Using locks in real-time audio processing, safely

14 APRIL 2020 / TIMUR DOUMLER / 13 COMMENTS

When developing music software, you are operating under tight time constraints. The time between subsequent audio processing callbacks is typically between 1-10 ms. A common default setting is a buffer size of 128 samples at a sample rate of 44,100 Hz, which translates to 2.9 ms in between callbacks. If your process does not compute its audio output and write it into the provided buffer before this deadline, you will get an audible glitch, rendering your product worthless for professional use.

It is therefore commonly taught that you cannot do anything on the audio thread that might block the thread or otherwise take an unknown amount of time, such as allocating memory, performing any system call, or doing any I/O. This includes waiting on a mutex, which not only blocks the audio thread but also leads to priority inversion. This is well known and covered in many articles and talks, such as Ross Bencina's seminal article, my own CppCon 2015 talk, and more recently a brilliant series of talks by Dave Rowland and Fabian Renn-Giles (part 1, part 2).

So how do we synchronise the audio thread with the rest of our application?

In the simplest case, a single numerical value needs to be shared between threads. For example, the user might turn a knob, which will update a parameter in your DSP algorithm. For this, you can use a `std::atomic<T>` variable, where `T` is a built-in numeric type such as `int` or `float`; this will be lock-free on modern platforms.

If you have a stream of objects flowing from one thread to the other, such as MIDI messages, you can use a lock-free single-producer single-consumer FIFO (`boost::spsc_queue` is a good implementation).

But sometimes we find ourselves in a more complicated situation. The audio thread needs to access a bigger data structure describing some part of the system, such as a `std::vector` or a directed graph or perhaps a linked list. Typically, the audio thread will be traversing that structure to do its work, while another thread might be modifying or re-generating that structure at the same time. How do we deal with this situation?

The correct answer, in my opinion, is to design your audio engine in such a way that this case never occurs. This can be achieved using immutable data structures. Instead of modifying the data structure in-place, the message thread peels off a copy that contains the modification, while the audio thread still looks at the previous version for however long it needs to. When done correctly, this will lead to a system that is thread-safe by design, and you will never need locks on the audio thread.

Unfortunately, not everyone is in the position where they can re-engineer their audio engine to follow this pattern. You might be stuck with data structures accessed by multiple threads simultaneously. In this situation, many developers reach for locks, and that's exactly what we will be looking at in this article.

Use cases

Looking at the codebase I am currently working with, I have found various places where locks on the audio thread are used to avoid data races. These use cases are all variations on the same theme. Here are just three examples:

- The audio thread is searching a list of audio samples, trying to find the one that matches the currently played note; another thread loads more samples and inserts them into that list.
- The audio thread is searching for a free polyphonic synthesiser voice that it can use; another thread allocates more such voices.
- The audio thread is processing a whole audiograph; another thread re-builds this graph.

You might notice that the only real difference between all those use cases is how large the data structure in question is, and how big the portion of the audio graph is which depends on the current state of that data structure.

Now (putting aside for a moment the fact that all of these can be implemented without any locks at all if we use more clever data structures), let's look at the types of locks chosen. It seems that some developers intuitively choose different types of locks depending on the perceived (but often not actually measured) duration of the code under lock. If there is a lot of code, they might reach for `std::mutex` or something similar. If, on the other hand, it looks like a relatively quick operation, they might opt for a spinlock instead.

Below, we look at both of those options.

Don't use std::mutex::try_lock!

It is well-known by now that you should not lock a mutex on the audio thread. At this point, it might seem like a good idea to resort to a different strategy: using `try_lock` instead. In fact, I saw this being recommended in tutorials. If you're using C++17 and the C++ standard library, it might look like this:

```
if (std::unique_lock lock(mtx, std::try_to_lock); lock.owns_lock()) {
    // do your audio processing
}
else {
    // use a fallback strategy
}
```

In this code, the audio thread is trying to obtain the mutex. If this fails, because the mutex is being held by another thread, the audio thread doesn't wait (it's not allowed to). Instead, we fall back to an alternative strategy such that we can keep the audio running without accessing the data that is being used by the other thread. If you're lucky, the object can use the last known values of the needed parameters instead, or do something else that the user won't notice. If you're less lucky, you'll have to switch your audio effect to bypass mode, or perhaps fade out, or some similar fallback strategy. It's probably not ideal, but in many cases still better than a glitch.

At first glance, this seems like a good strategy. After all, the C++ standard says that `try_lock` doesn't block and returns immediately. So it's safe to call on the audio thread. That is true. But unfortunately, this strategy has a fatal flaw.

If we look more closely at what the above code snipped actually does, it turns out that `std::mutex::try_lock()` is not the only call that is being made by `std::unique_lock`. After all, it's an RAII class, so when it goes out of scope, it will call `std::mutex::unlock()` in its destructor. Most of the time, that's not going to do very much. But if there is another thread waiting to acquire the mutex, the audio thread will have to interact with the OS thread scheduler at that point so that that other thread can then be woken up. And that's a system call. It's not realtime-safe to do that. So you should not use `std::mutex` on the audio thread, not even with `try_lock`.

What about spinlocks?

The alternative to mutexes are spinlocks. For example, a widely used implementation is juce::SpinLock. It is probably being used in half of the VST plug-ins out there, and the other half will use something very much like it. I am not a huge fan of this class, because its API is not STL-compatible (so you can't use it with `std::scoped_lock` and `std::unique_lock`), and it doesn't use the optimal memory order flags in the implementation. But we can easily fix that and put together a more standard C++ style version of this type of lock:

```
struct spin_mutex {
   void lock() noexcept {
       if (!try_lock())
           /* spin */;
   }

   bool try_lock() noexcept {
       return !flag.test_and_set(std::memory_order_acquire);
   }

   void unlock() noexcept {
       flag.clear(std::memory_order_release);
   }

private:
   std::atomic_flag flag = ATOMIC_FLAG_INIT;
};
```

This can now be used as a drop-in replacement for `std::mutex`, except that it now has a realtime-safe `unlock()` method. Because of this, it is much, much better than `std::mutex`. But is it good enough?

On the audio thread, we will only be using `try_lock` and `unlock` (via a `std::unique_lock` wrapper like in the first example), and never `lock`. As we can see, both of these will only do a single atomic operation, so it's perfectly safe.

On the other thread however, whenever we try to access the lock while the audio thread has it, we will use `lock` (via one of the RAII wrappers). Therefore, the non-audio thread will be spending some time in that spin loop. It will keep hammering that atomic flag again and again and again, about 200,000,000 times a second (on my machine), until the audio thread is done and the atomic test-and-set finally succeeds. This loop will completely max out the CPU load of the core it will be on. It's like using a continuous Sheldon's knock as a method to determine when someone returns home from vacation.

If this happens rarely, or if you are on a powerful machine with many cores, that's probably not a problem. In a typical audio application, apart from the audio processing itself, the most CPU time goes into rendering the GUI and doing file I/O. A short spinloop shouldn't cause you much trouble in the overall picture.

But if the lock is around your whole processing callback or a significant part of it (like in some of the use cases above), and if you are doing a lot of audio processing (let's say you're at 90% CPU load on the audio thread), depending on where you use the spinlock, you may end up waiting on it longer and more often. If you're then on a machine with not too many cores, or worse, a mobile device where energy consumption becomes really important, spinlocks like the above suddenly start looking like not a great strategy at all. Not only do they waste a lot of CPU cycles with useless instructions, sucking your battery dry, but they also keep any other thread from doing any work on that core. So if the audio thread is doing a lot and is near high load, and the message thread is spinning on that lock at the same time, any additional threads in your audio app trying to do meaningful work will possibly not have a great time.

Exponential back-off

When stuck with a problem like this, it is often helpful to look outside of the audio industry for solutions. Spinlocks are used in other applications that demand low latency and high efficiency. If we search the literature, we will find a strategy called exponential back-off. In his recent talk about the C++20 synchronisation library, Bryce Adelstein Lelbach shows an example implementation of such a strategy (*starting at 25:50 mins in the video*):

```
struct spin_mutex {
    void lock() noexcept {
        for (std::uint64_t k = 0; !try_lock(); ++k) {
            if (k < 4) ;
            else if (k < 16) __asm__ __volatile__("rep; nop" : : : "memory");
            else if (k < 64) sched_yield();
            else {
                timespec rqtp = {0, 0};
                rqtp.tv_sec = 0; rqtp.tv_nsec = 1000;
                nanosleep(&rqtp, nullptr);
```

```
            }
        }
    }

    bool try_lock() noexcept {
        return !flag.test_and_set(std::memory_order_acquire);
    }

    void unlock() noexcept {
        flag.clear(std::memory_order_release);
    }

private:
    std::atomic_flag flag = ATOMIC_FLAG_INIT;
};
```

Now, there is quite a lot to unpick here, and unfortunately the code above is specific to x86 gcc and non-portable. But the idea is to use progressively longer wait times the longer you fail to acquire the lock. First, we try to spin four times; if we can't get the lock, we insert a CPU pause instruction for the next 12 times, which will consume less energy; then if we still can't get the lock, we yield and tell the thread scheduler to put the thread back into the run-queue; and finally, if we still fail to get the lock after a series of such yields, we start putting the thread to sleep for some amount of time.

Of course, no one wants to be re-implementing the above by hand for all the platforms and compilers you care about, and then tuning the numbers for each one. Bryce suggests to instead use the new `std::atomic::wait/notify` operations in C++20 as a portable solution:

```
struct spin_mutex {
    void lock() noexcept {
        while (!try_lock())
            flag.wait(true, std::memory_order_relaxed);
    }

    bool try_lock() noexcept {
        return !flag.test_and_set(std::memory_order_acquire);
    }

    void unlock() noexcept {
        flag.clear(std::memory_order_release);
        flag.notify_one();
    }

private:
    std::atomic_flag flag = ATOMIC_FLAG_INIT;
};
```

In general, we can expect this to be low-latency. Unfortunately, it's up to the standard library implementation to decide how to implement this. For example, `std::atomic::notify_one()` might be implemented such that it does a system call to wake up another thread waiting for the lock, so (like with most of the C++20 synchronisation library) we have no guarantees that this would be safe to use for real-time audio. It is therefore not really helpful here.

So let's look again at the exponential back-off strategy. Upon closer inspection, it turns out that the above strategy is tuned for a scenario where you have many cores, and many threads (dozens, maybe hundreds) are contending for the same lock. In such situations with high concurrency and high contention, exponential back-off is really great. However, in audio, we are in a very different situation. In all relevant cases I have seen, there were only two threads involved in trying to acquire the lock: the audio processing thread, accessing some object or data structure, and a second thread modifying it (either the message thread or some other thread loading stuff in the background). And the product will typically run on consumer hardware such as a phone or a mid-range laptop.

So what can we do for the audio use case? Are any bits of the exponential back-off strategy useful for us? Let's find out. First, let's try to figure out what the different phases in the loop actually do, and measure how long they take.

Look and measure

First of all, many audio apps are cross-platform, and therefore we need to come up with something more portable than the Linux-specific thing above. I don't have a mobile development setup at home at the moment, so I can't test this on ARM, but at least I have two different Intel x64 machines with three different operating systems and three different C++ compilers, so let's see if we can get something to work on all three of them.

The first stage of the back-off strategy is just an empty spinloop like before – this will work everywhere.

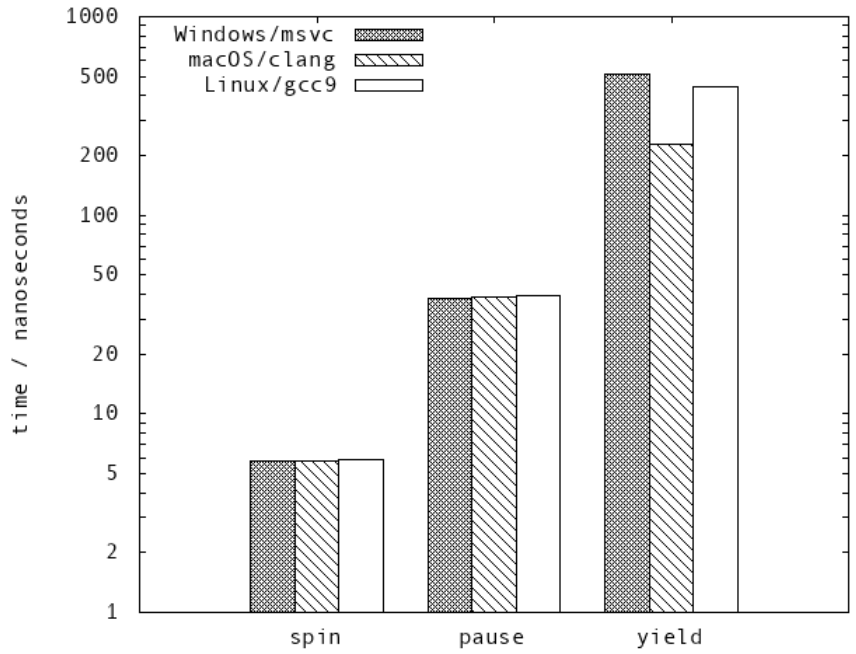
In the second stage, Bryce uses inline assembly to introduce a `rep; nop` CPU instruction, which according to Intel's Software Developer Manual (a truly great resource!) is equivalent to a `pause` instruction, effectively causing the CPU to briefly idle. This is great on modern Intel CPUs, because not only does it save energy compared to the busy wait, but it also allows another thread to make progress on the same core thanks to hyperthreading. It's effectively CPU magic for implementing efficient spinlocks.

The code as written above is unfortunately not portable, because MSVC no longer allows inline assembly (thanks, Microsoft). But it turns out there is a portable way to do this: we can call an x86 intrinsic called `_mm_pause()`, which you can include via `#include <emmintrin.h>`. Even though it's not part of the C++ standard, every major compiler targeting Intel x64 will understand this and turn it into the right instruction, which is documented here.

On ARM, it will look slightly different – you could perhaps use the WFE instruction instead to achieve a similar effect. I won't go into ARM details this time, as this blog post focuses on Intel, but it's on my to do list.

The third stage after the CPU pause is a thread yield, which can be done in standard C++ as well: just call `std::this_thread::yield()` for the same effect. The fourth stage introduces sleeping, but as we will see later, this is less interesting for us, so let's ignore sleeping for now.

Let's take the three stages we got so far, and measure how much time, on average, they consume – at least, on my two machines – when used in a spinloop:



Average time for a single spinlock loop iteration using different back-off levels, across three different operating systems, three different compilers, and two different machines (Dell XPS 15″ and MacBook Pro, both with a 6-core 2.9 GHz Intel Core i9 processor and 32 GB RAM).

Note that I used a logarithmic scale for the time. Every stage takes roughly one order of magnitude longer per loop iteration than the previous one. The linearity of the plot in log scale demonstrates the exponential nature of the different stages.

The spin, which is basically just checking the atomic flag and increasing the loop counter, takes about 5 ns on both machines and all three OSes. The `_mm_pause()` takes about 39 ns. Finally, the yield consumes between 230 and 510 ns, depending on the OS, showing some variance across OSes. This is expected, because every OS implements their

thread scheduler differently.

It's worth looking at the yield more closely.

Don't yield in a loop

In the scenario above, I would have expected the yield to take much longer – after all, a timeslice of the thread scheduler is typically something between 10 and 100 ms. And indeed, this happens sometimes.

For my next test, I recreated a scenario typical of high-load music software usage. I let the above yield-in-a-loop spin inside a plug-in that synthesises some sound, which was loaded inside a DAW with a whole bunch of other plug-ins and tracks going on, including something CPU-intensive like a convolution reverb with a very long IR, and various other apps and processes running at the same time.

And indeed, in this scenario, a single iteration of the yield loop would sometimes take 10 or even 30 ms. This is an indicator that our waiting thread was actually scheduled to the back of the run-queue and another thread jumped in. But it was still happening very rarely, perhaps once every few seconds, and only if I drive my machine very hard. I imagine that on a less powerful machine (both my laptops have a 2.9 GHz i9 with six cores) under high load, the re-schedulings might happen more often , and/or take longer. That's when you will start dropping GUI frames, but this is actually expected in this scenario: under high load, what we want is the audio to not glitch and other threads doing important work to be able to progress.

But in general, almost all of the time, our message or I/O thread would just spin in that loop, repeatedly calling `std::thread::yield()` with nothing else to do. Each such call triggers a chain of events. It summons the thread scheduler, at which point the OS kernel does the whole dance around suspending the thread, putting it all the way back into the run-queue, then realising it's the only thread in the queue anyway, and then resuming it again. That's what takes 230-510 ns in the graph above. It is doing a lot of work, actually even worse than the busy waiting in a certain way, because it also keeps the OS kernel busy. And since there is no contention for that lock in our scenario, doing a yield in a loop seems like a bad idea: we are back to burning unnecessary energy, bumping that CPU core up to a constant 100% load again, without significant benefit. We don't want that!

Don't sleep

Now that we figured out we probably don't want to yield repeatedly, it doesn't make much sense to consider the next possible back-off stage: putting the thread to sleep. Sleeping implies yielding as well, and both yielding and sleeping are scheduling techniques rather than wait-techniques. But remember, we don't have contention on that spinlock. If the audio thread has it, there is typically only one other thread waiting for it (and that's the one we're in).

Moreover, sleeping creates an additional problem. If the CPU load is high, the audio thread might spend 90% or more of its time processing audio. If the app is architected in a particularly unfortunate way, most of that might be under the spinlock. So, if the callback period is every 3 ms (a typical value), the other thread has only a 300 µs time window, or less than that, to jump in and do its work. If it's sleeping, it might repeatedly miss that narrow window of time and not proceed at all. This won't cause any audio glitches, but it is a recipe for a GUI that keeps dropping frames and feels unresponsive. On the one hand, we don't want to burn unnecessary energy; on the other hand, we also want the message thread to jump in and acquire the lock from the audio thread as soon as possible so that things keep flowing smoothly. How can we achieve that?

Let's tune this for audio

To get to a suitable implementation, we need to figure out which of the possible back-off stages we should use for our spinlock, and how often. Instead of guessing, let's go back to our original use cases and measure how long they take.

It turns out that this is spread out over a relatively broad spectrum. If the audio thread only does a quick lookup, perhaps inside a `std::vector` which it finds to be empty, such things typically take 100 ns or less (for my audio plug-in on my machine). If we have to traverse only a few elements, this will typically be under 1 µs. On the other end of the spectrum, if the lock is around the whole audiograph, and our instrument is quite CPU-heavy, one round of processing the graph and producing an audio buffer can take up to 1 ms.

In other words, the time it takes until the audio thread releases the lock can vary over five orders of magnitude. We need a spinlock that will work well for all these cases. For this, we need to choose appropriate values for how often each back-off stage should get executed on the waiting thread while it stays inside the loop.

Let's start with the first stage, the spin. I chose to do this five times, which brings us up to 25 ns. This is probably the shortest amount of time in which the audio thread might complete any kind of meaningful operation. In all honesty, this is the most arbitrary of the values I picked. I might just as well have chosen four or three. If the first spin fails, most likely the next couple of spins will fail as well.

The next stage is to do `_mm_pause()` in a loop, which takes 39 ns each time around. I conducted some additional measurements and found that of those 39 ns, about 90% are spent with the CPU actually paused, and the remaining 10% are spent checking the atomic flag and incrementing the loop counter. 10% overhead is not bad. But we are still touching memory and the CPU's integer unit every 39 ns in those 10%, and we probably can do better. By doing ten subsequent `_mm_pause()`'s, we can bring that number down from 10% to 1%. Each such iteration of 10 `_mm_pause()`s now takes about 350 ns. So, as the second back-off stage, I chose 10 iterations of a single `_mm_pause()`, and then after that I switch to the third back-off stage, which does 10 `_mm_pause()`s at a time.

Annoyingly, I actually had to write out the 10 `_mm_pause()`'s by hand, because when checking the codegen in godbolt, I found that if I stick them into an inner loop, the compilers don't unroll that loop – instead, they insert a loop counter that then gets incremented and checked, which defeats the purpose of putting the core to sleep.

Now the waiting thread will spend most of its time in the stage that does 10 `_mm_pause()`'s at a time. This is exactly where we want it to be: on a spinlock that consumes very little energy and allows other threads to progress. At the same time, the time between each check of the lock (350 ns) is still short enough that the waiting thread won't miss its opportunity of acquiring the lock after the audio thread is done, even with small buffer sizes and under high audio processing load.

But when we reach a waiting time around 1ms (or about 3000 iterations of the 10 `_mm_pause()`s-loop), we should begin to worry. This is getting close to the maximum amount of time that the audio thread will keep itself busy during a single processing callback. If we are here, the audio thread is operating at high CPU load. It might get close to missing its deadline (and then we're in bigger trouble), or perhaps the system is just generally clogged up with threads trying to do other things, or something weird might be going on.

In this case, we choose to call `std::this_thread::yield()` once, to give the scheduler a chance to sort things out (the waiting thread is not the priority in this situation!). But remember, we don't want to be yielding in a loop as it would increase CPU load even further. Instead, after that single yield, we immediately go back into the previous stage of looping in the low-energy 10 `_mm_pause()`s stage. We keep going back and forth between these two states until the waiting thread finally gets the lock.

Considering all of the above, this strategy seems to me like the most efficient way to implement a spinlock for a typical audio plug-in.

Implementation

Let's turn the above strategy into C++. It turned out that sticking all stages into a giant for-loop like in Bryce's talk isn't actually that great for our use case. The optimiser and branch predictor can do a much better job if we break things up by stage and make a separate loop for each one, instead of a single loop with four different branches in it. It makes our intent clearer and the code more readable as well. In the end, I arrived at the implementation below, which I will use in my code from now on:

```
#include <array>
#include <thread>
#include <atomic>
#include <emmintrin.h>

struct audio_spin_mutex {
    void lock() noexcept {
        // approx. 5x5 ns (= 25 ns), 10x40 ns (= 400 ns), and 3000x350 ns
        // (~ 1 ms), respectively, when measured on a 2.9 GHz Intel i9
        constexpr std::array iterations = {5, 10, 3000};
```

```
            for (int i = 0; i < iterations[0]; ++i) {
                if (try_lock())
                    return;
            }

            for (int i = 0; i < iterations[1]; ++i) {
                if (try_lock())
                    return;

                _mm_pause();
            }

            while (true) {
                for (int i = 0; i < iterations[2]; ++i) {
                    if (try_lock())
                        return;

                    _mm_pause();
                    _mm_pause();
                    _mm_pause();
                    _mm_pause();
                    _mm_pause();
                    _mm_pause();
                    _mm_pause();
                    _mm_pause();
                    _mm_pause();
                    _mm_pause();
                }

                // waiting longer than we should, let's give other threads
                // a chance to recover
                std::this_thread::yield();
            }
        }

        bool try_lock() noexcept {
            return !flag.test_and_set(std::memory_order_acquire);
        }

        void unlock() noexcept {
            flag.clear(std::memory_order_release);
        }

    private:
        std::atomic_flag flag = ATOMIC_FLAG_INIT;
    };
```

Summary

Ideally, you should avoid having to do any non-trivial thread synchronisation in your audio code. Instead, use `std::atomic`s, lock-free FIFOs, immutable data structures, and separate your processing code from your data model as much as possible.

But if you have to use locks, never use `std::mutex` on the audio thread, not even with `try_lock()`. While that call is realtime-safe, the subsequent `unlock()` is not – hard to see because it will be hidden behind a `std::unique_lock` or similar RAII wrapper.

Spinlocks are a good fallback solution, but if you find they waste too much energy and CPU time, you can back off from the busy-wait spin loop to progressively longer sequences of CPU pause instructions (`pause` on x64, `WFE` on ARM). Once in a while, yield the waiting thread to allow the thread scheduler to do its job.

This wait loop should never run on the audio thread; the audio thread should only ever call `try_lock()` and fall back to an alternative strategy on failure.

If you choose to use a progressive back-off implementation using pause and yield, measure what your code is doing, and tune the numbers for your use case. For mine, the above implementation seems to do the job.

Thanks much to Dave Rowland and Gašper Ažman, who helped me a lot with putting all this together.

C++

PREVIOUS POST
Trip report: February 2020 ISO C++ committee meeting, Prague

NEXT POST
Trip report: C++ Siberia 2020

13 Comments

Sander Bouwhuis
15 April 2020 at 09:36

Thanks for this insight! Although I don't do anything with audio, I do need mutex-like behaviour without the mutex problems.

REPLY

Julien
15 April 2020 at 10:07

Very interesting post, Timur. This is the first I hear about the non realtime-safety of mutex unlock(), and I had never thought about it, I always assumed that try_lock was a decent choice.

REPLY

Bret Alfieri
15 April 2020 at 21:10

Keep in mind that _mm_pause() can vary among vendors and SKUs. Consider exponential back-off with pauses bounded by cycles counts from RDTSC, which will provide a more adaptive backoff strategy.

REPLY

Jon Chesterfield
15 April 2020 at 23:44

Pinning the back off timing to expected fractions of the audio frame is interesting. That probably maps onto non-audio work where the kernel time slice or other minimum latency (network, cache miss) is approximately known.

Potentially worth running a benchmark at build or init time to choose magic numbers appropriate to the machine one is running on.

REPLY

Timur Doumler *(Post author)*
16 April 2020 at 00:04

Great comment. Yeah I thought about running a benchmark on init time but I decided it was overkill for my current use case. Another thing would be to pass the current buffer size and

sample rate (from which you can derive the audio processing callback time – it's a runtime variable that can change very occasionally when the user changes their settings) into the spinlock, so instead of "something around 1 ms" we can use a more precise number for the last backoff stage.

REPLY

Erik Rigtorp

21 April 2020 at 06:43

That's a very poor spin lock implementation. TAS instead of TTAS: https://en.wikipedia.org/wiki/Test_and_test-and-set. Backoff in lightweight mutexs is used to increase throughput by reducing number of context switches, but will increase worst case latency. It can also be used to improve scalability of spinlocks by reducing pressure on the memory bus. But scalability can be addressed by using a spinlock with better scalability such as ticket lock or even better a queueing lock as used in the linux kernel.

std::mutex::unlock() and atomic::notify_one() uses the same underlying syscall futex(FUTEX_WAKE) and equivalent on Windows. This syscall is "real-time safe" on linux since it is O(1) and should only take a microsecond or so. It also supports priority inheritance which is what you want for your use case. It seems like the issue you are having is that the real-time processing thread is not configured with high enough priority. If you set SCHED_FIFO with a priority less than the audio input interrupt handler but higher than all the other kernel threads and use a priority inheriting mutex. Then when your real-time thread is blocked by the low-prio UI thread (for example) it's priority will be boosted above for example mouse input etc. If you have high contention you can construct a mutex or spinlock such that different threads have different priorities when acquiring the lock.

Spinlocks are really only good in non-preemtible contexts inside for example a kernel or DPDK packet processor where there is only one runnable thread per core.

REPLY

Timur Doumler *(Post author)*

10 July 2020 at 10:08

Thanks for your comment!

Regarding TAS vs. TTAS: Good point. TTAS is impossible in C++ with `std::atomic_flag`, because it doesn't let you test the flag without setting it. TTAS could be implemented by choosing `std::atomic<bool>` instead. This makes it not portably lock-free, but we are writing x86-specific code here anyway, so that's probably okay, we can add a `static_assert(std::atomic<bool>::is_always_lock_free);` which will always be `true` on x86.

Regarding `unlock()` and `notify_one()` being real-time safe: it might be the case on Linux if you are familiar with that specific implementation of that syscall, but what about Windows and macOS? You don't know what's happening inside their implementations. We are talking about cross-platform code, and as far as I know you cannot rely on that syscall being real-time safe across all operating systems that support C++ and x86. So the only reasonable advice in my opinion is to generally avoid any type of syscall in a piece of code that is supposed to be real-time safe across platforms.

This is pretty much the reason why I advise against using any of the facilities in the new C++20 synchronisation library in real-time audio code, which also specifically includes `std::atomic::notify_one()`.

REPLY

Fabian

12 July 2020 at 13:52

> This syscall is “real-time safe” on linux

I’ve been searching for some definite information on how real-time safe futex(FUTEX_WAKE) really is. I’ve been avoiding using std::mutex::unlock() (and hence also try locks) for this very reason in realtime safe code.

I was always under the impression that it will not wait/block if no other thread is waiting for the lock, but if there are other threads waiting for the lock to become free (in particular – higher priority threads) then a futex(FUTEX_WAKE) could cause a context switch to another thread. Is this true?

From the “Futexes Are Tricky” paper [1], it says the following about FUTEX_WAKE:

“The normal futexes are not realtime-safe. There might be extensions in future which are, though. Whether the woken thread gets executed right away or the thread waking up the others continues to run is an implementation detail and cannot be relied on.”

If someone could help me clarify this once and for all, I’d be super happy!

[1]: https://akkadia.org/drepper/futex.pdf

REPLY

Emile

8 August 2021 at 18:51

Great article, thank you!

I would love to see a simple example of a lock-free FIFO with immutable datastructures, or something along the lines of what you would consider the correct solution to this problem.

When I think about this, I quickly encounter issues and I seem to only be able to come up with rather complicated solutions, would love to see a worked out/elegant solution that is minimal but a good starting point (i.e. covering common basic usecases) for this particular dual thread combo, with a GUI/messaging thread (relatively slow and can wait) and audio thread (no block/wait) trying to communicate data.

REPLY

nyanpasu64

1 April 2023 at 11:35

My personal preference is to use either a lock free FIFO queue for sequential data preserving all items pushed, or an atomic or triple buffer for publishing the latest state only, overwriting older values not observed (triple buffers are SPSC only). They have nice theoretical properties, but I’m not sure about the CPU overhead and cache misses etc. (triple buffers have a 3x memory overhead, but that’s nothing compared to an Electron app).

REPLY

Evan Murray

16 October 2021 at 23:15

Hi Timur,

I just wanted to say, this is an insightful article; I still read it today to help me understand how locks work and the implications they come with.

I was curious, are there any examples you know with the audio thread where deadlocks and/or live locks would happen?

Thanks again for your insight!

REPLY

Timur Doumler *(Post author)*
17 October 2021 at 12:01

Can't really think of an example because these are things you want to avoid doing on the audio thread. I have seen production code where the audio thread would wait for a lock, usually unintentionally. Sometimes this will cause audible glitches, other times it goes undetected.

REPLY

Hammi
10 January 2022 at 22:21

Hi Timur
I just watched your video (https://www.youtube.com/watch?v=Tof5pRedskI)
I'm working on video games and for fast random generator numbers, we are using recalculated random numbers. I will post the code here. I'm sure you can modify it to compile on your machine since the code is super simple. In our project, we are using 10000 pre-calculated normalized random numbers and we have another table for int numbers.
We wrote our table in a cpp file like:
static const F32 g_fStaticRandomArrayArray[] = {
0.000022f,0.085032f,0.601353f,0.891611f,0.967956f…};

static const U32 g_uStaticRandomArrayArray[] = {
13287737,395198432,1722090732,1618700180,70927263,…}

For our use case, the overhead is almost zero and it is good enough when the quality of random numbers is not important.

```
#pragma once

///code used to produce these number
/*CRandomNumber rnd;

rnd.Seed(1);

std::string txt = "static const F32 g_fStaticRandomArrayArray[]={";

for (size_t i = 0; i < 10000; i++)
{
if (i)
{
txt += ",";
}
if (i % 10 == 0)
{
txt += "\n";
}
```

```
txt += std::to_string(rnd.GetRandomFloat())+"f";
}
txt += "\n};";

txt += "\n\n\nstatic const U32 g_uStaticRandomArrayArray[]={";

for (size_t i = 0; i < 10000; i++)
{
if (i)
{
txt += ",";
}
if (i % 10 == 0)
{
txt += "\n";
}

txt += std::to_string(static_cast(rnd.Rand()));
}
txt += "\n};";

FILE* F = fopen("d://array.txt", "w");
fwrite(txt.c_str(), 1, txt.length(), F);
fclose(F);
*/

///a list 10000 of random numbers already calculated.only use it when quality of random number do not
matter
class CFastStaticRandomNumber
{
public:
CFastStaticRandomNumber();
~CFastStaticRandomNumber();

inline void Seed(U32 seed)
{
m_index = seed % EArraySize::ESize;
}

///randomize the seed
void RandomizeSeed();

///get a random float
inline F32 GetRandomFloat(F32 fMin, F32 fMax)
{
F32 rnd = GetANormalizedRandomNumber();
return fMin + rnd * (fMax – fMin);
}

///get a random number[0,1]
inline F32 GetANormalizedRandomNumber()
{
return m_array[GetNextIndex()];
}

///rand int
inline U32 Rand()
{
return m_uArray[GetNextIndex()];
```

```
}

/// get random in range
inline I32 GetRandomIntForRange(I32 minI, I32 maxI)
{
if (minI == maxI)
{
return minI;
}

if (minI > maxI)
{
std::swap(minI, maxI);
}

I32 range = maxI – minI;

U32 N = Rand() % static_cast(range+1);

return static_cast(N) + minI;
}

///get a vect4 random
inline vect4 GetRandomVector4(const vect4 & vMin, const vect4 &vMax)
{
return vect4(GetRandomFloat(vMin.x, vMax.x),
GetRandomFloat(vMin.y, vMax.y),
GetRandomFloat(vMin.z, vMax.z),
GetRandomFloat(vMin.w, vMax.w));
}

private:
///arraysize
enum EArraySize
{
ESize = 10000
};

///get next index
inline U32 GetNextIndex()
{
++m_index;
if (m_index >= EArraySize::ESize)
{
m_index = 0;
}
return m_index;
}

///index
U32 m_index;

///array
const F32* m_array;

///u array
const U32* m_uArray;

};
```

```
CFastStaticRandomNumber::CFastStaticRandomNumber()
:m_index(0)
{

m_array = g_fStaticRandomArrayArray;
m_uArray = g_uStaticRandomArrayArray;

}
CFastStaticRandomNumber::~CFastStaticRandomNumber()
{

}

/************************
///randomize the seed
************************/
void CFastStaticRandomNumber::RandomizeSeed()
{

std::random_device RD;
Seed(RD());

}
```

REPLY

Leave a Reply

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

Comment *

Name *

Email *

Website

Post Comment