FluidSynth real-time and thread safety challenges

David Henningsson
FluidSynth Developer Team
diwic@ubuntu.com

Abstract

FluidSynth takes soundfonts and MIDI data as input, and gives rendered audio samples as output. On the surface, this might sound simple, but doing it with hard real-time guarantees, perfect timing, and in a thread safe way, is difficult.

This paper discusses the different approaches that have been used in FluidSynth to solve that problem both in the past, present, as well as suggestions for the future.

Keywords

FluidSynth, real-time, thread safety, soundfont, MIDI.

1 Introduction to FluidSynth

FluidSynth is one of the more common software synthesizers in Linux today. It features a high level of compliance to the SoundFont (SF2) standard, as well as good performance. The design is modular enough to suit a variety of use cases.

FluidSynth does not come bundled with a GUI, but several front-ends exist. It does however come with several drivers for MIDI and audio, e.g. JACK, ALSA, PulseAudio, OSS, CoreAudio/ CoreMidi (MacOSX), and DirectSound (Windows).

1.1 FluidSynth's use cases

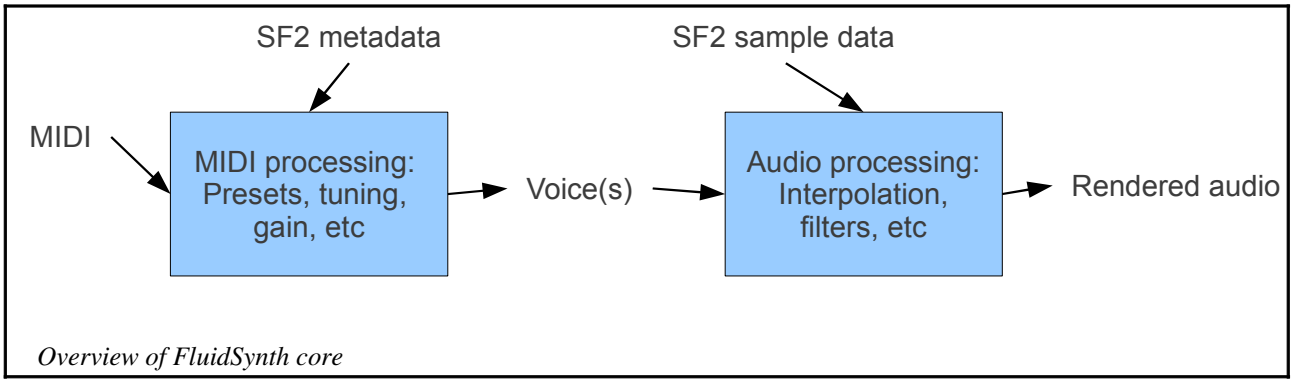
FluidSynth is not only a command-line application, but also a library used by more than 15 other applications [1], all putting their requirements on the FluidSynth engine. Requirements include:

- low-latency guarantees, e.g. when playing live on a keyboard.
- fast rendering[1], e.g. when rendering a MIDI file to disk.
- configurability, such as loading and changing soundfonts on the fly.
- monitoring current state and what's currently happening inside the engine, needed by GUI front-ends and soundfont editors.

1.2 Introduction to SoundFonts

SoundFont (SF2) files contains samples and instructions for how to play them, just like similar formats such as DLS, Gigastudio and Akai. A soundfont renderer must implement features such as cut-off and resonance filters, ADSR envelopes, LFOs (Low-Frequency Oscillators), reverb and chorus, and a flexible system for how MIDI

[1] Sometimes known as “batch processing”, a mode of operation where throughput matters more than latency.



Overview of FluidSynth core

messages affect the different parameters of these features.

1.3 More background information

1.3.1 Buffer management

FluidSynth internally processes data in blocks of 64 samples[2]. It is between these blocks the rendering engine can recalculate parameters, such as e.g. current LFO values and how they affect pitch, volume, etc.

There is also the concept of the audio buffer size, which controls the latency: the audio driver uses this size parameter to determine how often the system should wake up, executing one or more internal block rendering cycles, and write the result to the sound card's buffer.

1.3.2 MIDI processing latency

To understand some of the problems faced below, it is also important to understand the difficulty of handling all MIDI messages in a timely fashion:

- Loading soundfonts or MIDI files from disk are worst, and are not guaranteed to execute within an acceptable amount of time due to disk accesses.
- MIDI Program change messages are troublesome, somewhat depending on the current API allowing custom soundfont and preset loaders.
- Other MIDI messages, while they are not calling into other libraries (and thus unknown code latency-wise), still take some time to process, compared to just rendering a block.

[2] It is known as the FLUID_BUFSIZE constant in the code, and I have never seen anyone change it.

2 Architecture before 1.1.0

FluidSynth has always had a multi-threaded architecture: One or more MIDI threads produce MIDI input to the synthesizer, and the audio driver thread is asking for more samples. Other threads would set and get the current gain, or load new soundfonts.

2.1 Thread safety versus low latency

When the author got involved with the FluidSynth project, a few years ago, thread safety was not being actively maintained, or at least not documented properly. There weren't any clear directions for users of FluidSynth's API on what could be done in parallel.

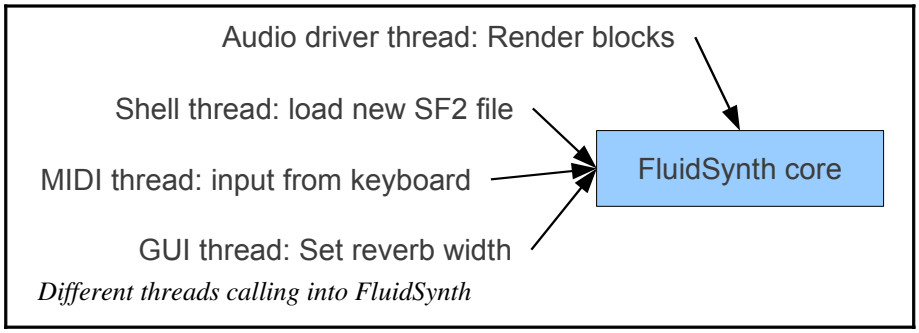
Yet there seems to have been some kind of balance: Unless you stress tested it, it wouldn't crash that often – even though several race conditions could be found by looking at the source code. At the same time, latency performance was acceptable – again, unless you stress tested it, it wouldn't underrun that often.

This “balance” was likely caused by carefully selecting places for locking a mutex – the more MIDI messages and API calls protected by this mutex, the better thread safety, but worse latency performance. In several places in the code, one could see this mutex locking code commented out.

2.2 The “drunk drummer” problem

An additional problem was the timing source: The only timing source was the system timer, i.e. timing based on the computer's internal clock. This had two consequences.

The first: All rendering, even rendering to disk, took as long time as the playing time of the song, so if a MIDI file was three minutes long, rendering



Different threads calling into FluidSynth

that song would take three minutes, with the computer idling most of the time.

The second: With larger audio buffer/block sizes[3], timing got increasingly worse. Since audio was rendered one audio buffer at a time, MIDI messages could only be inserted between these buffer blocks. All notes and other MIDI events therefore became quantized to the audio block size. (Note that this quantization is not at all related to the intended timing of the music!)

This problem was labelled “the drunk drummer problem”, since listeners were especially sensitive to the drum track having bad timing (even though the same bad timing was applied to all channels).

3 Architecture in 1.1.0 and 1.1.1

3.1 Queuing input

To make FluidSynth thread safe, it was decided to queue MIDI messages as well as those API calls setting parameters in the engine. This was implemented as lock-free queues – the MIDI thread would insert the message into the queue, and the audio thread would be responsible for processing all pending MIDI messages before rendering the next block.

3.2 The sample timer

To make the drunk drummer sober again, the “sample timer” was added – that uses the number of rendered samples as a timing source instead of the system timer. This also allowed features such as fast MIDI-file rendering to be added. This was implemented so that on every 64th sample, a callback was made to the MIDI player so that it could process new MIDI messages.

3.3 Problems with the overhaul

3.3.1 Worse latency

As the audio thread was now expected to process all MIDI messages, this meant more pressure on the MIDI messages to return timely, and audio latency now had to take MIDI processing into account as well. The sample timer made this even worse, as all MIDI file loading and parsing now also happened in the audio thread.

3.3.2 Reordering issues

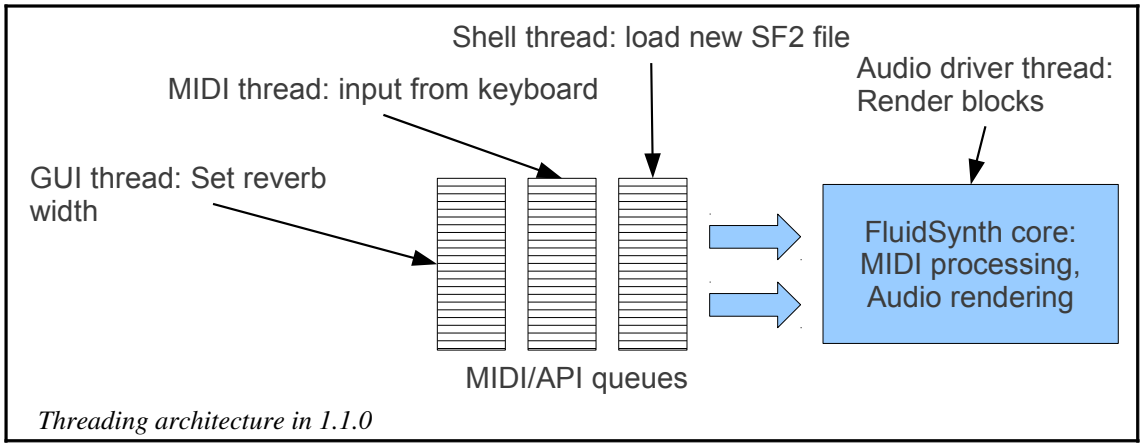
To aid the now tougher task of the audio thread, program change messages were still processed in the MIDI thread, queueing the loaded preset instead of the MIDI message. However, this also meant that bank messages had to be processed immediately, or the program change would load the wrong preset. In combination with API calls for loading soundfonts, this became tricky and there always seemed to be some combination order not being handled correctly.

3.3.3 Not getting out what you're putting in

Since API calls were now being queued until the next rendering, this broke API users expecting to be able to read back what they just wrote. E g if a GUI front-end set the gain, and then read it back, it would not read the previous set value as that value had not yet been processed by the audio thread.

This was somewhat worked around by providing a separate set of variables that were updated immediately, but since these variables could be simultaneously written by several threads, writes and reads had to be atomic, which became difficult

[3] In high-latency scenarios, such as a MIDI file player, you would typically want as large buffer as possible, both to avoid underruns and to improve overall performance.



Threading architecture in 1.1.0

when writes and reads spanned several variables internally.

4 Architecture in 1.1.2 and later

To overcome the problems introduced with 1.1.0, the thread safety architecture was once again rewritten in 1.1.2. This time, it was decided to split the engine into two parts: One for handling MIDI and one for handling audio. Hard real-time is guaranteed for the audio thread only, in order not to miss a deadline and cause underruns as a result.

For MIDI, the synth no longer has an input queue, but is instead mutex protected[4]. This means, that if one thread calls into the API to do something time intensive, such as loading a new soundfont, other MIDI threads will be delayed in the meantime and will have to wait until soundfont loading is finished.

4.1 The new queue

Instead of having a MIDI input queue, the queue has now moved to being between the MIDI handling and the audio thread. Instead of queuing the MIDI messages themselves, the outcome of the MIDI processing is queued to the audio thread. This releases pressure on the audio thread to handle MIDI processing, so audio latency is improved. If MIDI processing is lengthy, the end result will be that the result of that event is delayed – as compared to 1.1.0, where the result would have been an underrun.

[4] The mutex can optionally be turned off for cases where the API user can guarantee serialized calls into the API, e.g. in some embedded use cases.

4.2 Return information

A queue with return information also had to be added, with information flowing from the Audio rendering thread to the MIDI threads. This is used to notify the MIDI processing when a voice has finished, so that the voice can be reallocated at the next MIDI note-on event. This return information queue is processed right after the mutex is locked.

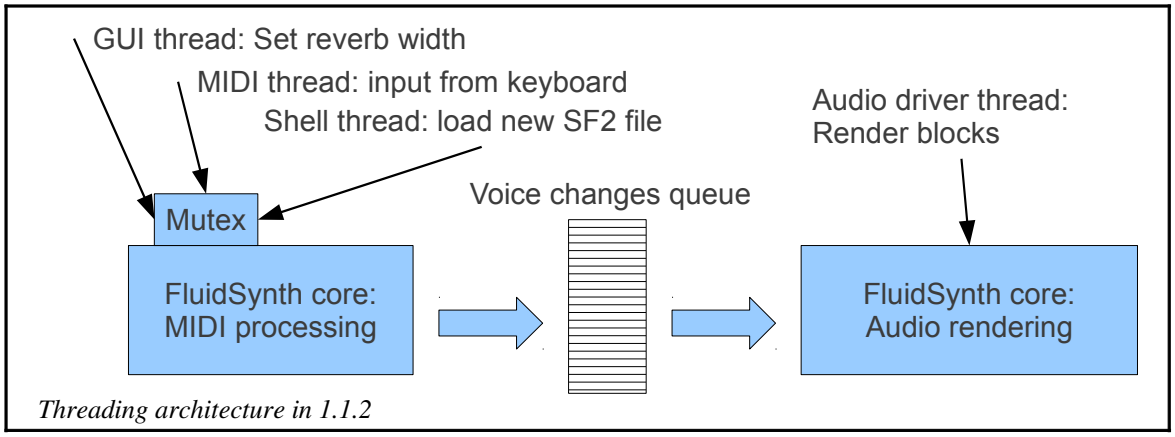
5 Conclusion and suggestions for the future

While the architecture in 1.1.2 seems to have been more successful than the previous attempts in terms of stability, it is still not optimal. There is still work that could be done to improve the thread safety and real-time performance further.

5.1 Sample timers

Given the new architecture, the sample timer mechanism needs to be rewritten to work optimal under low latency conditions: as it currently stands, the audio thread triggers the sample timer, which in turn performs potentially lengthy MIDI processing.

To solve this problem without regressing back to the “drunk drummer”, one would need to add a “mini sequencer” into the event queue so that events can be added to be processed by the audio thread not before all 64 sample blocks are processed, but also between them. This would further require a time aware MIDI part of the synth – so the MIDI part would know where into insert the new queue item. Also the MIDI player needs to have a separate thread, monitoring the progress of the audio stream and adding more MIDI events as necessary.



Threading architecture in 1.1.2

5.2 Synchronous MIDI/Audio

In particular, synchronous MIDI and audio can be a problem when using JACK MIDI in conjunction with JACK audio – because JACK calls into both MIDI and audio callbacks synchronously. To try to avoid MIDI blocking audio rendering, MIDI input could be queued to a lower priority thread, and processed as time permits. Caution must be taken to make that this does not happen when JACK is running in its "freewheeling"[5] mode, where MIDI and audio callbacks should be processed in the exact order they arrive.

5.3 Monitoring the audio thread

A sometimes asked for feature, in particular by soundfont editors and sometimes by other GUI frontends, is to be able to monitor the progress of the audio rendering.

This could be e.g. to see the current sample position of a particular voice, or to be able to receive callbacks whenever something happens in the audio thread, e.g. when a voice enters a new envelope stage. This is currently difficult as information is optimized to flow from the MIDI part to the audio thread, not the other way around.

One solution to this problem would be for the audio thread to continously write down relevant information into a buffer, that the MIDI part could read upon request. Caution must be taken in order not to have the MIDI part read partially updated information (and thus get a potentially inconsequent view), but at the same time an ongoing read should not block a new write. This can be done with some clever atomic pointer exchanges.

The audio thread's write-down operations could potentially be turned on and off, in order not to hurt performance for users not needing the feature.

[5] This is a JACK term indicating that the JACK server is currently rendering as fast as CPU permits, e.g. when performing a mixdown to disk, not unlike the "fast rendering" mode of FluidSynth.

6 Acknowledgements

Thanks go to the rest of FluidSynth developer team (Joshua "Element" Green and Pedro Lopez-Cabanillas), as well as all other contributors to FluidSynth, including patch writers, testers, and people helping out on the FluidSynth development mailing list.

References

[1] http://sourceforge.net/apps/trac/fluidsynth/wiki/Applications