Scheduler testing using synthetic delays

skyostil@, brianderson@

Overview

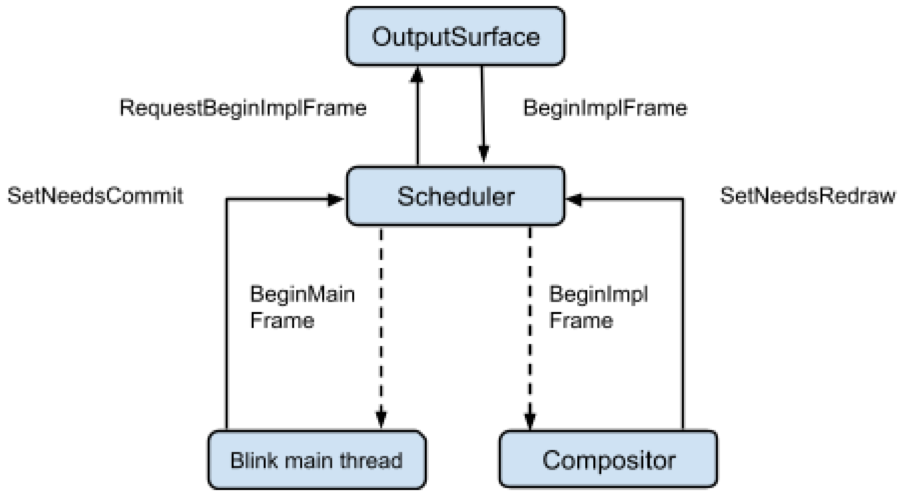
This document describes a way for testing the effect of Chrome's compositor scheduler on input latency, frame rate and animation smoothness under various interaction scenarios. The goal is to measure these metrics with synthetic tests that mimic simple and more complex web content. This test suite will serve to guide optimization work and guard against regressions.

Scheduler

The scheduler is a component in Chrome's compositor that decides what action the compositor should perform at any given time. These actions range from mundane things such as making sure the output surface for the compositor is initialized to more time-critical operations like drawing a new graphics frame.

Often the scheduler is only being asked to do a single thing at a time. In this case the choice for the next action is obvious. However, when there are multiple pending operations, the scheduler has more freedom to choose actions in a way that minimizes input latency or ensures animation smoothness.

One example of such a multiple choice situation is the handling of the BeginFrame signal:



This signal is used to drive rendering and can be requested and consumed by several entities – most notably the main Blink thread and the compositor itself. Each of the consumers can use this signal to trigger the drawing of a new graphics frame at some point in the near future.

The basic problem for the scheduler is to decide whether to let one entity produce the graphics frame greedily or to wait until all parties have had a chance to contribute to the new frame. This algorithm is called deadline scheduling – named after the time limit imposed by the display refresh interval – and is guided by factors such as the time left until the next display refresh, the estimated drawing times of each subsystem and the type of the active touch gesture.

The aim of the testing framework described below is to exercise different varieties of the BeginFrame signaling and measure the resulting performance of the browser.

Synthetic delays

In order to simulate different loads on various parts of the system, we introduce a generic synthetic delay framework. First, the code is instrumented with named delay points, e.g.

```
TRACE_EVENT_SYNTHETIC_DELAY(“cc.BeginMainFrame”);
```

Asynchronous delay points that span multiple scopes can be defined in two parts:

```
TRACE_EVENT_SYNTHETIC_DELAY_BEGIN(“cc.RasterPendingTree”);
...
TRACE_EVENT_SYNTHETIC_DELAY_END(“cc.RasterPendingTree”);
```

Here `BEGIN` establishes the start time for the delay and `END` executes the delay based on the remaining time. `BEGIN` may be called one or multiple times before `END`. Only the first call will have an effect. If `BEGIN` hasn’t been called since the last call to `END`, `END` will be a no-op. These semantics are designed to make it easy to instrument delays into multi-step processes such as texture uploads where we want an entire burst of activity to take a specific time instead of each individual step.

It is also possible to run several instances of the same delay in parallel:

```
TraceEventSyntheticDelay::BeginParallel(base::TimeTicks* out_end_time);
TraceEventSyntheticDelay::EndParallel(base::TimeTicks end_time);
```

Here the delay end point calculated by `BeginParallel()` is stored by the client and passed to `EndParallel()`.

Each delay point can be configured to four different modes:

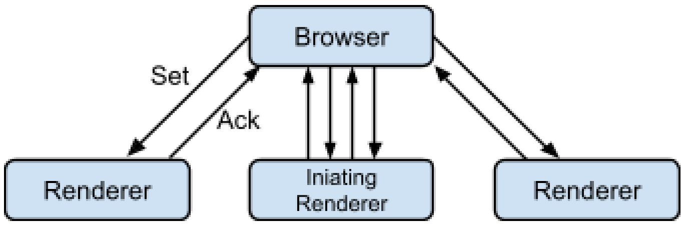
- **No-op** (default)
- **Delay to T ms** – uses a busy loop to ensure that the execution of the scope where the delay point is defined takes at least T ms.

- **One shot delay to T ms** – same as above but resets the delay to zero after one iteration.
- **Alternating between zero and T ms** – every second iteration uses the configured delay, every other one adds no delay.

For a given benchmark, synthetic delays are configured using directives in the Telemetry page set. For example:

```
      "url": "file://tough_scheduling_cases/simple_text_page.html?medium_layers",
      "why": "Simulate a page with a a few graphics layers",
      "synthetic_delays": {
        "cc.BeginMainFrame": { "target_duration": 0.004 },
        "cc.DrawAndSwap": { "target_duration": 0.004 },
        "gpu.SwapBuffers": { "target_duration": 0.004 }
      }
    },
```

When running the benchmark, Telemetry serializes the given delay configuration and sends it to the browser, which in turn proceeds to broadcast the configuration to all child processes. Once all the recipients have acknowledged, the browser sends back a notification to the originating renderer, causing the configuration callback to get called and the test to proceed.

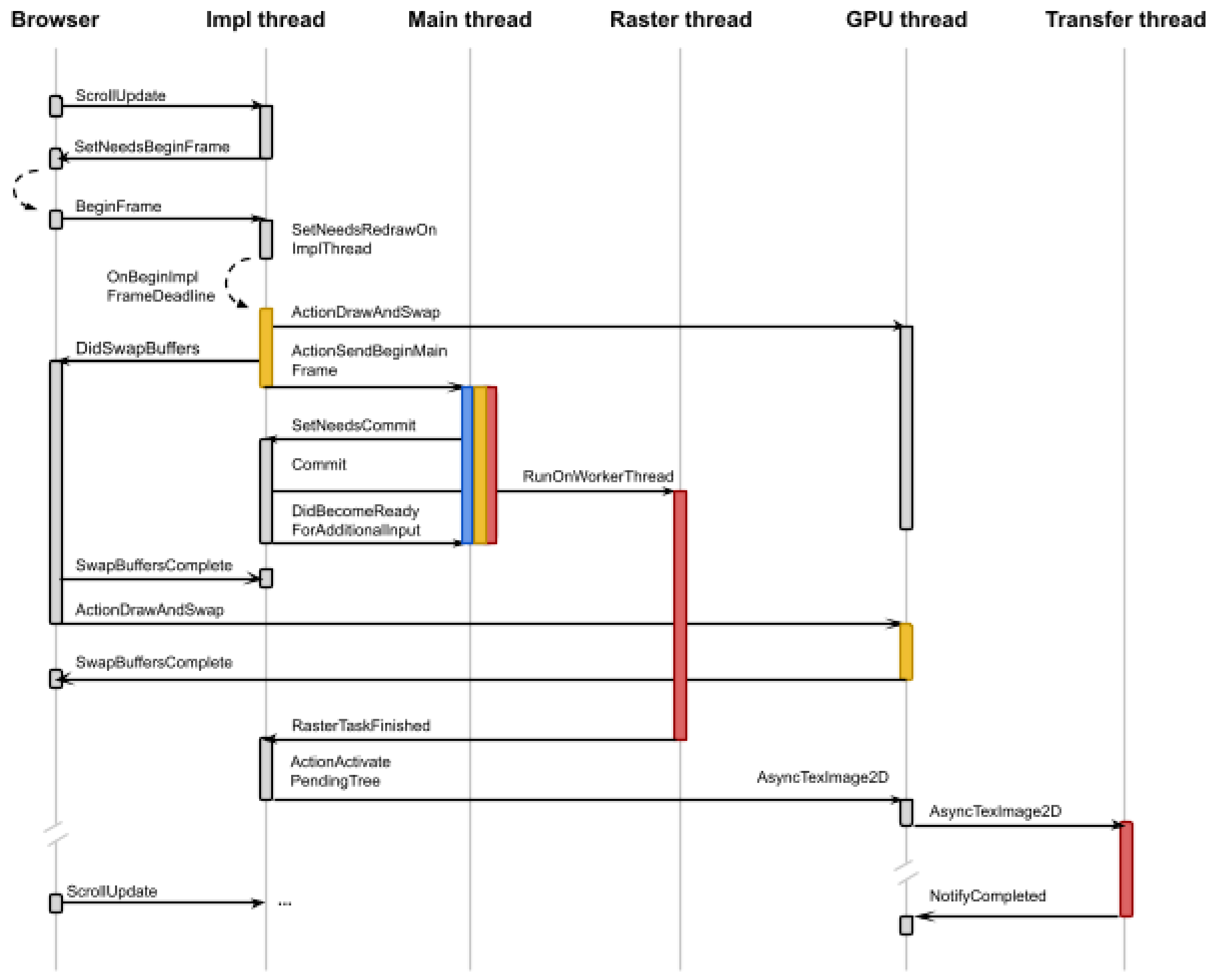


Test cases

This section describes the main scheduler test scenarios and their different variants. Note that the sequence diagrams show the roughly ideal ordering of operations for optimal latency. If, for instance, the renderer or browser schedulers have entered high latency mode, the actual order of events will be different.

The telemetry page set for these tests is defined in tough_scheduling_cases.json. The names in brackets indicate the corresponding test page URL.

1. Scrolling a basic page [simple_text_page.html]



This page represents scrolling a simple page with no special touch handlers or DOM manipulation.

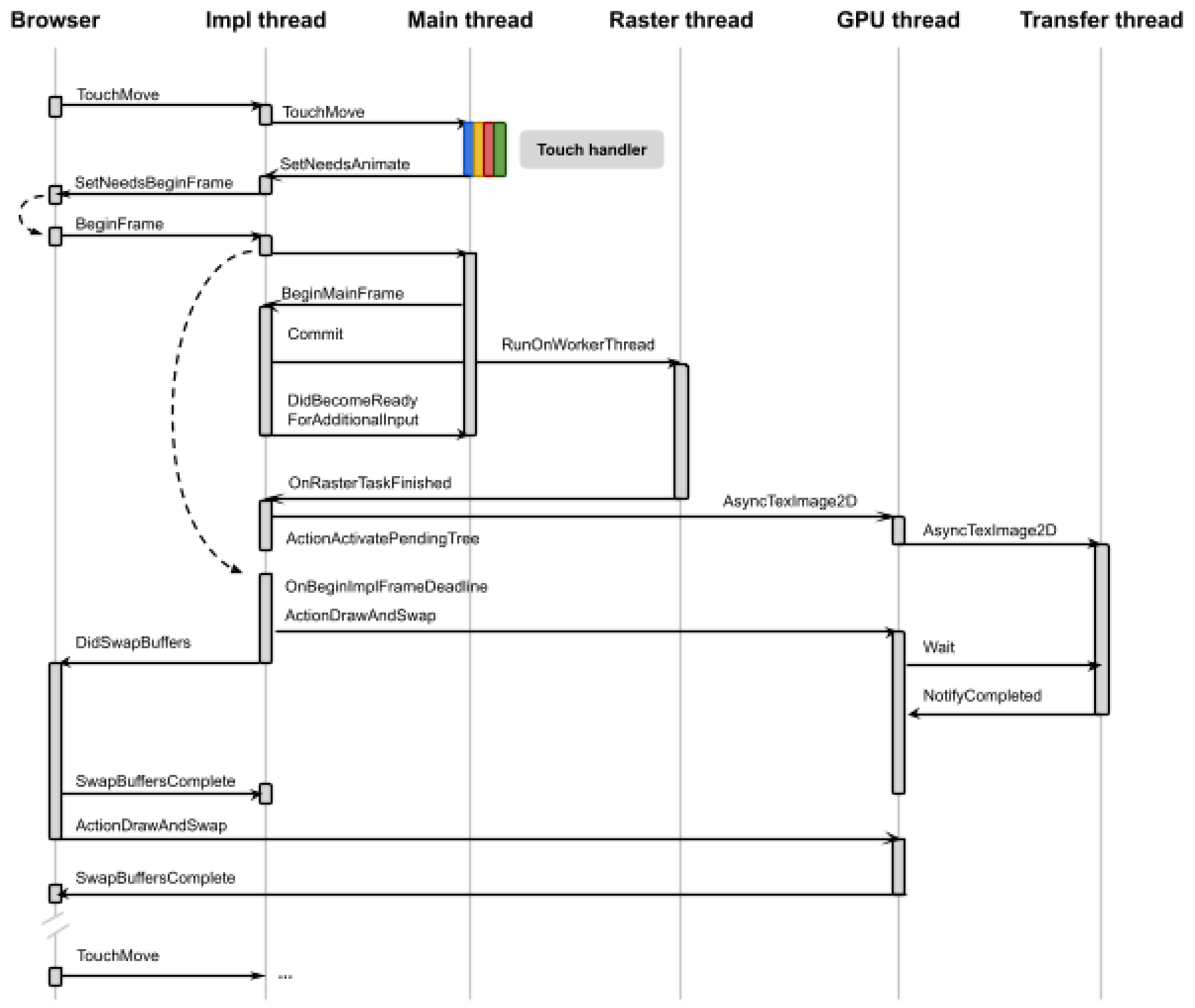
Variants

a. ⬛ Test differing amounts of main thread delay [main_busy, main_very_busy]
   - Simulate the main thread being oversubscribed by delaying BeginMainFrame to 8 ms and 24 ms.

b. ⬛ Lots of layers [medium_layers, many_layers]
   - Simulate a more complex page by delaying BeginMainFrame, compositor drawing and the swap ack all to 4 ms and 12 ms.

c. ⬛ Expensive recording and rasterization [medium_raster, heavy_raster]
   - Simulate a page with complex contents by delaying BeginMainFrame, raster tasks tree activation and texture uploads to 4 and 24 ms.

2. Touch handler driven scrolling [touch_handler_scrolling.html]



This page represents the case where a JavaScript handler intercepts all touch events and implements custom scrolling interaction by manipulating the DOM.

Variants:

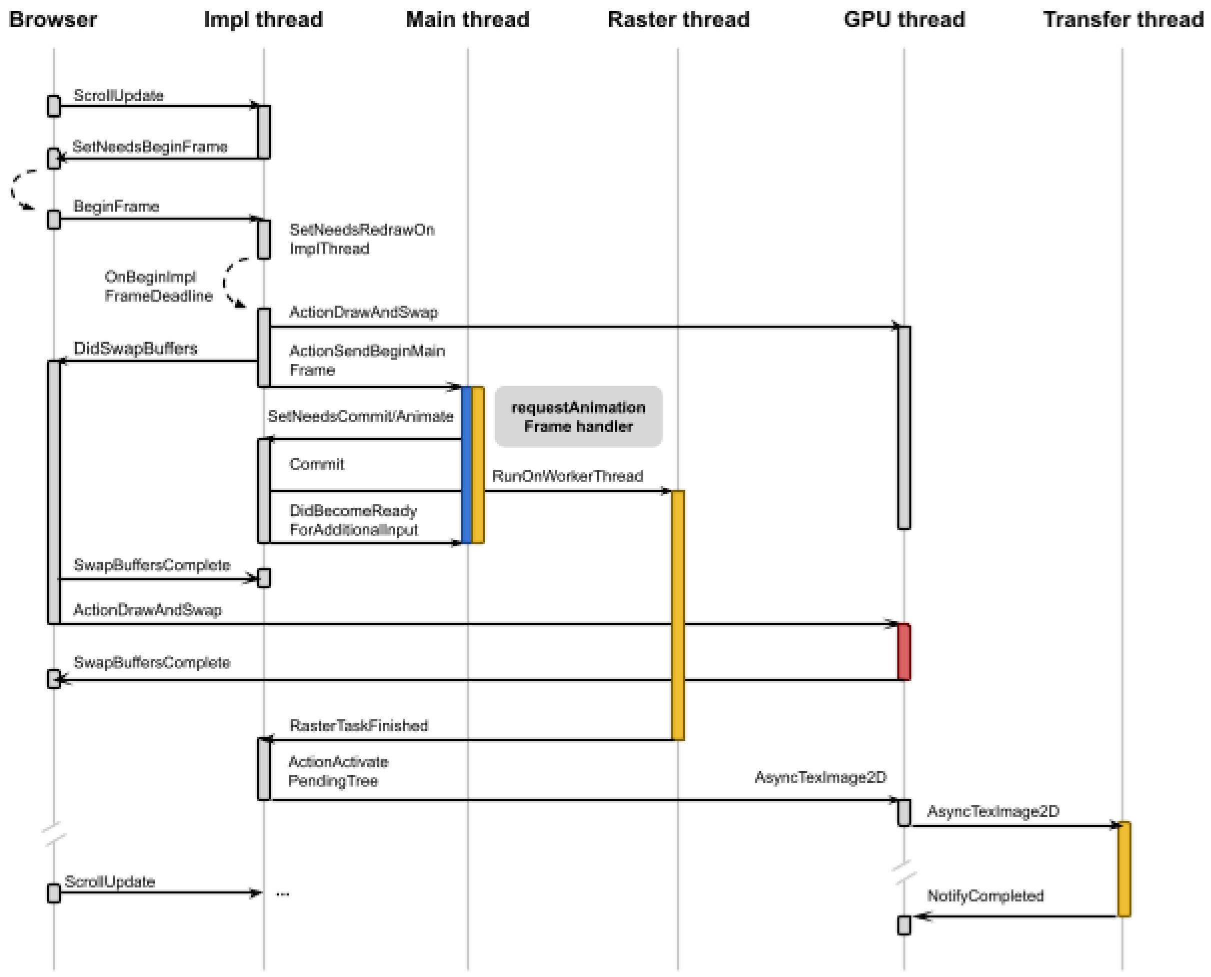
a. ⬛ Expensive touch handler [medium_handler, slow_handler]
   - Delay touch handler to 8 or 24 ms.

b. ⬛ Consistently janky touch handler [janky_handler]
   - Main thread driven scrolling where the touch handler is delayed to 24 ms every other frame. Test if we can hit 60 fps or throttle to 30. Also see if each frame gets a consistent interval of input.

c. ⬛ Occasionally janky touch handler [occasionally_janky_handler]
   - Delay touch handler once to 24 ms. Test entering and recovering from high latency modes.

d. ⬛ Super expensive touch handler [super_slow_handler]
   - Delay touch handler to 200 ms. Test browser scrolling timeout mechanism.

3. Scrolling a page with a requestAnimationFrame handler [raf.html]

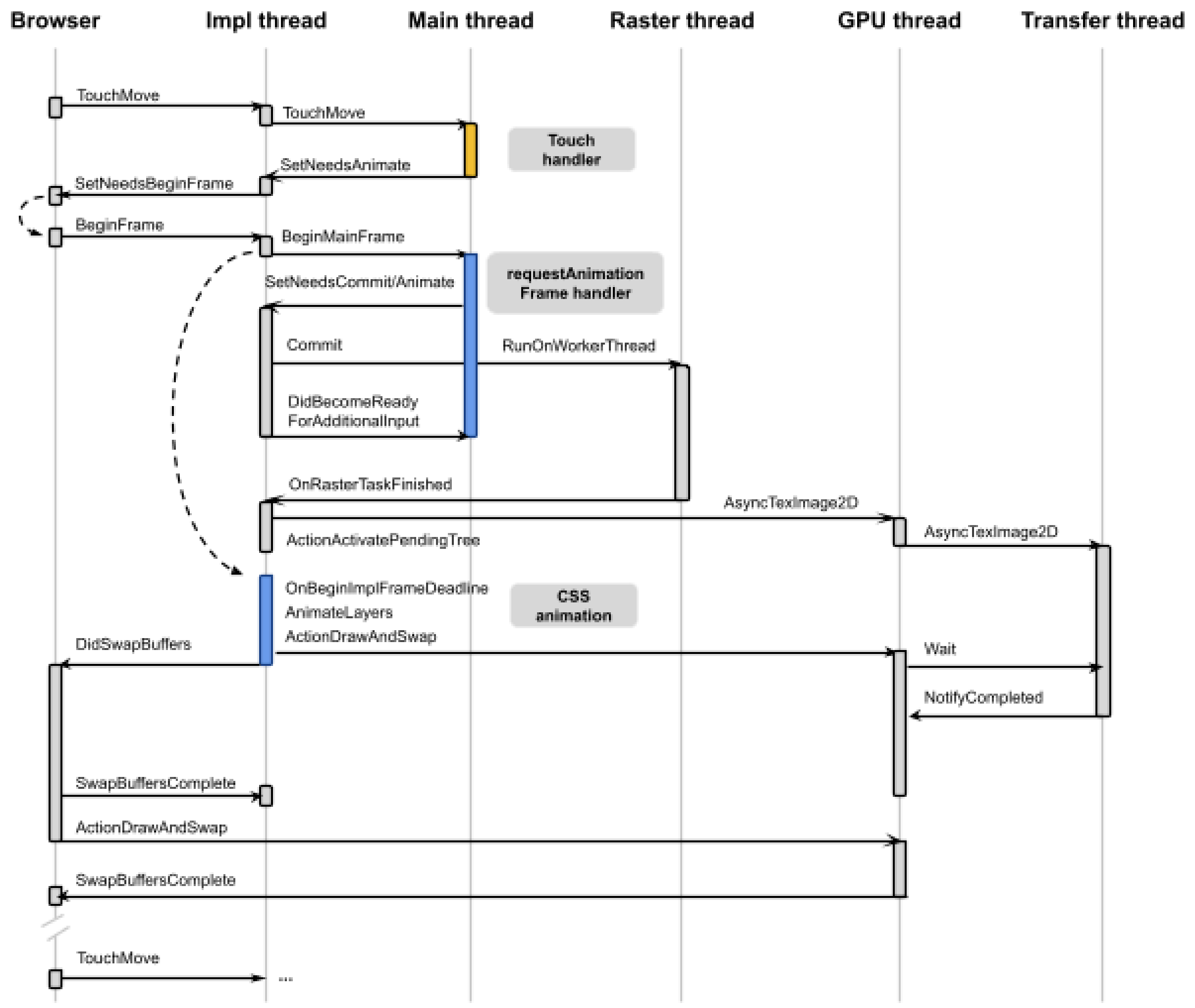


In this test the page has a constantly running requestAnimationFrame handler that modifies the DOM and the page is also being scrolled. This means that there are two sources of graphics frames – the DOM updates and the scroll gesture – and the scheduler must load balance between both.

Variants

a. ⬛ requestAnimationFrame with canvas rendering [raf_canvas.html]
   - Test canvas blocking behavior
b. ⬛ Expensive requestAnimationFrame handler [medium_handler, heavy_handler]
   - Test more complex content by delaying the requestAnimationFrame handler (BeginMainFrame), raster task tree activation and texture uploads to 4 ms and 24 ms.
c. ⬛ Heavily GPU-bound page [gpu_bound]
   - Simulate a heavily GPU-bound page (e.g., MapsGL) by delaying swap acks to 100 ms.

4. Scroll page with touch handler, requestAnimationFrame and CSS animation [raf_touch_animation.html]



This test is designed to stress the deadline estimation mechanism of the scheduler. The scheduler needs to deal with three different frame producers: touch handler DOM modifications, requestAnimationFrame DOM modifications and a CSS animation.

Variants

a. ⬛ Heavy page version [medium, heavy]
   - Simulate a more complex page by delaying BeginMainFrame and compositor drawing to 4 ms and 12 ms.

b. ⬛ DIV touch handler [div_touch_handler.html]
   - Scroll a page with a super expensive touch handler that only occupies a part of the page. Makes sure the compositor performs hit testing before invoking the handler.

Future work

- Introduce latency tests for browser compositor corner cases.
- Test latency of non-BeginFrame scenarios (e.g. GPU context loss).

References

- Tracking bug for synthetic delays: https://code.google.com/p/chromium/issues/detail?id=307841
- Synthetic delay framework: https://codereview.chromium.org/53923005
- Tough scheduler test cases: https://codereview.chromium.org/68203031