Browser IO thread scheduling for input latency

PUBLIC DOCUMENT

Authors: eseckler@
Reviewed by: skyostil@
Last Updated: 2018-08-23
Tracking bug: https://crbug.com/861708
Design review: https://crbug.com/870325

[Update 2018-08-23]

We've reconsidered our proposal to add a scheduler on the Browser IO thread in light of the improvements that the network and viz services will bring.

When enabling the network service, the congestion on the IO thread is reduced, particularly on Android. With viz, most of the critical path for input/compositing also no longer flows through the browser IO thread. With both features enabled, we could not reproduce situations similar to those mentioned below.

For the time being, we'll pause our efforts to add scheduling of tasks on the IO thread. Instead, we will focus our efforts on scheduling UI thread tasks. We also plan to collect more metrics for tasks executing on the IO threads - in the browser, renderers, network service, and elsewhere - to inform any future decisions around scheduling on IO threads.

Details:

With viz, renderers receive vsync signals and submit frames directly from/to the display service, which runs in the GPU process and thus uses its own IO thread (even when in-process, there is a separate IO thread). The only important compositing messages that still route through the browser's IO thread are vsync messages to the browser, which are used to start browser-side surface updates, such as updates to the omnibox/bar or changes to a window or frame size.

On android (which this proposal focuses on), the frame size changes only for onscreen keyboard and omnibar hiding during scroll. These use cases are already more limited than originally anticipated. Furthermore, they are much more likely to be affected by congestion on the browser UI thread (where e.g. vsync events from the OS are received and actual browser compositing work happens) than on the IO thread.

The vsync messages mentioned above are received on the IO thread inside a file descriptor event handler (libevent / IO completion) and immediately passed on to the UI thread. Since these file descriptor event handlers already have a high priority (on posix, all active libevent handlers execute in between each MessageLoop task), the vsync messages would only benefit from prioritization between different file descriptors (e.g. to prioritize the file descriptor of the mojo connection to the GPU process).

With the network service, the congestion on the browser IO thread is further reduced significantly. In particular, socket file descriptors move out of the thread. When enabling the network service in local tests,

we did no longer observe long queuing delays on the IO thread which would affect the vsync messages. Only when enabling out-of-process iframes, did we ever see small congestions.

We further implemented a prototype to prioritize the file descriptor of the mojo connection to the GPU process (making this production worthy would require considerable changes to mojo internals). We then verified its impact using system health benchmarks on perf bots (Nexus 5). We compared configurations with network service, viz, and out-of-process iframes enabled and disabled. None of the benchmarks showed a significant change in input latency metrics. While this result is not conclusive, it's an indication that the browser IO thread is not currently the bottleneck for input latency.

Because of the above, we decided to put IO thread scheduling on hold and focus on scheduling UI thread tasks for the time being. We will also collect more metrics from the field to get a better understanding of the queuing delays potentially introduced on the IO thread.

[ORIGINAL OBSOLETE PROPOSAL]

One-page overview

Summary

99%ile Input latency is quite terrible, particularly on low-end Android devices (>1s for ScrollBegin). High input latency is often caused or exacerbated by unrelated tasks that delay the execution of input and compositing tasks, particularly on the browser's IO and UI threads.

We propose to add a SequenceManager-based scheduler to the browser's IO thread to avoid running low-priority tasks while high-priority tasks (such as input-related tasks) are pending. Furthermore, we plan to add prioritization to the execution of different file descriptor I/O event handlers, to avoid delays caused by concurrent low-priority socket operations.

In the future, we intend to use this scheduler to prioritize tasks not only based on task type, but also based on a task's associated frame and context, for example, to prioritize network fetches for the current foreground tab or main frame over background tabs or out-of-process iframes.

Related to this proposal, we are also planning to introduce scheduling on the UI thread.

Platforms

All, focused on Android.

Team

scheduler-dev@chromium.org

Bug

https://crbug.com/861708

Code affected

Some code under //base, especially related to tasks, base::Thread, base::MessageLoop/Pump.
Call sites that post tasks to the browser's IO thread will be gradually annotated with priority-defining attributes (or redirected to corresponding TaskRunners), such as task types and associated frame IDs.
Initially, we will add such annotations for high-priority IPC/mojo tasks, input tasks, and I/O events.

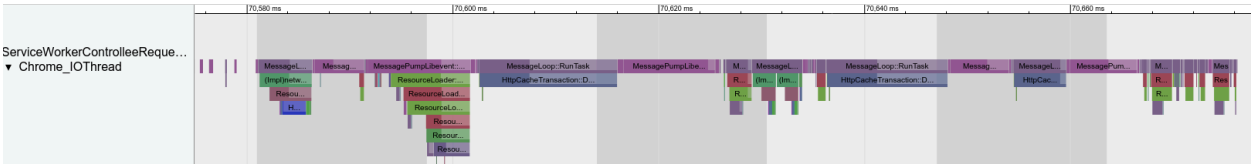
Browser IO thread congestion - motivating examples

We've collected a few example traces from an Android Go device that show how unfortunate scheduling of IO thread tasks can lead to frame drop.

Congestion on the IO thread has two primary effects:

1) On platforms that route input events from the OS through the IO thread (primarily desktop), the input event may be received late.
2) On all platforms, mojo IPCs to any browser thread and legacy IPCs to and from the browser threads are delayed.

Example 1: Unrelated tasks, e.g. resource loading, cache access & maintenance

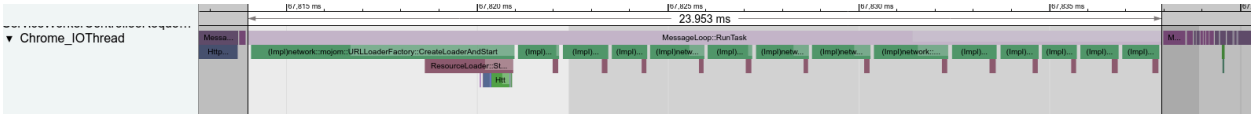


Generally, any concurrent tasks can delay the execution of compositing/input-related tasks, since tasks are executed in FIFO order. In the above example, multiple HTTP cache transactions and resource loads occur concurrently to a scroll and are causing congestion. Furthermore, the cache tasks often take >10ms, thus could cause a frame miss even if the thread wasn't as congested as in the example above.

Example 2: Socket I/O event handlers

This trace shows how a legacy IPC to the GPU process is delayed for > 40ms by multiple consecutive socket reads. It highlights a problematic property of MessagePumpLibevent: In each iteration of the message loop, the pump executes the event handlers for all signalled file descriptors in FIFO order. In the trace above, multiple sockets have data available to read, thus all their handlers are executed before processing any other tasks.

Example 3: Long resource loader tasks



Similar to the last example, when a mojo handle has new data available, a task is posted which reads and executes all available messages from the handle. In the case of resource loader messages, this task executes on the IO thread, and can delay execution of other tasks for a long time (in this case, for more than 20ms).

Design

We suggest the following changes to address the above issues:

1. Introduce prioritization for tasks executing on the IO thread's MessageLoop, by posting tasks to different TaskQueues managed by a SequenceManager and prioritized by a new BrowserIOThreadScheduler.
2. Introduce prioritization for I/O events on file descriptors and delay low-priority event handlers for high-priority events or MessageLoop tasks. We can accomplish this by running file descriptor event handlers as separate (prioritizable) tasks on the MessageLoop.
3. Splitting mojo message handler tasks on the IO thread into multiple smaller tasks, by posting a separate task for each message.

MessageLoop task prioritization

Introducing a SequenceManager and BrowserIOThreadScheduler on the IO thread

Prototype patch: https://chromium-review.googlesource.com/c/chromium/src/+/1140330

Adding a SequenceManager on the IO thread will allow us to assign tasks posted to it onto different TaskQueues, which can be prioritized independently. For example, network-related tasks may be scheduled on one queue and input- or compositing-related tasks on different ones.

The BrowserIOThreadScheduler can then dynamically change the priorities of these queues, so that e.g. compositing tasks gain a higher priority during scrolling. Initially, we intend to introduce the following queues:

- **IO:** I/O event handlers (highest priority)
- **Input**: Input tasks (high priority)
- **GPU**: Compositing and GPU tasks (high priority)
- **Network**: Network tasks (TBD: default priority)
- **Idle**: Background/maintenance tasks (best effort priority)
- **Default**: Other tasks (default priority)

We will add prioritization to these queues gradually and gated by finch experiments. We will start with preserving the current implicit task order by applying a simple FIFO scheduling policy over all queues except the IO queue and prioritizing the IO queue over all other queues. We then plan to run experiments

to add static prioritization amongst the other queues and later add dynamic prioritization based on detected use cases.

Eventually, we plan to add more task queues, e.g. to separate tasks associated with different tabs or frames and I/O event handlers associated with different file descriptors.

Since the UI thread posts tasks to the IO thread before the latter has started, we will initialize the SequenceManager and BrowserIOThreadScheduler on the UI thread before startup of the IO thread. This way, we can override the IO thread's task runner with the scheduler's default TaskQueue before any tasks are posted to the IO thread.

Posting tasks to the IO thread's TaskQueues

Currently, developers use static BrowserThread::PostTask() methods and TaskRunners obtained from GetTaskRunnerForThread() to post tasks to the IO thread. We intend to replace this API with the base::PostTaskWithTraits()/GetTaskRunnerWithTraits() API, by introducing new content-layer-specific TaskTraits. This will allow annotating tasks with the target thread, task types, frame identifier, and other attributes, which the BrowserIOThreadScheduler can use to assign each task to the correct task queue. See this document for more details. Example:

```
base::PostTaskWithTraits(FROM_HERE,
                         {BrowserThread::IO, TaskType::kNetwork, ...},
                         base::Bind(...));
```

In the short term, while we get this new API ready, we will support obtaining a SingleThreadTaskRunner for a specific TaskQueue from the BrowserIOThreadScheduler. This TaskRunner can be used instead of BrowserThread::PostTask or IO-thread task runners to post a task to a specific queue. Example:

```
scoped_refptr<base::SingleThreadTaskRunner> task_runner =
    BrowserIOThreadScheduler::GetInstance()::GetTaskQueue(QueueType::kNetwork);
```

I/O event prioritization

Prototype patch: https://chromium-review.googlesource.com/c/chromium/src/+/1149366

Currently, I/O event handlers are executed by the platform's MessagePump. Their execution is interleaved with tasks running on the MessageLoop. On Posix (Android/Linux/Mac), MessagePumpLibevent executes all event handlers with active file descriptors before running the next MessageLoop task. On Windows, MessagePumpForIO (which uses an I/O completion port) executes one I/O handler before each MessageLoop task.

To add prioritization to these event handlers, we propose to run each event handler in a separate MessageLoop task. When a file descriptor activates, a new task that will execute the handler is posted to a TaskQueue, and new events on the file descriptor disabled until the handler runs. This way, high-priority MessageLoop tasks can execute before the event handlers. By making the TaskQueue handler-specific, we can also introduce prioritization between different (kinds of) handlers.

Note that this may introduce a small computation overhead by involving PostTask machinery for each event. We expect this overhead to be negligible. However, if experiments show otherwise, we may consider one of the alternative solutions described in the section below.

Splitting mojo message handler tasks

We will modify the mojo::Connector to execute messages in separate tasks, so that mojo code will yield to higher priority tasks more often. This should be relatively straight-forward, rockot@ is currently working on a patch that implements this.

If we find the overhead of task posting too high, we can consider batching multiple messages into smaller chunks, e.g. by adding a runtime-configurable batch size (similar to what Blink's MessagePort already implements).

Metrics

Success metrics

By prioritizing IO and GPU events/tasks, we primarily aim to improve the 99th percentile of these UMAs:

- Event.Latency.ScrollBegin.Touch.TimeToScrollUpdateSwapBegin4 and
- Event.Latency.ScrollUpdate.Touch.TimeToScrollUpdateSwapBegin4 on Android.

With future scheduling policies, we may also see an improvement in startup latency (Startup.Android.Cold.* UMAs).

Regression metrics

We will watch carefully to ensure there are no significant memory or performance regressions associated with the additional tasks for I/O event handlers and mojo message handlers. Likewise, we need to watch memory and performance metrics when we run the experiments described below.

Experiments

The initial scheduling policy will present no change from today's implicit policy (FIFO except for I/O event handlers). We plan to use Finch to run the following experiments:

- Statically prioritizing input and GPU/compositing tasks over other tasks, particularly network tasks and socket I/O event handlers.
- Detecting use cases, such as a scroll or page load, and prioritizing input vs. loading tasks accordingly.

Rollout plan

Standard experiment-controlled rollout.

Core principle considerations

Speed

This design is intended to directly improve speed metrics.

Security

We don't expect any security implications because this feature is not exposed to web content.

Privacy considerations

We are not handling or recording any user data (modulo UMA to measure impact) so this design should be privacy neutral.

Testing plan

Deferring non-essential work could potentially cause stability and UX regressions. We would like the testing team to be cognizant of this launch but we don't anticipate the need for special testing because the finch trials should reveal any performance or stability issues.

Follow-up work

We expect it will take some time to ensure every task is posted on the right queue. In the future, we also plan to split the proposed queues into separate queues for different frames and tabs, so that we can prioritize foreground tab activity over background tabs. Furthermore, we will investigate whether we can use combined scheduling policies on the IO and UI thread to improve startup performance on Android. Lastly, we may bring similar schedulers onto IO threads in other processes (e.g. GPU or renderer processes).

Parallel efforts

There are two servicification efforts underway, which may also impact the task load on the I/O thread:

The viz service refactor will move display compositing into the GPU process. As a result, most compositing-related messages that are currently handled in the browser process will be handled in the GPU process instead. Thus, these messages will no longer be blocked by concurrent tasks on the browser's IO thread. This will limit the usefulness of the IO thread scheduler with regards to compositing.

The network service refactor will replace the IPC filters on the IO thread with mojo interfaces. At first, these will continue to run on the IO thread. However, it may eventually be possible to move the network service out-of-process, which would significantly reduce the task load on the browser's IO thread and thereby limit the usefulness of the IO thread scheduler. Yet, adding a separate networking process on low-end Android devices may not be feasible.

Alternatives

Add prioritization for I/O handlers directly in the MessagePump

Instead of executing each I/O event handler in a separate task, we could make the MessagePumps aware of the priorities of individual file descriptors and add logic to their implementations to preempt processing of low-priority events for higher priority events and MessageLoop tasks.

For posix, libevent actually supports priorities for different events, i.e. file descriptors. In the event loop, higher priority events will then be processed first. We could add logic that stops the event loop after high priority events were processed if there are high-priority MessageLoop tasks pending.

For windows, I/O completion ports sadly do not support prioritization - We would probably have to split the file handles into multiple groups, each associated with a separate I/O completion port, and then only poll the high-priority port(s) if there are high-priority MessageLoop tasks pending.

This approach would require scheduling-awareness outside the SequenceManager and BrowserIOThreadScheduler, in the MessagePump. We'd like to avoid this if possible.

Disable I/O event processing when pending high-priority MessageLoop tasks exist

Instead of prioritizing I/O events, we could simply preempt them for high-priority MessageLoop tasks. However, this does not support prioritization amongst different file descriptors. Thus, incoming high-priority IPCs could still be delayed by other I/O events such as socket I/O. We would prefer an approach that allows us to prioritize such events differently.

Implementation Notes

- Plumbing a TaskRunner to mojo tasks:
    - Mojo SimpleWatcher can receive a TaskRunner.
        - Allows prioritization per "channel", i.e. interface + associated interfaces.
        - Should work without adding additional PostTasks.
    - Mojo bindings can receive a TaskRunner, which is plumbed through to MultiplexRouter.
        - Allows prioritization per interface.
        - But requires that each incoming message is posted to another TaskRunner after having been deserialized by Connector/MultiplexRouter.
        - How big would this overhead be?
- Plumbing a TaskRunner to legacy IPC messages:
    - No way to prioritize different legacy IPC interfaces on IO thread.
    - All go through same mojo interface (MessagePipeReader::Receive).
    - Can only prioritize different IPC message pipes, i.e. different endpoints.
    - Could think about prioritizing messages from GPU process?
    - Maybe not that important anyway since viz and network service are both coming soon.
        - Resource fetching is mojo already.