

Multithreading in GStreamer

How do flushes work

Alicia Boya García

Igalia

GStreamer Conference 2026 Prague

About me

- ▶ I work at Igalia
- ▶ Maintainer of the MediaSource Extensions stack for WebKit
 - ▶ WebKitGTK (often embedded in desktop applications)
 - ▶ WPE (WebKit for embedded Linux hardware)
- ▶ You often see me working in demuxers and with threading bugs



Outline of the talk

Threading and data flow in GStreamer

Locks in GStreamer

Flushes in GStreamer

Aftermath of the flush: misc. details



Section 1

Threading and data flow in GStreamer



Lower level synchronization objects used

These are the synchronization primitives used in GStreamer core:

- ▶ Threads
- ▶ Atomics
- ▶ Mutex
- ▶ Condition variables
- ▶ Recursive mutex

An introduction of those is beyond the scope of this talk for time and scope reasons.



The streaming threads

- ▶ Certain pads *start tasks* (threads) when *activated* (which happens automatically on READY_TO_PAUSED). Examples:
 - ▶ In **push** mode, the **srcpads of source** elements
 - ▶ In **pull** mode, the **sinkpads of the sink** elements
 - ▶ The **srcpads of queue** elements (e.g. queue, multiqueue...)
- ▶ For an example, see `gst_base_src_activate_mode()`
- ▶ The GstTask thread will run the *task callback* in a loop for as long as the task is in STARTED state
 - ▶ As a convention, task callbacks names end with **_loop()**, e.g. `gst_base_src_loop()`, `gst_qtdemux_loop()`
- ▶ Pads with tasks are shown in pipeline dumps with [T]



Refcounting

- ▶ GObject, GstObject and GstMiniObject are **thread-safe refcounted**
 - ▶ The object is destroyed by the thread that does the **last unref**
- ▶ Beware: **If the calling thread is not holding a reference, the object may be destroyed while in use.**
 - ▶ Rust bindings will always force you to own or borrow a reference, avoiding that risk
 - ▶ C does not enforce this
 - ▶ If you don't ensure this, it's very easy to get a **racy use-after-free**

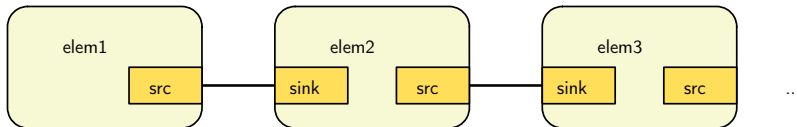


Refcounting: data mutability

- ▶ `GstMiniObjects` (e.g. `GstBuffer`) are only meant to be directly modified if you're the sole owner of the object.
- ▶ `gst_mini_object_make_writable()` takes a reference to the object and returns the same object if you were the sole owner, or a copy otherwise.

Data flow (push mode) (I)

- ▶ `gst_pad_chain(elem1:sink, buf1)` causes `elem1:sink` to process `buf1`, *synchronously*
- ▶ As part of that processing, typically some new `buf2` is created and `gst_pad_push(elem1:src, buf2)` is called, synchronously
 - ▶ This **in turn** synchronously calls `gst_pad_chain(elem2:sink, buf2)`



Data flow (push mode) (II)

The call chain will not stop growing **until some sinkpad** just consumes the buffer and **returns**, for example:

- ▶ ... because it is a pipeline sink and it **finished rendering** the frame
- ▶ ... because some element in the middle needs more data to be able to output a buffer
 - ▶ e.g. a demuxer needing another chunk of bytes from upstream to complete a frame
- ▶ ... (in a queue element or similar) because it stored the buffer in a **non-full queue** for another thread to consume

Data flow (push mode) (III)

The call chain will block if **any of the chain functions blocks**

- ▶ E.g. in a queue element, when the queue is full
- ▶ This is by design: it is how GStreamer handles *backpressure* (i.e. not overloading any element with too much data)



Data flow (push mode) (IV)

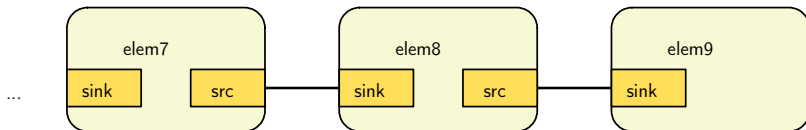
For events, it is the same, but the functions have different names:

- ▶ `gst_pad_send_event()` is akin to `chain()`
- ▶ `gst_pad_push_event()` is akin to `push()`



Data flow (pull mode)

- ▶ Same pattern as push mode, different functions:
 - ▶ `gst_pad_get_range(elem8:src, offset, size, &buf1)`
 - ▶ `gst_pad_pull_range(elem8:sink, offset, size, &buf2)`
 - ▶ `gst_pad_get_range(elem7:src, offset, size, &buf2)`
 - ▶ ... eventually the leaf returns and buf2 is filled
 - ▶ ... buf2 is transformed and to fill buf1



Serialized vs non-serialized flow (I)

- ▶ Serialized objects are processed in **strict FIFO** – will see how shortly
- ▶ Processing of **serialized** objects can **stall due to backpressure** in their normal operation
- ▶ Processing of **non-serialized** objects **must never stall** for any user-noticeable amount of time

Serialized vs non-serialized flow (II)

- ▶ **Serialized** objects
 - ▶ **Buffers**
 - ▶ Serialized **events**: STREAM_START, CAPS, SEGMENT, FLUSH_STOP, GAP, TAG, EOS
 - ▶ Serialized **queries**: ALLOCATION, DRAIN
- ▶ **Non-serialized** objects
 - ▶ Non-serialized **events**: SEEK, FLUSH_START, RECONFIGURE, INSTANT_RATE_CHANGE
 - ▶ Non-serialized **queries** (questions without side effects): POSITION, DURATION, LATENCY, SEGMENT, FORMATS, CONTEXT, BUFFERING, CAPS, ACCEPT_CAPS...

Section 2

Locks in GStreamer



Pad stream lock (I)

- ▶ `GST_PAD_STREAM_LOCK(pad)`
- ▶ Enforces that **serialized objects** are handled in strict FIFO
 - ▶ Taken before processing any serialized object
- ▶ For a pad that starts a **task** (e.g. in a src element), that *pad stream lock* becomes the *task lock* (see `gst_task_func()`)
 - ▶ This ensures the task function — e.g. `gst_base_src_loop()` — runs with the pad **stream lock taken**
 - ▶ The stream lock is not released **until the task stops running** (e.g. because of a flush or state change)

Pad stream lock (II)

- ▶ `gst_pad_chain(elem2:sink)` takes the stream lock of that pad
 - ▶ As the call chain grows, each involved sinkpad gets its stream lock locked
- ▶ `gst_pad_push(elem2:src)` does *not* take any stream lock!
 - ▶ Only srcpads with a task take the stream lock

Pad stream lock (III)

- ▶ The pad stream lock is a **recursive mutex**, and this is relied upon in core
- ▶ Because of the stream lock, GStreamer can tolerate two threads chaining on the same sinkpad concurrently
 - ▶ The two chain operations may run in any order, but they won't interleave.
 - ▶ (The use cases where this is safe and useful are niche, but it is nonetheless a helpful realization for understanding the model)

Object lock

- ▶ `GST_OBJECT_LOCK()`
- ▶ Exists on every `GstObject`
- ▶ **Non-recursive** mutex
- ▶ Protects:
 - ▶ `GstObject`: name, parent, flags
 - ▶ `GstPad`: probe list, offset, task
 - ▶ `GstElement`: state (e.g. `PLAYING`), bus, clock, pad list, contexts
- ▶ In C code it's also used in derived classes for protecting miscellaneous **state of the object** such as values of properties

Element-specific locks

- ▶ In **Rust elements**, the object lock is reserved for core. Pad subclasses instead create their own mutexes, typically named:
 - ▶ state for processing their element-specific state
 - ▶ settings for storing their own properties
- ▶ Thankfully, Rust's `Mutex<>` enforces access only occurs while locked
 - ▶ Sadly, since this is **not enforced in C** code, it's not always done consistently, which can result in **undefined behavior**
- ▶ C elements often have **their own** processing state mutex too, e.g. `_GstAppSrcPrivate::mutex`, but **other times** they just reuse `GST_OBJECT_LOCK()`



Must lock briefly

Both the object lock and element-specific locks must only be locked **briefly**

- ▶ Must **never** push, pull, chain or send anything while the object lock is taken!
- ▶ **Flushes** and **non-serialized** object handling code will usually **need** to take the object **lock** (e.g. as part of `gst_pad_send_event()`) and some element-specific lock — if you push while holding either, they will **stall**



Section 3

Flushes in GStreamer



Flushes

- ▶ A flush **ceases processing of serialized objects** downstream (push mode) or upstream (pull mode) of the flushing pad
- ▶ The most common **use case** is a **seek**:
 - ▶ When the user seeks to a new time it would be unacceptable to wait to play the new media until all the remaining buffered media has been played
- ▶ Flushes are a **general concept** of pipelined architectures, and by extension multimedia code, not just GStreamer
- ▶ Flushes are by definition edge cases and **tend to reveal bugs** not seen during buffered playback

Flushing thread

Flushes can be performed by **any thread**.

- ▶ The design of GStreamer guarantees this
- ▶ This is normally done by a **non-streaming thread**
 - ▶ Typically the **main thread** or UI thread requesting the seek
 - ▶ If the elements are not broken, this is safe



Example seek: pushmode qtdemux (I)

- ▶ Main thread calls `gst_element_send_event(playbin, time_seek)`
- ▶ The pipeline `send_event` handler **delegates** to the `send_event` handler of the **sink element(s)**
 - ▶ See `gst_bin_send_event()`
 - ▶ See `gst_play_sink_send_event()`
- ▶ A **non-bin** sink element `send_event` handler pushes the seek event upstream of its sinkpad
 - ▶ See `gst_element_default_send_event()`

Example seek: pushmode qtdemux (II)

- ▶ The seek event eventually reaches the **demuxer**, which **translates the TIME seek into a BYTES seek**, and sends the new seek event upstream
 - ▶ See `gst_qtdemux_handle_src_event()`
- ▶ The BYTES seek reaches the **source element**
 - ▶ See `gst_base_src_perform_seek()`
 - ▶ `basesrc:src` synchronously sends **FLUSH_START downstream**
 - ▶ `basesrc:src` synchronously sends **FLUSH_STOP downstream**
- ▶ The processing of the BYTES seek for the src element **returns**
- ▶ The processing of the TIME seek for the demuxer returns, and so on

Two-step flush

- ▶ GStreamer splits flush in two steps
 - ▶ **FLUSH_START** event (**non-serialized**)
 - ▶ **FLUSH_STOP** event (**serialized**)
- ▶ This is a sound design that works for pipelines of any complexity!
 - ▶ Bugs end up scoped to specific elements
 - ▶ When a buggy element causes a deadlock, it is reasonably easy to track from the backtrace

FLUSH_START, flushing thread (I)

What the event handler should do:

- ▶ **Lock** the **element-specific mutex**
- ▶ Update some **element-specific field** to indicate we're now flushing
- ▶ **Unlock and notify** all condition variables the processing code could be waiting on
- ▶ **Delegate** to the parent event handler
 - ▶ This will in turn **forward** the FLUSH_START downstream

FLUSH_START, flushing thread (II)

- ▶ By the time `gst_pad_send_event(pad, FLUSH_START)` returns, **all pads downstream** have been **signaled**
 - ▶ The streaming threads may not have wound down yet, but we have asked them to, and they will wind down shortly
- ▶ In core, `FLUSH_START` also sets the **FLUSHING** flag of `GstPad`
 - ▶ See `gst_pad_send_event_unchecked()`
 - ▶ When this flag is on, a pad will **refuse** any serialized object.
 - ▶ That is, we're not only stopping what we're currently processing, but also preventing any new processing that could otherwise undo our effort.

FLUSH_START, flushed thread

- ▶ Case 1: flushed thread is **busy**:
 - ▶ Once it's done with any element-specific **uninterruptible processing**...
 - ▶ Flushes are only handled in programmer defined locations
 - ▶ **Lock** the element-specific mutex
 - ▶ Find the element-specific flushing flag set
 - ▶ **Unlock and return**
- ▶ Case 2: flushed thread is **blocked waiting**:
 - ▶ **Wake** due to condition variable being signaled
 - ▶ Find the element-specific flushing flag set
 - ▶ **Unlock and return**

FLUSH_START, task owning pads

- ▶ At the beginning of the flush, the streaming threads will have **pad stream locks** taken — we need them to relinquish it
- ▶ `gst_pad_chain()` will **unlock** the pad stream lock **on return**
- ▶ Pad task callbacks must also use `gst_pad_pause_task()` in response to a FLUSH_START before returning — otherwise the task callback would be run again
 - ▶ Examples: `gst_queue_handle_sink_event()`, `gst_base_src_loop()`
 - ▶ A paused task **preserves its thread** but doesn't run the callback until set back to STARTED
 - ▶ The task lock (i.e. srcpad streaming lock) is **unlocked** while the task is PAUSED **and** the task function is not running



FLUSH_STOP, flushing thread (I)

- ▶ Because it is serialized, it cannot flow while the streaming threads are processing serialized data
 - ▶ **Corollary:** by the time a FLUSH_STOP has flown all the way and returned we are guaranteed that all the streaming threads have **stopped processing** serialized data



FLUSH_STOP, flushing thread (II)

What the event handler should do:

- ▶ **Lock** the **element-specific mutex**
- ▶ Update some **element-specific field** to indicate we're no longer flushing
- ▶ Reset element-specific state to prepare to handle new content, e.g. clearing queues, resetting counters...
- ▶ **Unlock** the element-specific mutex
- ▶ **Delegate** to the parent event handler
 - ▶ This will in turn **forward** the FLUSH_STOP downstream

FLUSH_STOP, flushing thread (III)

- ▶ During the handling:
 - ▶ Elements must **not send any serialized objects other than FLUSH_STOP**, as that could block and stall or **deadlock** the flushing thread
- ▶ Inside core, the **FLUSHING** flag on GstPad is also reset, allowing it to **accept serialized objects again**
- ▶ If the pad owns a task, the handler must also call `gst_pad_start_task()` so that it can later resume
 - ▶ The task callback cannot start running until the FLUSH_STOP returns because until then the flushing thread owns the pad stream lock

FLUSH_STOP, flushed thread

- ▶ On task resume, the flushed thread waits to lock the **task lock**, which is also the **pad stream lock**
- ▶ Once FLUSH_STOP returns, it successfully locks and calls the task callback
- ▶ The task callback encounters a somewhat clean element-specific state, as left by the FLUSH_STOP event handler
- ▶ Processing and streaming starts anew

Tada!!

We flushed and everything went alright!



Section 4

Aftermath of the flush: misc. details



Sticky events cleared by flushes

- ▶ **EOS, SEGMENT, STREAM_GROUP_DONE**
 - ▶ See `gst_pad_send_event_unchecked()`
- ▶ **All other sticky events are preserved by default** (e.g. `STREAM_START`, `CAPS`)



Reset-time vs non-reset-time flushes

- ▶ FLUSH_STOP had a boolean `reset_time` argument, used by sinks
- ▶ If true:
 - ▶ Running time is **reset to 0** after the flush
 - ▶ This is what you want for **seeks**
- ▶ If false:
 - ▶ The running time from previous streaming is still accumulated
 - ▶ This is what you want for user-requested **quality changes**

Async lost states (I)

- ▶ When **basesink** has `async=true` (e.g. for playback) it will call `gst_element_lost_state()` in response to **FLUSH_START**
- ▶ This will propagate **STATE_CHANGED** and **ASYNC_START** messages to parent bins, eventually reaching the pipeline
- ▶ Assuming we were in **PLAYING** state, the sink element and parent bins **element state** becomes:
 - ▶ Current: **PAUSED**, Pending: **PAUSED**, Target: **PLAYING**

Async lost states (II)

- ▶ The **running time is frozen**. See `gst_base_sink_update_start_time()`
 - ▶ That running time is later discarded in the case of a `FLUSH_STOP` with `reset_time=true`
- ▶ Later, on preroll, the `ASYNC_DONE` handler sets the elements back to `PLAYING`. See `bin_handle_async_done()`
- ▶ The change from `PAUSED` to `PLAYING` also makes the **running time resume**
 - ▶ See `gst_pipeline_change_state()`, particularly its use of `gst_element_set_base_time()`

Questions?