Trustable Future Sync Points

@brianderson, @jbauman

Status: Public. Work in progress. Seeking feedback.

Introduction

Future sync points allow clients to make a promise that other clients can depend on before that promise has been fulfilled, which can be used to enable more concurrency, reduce memory usage, and reduce latency through dynamic channel prioritization or by deferring decisions to occur later in a pipeline.

Future sync points would also allow any operation (allocation, swizzling, cache maintenance, etc.) to function as a service, where the order of operations and dependencies are defined up front asynchronously. If a device were to support future sync points in hardware, the CPU could chain many operations together in advance and shut down to save power.

This document focuses on describing a future sync point API and getting feedback on it's usability and completeness for Chrome, especially regarding any security concerns and error conditions since they may result in broken promises.

This document does not focus on the implementation details, however it does try to make the API easy to implement in a way that most calls can be asynchronous or, if they must be synchronous, that they don't need to be called as part of a critical path. An attempt is also made to make this API fully implementable in hardware, but also fully in software if hardware support is not available.

Background

Futures and promises are tried and proven concepts that are already available in many programming languages. For reference, this wikipedia article about futures is a good starting point.

Related OpenGL extensions: http://www.opengl.org/wiki/Sync_Object

Related documentation:

- GPU Service Scheduling Latency. Note: Instead of differentiating between weak vs. strong future sync points, this document introduces a prioritization scheme that applies to any kind of sync point.
- Sync Point Shader Arguments - Make late-binding decisions based on sync point state in shaders.
- Android sync driver - An Android driver that implements explicit sync points and fences.

- dma_fence driver - A Linux driver for fences. There is work to implement Android's sync driver on top of dma_fence.
- Windows GPU pre-emption - Devices already support pre-emption on Windows, which is a strong indicator that the prioritization aspects of this proposal can be implemented for other platforms.

Vocabulary

Different groups use different terms to refer to similar concepts. In an attempt to make this document easier to read for those not familiar with Chrome, here are a list of definitions for words used in this document.

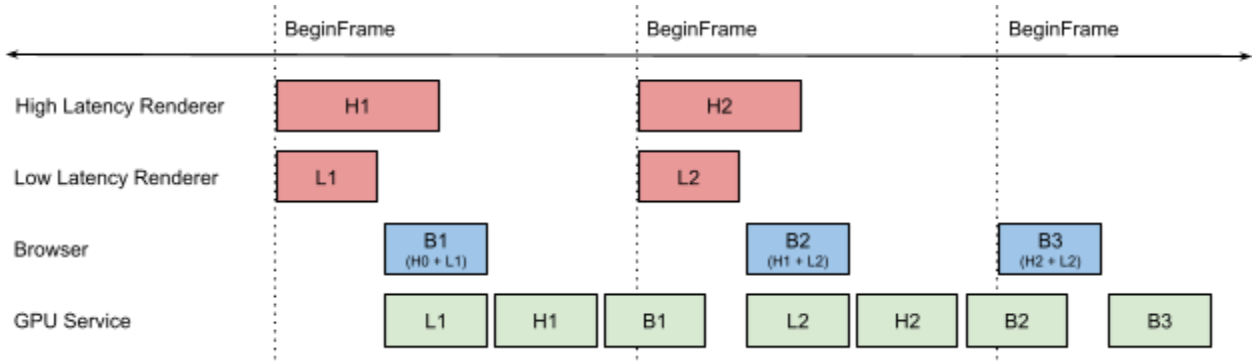
- **Renderer** - an *untrusted* process that renders the contents of a web page by producing GPU commands targeting one or more offscreen buffers.
- **Browser** - a *trusted* process that produces GPU commands to composite the offscreen buffers of multiple Renderers.
- **GPU Service** - a *trusted* process that consumes the GPU commands from the Browser and Renderers and sends them to the GPU driver/hardware.
- **Channel** - a queue that executes in FIFO order, but may be executed out of order or interleaved with respect to other channels.
- **Timeline** - a dimension on which sync points are created in a strictly monotonically increasing way. May correspond to positions within a channel or may correspond to a virtual class of events that are known to execute in FIFO order.
- **Sync point** - a representation of a specific position on a timeline.
- **Client** - pushes commands onto a channel.
- **Service** - executes commands from one or more channels.
- **Arbiter** - decides which channels the service should execute based on outstanding dependencies and current channel priorities.
- **Retired sync point** - all commands before the sync point have been executed on the service.
- **Scheduled sync point** - all commands before the sync point have been queued to the service.
- **Promised sync point** - a sync point that a client has promised it will push on to a channel, but hasn't yet.
- **Future sync point** - a sync point representing a client's promised sync point that may or may not be scheduled yet.

Motivation

This section describes 7 use cases that future sync points could be used for. Explanations of how future sync points could solve these problems are described in more detail in the Examples section after the API description.

Use case 1: Deprioritize a high latency Renderer's context

In the diagram below, we want to make sure that H1 doesn't execute before B1 on the GPU Service since B1 doesn't need H1. Executing the high latency Renderer's context before it's actually needed would only hurt the latency of everything else since the Browser doesn't actually consume H1 until B2.



Use case 2: Prioritize a specific Renderer that the Browser depends on

The basic functionality here involves being able to detect and correct for priority inversion, when a high priority channel depends on a low priority channel. We can go further than just detecting priority inversion last minute if there is a mechanism to proactively bump the priority of another channel.

Fine grained prioritization of contexts is meant to be a generic solution to compositor pre-emption. Windows has fine-grained pre-emption in their driver model; it would be nice to expose that functionality to other platforms as well.

Use case 3: Return a buffer to the Renderer before it's actually available

Future sync points will allow us to reduce memory usage of Renderers that are operating in a high latency mode without artificially limiting concurrency, by enabling a client to behave as if it is triple buffered even if it is only double buffered.

Use case 4: Synchronize multiple nested embedders efficiently

Today, Chrome has a long critical path when there are nested embedders since they must run serially in time. Future sync points would allow all embedders to run concurrently while still providing mechanisms to synchronize their embeddees.

Use case 5: Defer decisions to occur later in the pipeline

If there is a way to query whether or not a future sync point has retired, decisions could be deferred to occur at command execution time rather than command queue time to reduce latency.

Use case 6: Allow everything to be a service

Future sync points could be used to set up, for example, cache maintenance or DMA operations that run concurrently with other CPU or GPU operations. Instead of performing cache maintenance synchronously at a call site, it could be set up in advance to run asynchronously on its own virtual timeline using future sync points. The service might be able to perform optimizations by batching work from its queues.

Use case 7: Save power by chaining services and shutting down the CPU

If a workload, such as media playback or a simple animation, is mostly hardware accelerated and requires comparatively little CPU work, the CPU could batch up a chunk of its work and then chain many frames together.

Note: Chaining so far in advance without CPU intervention would require a) hardware support for future sync points and b) some kind of cancellation mechanism to maintain device interactivity. A cancellation mechanism will greatly complicate the API, so is left TBD, potentially as a separate proposal.

The API

Structs

```
struct SyncPoint {
      int32 domain; // TBD. See discussion.
      int32 timeline;
      int32 id;
}
```

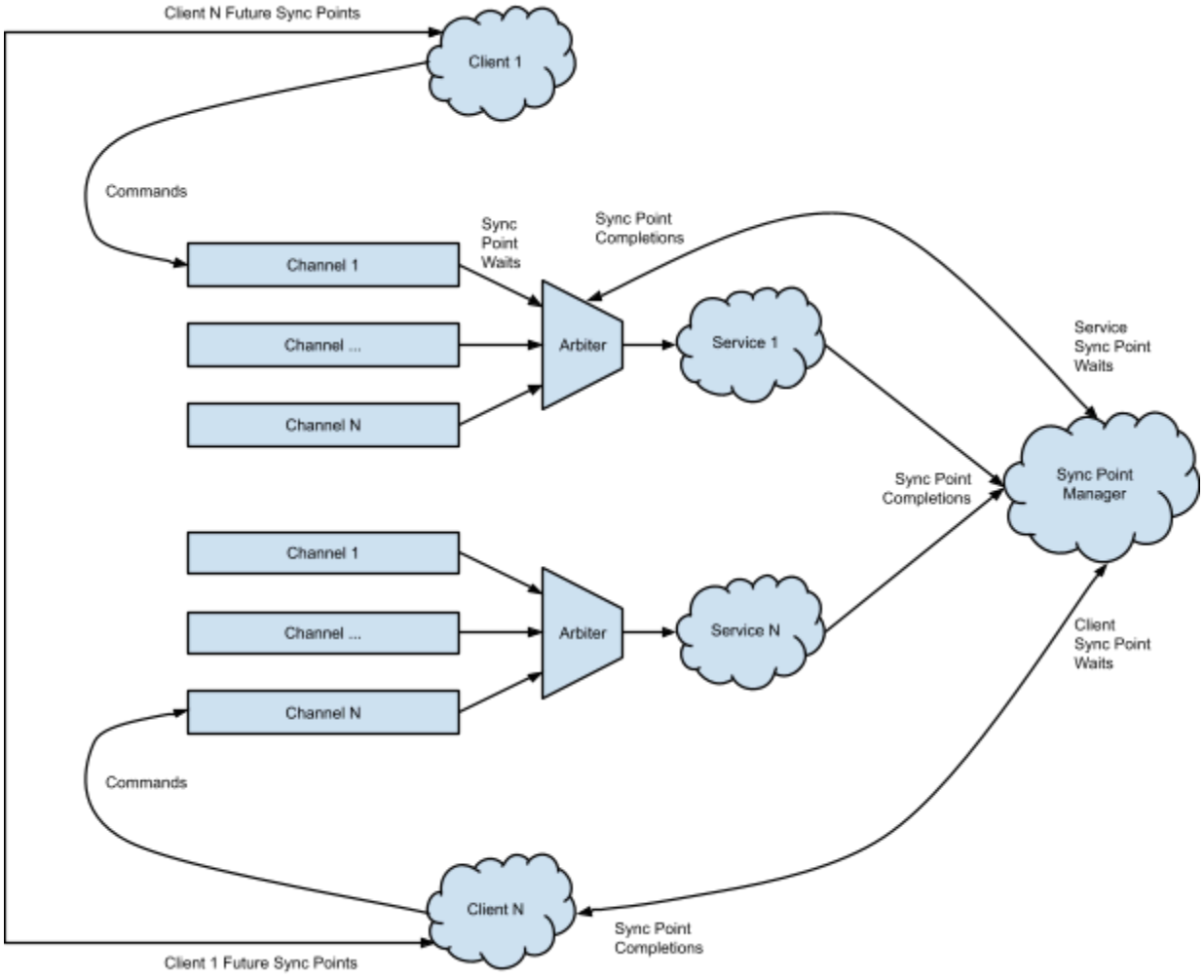
By including a domain and/or timeline as part of the sync point a client can create sync points without any synchronous round trips to the GPU Service.

Basic Ordering

The following diagram shows how different concepts interact with each other to enforce basic ordering of commands with future sync points, both client side and service side:

1. **Client side:** Client N wants to read the results associated with a sync point from client 1. In order to guarantee client visibility, client N does a client-side wait.
2. **Service side:** Client 1 trusts client N and has a service side command that depends on a result from Service N. In this case, client 1 can insert a service-side wait on the sync point from client N.

Note: This document assumes every service has arbitration logic that decides which channel to execute and is represented in the diagram as an “Arbiter”. This document also assumes there is a sync point manager that is aware of all sync points in the system. Although these concepts are depicted as single blocks for the sake of simplicity, it’s conceivable that implementations could be more distributed than the diagrams imply.



Basic Ordering: Sync points

- `SyncPoint InsertSyncPoint()`

    - Inserts a sync point on the current channel and returns its handle.
    - All sync points inserted with this method retire in FIFO order.
    - Similar to glFenceSync.
- `SyncPointWait(SyncPoint)`
    - Queues a command on the current channel to perform a server-side wait on the sync point provided.
    - Similar to glWaitSync.
- `void SyncPointClientWaitUntilComplete(SyncPoint, Callback)`
    - A client-side wait that notifies the client of completion asynchronously via a callback.
    - Useful for knowing when service-side results are available.
    - Similar to glClientWaitSync.

Basic Ordering: Future sync points

Future sync points are created (promised) and retired (fulfilled) as separate actions so other clients can use or wait on them before they've been retired. Every sync point can be thought of as transitioning through the following three states:

1. **Promised**: A client has created a sync point and intends to retire it.
2. **Scheduled**: The sync point will retire in finite time because all dependencies are known server-side.
3. **Retired**: Execution of the command stream has passed the sync point.

Often times, there will be a class of future sync points that we know will retire in the order they are created. Instead of allowing all future sync points to retire out of order with respect to each other, we create virtual timelines for them to formalize classes of sync points that are known to retire in the order they are created. Using virtual timelines should also simplify how future sync points are managed, since a given timeline could use a simple counter to track retirement.

- `int32 CreateFutureSyncPointTimeline()`
    - Creates and returns a virtual sync point timeline that is strongly associated with the current channel. Associating a timeline with a single channel makes it possible to detect priority inversion and correct it.
- `SyncPoint PromiseSyncPoint(int32 timeline)`
    - Creates a future sync point and "queues" it on the given timeline to be fulfilled at a later time.
- `void QueueSyncPoint(int32 timeline)`
    - Queues a command on the associated channel to retire the oldest future sync point still promised on the given timeline.
- `void DestroyFutureSyncPointTimeline(int32 channel)`

By having sync points associated with a channel at creation time, the server side can detect priority inversion and know which channel to increase the priority of if it needs to. See “Scheduling and Priorities” for more detail.

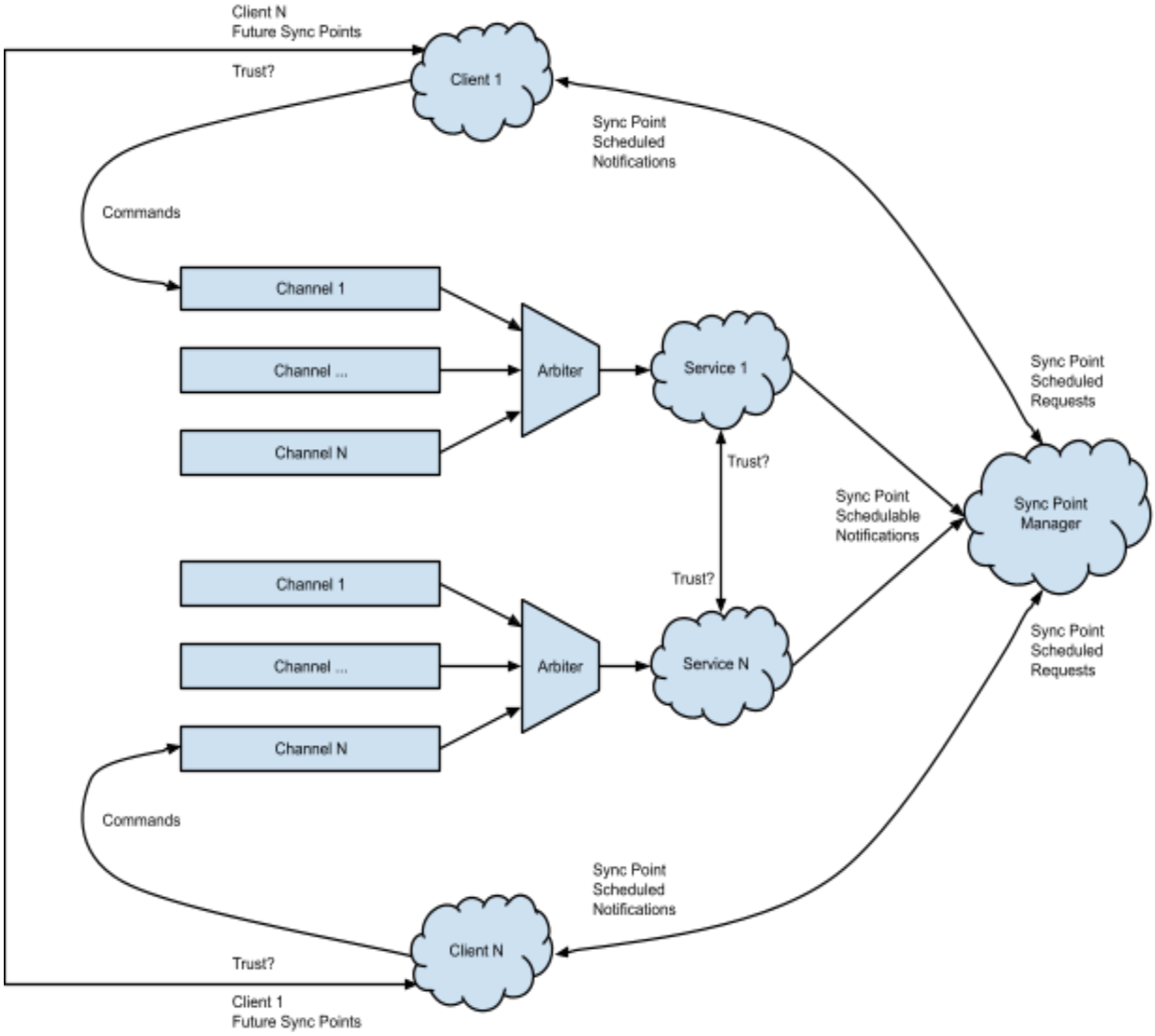
By creating a class of sync points that we know will retire in the order they are created, we can simplify implementations on certain hardware platforms that support synchronization primitives that you can wait on to be >= some value. On such platforms, a synchronization primitive would only need to be allocated once per channel creation, instead of once per sync point.

Note: Timelines are analogous to *timelines* in Android’s sync driver and *contexts* in LInux’s dma_fence driver.

Deadlock Avoidance and Trust

By providing mechanisms to wait for a sync point to enter the scheduled state, we can provide stronger guarantees against deadlock when waiting on a sync point from an untrusted source.

The following diagram illustrates how clients may or may not trust each other and services may or may not trust each other as well, which affects how to determine if a client would consider a sync point something which it can reliably service-side wait on.



Deadlock Avoidance and Trust: Wait until Schedulable

- `void SyncPointClientWaitUntilSchedulable(SyncPoint sync_point, Callback callback, SyncPoint* assume_scheduled, int assume_scheduled_size)`
    - A non-blocking client-side wait that only waits until the SyncPointManager knows that the given sync point will retire in finite time.
    - Returns earlier than SyncPointClientWait because it doesn't wait for the commands to actually finish.
    - Could be used to wait for an untrusted client to submit all its commands before inserting a server-side SyncPointWait.

    - assume_scheduled and assume_scheduled_size describe a set of sync points the call should assume will retire. This is useful, for example, when the sync points will be retired immediately after the callback returns.
    - A sync point provided in assume_scheduled for a given channel implies all sync points before it have also retired.
        - Note: This can only apply if sync points of a channel retire in FIFO order.

Deadlock Avoidance and Trust: Trusted vs Untrusted Services

A single service that manages/owns all sync points it encounters can know with certainty whether or not a sync point will retire in finite time and it is straightforward to implement SyncPointClientWaitUntilSchedulable.

Implementing SyncPointClientWaitUntilSchedulable in the context of multiple services, however, becomes more complicated if each service only trusts a subset of other services. If an untrusted service reports that a sync point is scheduled, but doesn't actually retire it, then there will be deadlock if a client waits on the sync point.

Assuming each sync point has a domain element, each service could be assigned a domain and services can specify which domains they trust with the following function:

- `void AddTrustedService(int32 domain)`
    - TBD: Should this define what services a client trusts, or should it define what services a service trusts?
    - If the service corresponding to the domain indicates that a sync point is schedulable, SyncPointClientWaitUntilSchedulable should consider it schedulable.
    - For untrusted services, SyncPointClientWaitUntilSchedulable should only consider sync points schedulable if the service has indicated the sync point is retired.

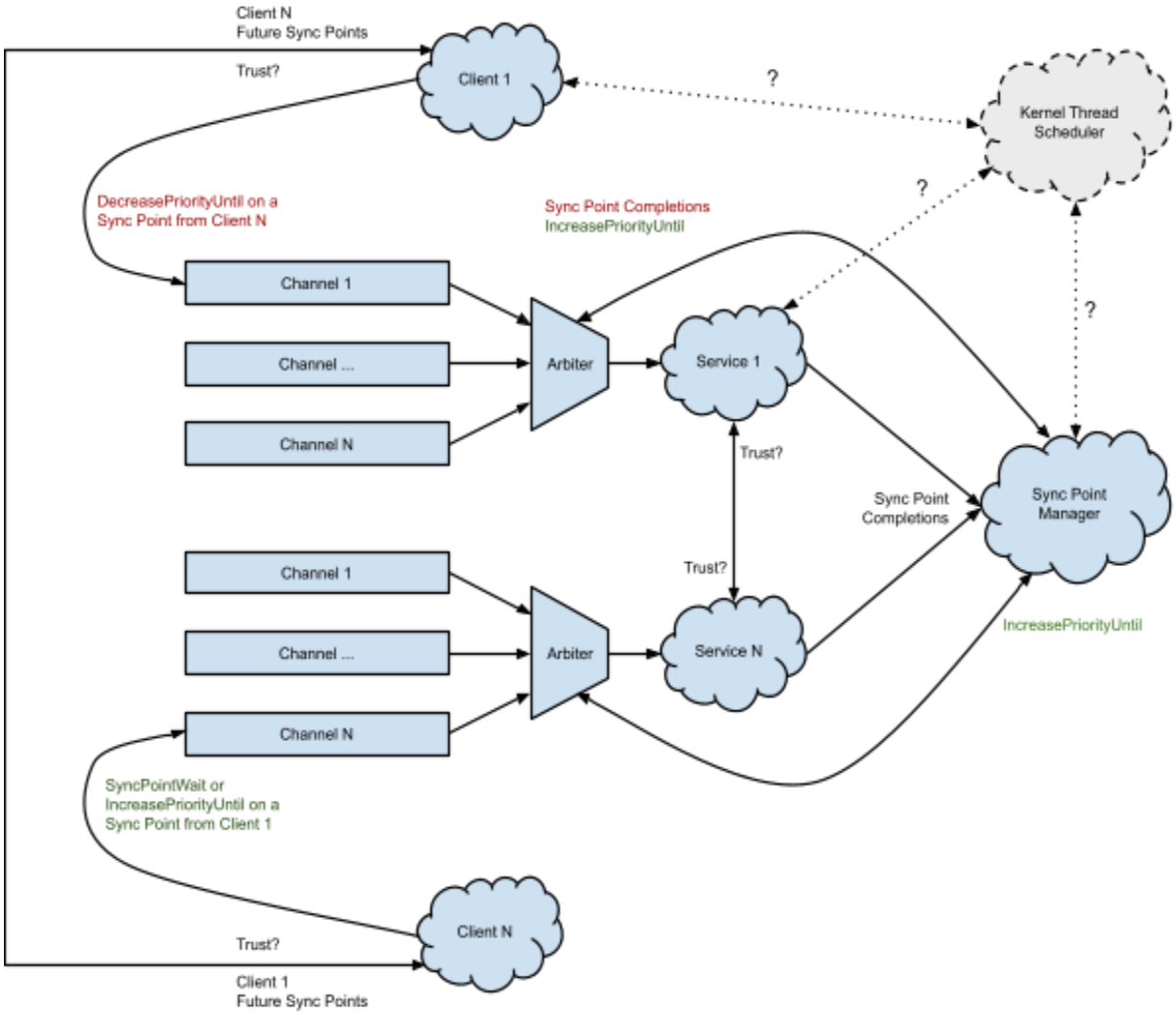
Scheduling and Priorities

There are three use cases where fine-grained channel priorities would be beneficial:

1. Avoiding priority inversion in the context of a hard dependency.
2. Avoiding priority inversion in the context of a soft dependency.
3. Ensuring a low-latency producer takes priority over a high-latency producer.

The diagram below shows how priority inversion (1 & 2) can be addressed in green and shows how a high-latency producer (3) can reduce its priority in red. The Arbiter takes into account priority changes in order to determine which channel to run or preempt.

It might help to dynamically adjust the thread priorities of the client and service if they are implemented in software, however that is left as a TBD. This document only focuses on channel priorities and does not address thread priorities.



Priority inversion in the context of a hard dependency

Detecting and avoiding priority inversion, where a high-priority channel has a SyncPointWait on a low-priority channel, does not require any additional API functions. However, it does require that sync points are strongly associated with a channel so that the SyncPointManager knows which channel to bump the priority of, given the sync point provided to SyncPointWait.

Priority inversion in the context of a soft dependency

There may be cases where a high priority queue only has a soft dependency on a low-priority queue; for example, if there are conditional code paths based on the state of a sync point. In

these cases, it could help to have a function that bumps the priority of a low-priority queue that a high-priority queue might want to wait on:

- `void IncreasePriorityUntil(SyncPoint target, int requested_priority)`
    - Queues a command on the current channel to increase the priority of the channel the target sync point belongs to up.
    - The priority increase remains valid only up until the sync point retires.
    - The priority of the target channel will become equal to the max of:
        - min(requested_priority, current_channel_priority)
        - max(requested_priority, target_channel_priority)

Additional benefits of `IncreasePriorityUntil` over relying on SynPointWait to avoid priority inverson are: 1) it can proactively bump the priority of low-priority channel in anticipation of a hard dependency and 2) it can increase the priority of multiple dependencies in parallel rather than only in serial since it is non blocking.

Decreasing the priority of a high-latency producer

If a Renderer wants to play nice with the system, it could decrease its priority once it's done with a frame, so that its subsequent frame doesn't fight for resources when other Renderers are still trying to get their first frames up.

- `SyncPoint DecreasePriorityUntil(SyncPoint dependency, int adjustment)`
    - Queues a command on the current channel to decrease the priority of the current channel by the specified amount, until the specified dependency sync point (from another channel) is reached.
    - Returns a new sync point that can be used to cancel the adjustment.
    - Decrements accumulate on top of previous decrements so that clients can automatically punish themselves more and more if they are producing faster than other clients.
- `void CancelPriorityLevelDecreases()`
    - Temporarily disable previous calls to DecreasePriorityUntil immediately/asynchronously.
    - Queues a command on the current channel to make previous calls to DecreasePriorityUntil effective again.
    - Helps avoid deadlock when a synchronous call, such as glReadPixels, must complete and the priority adjustment is implemented as a deferral.
    - Even if the priority adjustment isn't implemented as a deferral, canceling the priority adjustment can make a synchronous call return sooner.

IncreasePriorityUntil vs. DecreasePriorityUntil

From a security perspective, `DecreasePriorityUntil` prevents channels from punishing other channels since a channel can only punish itself, and `IncreasePriorityUntil` allows a queue with high-priority permissions to bump the priority of a low-priority queue that doesn't have permission to bump itself as high.

With `IncreasePriorityUntil`, many queues can increase the priority of another queue. `DecreasePriorityUntil`, however, doesn't have the ability to decrease the priority of a queue based on the state of multiple other queues, unless compound sync points exist.

Compound Sync Points

A **compound sync point** is a sync point that is a logical combination of other sync points. Compound sync points could be implemented entirely client-side, but could be made more efficient if represented server-side since it would reduce the number of interrupts, messages, or queries required to determine completion of a compound sync point.

Sync points should only ever transition in one direction from created to retired. So, to prevent unexpected transitions backwards, the only logical combinations allowed are ANDs and ORs.

- `SyncPoint PromiseCompoundSyncPoint()`
    - Separating the creation vs definition of a compound sync point allows clients to wait on the compound sync point before it's been decided what the compound sync point will look like.
    - Implementations can avoid synchronous round trips to the sync point manager by pre-allocating compound sync points.
- `void QueueCompoundSyncPointAll(SyncPoint* sync_points, int count)`
    - Defines a compound sync point that is the logical AND of all sync points provided.
    - Can be used to emulate barriers but is more versatile than barriers since the points at which you "decrement" the compound sync point count don't also have to be the points at which you wait.
- `void QueueCompoundSyncPointAny(SyncPoint* sync_points, int count)`
    - Defines a compound sync point that is the logical OR of all sync points provided.
    - Useful for to triggering updates that combine multiple results asynchronously.
- `bool QueueCompoundSyncPointExpression(char* expression, SyncPoint* sync_points, int count)`
    - Returns true on success, false on failure. Failure may occur if the expression is malformed.

- expression: A string representing the logical expression of sync points. Sync points are referenced by their zero-based index in the sync_points array. For example: “(0&1&2)|(3&4&5)” would retire when all of the first three sync points have retired or when all of the last three sync points have retired.

Examples

Use case 1: Deprioritize a high latency Renderer’s context

The Browser could create one timeline per Renderer and then make promises to each Renderer that it will consume the corresponding Renderer’s output. The Renderer could then lower its priority level (via DecreasePriorityUntil) until the Browser fulfills its promise that it will consume the output. The priority will give the GPU Service enough information to defer or de-prioritize Renderers that are operating in a high latency mode relative to other Renderers.

To prevent deadlock, clients must cancel the priority adjustment when they make synchronous calls like glReadPixels.

Use case 2: Prioritize a specific Renderer that the Browser depends on

Once the Browser is notified that a particular Renderer is scheduled, it can bump the priority of that Renderer via IncreasePriorityUntil, giving the GPU scheduler a hint about which Renderer to execute first.

Use case 3: Return a buffer to the Renderer before it’s actually available

The Browser could create one timeline per Renderer and then make promises to each Renderer that it will return a buffer for it to draw into. The Browser could then return a buffer as soon as it’s received it, but only fulfill its promise after actually consuming and subsequently releasing it.

Use case 4: Synchronize multiple nested embedders efficiently

Each embeddee makes a promise to it’s embedder that it will be ready to swap. The embedder, in turn, promises a compound sync point representing a collection of embeddees (and potentially itself) that must be swapped simultaneously. The embedder then queues a single swap containing all clients that must synchronized. That swap is triggered once the compound sync point is schedulable. The embedees must not swap themselves unless they are running asynchronously.

With all of the dependencies defined up front, arbitrarily nested embedders can run concurrently in time rather than nested in time.

Use case 5: Defer decisions to occur later in the pipeline

See this doc.

Use case 6: Allow everything to be a service

Operations like texture uploads, cache flushing, cache invalidation, swizzling, buffer allocation/deallocation, etc can be given their own services with dedicated channels for each graphics context and corresponding future sync point timelines.

A texture upload could return two future sync points, one representing a promise that reading of the source data is complete and a second one indicating that the texture is visible to the GPU.

In-place swizzles could return a future sync point representing a promise of completion.

GPU writes to a buffer could return two future sync points, one for when the GPU is complete and another for when the results are CPU visible.

Coupled with the prioritization adjustment feature, all services could intelligently arbitrate amongst the channels.

Implementation Considerations

The goal: Pure hardware implementation

TBD

The starting point: Pure software implementation

TBD

Discussion

1. **Domains:** Should SyncPoint include a “domain” element?
    a. A domain would be able to create channels and coordinate channels within its domain without communicating with another entity.
    b. Coordinating channels across domains would require a higher level of synchronization.
    c. This could be used, for example, to set up synchronization between multiple devices (physical or virtual) ahead of time.
2. **Hashing:** Should we make the series of sync points hard to guess using hashes?
    a. Hash just the id?
    b. Hash the whole field?
    c. Are SyncPointClientWaitUntilTrustable and SyncPointClientWaitUntilCompelete enough to guard against all negative outcomes of a bad client guessing another channel’s sync points?

3. **Async memory management:** Separate future sync point channels seem like a good way to formalize scheduling and notification of asynchronous completion steps such as cache maintenance or other types of memory cleanup.
    a. Should we use future sync point channels for such cases?
    b. For example, this would allow a client to differentiate between all of the following:
        i. When GPU result are visible to the CPU.
        ii. When CPU results are visible to the GPU.
        iii. When it's okay to queue new GPU commands that depend on a previous GPU result. Can be earlier than (i).
    c. Note: Even cache coherent systems might only be coherent in one direction (GPU can snoop CPU, but CPU can't snoop GPU), which ARM calls I/O coherent. Some fun reading. Also, some Intel systems are only coherent with the CPU's L3 cache [citation needed].