boliu@

tl;dr
Preparing to make Android WebVIew multi-process, allow the UI thread to send synchronous IPC messages to renderer compositor thread to maintain synchronous requirements. Rough but mostly working prototype CL here.

Background

Android WebView inherited a lot of synchronous rendering requirements due to being an Android View. Many existing apps rely on this synchronous behavior, so breaking this behavior is not an option.

High level synchronous calls in Android WebView

- Root layer scroll state (scroll offset, zoom level, bounds, etc). Android Views have APIs like scrollTo that are expected to be synchronous on the UI thread.
- Input events that are handled directly on the compositor thread. These are usually scroll or zoom events that updates the root layer scroll state.
- Draw. It's a synchronous call on the UI thread.

Webview currently achieves synchronous behavior by being in single-process and merging the UI thread and the renderer compositor thread. This way UI thread can directly access and manipulate state that's normally in the renderer compositor. This code currently lives in content/browser/android/in_process

Solution for multi-process

Split UI and renderer compositor threads across processes like chrome. Allow synchronous IPC from UI thread to renderer compositor thread. Synchronize any state that's synchronous in every sync IPC and reply.

Renderer to browser direct call can be partially simulated with the renderer response to sync IPCs. For example, scroll state in browser must be updated synchronously in response to input event, which can be done by sending the new scroll state in the response of the input sync IPC. This covers the majority of renderer to browser calls that can break compatibility. The hope is everything else is not important for compatibility and can be asynchronous.

A working prototype of this here. The CL is not meant for low level scrutiny, though the messages header may be worth a glance.

Sync IPCs are BAD!

Yes. Doubly bad for browser to renderer. Triply bad for blocking UI thread. But the security benefit of sandboxing the renderer process is worth these downsides.

Some early alternative ideas considered are here. They all involved some kind of UI blocking behavior, and are much more invasive to the chromium code base.

Avoiding deadlocks and long term burden

The existing thread blocking restrictions are represented in the table below.

IO threads
GPU thread, Android Render Thread (webview only, Android's GPU thread)
UI/renderer compositor thread (same thread)
Renderer main thread (Blink runs on this thread)

A thread can only block on threads in cells above it. This way there are no blocking cycles that can cause a deadlock.

With the proposed change, renderer compositor thread is lifted up to its own level. UI is allowed to block on renderer compositor, and renderer compositor thread is restricted from blocking on UI thread.

IO threads
GPU thread, Android Render Thread (webview only, Android's GPU thread)
renderer compositor thread
UI thread
Renderer main thread (Blink runs on this thread)

Compositor thread restricted from blocking on UI thread is an on-going requirement on the entire chromium community: with merged-thread, chrome can add a sync IPC with merged threads, which can be re-implemented as a direct function call in webview without much difficulty; this approach will not be possible with split threads.

Note synchronous IPC from renderer main thread to browser process is still allowed. Renderer compositor thread never blocks on main thread so there is no wait cycle. And synchronous IPCs only block the sending thread, not the IO threads.

Hopefully this should not be a big issue, since merge-thread already requires that there can be no synchronous IPC between UI and compositor logical threads (in code used by webview), because they are the same physical thread.

Results from prototype

Everything worked as expected. Did not find any major compatibility issues from trying a few google apps (gmail, search) that rely heavily on synchronous behavior.

From the overhead of a sync IPC round trip, is on the order of 0.5ms on nexus 5; tested by looking at chrome trace of a no-op sync IPC while idle. There are up to 4 synchronous IPCs for each frame: input, vsync, animate fling scroll, draw. User visible performance is on-par with merged threads because there is enough headroom on the UI thread. Lots of potential for optimizations in the future, but aim to keep the same existing calls for now to avoid too many things changing at once.

Other technical notes (seeking advice/comment from eng elders)

Browser sender and renderer receiver have different lifetimes. Need to make sure that browser side never deadlock due to an unhandled sync IPC in renderer. In the prototype, renderer side sends a failure reply to any unhandled synchronous IPC.

SyncChannel takes a shutdown event. On renderer side, similar event is signaled on the main thread. Not sure how this work when main/UI thread is doing the blocking in this case. Ignored this problem in prototype since Android never does a clean shutdown.

Messages from renderer can arrive on UI out of order. A reply to a synchronous IPC can be handled on the UI thread before an asynchronous IPC from renderer. In prototype, handled this by adding a version number to renderer state, and old state is ignored.