffers to download into and is usually heavyweight
- Conflicting requirements, the traditional solution is to run with very small audio buffers, calling every client per buffer
  - Very expensive, on virtual memory systems
  - Background must also double buffer. From natural decode buffers to small low latency buffers

Observations

- Observe that audio plays at a known and constant rate
- Also most modern audio hardware has advanced DMA
- Hardware reads data when it needs it, not when DMA is scheduled
- Audio hardware often allows for free-running DMA program, that is the engine automatically loops back to the first DMA
- Since the audio rate is known it is possible to determine which sample is playing
  - Any buffer size can be simulated using timer interrupts instead of primary audio interrupts and each client can use any buffer size that suits their algorithm
- Notice that the DMA runs all the time, with or without data

The audio ‘framebuffer’

- Audio has some similarities with video
  - A piece of hardware automatically reads memory
  - ... and outputs the data through a set of DACs
  - Compositing layers graphics into video memory
- We can consider mixing of an audio stream to be a compositing operation
- Instead of using xy-coordinates, an audio mixbuffer is addressed with time-coordinates
- A client plays audio by
  - presenting a buffer to the driver with a start time
  - driver translates time to an address and accumlates onto the specified location

In kernel

- You may have noticed a hole in this design
  - Unlike video audio compositing is inherently additive
  - The buffers must be zeroed after they have been played
  - Some already played data can be convenient for echo cancelation
- The kernel driver's task is to:-
  - Build a time/sample mapping, Interrupt timestamps
  - Accumulate each client stream into the mixbuffer
  - Convert the mixbuffer format into hardware samples
  - At some point after samples are played, zero the mixbuffer out
  - In case of a kernel panic, zero the sample buffers

Userland audio?

- You may have noticed that this design would work as well in user land, by mapping the mixbuffer into every address space
- Unfortunately mixing, which is accumulative (+=), must be serialised
- We realised that taking and dropping an interprocess mutex required 2 kernel roundtrips plus one more for the data transfer and finally a call to sleep the thread
- But doing the serialisation, mixing, sleeping & input data copying in the kernel only had a single kernel round trip
- What a shame, basic rule for everybody else is to stay out of my kernel!
- Have you seen what we missed, at the time?

Human sensitivity

- Early in the design process it was apparent that humans are very sensitive to missed samples
- The user land audio library team convinced us that they must have the highest priority in the system to provide glitch free sound
- We warned them that this is "just soooooo" dangerous
  - Infinite loops would not be debuggable
  - You must give up the processor occasionally to allow the system to run
- We caved and the audio threads in OS X are the highest priority in the system, only primary interrupts run at a higher priority

Audio duty cycle

- Register a pair of input and output buffers with the kernel
  1. Fill output buffer
  2. Call kernel driver with output buffer and a deadline
  3. Compute starting sample of buffer
  4. Mix data into floating point buffer and clip into sample buffer if required
  5. Block thread until deadline passes expires
  6. Convert input samples to floating point
  7. Return input data to user land
  8. Repeat

Kernel responsibility

- Track number of registered client processes
- Accumulate each client's data into the mixbuffer
- Track buffer start times precisely using CPU counters
  - On interrupt a sample counter is read from the hardware
- Convert from CPU time to sample clock and back
- When last client buffer is presented for a time period clip the floating point data into hardware samples
- Setup a timer interrupt to erase the outgoing data
- We chose to use 3 x 1/8s buffers for most drivers

Implementation

- Mac OS X was a brand new system. It was not necessary to maintain backwards compatibility with classic mac's 68K driver model
- This design was only practical by keeping very tight control over primary interrupts
- But we were free to define an I/O model that uses threads to prioritise interrupts
- During the primary interrupt the interrupt line is disabled until the workloop calls the interrupt handler
- Most Apple hardware does NOT share interrupts, this means that only add-on PCI drivers are *required* to implement primary interrupt filters

Intermission

- Consider the implications of audio processing being at such a high priority compared to drivers and the window manager

Highest priority

- The first problem we had to fix was the infinite loop debugging
  - Scheduler enforces a maximum thread run-time and reclassifies a real-time thread to the time-share band
- What about the I/O threads?
  - No easy solution, the audio engine must regulate its system use and give up the processor regularly
- But even giving up the processor may not be sufficient, what about audio input from one of the serial buses?
- In this hour I will describe two completely different solutions to solve the same basic priority inversion

Thread priority bands

Primary interrupts
Time constrained band
thread_call timers
Page-out thread
IOWorkLoops & kernel_threads
Top user band & Window Manager events
Carbon Async threads
user time-share threads
Idle threads (one per CPU)

System threads

- Audio threads are in the real-time band
- OS X has a simple dispatcher
  - The highest priority run-able thread is dispatched
  - Continues till end of time-slice
  - Round-robin within each priority
- What does this mean?
  - Audio threads can starve the rest of the system
  - OS X's scheduler is not 'fair'.
  - Real-time band have responsibilities
  - ..but they only operate with limited information

Serial bus input problem

- Input data from these buses are not laid down contiguously
  - 44100smpl/s uses 9 x 44 and 1 x 45 sample 1ms buffers
- Under heavy system load the firewire and usb bus work loops are held off
- When an audio DMA chain completes the buffers are converted to floating point and copied into the input buffer
- The data is on the system but can not be accessed until workloop is scheduled, usually too late for input processing

USB audio input

- All IOKit drivers use workloops to process interrupts
- The frame buffer design works fine for output
- Isoch DMA is not very time dependant, it reads data as necessary. Data need not be present at DMA program time
- Input is a different problem
  - With output the host can sprinkles the occasional larger DMA request on contiguous data as needed
  - For input however the DMA must provide 45 sample buffers for every millisecond, even though 90% of buffers are 44 samples
  - Input must be copied into a contiguous buffer, combined with conversion to floating point

Input buffer size

- These holes are inconvenient
- Consider the public address use case, audio data needs to be copied from input to output every few milliseconds
- But the input DMA's don't complete until the end of the 125ms DMA chain
  - At which point the convert from input samples to floating is performed
- Even though the data is already on the system it needs to be copied into the input floating point buffer to be useful
- But this copy only occurs infrequently and is easily held off by higher priority threads

USB solution

- The USB family was enhanced with some high-performance isoch APIs
- The Audio driver can label an input stream as 'high-performance'
- The USB bus driver adds interrupts after every 1ms I/O frame
- But limits itself to marking completed USB frames nothing further
- On the client thread the audio driver can walk its input DMA chain and copy and convert any available data

USB critique

- Great
  - Converting input data to floating point is done lazily
    - Either at 8Hz by driver workloop
    - ... or by high-priority client as needed
- Terrible
  - 1000 interrupts per second!
    - The primary interrupt is used for more than scheduling a client thread
    - All other interrupts are disabled while the USB controller does its thing
  - A lot of code needed to be written to support new API

Firewire

- Like USB that is input data layout is determined by the data source
- The DMA queues need to be processed and completed
- We needed a different solution, the USB solution was less than ideal
- The sneaky solution: Call Firewire driver interrupt handler directly while on the client thread
  - All drivers must be able to cope with ghost interrupts
  - Most kernel threads are preemptible
  - OS X implements priority hand-off on mutexes
  - But all completion routines are run, including the non-audio requests

Firewire implementation

- The change was easy for the Firewire family & IOKit
- IOKit:
  - modify IOWorkLoop to publish the interrupt handler loop as a function
- Firewire
  - An API that allows a client to call this function
- The firewire workloop can be in three states
  - Not running: new API simulates a ghost interrupt
  - Running and pre-empted: At workloop mutex promote thread to caller priority till mutex dropped
  - Running: Do not allow running thread to be halted till it drops the mutex

Firewire driver

- We found that no completions ran long, as a result not required to preferentially handle isoch completions
  - Most of the other firewire clients do little work, they usually just wake up another thread
    - Network packets wake NetISR
    - Disk packets wake pager or client disk IO thread
- The change to the Firewire bus driver was small
- We didn't have to change the workloop threads priority!
- Firewire bus driver cleanly transitions to high-priority when a client requires it
- This few lines of change makes all data that has arrived available to the audio client

Firewire Audio changes

- Small mod to IOWorkLoop to export workloop function
- Small mod to IOFirewireDevice to re-export it
- On entry to input fetch function call interrupt handler loop
- On function completion, the audio driver knows exactly what data is available and can copy and convert it immediately
- Done

Clean solution

- High-priority threads are a scarce and expensive resource and should only be used when absolutely necessary
- Did you notice that the firewire workloop can continue running at its normal priority?
- Only when low-latency audio input is required does it ever run at a high priority
- Sad kernel person alert, I think this solution is really cool

Summary

- The basic audio mixbuffer design using 8Hz input processing is hopeless for input to output echoing
- Both USB and Firewire audio protocols require input copying and conversion to floating point contiguous data
- USB's Primary interrupt solution required extensive, dangerous and expensive development in bus driver
- Firewire's Priority handoff solution, required a 2 line function change to the bus driver and a slightly bigger change to IOKit
- Sometimes the sneaky solution is worth looking for!

Problems with Audio

- Too much code needs to run in kernel. How much is too much?
  - Any at all! Stay out of ***my*** kernel!
- Playing system sounds requires a client to spin up the entire audio infrastructure
- The audio thread's responsibility to give up some time for lower priority threads is unreliable and won't scale
- Have you seen a solution to audio design we missed?
- We'll look at audio v2 in the next hour.

Audio v2

- How would you address these problems?
- I'd like each team of you to discuss them and when we restart we will design a more elegant solution

What must be in kernel

- Any resource that publishes via standard BSD APIs, such as the file system or network stacks
- Any driver that requires primary interrupts
- Any driver that talks directly to hardware registers
- Any resource that can be shared by many different processes
- The last step in performance enhancement, that is when too many context switches on data path

coreaudiod

- Requires client process to register double buffers with daemon
- Daemon runs in time-constrained band, probably at slightly higher priority for the floating point to sample clip
- Mix each client buffer when it must be valid, determined by 'play/record' head position and each buffer's deadline
- Daemon can play 'simple' audio streams
- In principle the daemon is nearly as efficient as a kernel driver
- Publish a userland plugin model for non-standard mix-clip and sample conversion

Kernel support

- Any driver that could not be bothered to rewrite for new driver model
  - Over time Apple will port drivers to new model as hardware is released
- The output sample-buffer must always be mapped into kernel address space in case of a panic
  - Playing last 3/8ths of a second in a loop is unpleasant to say the least
- Interrupt time-stamping probably needs to remain in kernel
  - Driver could, in principle, wake a daemon interrupt thread

Further issues

- We have pulled a lot of complicated code out of the kernel
- Some drivers are more difficult to develop as they need to provide two different modules
  - mix-clip/conversion plugin to coreaudiod
  - driver that maps buffers and gets interrupts
    - new drivers are easier; only needs to support a single client, coreaudiod.
- Still has a problem with developer responsibility to give up some CPU, i.e. transitory priority inversions
- Very slightly higher CPU expenditure for one more address space transition