Real-time audio programming 101: time waits for nothing

> "The audio processing thread is stalling because the client's implementation of some XAudio2 callback is doing things that can block the thread, such as accessing the disk, synchronizing with other threads, or calling other functions that may block. Such tasks should be performed by a lower-priority background thread that the callback can signal."
>
> – Microsoft XAudio2 documentation

> "Your IOProc is making blocking calls to the HAL and it is calling NSLog which allocates memory and blocks in fun and unexpected ways. You absolutely cannot be making these calls from inside your IOProc. You also cannot be making calls to any ObjC or CF objects from inside your IOProc. Doing any of these will eventually cause glitching."
>
> – Jeff Moore, Apple Computer, on the CoreAudio mailing list

> "The code in the supplied function must be suitable for real-time execution. That means that it cannot call functions that might block for a long time. This includes malloc, free, printf, pthread_mutex_lock, sleep, wait, poll, select, pthread_join, pthread_cond_wait, etc, etc."
>
> – JACK audio API documentation for jack_set_process_callback()

As you may have gathered from the quotes above, writing real-time audio software for general purpose operating systems requires adherence to principles that may not be obvious if you're used to writing "normal" non real-time code. Some of these principles apply to all real-time programming, while others are specific to getting stable real-time audio behavior on systems that are not specifically designed or configured for real-time operation (i.e. most general purpose operating systems and kernels).

These principles are not platform-specific. The ideas in this post apply equally to real-time audio programming on Windows, Mac OS X, iOS, and Linux using any number of APIs including JACK, ASIO, ALSA, CoreAudio AUHAL, RemoteIO, WASAPI, or portable APIs such as SDL, PortAudio and RTAudio.

I'll get on to the programming specifics in a moment, but first let's review a couple of basic facts: (1) You do not want your software's audio to glitch, and (2) real-time waits for nothing.

You do not want your software's audio to glitch

You don't want your software's audio to glitch. Especially if your software is going to be used to perform to a stadium full of fans, or to watch a movie in a home theater, or to listen to a symphony in your car while you're driving down the freeway, or anywhere really. It's so fundamental I'll say it again: you do not want your audio to glitch. Period.

Search

Tags

android async asynchronous programming audio-programming code computer-music conferences dsp gigs latency lock-free ma math melbourne message passing music ndk Necessitas network-music Qt real-time supercollider travel user interface

Archives

January 2015
July 2014
August 2013
February 2013
January 2013
December 2012
July 2012
December 2011
July 2011
June 2011
April 2011
March 2011
February 2011

Sitemap

Subscribe!

Recent Tweets

@MovingFurniture @audiomulch Hi Sietse, news: not right now, updates: there will be some, yes. Progress is being made.
4 months ago

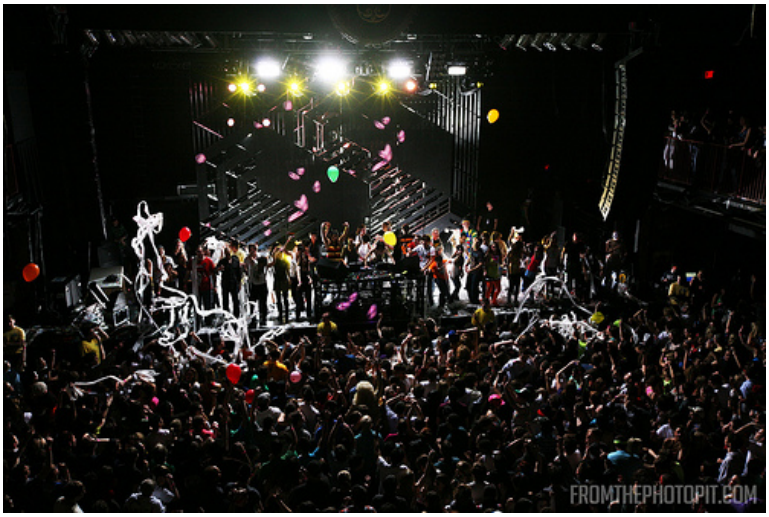
@nitsanw Err... "A new approach for the design and fabrication of an acoustic instrument is presented..." https://t.co/iRtGlBrH7G
5 months ago

@STChelvam Sorry, fairly swamped here. Have replied to your emails now.
6 months ago

@EvanBalster how about this: https://t.co/wMGfcqWEWJ
7 months ago

@mister_borogove Hey Russell, no it didn't. There's still a chance it could happen once all this darn yak shaving is done with.
10 months ago

RT @annaxambo: How do you create and perform live audiovisual compositions? Share your thoughts with our online survey:… https://t.co/Hqio8STEWX
11 months ago



That's Girl Talk performing using a laptop running my AudioMulch software up there (Photo by fromthephotopit on Flickr). I don't want AudioMulch's audio to glitch. Girl Talk doesn't want his audio to glitch either.

Real-time waits for nothing

Digital audio works by playing a constant stream of audio samples (numbers) to the digital to analog converter (DAC) of your sound card or audio interface. The samples are played out at a constant rate known as the sampling rate. For a CD player the sampling rate is 44100Hz, that's 44100 stereo sample frames every second. Every second at the same rate. Not faster, not slower. 44100 sample frames every second. If your sound card doesn't have the next sample when the DAC needs it, your audio will glitch.

On general purpose operating systems such as iOS, Android, Windows, Mac OS X or Linux your software usually won't be delivering individual samples to the DAC, it will be providing buffers of samples to the driver or an intermediate OS layer. For example it might process buffers of 256 samples at a rate of 179.26Hz (that's 44100 / 256). The lower levels of the system then feed the individual samples from each buffer to the DAC at 44100Hz.

It isn't always the case, but for the purpose of this post I'll assume that the audio API periodically requests a buffer of samples by calling a callback function in your code. In the above example your callback would have to compute each and every buffer in less than 5.8 milliseconds. No matter how your code is invoked your software has to provide those samples within 5.8ms. Each and every buffer. No exceptions. Real-time does not wait for latecomers.

Sidebar: real-world buffer sizes and latency values

To put things into context, just to make sure we're all on the same page here: To many users today 5ms is considered a large buffer size. ~1ms buffers (say 64 samples at 44100Hz) would be considered pretty good but not necessarily ideal. Applications where low latency is especially important are (1) interactive audio systems (such as musical instruments or DJ tools) where the UI needs to be responsive to the performer, and (2) real-time audio effects, where the system needs to process analog input (say from a guitar) and output the processed signal without noticeable delay. What is considered "noticeable delay" depends on a number of factors, but an end-to-end system response of less than 8ms including all system latencies (not just audio latency) is not an unreasonable ballpark number for interactive systems. For live audio effects processing many users would prefer latency to be much lower than this.

For the purposes of this post I consider buffer sizes somewhere in the 1-5ms range to be normal and achievable today on most computers running Windows with WDM, WASAPI or ASIO drivers, with Mac OS X's native CoreAudio system and with ALSA or JACK on Linux. Some audio hardware now supports buffer sizes down to 16 samples (about 1/3 of a millisecond at 44.1kHz sampling rate) or even lower. I assume that you want to write low-latency audio software for one or more of these platforms. Even if you're targeting a platform that only supports high-latency, like Android's ~50ms or Windows' legacy

Making and performing electronic music? Check out my AudioMulch software.

WMME API's ~30ms the same principles apply. You don't want to glitch.

Sources of glitches

We've established that you don't want to glitch, and your buffer period is around 5ms or less. Your code has to deliver each and every buffer of audio in a time shorter than one buffer period.

All sources of audio glitches within your code boil down to doing something that takes longer than the buffer period. This could be due to "inefficient code" that's too slow to run in real-time. That's not the main thing we're interested in here though. I assume that your code is efficient enough to run in real-time, or that you can profile and optimise it so it's fast enough. If not, the internet is full of resources to help you write faster code.

The main problems I'm concerned with here are with code that runs with unpredictable or un-bounded execution time. That is, you're unable to predict in advance how long a function or algorithm will take to complete. Perhaps this is because the algorithm you chose isn't appropriate, or perhaps it's because you don't understand the temporal behavior of the code you're calling. Whatever the cause, the result is the same: sooner or later your code will take longer than the buffer period and your audio will glitch.

Therefore, we can state the cardinal rule of real-time audio programming simply as follows:

If you don't know how long it will take, don't do it.

There are a number of things your program could do that fall into the general category of "unbounded time operations". Many are mentioned in the quotes at the start of this post. I explore them in more detail below. As you might guess, some are more obvious than others.

"Blocking"

You may hear someone say "don't do anything that blocks the audio callback thread." Used like this it's a general term. Doing anything that makes your audio code wait for something else in the system would be blocking. This could be acquiring a mutex, waiting for a resource such as a semaphore, waiting for some other thread or process to do something, waiting for data to be read from disk, waiting for a network socket. It's pretty clear that some of this waiting could take an indeterminate, or even indefinite amount of time, and certainly longer than a few milliseconds. I discuss some of these specific types of blocking in more detail below.

Keep in mind that not only do you want to avoid directly writing code that blocks, it is critical that you avoid calling 3rd-party or operating system code that could block internally.

Poor worst-case complexity algorithms

You've written every line of code in your audio callback yourself to avoid blocking. No calls to any system or 3rd-party code that could block. Even so, you might still have a problem: software efficiency is often analysed in terms of average-case (amortized) complexity. For example, in many applications, an algorithm that runs super-fast 99.9% of the time but every now and then takes 1000 times as long might still be considered "the fastest algorithm available." This is often referred to as "optimising for throughput instead of latency." But remember: *real-time waits for nothing*, certainly not for that 1000-times-normal processing spike that happens every now and then. If you do something like that in your audio callback, you may get a glitch. For this reason, you should always consider the worst-case execution time of your code.

A simple example would be zeroing a delay line to reset it (say using a for-loop to zero every element, or perhaps by calling memset). You might think that using memset to clear a buffer of memory is fast, but if the delay line is large the memset call is going to take quite a long time — and if the time taken is longer than you have available you'll glitch. It's usually better to use an algorithm that spreads the load across many samples/callbacks, even if it ends up burning a few more cycles overall. Of course if these spikes in processor usage are small, and you know that they'll be bounded (e.g. you know the maximum size of the delay line you'll need to zero), you might be OK. But if your code is full of these little occasional spikes,

you'd better hope you can predict how they all interact and sum together. If they all occur at the same time you could still get a glitch, and you don't want that.

Another thing to keep in mind here is that many operating system and library functions are implemented using average-case optimised algorithms. In C++ many STL container methods fall in to this category (more on that below). General purpose memory allocation algorithms and garbage collectors also have unpredictable time behaviour, even if they don't use locks.

Locking

It's difficult avoid concurrency when you have a GUI or text interface, network or disk i/o and a real-time hardware-driven audio callback. Assuming that your GUI is somehow controlling the audio process you'll need to communicate between the GUI thread and the audio callback. Perhaps you'll want to display graphics that reflect the state of audio processing (level meters for example). Similar requirements may arise if you need to control audio via a network socket or MIDI.

The first thing you might think of is "I need a lock or mutex" to protect concurrent access to data shared between the GUI and the audio thread. This is a common response. I remember having it too. Actually when I first went looking for a lock, there wasn't one available. On Mac OS 7, with the SndPlayDoubleBuffer API there was no OS mechanism to implement a lock since the callback could occur at interrupt time. Of course modern operating systems do provide locks (mutexes) to protect against concurrent access to shared state. You should not use them within an audio callback though. Here are three reasons why:

#1 reason to not use locks: priority inversion

Let's say your GUI thread is holding a shared lock when the audio callback runs. In order for your audio callback to return the buffer on time it first needs to wait for your GUI thread to release the lock. Your GUI thread will be running with a much lower priority than the audio thread, so it could be interrupted by pretty much any other process on the system, and the callback will have to first wait for this other process, and then the GUI thread to finish and release the lock before the audio callback can finish computing the buffer — even though the audio thread may have the highest priority on the system. This is called priority inversion.

Real-time operating systems implement special mechanisms to avoid priority inversion. For example by temporarily elevating the priority of the lock holder to the priority of the highest thread waiting for the lock. On Linux this is available by using a patched kernel with the the RT preempt patch. But if you want your code to be portable to all general purpose operating systems, then you can't rely on real-time OS features like priority inheritance protocols. (Update: On Linux, user-space priority inheritance mutexes (PTHREAD_PRIO_INHERIT) have been available since kernel version 2.6.18 together with Glibc 2.5. Released September 19, 2006. Used in Debian 4.0 etch, 8 April 2007. Thanks to Helge for pointing this out in the comments.)

Keep in mind that any code in your audio callback that waits for something to happen in another thread could be subject to priority inversion. Rolling your own "spin lock" that busy-waits polling for something to complete in another thread, in addition to being inefficient, will have the same priority inversion problem if the thread you're waiting for has lower priority and gets preempted by another thread in the system.

#2 reason to not use locks: risk of accidentally calling code with unbounded execution time

I know you're not going to use a lock, because I just explained that priority inversion will mess with you even if your code simply locks a mutex to set one flag and then releases the lock:

```
mutex.lock();
flag = true; // subject to priority inversion
mutex.unlock();
```

But if you're still contemplating calling code that acquires a lock, consider this: the audio callback will have to wait for all of the code that is protected by the lock (the "critical section") to complete before the

audio callback can proceed. Effectively, in addition to the thread context switching costs, you're executing all that code sequentially with your audio callback. Do you know how long that will take? in all cases? Remember we're talking about worst-case time here, not average-case. Any code path inside a critical section that's shared with the real-time audio thread would have to follow all of the rules we're outlining here. That's asking for a lot of discipline from you and your fellow developers. It would be easy for bugs to creep in. In C++ you wouldn't want to do this for example:

```
mutex.lock();
my_data_vector.push_back( 10 ); // could allocate memory and copy mucho data
mutex.unlock();
```

If my_data_vector is a std::vector, calling push_back() when the vector's internal storage is full will cause memory to be allocated and all existing elements to be copied into the new memory. That's obviously going to cause a processing time spike. Most non-real-time code behaves like this some of the time. It's easy for simple-looking code to have non-deterministic time behaviour. You don't want it creeping into your critical sections.

#3 reason to not use locks: justified scheduler paranoia

> "you're definitely opening things up to a world of hurt when the system gets stressed"
>
> – Jeff Moore to James McCartney on the CoreAudio list in 2001

Priority inversion and unbounded execution time inside critical sections aren't the only reasons to avoid locks and other concurrency primitives. In the case of proprietary operating systems, few people know exactly how the schedulers are implemented. No matter what the OS, scheduler implementations may change with each OS release. These general purpose operating system schedulers are not required or guaranteed to exhibit real-time behavior. They're not certified for use in avionics systems or medical equipment. There are no governments or judiciaries to hold their real-timeness to account.

My general position on this is that you should avoid any kind of interaction with the OS thread scheduler. Avoid calling any synchronisation functions in your audio callback. Schedulers employ complex and diverse algorithms and you don't want to give them extra reasons to de-schedule your real-time audio thread.

Of course you can't escape depending on the scheduler to invoke your audio callback on time, and with high priority. You might also consider a few other hard-core scheduler interactions, such as signalling condition variables or semaphores to wake other threads. Some low-level audio libraries such as JACK or CoreAudio use these techniques internally, but you need to be sure you know what you're doing, that you understand your thread priorities and the exact scheduler behavior on each target operating system (and OS kernel version). Don't extrapolate or make assumptions. For example, last I checked the pthreads condition variable implementation on Windows employed a mutex internally to give correct POSIX semantics — that's definitely not something you want to get involved with (you could perhaps use the native Windows SetEvent API though).

Side note: trylocks

One related option that may be open to you is to use a non-blocking "try-lock" function that simply returns a failure code instead of blocking if the lock can't be acquired (pthread_mutex_trylock() on POSIX, TryEnterCriticalSection() on Windows). The downside is that since acquisition of the lock is not guaranteed, you can't use it to protect anything that you must access at every callback. You're gambling that you'll be able to acquire the lock at least some of the time — although depending on the behavior of the other lockers, in the worst case, your code may never manage to acquire the lock.

Memory allocation

I've touched on this point above, but it's worth re-iterating: you should not allocate memory in your audio callback. For example you shouldn't call malloc() or free() or C++'s new or delete, or any operating-system specific memory allocator functions, or any routine that might call these functions. The three

obvious reasons are:

- The memory allocator may use a lock to protect some data it shares between threads. Aside from priority inversion, trying to lock a mutex that's potentially contended by every other thread that allocates memory is clearly not a good idea.
- The memory allocator may have to ask the OS for more memory. The OS may also have it's own locks, or worse, it may decide to page some memory to/from disk and make you wait while it happens.
- The memory allocator may use algorithms that take unpredictable amounts of time to decide how to allocate a block — and you know you don't want that.

Some obvious solutions are:

- Pre-allocate all your data.
- Only perform dynamic allocation in a non-real-time thread where it isn't time-critical.
- Pre-allocate a big chunk of memory and implement your own deterministic dynamic allocator that's only invoked from the audio callback (and hence doesn't need locks).

For example, SuperCollider has an implementation of Doug Lea's memory allocator algorithm that is only used in the audio callback. The current version of AudioMulch uses per-thread memory pools for dynamic allocation. For tasks with unbounded execution time such as plugin loading, AudioMulch performs them in the UI thread and then sends the results to the audio callback when they're ready — sometimes it's OK to make the user wait, so long as the audio callback doesn't have to.

Invisible things: garbage collection, page faults

Even if you avoid all of the above in your own code, there are still a few environmental issues. If you're using a garbage collected language such as Java or C# and the GC kicks in while your audio thread is running you will probably be in trouble unless you can disable GC or use an environment with deterministic real-time GC. This isn't my area of expertise so I won't say more about this other than that there are Java implementations with real-time GC. Otherwise you may have to settle for higher latency and additional intermediate audio buffering.

Even in a non-garbage-collected language such as C, the OS virtual memory system could page out memory that you reference in your audio callback. This would cause the thread to stall waiting for memory to be paged in from disk. I've never had problems with this myself — usually if the memory pages are kept hot by accessing them regularly the OS will keep them in physical RAM. But if your program is pushing the limits of available RAM, uses data that's only referenced infrequently, or expect other memory-intensive tasks to be going on in parallel to your audio processing, you may want to investigate locking your real-time data into RAM (using operating system specific mechanisms such as mlock()/munlock() on OS X and Linux, VirtualLock()/VirtualUnlock() on Windows).

Waiting for hardware or other "external" events

You're probably not writing code that directly waits for hardware. But disk i/o often has to wait for the hard disk head to seek to the right position, and that can take a while (averages ~8ms for consumer disks). This means performing file i/o from an audio callback is a no-go, even ignoring the fact that file i/o may lock a mutex while accessing file system data structures, allocate memory, or otherwise use poor worst-case performing algorithms. Similar rules apply to other tasks like syncing with the vertical interrupt on the graphics card or performing network i/o. As I said earlier: If you don't know how long it will take, don't do it.

Combinations of all of the above: code that doesn't make real-time guarantees

If you use a closed-source operating system you probably don't know what every API call does under the hood, but it's a safe bet that most general purpose computing code is not written with all of the above real-time considerations in mind. Even if you do have access to your operating system source code, unless the system guarantees real-time behavior, it's hard to say for certain that code that doesn't use a

lock today won't use one in the future, or that an implementation won't change to one with better average-case performance but worse worst-case performance. Memory paging issues are almost unavoidable since you can't lock every OS-managed page in RAM without disabling virtual memory entirely.

For these reasons I suggest that you assume that *all* system API functions could allocate memory, use locks or employ algorithms with poor worst-case time behavior. Some may perform disk i/o either directly, or indirectly by triggering page faults when there is a pressure on the memory subsystem. Many functions will allocate memory — even just temporarily as scratch-space.

Paranoia is justifiable, since you really don't want to glitch. I generally avoid OS API functions with very few exceptions and only when absolutely necessary. The only exceptions I can think of off the top of my head are QueryPerformanceCounter() on Windows, and mach_absolute_time() on OS X. Unfortunately I'm not 100% confident of the real-time behaviour of either.

Similar paranoia should be applied to 3rd-party libraries such as those for soundfile i/o. Unless code claims to have non-blocking real-time behavior, don't assume that it does.

On OS X, Apple advises: don't call the BSD layer from your IOProc. Objective C code (since the objective-C dispatcher may use locks) and CoreFrameworks are also out. Microsoft don't typically document which, if any, of their APIs have real-time semantics.

Lands of confusion

The ideas in this post are not secret knowledge although they are in-part folklore. They come up periodically on all the audio development mailing lists I've ever subscribed to and I know they come up elsewhere. The thing is, avoiding locks goes against best-practice advice in "normal" concurrent programming. As a result, sometimes things get confused. Here are two examples of bad advice I've found online that illustrate the confusion:

***** DO NOT TAKE THE FOLLOWING QUOTED ADVICE *****

> "you should use a mutex or other synchronization mechanism to control access to any variables shared between the application and the callback handler"
>
> – ~~Open SL ES audio API documentation for Android~~

Well, it is true that you should use a synchronisation mechanism, just not a mutex. My thoughts on the above quote are on the public record here.

> "Both read and write locks the buffer so it the pointers of the buffer will be maintained consistent"
>
> – ~~JACK Wiki description of using ringbuffers~~

(The JACK guys do know better than this, I've reported the error, it's probably fixed by now).

If these snippets were referring to normal multi-threaded code then they would be absolutely correct. Except in extremely rare circumstances, you should not access shared data without protecting it with a mutex lock. However, for the reasons I've explained above, in real-time audio code, on general-purpose operating systems, using a mutex is not advisable. It is poor practice and widely frowned upon. The solutions that I advocate involve certain lock-free and atomic techniques that I mention below and that I hope to describe in more detail in future posts. [Update: but see the comments for other views.]

In summary

Boiling down the above discussion into a few rules of thumb for code that executes in a real-time audio callback:

- Don't allocate or deallocate memory

- Don't lock a mutex
- Don't read or write to the filesystem or otherwise perform i/o. (In case there's any doubt, this includes things like calling printf or NSLog, or GUI APIs.)
- Don't call OS functions that may block waiting for something
- Don't execute any code that has unpredictable or poor worst-case timing behavior
- Don't call *any* code that does or may do any of the above
- Don't call any code that you don't trust to follow these rules
- On Apple operating systems follow Apple's guidelines

There are a few things you should do where possible:

- Do use algorithms with good worst-case time complexity (ideally O(1) wost-case)
- Do amortize computation across many audio samples to smooth out CPU usage rather than using "bursty" algorithms that occasionally have long processing times
- Do pre-allocate or pre-compute data in a non-time-critical thread
- Do employ non-shared, audio-callback-only data structures so you don't need to think about sharing, concurrency and locks

Just remember: time waits for nothing and you don't want to glitch.

Coda

But wait you say, how am I supposed to communicate between a GUI and the audio callback if I can't do these things? How can I do file or network i/o? There are a few options, which I will explain in more detail in future posts, but for now I'll just say that the best practice is to use lock-free FIFO queues to communicate commands and data between real-time and non-real-time contexts (see my article about SuperCollider server internals for some ideas, PortAudio has an implementation of a lock-free queue you could use as a basis for your own queuing infrastructure). Other options include other types of non-blocking lock-free data structures, atomic data access (requires great care), or trylocks as I mentioned above. For another resource, these techniques are touched upon in the notes for a workshop that Roger Dannenberg and I ran in 2005 on real-time design patterns for computer music.

I'm not exactly sure where I picked up all these ideas. At a minimum I need to thank Phil Burk, Roger Dannenberg, Paul Davis and James McCartney for sharing their insights on various mailing lists over the years. The quotes above reveal that Jeff Moore has also been banging on about these issues on the CoreAudio mailing list for at least a decade.

I started writing this post a few weeks ago, but just yesterday I read Tim Blechmann's Masters thesis about his Supernova multi-core audio engine. It rehearses many of the ideas discussed here, and from there I learnt that the Linux RT Preempt patch implements priority inheritance as I mentioned above. Tim's thesis is definitely worth a read for another angle on this material, and a lot of ideas on multi-core audio too.

Finally, if you made it to here, please, if I got something wrong, you think I've missed something, or you know of other activities to avoid in an audio callback please post in the comments. Thanks for reading.

Lightly updated for clarity 13 August, 2011.

Posted on July 5, 2011 by rossb. This entry was posted in Code and tagged audio-programming, code, computer-music, lock-free, real-time. Bookmark the permalink.

« Reflections on Bret Victor's "Explorable Explanations" Dave Sparks on Android audio latency at Google I/O 2011 »

29 Comments

Dave Phillips
July 22, 2011 - 7:48 pm | Permalink 1

A good informative read. Thanks, Ross !

Reply

Harry van Haaren
July 22, 2011 - 9:33 pm | Permalink 2

Hey,

Thanks a lot for this article, its a great resource to refer to when pondering if something is acceptable or not…

Really looking forward to future articles on how to best provide lock-free inter-thread communication!
-Harry

Reply

Daniel
July 23, 2011 - 7:04 am | Permalink 3

Of course, atomic operations on x86 use the BUS LOCK# feature; which is an indeterminate* mutex at the hardware level….

* but somewhat bounded, time wise.

Reply

Paolo
July 23, 2011 - 5:39 pm | Permalink 4

Have you also tried a low-latency disk scheduler as BFQ (algo.ing.unimo.it/people/paolo/disk_sched/) to reduce the delay induced by other processes accessing the disk?

Reply

Glenn
July 24, 2011 - 3:24 am | Permalink 5

In your experience, how often does your audio application have to compete for CPU time with other regular system processes? You can attempt to do as much “scheduling” within your own thread (or perhaps a thread pool within the same address space), but sometimes the system (as a whole) as other things to do than just schedule your own application. Is the solution to simply to use multicore + CPU affinity + interrupt masking + POSIX SCHED_FIFO? Or, would you want a greater level of support from the OS? Also, how important is portability?

Also, if you’ve only just heard about PreemptRT, check out the SCHED_DEADLINE Linux patch. It has some interesting real-time properties such as bound deadline tardiness. This can allow for a high degree of confidence in adequate buffer sizes. Deadline scheduling can also allow greater levels of system utilization while guaranteeing deadlines are met, though static-priority scheduling (SCHED_FIFO) may be enough if your real-time threads have harmonic periods (which seems quite likely in an audio application).

By the way, Mac OS X already supports deadline scheduling as well as some other nifty non-POSIX options (see “THREAD_TIME_CONSTRAINT_POLICY” here:
developer.apple.com/library/mac/#documentation/Darwin/Conceptual/KernelProgramming/scheduler/scheduler.html).

Reply

rossb
July 24, 2011 - 2:16 pm | Permalink 6

Paolo:
Thanks for the tip about BFQ, it looks interesting. I agree that prioritising disk access is a good thing. I mainly develop for Windows and Mac where BFQ isn’t an option. I do use an in-process disk scheduler in AudioMulch to prioritise disk access within the app (e.g. reading samples from disk get lower priority than playing time-critical disk streams).

Glenn:
Usually audio processing threads are scheduled with high priority, so competion with regular system processes is low. Often the audio i/o API or subsystem is responsible for managing scheduling of audio buffer processing threads and callbacks. Apple’s CoreAudio uses THREAD_TIME_CONSTRAINT_POLICY threads for this purpose. Windows has time critical threads and MMCS. On Linux there are RT options as you note (thanks for

your input on this).

I usually assume that it's up to the OS or Audio driver to guarantee the primary audio callback is scheduled reliably and that the client's (caller's) job is to return audio by the required deadline. In that sense this post is about the audio application programmer doing the right thing.

Regarding portability: I aim to write portable code that will work reliably on consumer-grade operating systems. Irrespective of platform, the guidelines proposed in this post are required by many audio APIs for reliable operation. At the lowest level there may be non-portable but potentially reliable things you can do that go against what I've posted. I like to leave that kind of thing to the audio APIs an OS as much as possible. I admit it is not always possible to avoid low-level, non-portable code. For example non-portable code would probably be required to implement a multi-core real-time audio engine — that's definitely not a "101″ topic though, and in any case, each thread would still need to avoid priority inversion and unbounded computation.

Reply

Francesco Pretto 7

July 25, 2011 - 6:22 am | Permalink

Synchronization by means of monitor on condition variables can be good? Anyway, I actually don't agree much with the point of the article "don't use mutex because in some OS this can be bad", at least not from a religious stand. I mean: Mac OSX was and is appreciated because of innovation, can't understand why we shouldn't use new technologies and always leverage on common denominator. This reminds me discussions about un-portability of systemd, if you know what I mean. And incidentally the author of systemd is Poettering: the first to talk about glitch-free audio for a broader audience in the open source world.

Reply

Rob Fielding 8

October 11, 2014 - 1:08 am | Permalink

It's not "lowest common denominator". There is a function call available, and it has a signature. You pass in some pointers and get back a pointer, but the documentation says nothing about timing guarantees. So in some cases, you must supply an implementation that has such timing guarantees. If you implement enough of it, then you have a real-time framework (or as real-time as it can be while hosted in the OS).

On iOS phone/iPad, my stuff ran great until recent OS upgrades. Over time the pressure to show better "speed" numbers results in code getting optimized for throughput rather than jitter/latency. Ironically this means that the same software performs worse as "progress" is made…as the hardware actually gets faster. The people at Apple kind of understand real-time, the guys at Blackberry totally understand it (it's actually the best platform to port music code to because of QNX — except their business is all kinds of messed up), some people at MS understand it – but seem to be fighting against a majority that does not. And Android is something special… in some cases they have audio buffer sizes in the range of kilobytes; they are willfully ignoring this problem. Linux started off as a server OS and is highly throughput oriented.

At some point though, all mobile operating systems will need to be hard-real time and power-efficient systems. They are centered around sensors in a way that desktop systems are not, and have an urgency for real-time that servers and desktop systems don't have.

Reply

rossb 9

July 25, 2011 - 11:58 am | Permalink

Francesco:
All or almost all systems today have these limitations because desktop operating systems are not hard-real-time safe. Unless you know a monitor or condition variable is non-blocking why risk it? And even if it is non-blocking on one architecture and OS release, do you know it always will? These things are not usually specified unfortunately. Personally I think it would be interesting to have a real-time OS and/or real-time safe synchronisation primitives available for audio programming. But across all platforms we don't.

Yes I take a lowest common denominator position — because I want my software to work everywhere, not just on some specific tuned system. This post is not just about Open Source or Linux — that's just one small part of the user base and developer community for audio software.

I'm all for the use of innovative technologies if they solve the problems above. One personal interest is wait-free lock-free algorithms. I think they'd fit your interest in innovation and new technologies.

Reply

Glenn 10

July 25, 2011 - 4:16 pm | Permalink

Don't make the mistake of calling PREEMPT_RT a hard RTOS. Very few OSs can make this claim. Observe that all the OSs certified for avionics (ARINIC-653) are still (I am fairly certain) unicore (ex. VxWorks, LynxOS, QNX). These OS companies do have multicore variants, but they are not certified for avionics. What's the challenge? Shared caches are one. The processing on one core can evict cached data in use on the other, affecting bounds on worst-case execution time. You might have to pretend that you have a no-cache architecture when you

analytically prove you have built a hard RTOS. In doing so, you might have a higher performance, with respect to your analytical model, unicore system.

One limitation PREEMPT_RT currently has is that though interrupts are threaded, even threaded interrupts can still cause priority inversions. To avoid an inversion, the thread handling the interrupt should be scheduled with the same priority (i.e. priority inheritance) as the user/kernel thread that is dependent upon the interrupt processing. However, PREEMPT_RT does not do this (LynxOS does). I believe such features require deep integration into the device drivers, OS, and user application. For example, the device driver needs to identify the user-application which requires the interrupt handling so it can instruct the OS to modify the device driver thread's priority. This may be easy to do when a device is used by one application. This is harder when the device is shared (like a disk!). I doubt such complete integration will ever be feasible in Linux. Starvation of interrupt processing is a danger when if interrupt-handling threads cannot inherit priority. You can artificially boost their priority, but that defeats the purpose of threading them in the first place.

Reply

Helge
July 25, 2011 - 5:46 pm | Permalink 11

Your advise "use lock-free data structures instead of locking" is really complete and utter nonsense.

1. What you want are "wait-free data structures" (though I would contend that "obstruction-free data structures" would suffice unless you are aiming for a really perverted system design). How you achieve "wait-freedom" is immaterial.

2. Wait-freedom does not preclude locking — in fact, the widely available "priority inheritance" protocol can be regarded as a realization of the "helping" principle for construction of wait-free shared data structures (cf. for example Hohmuth, os.inf.tu-dresden.de/papers_ps/hohmuth-phd.pdf). Non-portability is a red herring, this is quite ubiquitous by now but if your platform of choice does not provide it, then you made a dumb platform decision. Tough luck.

3. Lock-freedom does not imply wait-freedom. Actually, almost all simple "lock-free" algorithms for data structures more complex than a bounded FIFO will contain some sort of "load – prepare – try-commit-with-CAS" loop, and are therefore by definition not wait-free. The non-simple ones that can operate with a bounded number of steps are a) very very very hard to implement and b) offer so piss-poor throughput over locked versions that you do not want to use them (cf. Attiya et al. citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.4076&rep=rep1&type=pdf). Means: locking with prio inheritance provides *stronger* real-time guarantees *and* better performance.

4. Even if you have a wait-free (lock-free) implementation of a data structure, implying bounded number of steps, your hardware will have a hard time guaranteeing that each of the steps takes only a bounded number of times: the worst case execution time scales with the number of CPUs contending for the same cache-line. Therefore you can actually only guarantee *anything* if you can make sure that the number of threads accessing the shared data is bounded (but in that case prio inheritance also provides the same guarantees).

Reply

rossb
July 25, 2011 - 6:51 pm | Permalink 12

Helge:
This is not a theoretical post. It is about practicalities on real-world desktop operating systems (Windows, Mac OSX, Linux). I am using "lock free" as a generic term. I understand the difference between non-blocking, lock free and wait free. If you want to argue detailed semantics I agree with your comments — I should have said "wait free" or "obstruction free." In practice it has been shown that some non-blocking algorithms perform better than truly wait-free algorithms in low-contention scenarios. If the contention behaviour is well understood I think this can be acceptable for the kinds of systems I'm discussing here.

I did mention priority inheritance in the second paragraph of "#1 reason to not use locks: priority inversion." My understanding is that this is not available on Windows. I have not found confirmation that it is available on OSX or Linux — if you have a link please share.

Your attitude "but if your platform of choice does not provide it, then you made a dumb platform decision. Tough luck." is a little strange. The reality is most musicians are using normal desktop computing platforms. I spend my time helping them make music, so do many other developers. The whole point of this post is about getting near soft-real time performance on platforms that don't have features such as priority inheritance.

Reply

bkor
July 25, 2011 - 10:22 pm | Permalink 13

Cool blogpost! It isn't useful to me (when I make something it is in Python), but really interesting to read and understand the difficulties + limitation that have to be dealth with.

Reply

rossb
July 26, 2011 - 1:36 am | Permalink 14

Regarding the applicability of POSIX priority inheritance mutexes on OSX, the following from the OSX Kernel Programming Guide page 73 may be relevant:

> The priority levels are divided into four bands… [Normal, system high priority, kernel mode only, Real-time threads]
>
> Threads can migrate between priority levels for a number of reasons, largely as an artifact of the time sharing algorithm used. *However, this migration is within a given band.*
>
> …some aspects of a thread's priority can be controlled from user space using the POSIX thread priority API. *The POSIX thread API is able to set thread priority only within the lowest priority band (0–63).*

(emphasis added)

developer.apple.com/library/mac/documentation/Darwin/Conceptual/KernelProgramming/KernelProgramming.pdf

The term "inheritance" is not used within the document in any context concerning priorities.

Reply

rossb
July 26, 2011 - 1:54 am | Permalink

15

On Linux PTHREAD_PRIO_INHERIT has been available in user-land since kernel version 2.6.18 together with Glibc 2.5. Released September 19, 2006 (Used in Debian 4.0 etch, 8 April 2007)

rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application

Reply

drw
July 29, 2011 - 3:04 pm | Permalink

16

HI Ross, you say

> Digital audio works by playing a constant stream of audio samples (numbers) to the digital to analog converter (DAC) of your sound card or audio interface. The samples are played out at a constant rate known as the sampling rate. For a CD player the sampling rate is 44100Hz, that's 44100 numbers per second. Every second at the same rate. Not faster, not slower. 44100 samples every second.

Perhaps one of your postees has mentioned this – I didn't read them all, but it is 88,200 numbers per second for a CD player. I tried playing a mono file on a CD player recently and it assumed it was interleaved stereo.

Reply

rossb
August 13, 2011 - 3:19 am | Permalink

17

Thanks for your attention to detail David. I've corrected the post accordingly.

Reply

Neil C Smith
August 2, 2011 - 4:16 am | Permalink

18

Hi Ross,

Great post! In many ways you're preaching to the converted with me, but it's great to see this written up in such a clear way in one place. In fact, I've just linked to it writing the help pages on code.google.com/p/java-audio-utils/ – hope you don't mind.

In reference to your comment about garbage collection, using one of Java's incremental garbage collectors I'm able to get JACK down to 3 periods of 128 samples with stability. There are improved concurrent garbage collectors coming too. Interestingly, you can get away with some object creation with Java due to its memory pooling, though not overboard obviously otherwise you're just creating more garbage!

One thing I've seen quite a bit in Java audio libraries is naive uses of a mutex (probably because Java makes it too easy to write synchronized) where an object is locked but not the whole audio pipeline, despite the fact that bits of the pipeline are expecting state not to change. The other good thing about a non-blocking queue would seem to be that it puts the audio thread in control of *when* events are processed without also blocking the calling thread.

Incidentally, in AudioMulch do you use the same mechanism to pass information back to the GUI too?

Best wishes, Neil

Reply

Pierre Lebeaupin
August 3, 2011 - 7:45 am | Permalink 19

Interesting. Well, I knew the real-time aspects, but not so much the best practices/folklore/how to cope with a non-RT OS when you want to do RT stuff.

There's just one thing though. Writing correct lock-free algorithms has just gotten a lot more complicated. Yup. Your proud existing lock-free queues/ringbuffers/whatevers? Unless they've prepared themselves for running on multi-core ARM, they are pretty much busted.

Audio programming is between a rock and a hard place: either use mutexes and pray they properly inherit priority, or use memory barriers and write code that no sane man or woman can determine is correct or not. Something will have to give: OSes will have to either provide a library of lock-free algorithms for use in audio contexts (a long shot), or provide mutexes which properly inherit priority (except on Mac OS X/iOS, a long shot too).

(I do not know whether Mac OS X/iOS mutexes properly inherit priority; I'll just note that never do people from Apple or Apple documentation recommend against mutexes themselves (only those where another thread could do unbounded work while holding the lock), and that Apple took a lot of care to provide some real-time-like behaviors, in particular this paper mentions that RT threads can degrade to non-RT (as he writes, that way it becomes possible to let normal users create RT threads since they will become non-RT if abused), therefore it is reasonable to think that, provided they support priority inheritance, mutexes do support priority inheritance that can raise a thread from non-RT to a priority in the RT band)

Reply

rossb
August 13, 2011 - 3:16 am | Permalink 20

Hi Pierre.

I agree, you need to know about memory barriers when implementing lock-free algorithms. Weak memory ordering was an issue on Power PC also — it's nothing new with ARM.

> I'll just note that never do people from Apple or Apple documentation recommend against mutexes themselves

That's not true. I give one example in the post, quoting Jeff Moore:: "you _really_ ought not block the IOThread on a mutex that some other thread may have"
lists.apple.com/archives/Coreaudio-api/2001/May/msg00032.html

> therefore it is reasonable to think that, provided they support priority inheritance, mutexes do support priority inheritance that can raise a thread from non-RT to a priority in the RT band

Personally I don't think it's reasonable to make such optimistic assumptions. But I would love to see proof that it's true. I did go and ask on darwin-dev. So far there has been no reply:
lists.apple.com/archives/darwin-dev/2011/Jul/msg00015.html

Reply

Pierre Lebeaupin
August 19, 2011 - 6:25 am | Permalink 21

In theory, multi-core ARM is indeed nothing new from multi-proc ppc. However, Olivier Guilyardi on the andraudio list said my blog post was the first time he heard of barrierless FIFO/ringbuffer code actually failing in practice (and this matches my experience preparing that blog post). So I was under the impression that available ringbuffer code was unprepared, but I may have judged too soon.

At least you are aware of barriers (it was unclear whether you were); I bet that many others are not, unfortunately.

Interesting, I had not read the full email. They seem to be entertaining the ambiguity everywhere else, on the other hand.

(to be clear, at least at the moment I'm not working in IOProcs, the closest I am is refilling buffers for an audio queue, operation which has not nearly as tight a deadline).

Reply

Druttis 22
September 13, 2011 - 12:23 am | Permalink

Good post Ross!

Cheers!

Reply

Albin 23
October 30, 2011 - 3:37 am | Permalink

Thank's for an excellent post!

Reply

Stefan 24
July 12, 2012 - 9:33 pm | Permalink

Thank you for the excellent info, Ross. And sorry for necroing an old post.
Everything you write rings true, but how do you deal with large files that need to be streamed, m y scenario being background music in a video game on Android smartphones.
I would like to avoid loading a whole song into RAM, since I already keep sound effects resident in it (due to latency woes).
Instead I decode songs to a cache file on disk during initialization (from vorbis to pcm), in order to avoid doing so during the audio callback later. During the callback I then fill an audio double buffer by reading from the cache file. Obviously I have to call read for that, but otherwise I try to avoid all kinds of computations inside of the callback.. Do you think leaving the audio buffer large enough (couple seconds of data, since latency is no issue for background music) would be a good enough approach? Or maybe I should decode the file during the callback, in order to keep the amount of data read small (since the difference between vorbis and pcm is quite significant). Any experiences you can share?

Reply

David Kirby 25
August 2, 2012 - 5:47 am | Permalink

Wow… You covered it all. EVEN TRYLOCKS! HAH Thanks for writing this.

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

EfwsBpeWXBRR *** Copy This Password ***

*** Type Or Paste Password Here ***

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code> <del datetime=""> <em> <i> <q cite="">`

<strike>

Post Comment