State of GPU Scheduling

sunnyps@

The aim of this document is to describe how scheduling in the GPU service works currently and capture all essential aspects that need to be retained in any changes made to GPU scheduling.

Flush Ordering of Contexts

Contexts on the same GPUChannel (which means same process at the moment) are ordered with the OrderingBarrierCHROMIUM, ShallowFlush and Flush commands. This is a consequence of the contexts sharing the same IPC channel and the GPU service processing all incoming IPC messages in order (with a very few exceptions). All of these update the put offset of the command buffer in different ways.

The underlying primitive for sending the put offset to the GPU service is called AsyncFlush. It uses an asynchronous IPC and gets intercepted by GpuCommandBufferStub on the service side. The commands may start running on the GPU process main thread if the GPU service is idle and the channel is not blocked for some other reason. The order in which contexts (belonging to the same GPUChannel) run on the GPU service depends on the order in which the AsyncFlush IPCs were sent.

OrderingBarrierCHROMIUM just updates the put offset on the client side and performs an AsyncFlush only when another command buffer does an AsyncFlush or when the command buffer is full.

ShallowFlush updates the put offset and performs an AsyncFlush immediately.

Flush is like ShallowFlush but also adds a glFlush command into the command buffer before updating the put offset.

Sync Points

Sync points are the primary synchronization primitive across channels. The way sync points are used is by calling InsertSyncPoint in one command buffer and adding a WaitSyncPoint command (with the return value of InsertSyncPoint) in another command buffer. This ensures that the latter command buffer does not execute past the WaitSyncPoint command until the former command buffer has executed past the InsertSyncPoint command.

The above properties and flush based ordering imply the following:

1. As soon as InsertSyncPoint returns a sync point identifier, any context can wait for that sync point immediately. This is because InsertSyncPoint uses (necessarily) a synchronous IPC.
2. Waiting on a sync point in a context blocks all work on the channel that context belongs to (except for synchronous IPC messages handled in GPUChannelMessageFilter) - the associated command buffer stub becomes “descheduled”. This is a consequence of flush based ordering.
3. Waiting on a non-existent sync point is always safe - the wait becomes a no-op.

Client Side Waits

There are a few ways for the client to block on execution of commands on the GPU service. These include:

1. InsertToken which allows the client to insert a token into the command buffer that can be waited on with WaitForToken/WaitForTokenInRange.
2. WaitForGetOffsetInRange which allows the client to wait until the get offset (which is the point until which the service has read) reached a point.

Command Buffer Descheduling

On the service side a command buffer can be descheduled if:

1. The command buffer is waiting for a sync point.
2. A command returns the kDeferCommandUntilLater error when run. The command is expected to deschedule the command buffer if it returns this error.

The command buffer can be preempted if its channel is preempted by another channel. In this case the command buffer isn’t considered to be descheduled.

In both the descheduled and the preempted states the command buffer doesn’t process any commands.

A command buffer can be scheduled again if:

1. The sync point the command buffer is waiting for is signalled.
2. The command that was deferred reschedules the command buffer.

Channel Preemption

Channel preemption exists to prevent a renderer context (say WebGL) from preventing a browser context from running. The channel preemption logic is shared between four classes: GPUChannel, GPUChannelMessageFilter, GPUCommandBufferStub and GPUScheduler.

Channels have two flags - a preempting flag and a preemption flag. The channel owns its preempting flag. A channel's preemption flag may be set to another channel's preempting flag which allows the latter channel to preempt the former channel. When a channel is said to be preempting it sets its preempting flag which allows it to preempt other channels.

Preempting other channels only makes sense if there are pending IPC messages to process in the channel - transition from IDLE to WAITING in the state diagram below.

Channels are not allowed to preempt other channels too often - a channel can only preempt other channels once every two frame intervals (kPreemptWaitTime) - transition from state WAITING to CHECKING.

When a channel preempts another channel it must be to service an IPC message from the client that's old - at least two frame intervals old (kPreemptWaitTime) - transitions from CHECKING to PREEMPTING/WOULD-PREEMPT-DESCHEDULED.

If any command buffer of the channel is descheduled (which implies it can't process any IPC messages) it itself is prevented from preempting other channels - transitions from CHECKING/PREEMPTING to WOULD-PREEMPT-DESCHEDULED.

If a channel doesn't have a message that's old enough (kStopPreemptThreshold) it stops preempting other channels - transitions from PREEMPTING/WOULD-PREEMPT-DESCHEDULED to IDLE.

State Diagram

kPreemptWaitTime = 2 x VSync

kMaxPreemptTime = VSync

kStopPreemptThreshold = VSync