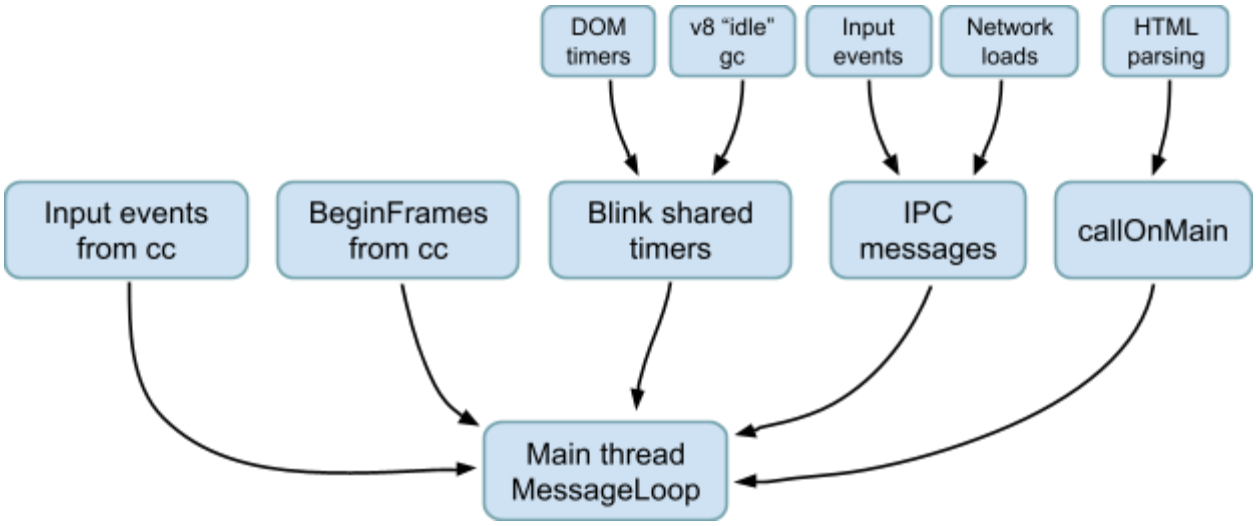
Blink Scheduler

{alexclarke, dominikg, eseidel, picksi, skyostil, rmcilroy}@chromium.org

(PUBLIC)

Introduction

The Blink main thread is getting crowded. Below is only a partial sampling of entities that schedule tasks to run in the main thread message loop.

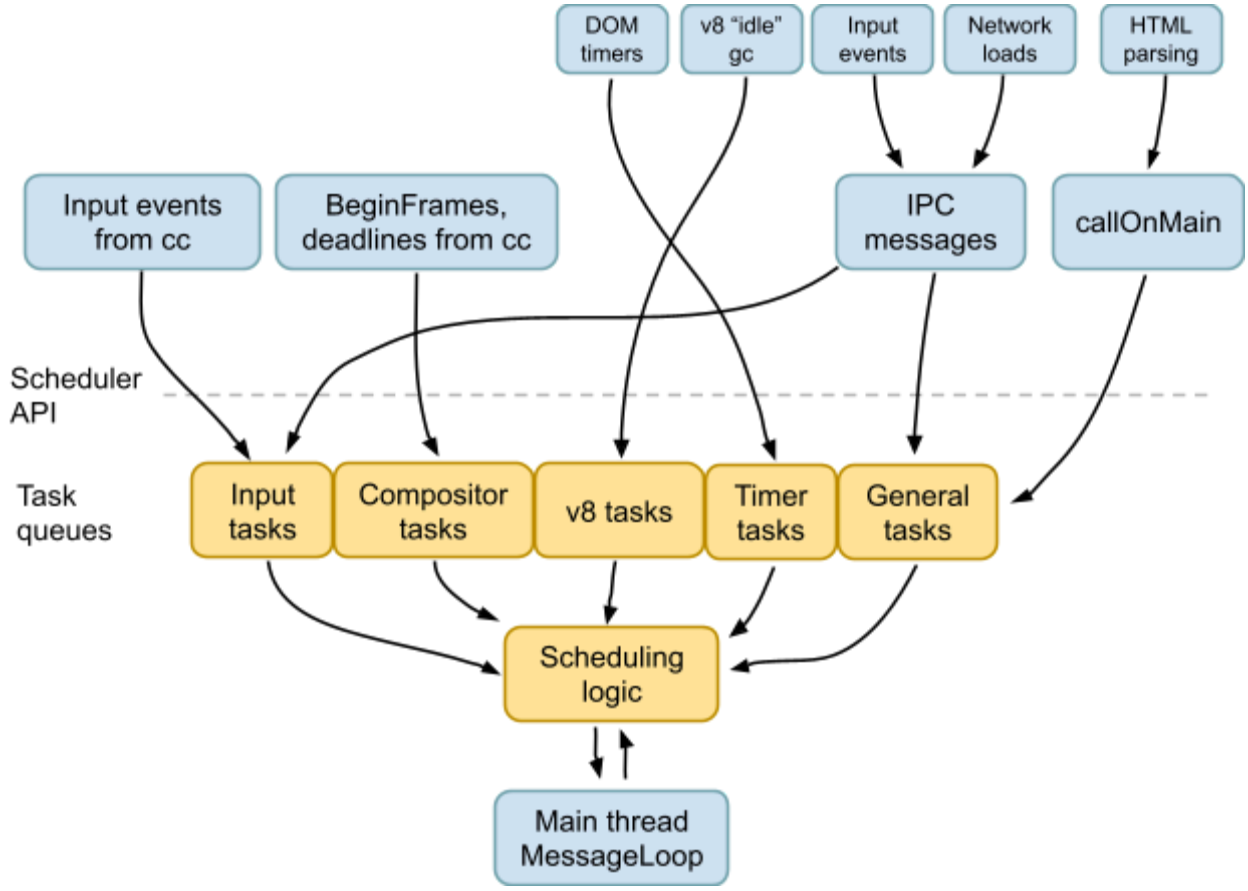


While allowing unfettered access to the main thread is convenient, it comes with several important downsides:

- **Limited prioritization** – work is either performed in the order it was posted, or can be explicitly delayed. This can cause important tasks such as input handling to get delayed ("queued") behind less time-sensitive tasks.
- **No coordination with the larger system** – the rest of the graphics pipeline has deep knowledge about input event delivery, display refresh timings and other platform constraints which are not being communicated to Blink.
- **No adaptability to different use cases** – running some tasks such as garbage collection may not be appropriate while the user is performing a touch gesture.

Proposed solution

This document describes the *Blink Scheduler*, which is designed to overcome the aforementioned issues by serving as an opinionated gateway to the main message loop:



The main purpose of the scheduler is to decide which task gets to execute on the main thread at any given time. To enable this, the scheduler provides higher level replacements for the APIs that are used to post tasks on the main thread.

The actual tasks posted to the scheduler are fairly concrete and high level instead of being opaque callbacks with abstract priorities. For example, input tasks are explicitly labeled as such and may be accompanied with additional metadata. This allows the scheduler to make more informed decisions about which tasks to run based on the system state; a static prioritization would not be rich enough to express constraints such as avoiding idle work while the user is actively interacting with the device.

Scheduler code organization

This section describes how the scheduler fits into the source tree and how it communicates with entities within Blink and the rest of the content layer.

Integration within Blink

The scheduler itself lives in `Source/platform/scheduler`. It has a single instance on the Blink main thread, which is accessible through a static accessor `blink::Scheduler::shared()`.

The `Scheduler` class provides a set of functions for posting different types of work items to its internal work queue. A key constraint is that the scheduler is allowed to choose the execution order of disjoint task types freely, but it can never reorder tasks of the same type. In particular, tasks from the same HTML task source should always run in FIFO order as required by the specification.

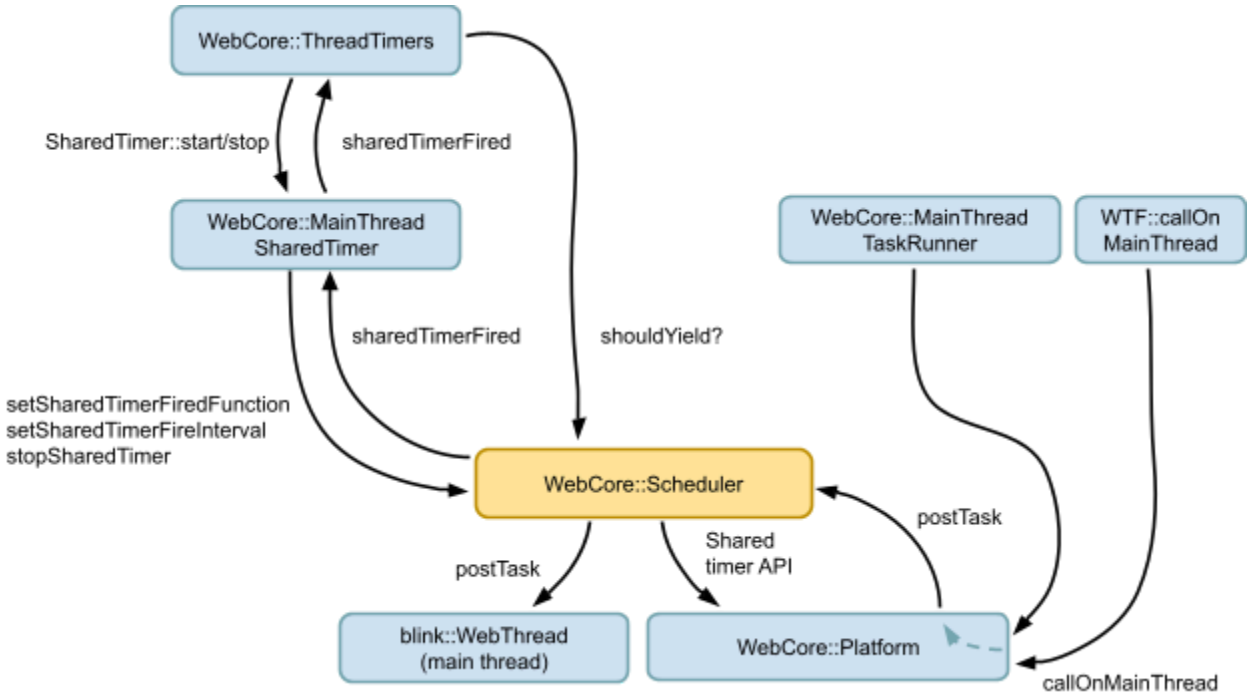
The initial set of task types is:

```
// Schedule a compositor or input task.
void postCompositorTask(blink::WebThread::Task*);

// Schedule a generic (low priority) task.
void postTask(blink::WebThread::Task*);
```

Internally each task type is placed into a separate task queue. Note that compositor and input tasks need to maintain their relative order, which is why they use the same entrypoint. Whenever there are pending tasks, the scheduler picks an appropriate one to run. To avoid starving other (non-Blink) clients of the main thread message loop, the scheduler will initially post one corresponding task to the main message loop for each incoming scheduled task and yield between task executions.

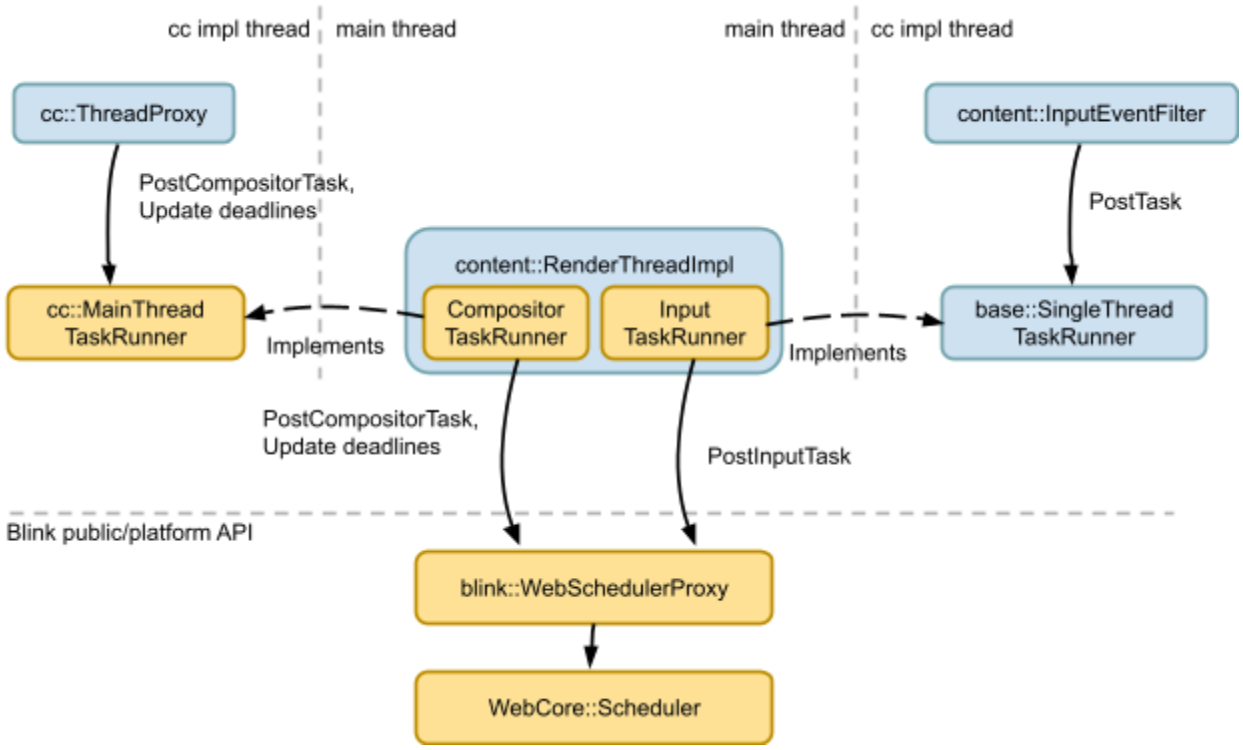
In the first phase, Blink's main thread `ThreadTimers` class will be wired into the Scheduler so that executing timer callbacks can be interrupted by higher priority work. Any work posted through `Platform::callOnMainThread` is also routed through the scheduler.



As a future improvement we are planning to fold `SharedTimer`, and by extension, `ThreadTimers` functionality into the scheduler by making timer tasks a first class concept (see Phase 2).

Compositor and content integration

The diagram below shows the interfaces exported by the scheduler to the compositor thread proxy and the input event filter, both of which run on the compositor thread. These interfaces allow posting of input, BeginFrame and generic compositor tasks.



V8 integration

The overall idea is to remove the V8 idle notification in favor of posting fine grained incremental tasks such as GC marking and letting the scheduler run them at appropriate times. V8 will provide a conservative estimate of the amount of time each GC task will take, in order to enable the scheduler to make an informed decision about how late in the idle window a GC task can safely be scheduled.

The first phase of V8 integration will involve V8 posting incremental marking GC tasks to the Blink scheduler during JavaScript execution, in preference to doing this incremental marking synchronously during JavaScript execution. The expectation is that these incremental marking tasks will be scheduled during the next idle period, and therefore the GC's incremental marking will generally "keep up" with modifications made to the heap by JS execution. However, in certain situations there will not be enough time in the idle window to perform the GC marking tasks. In order to avoid starvation of these tasks, either: the scheduler needs to detect if an

incremental marking task has been scheduled for too long, and schedule it independently of whether there is time in the idle window; or V8 will need to detect when GC marking is slowing down and then start synchronous incremental marking during JS execution (as is done now) in these situations.

Future phases may involve V8 posting full mark-sweep GC tasks during idle periods. Mark-compact GC tasks are much longer operations than incremental marking tasks, and cannot be preempted once started, so this will require a more sophisticated time estimation in order to be scheduled such that they don't introduce jank.

Benchmarking and tools

To validate our improvements and avoid regressions we will use the following metrics from the smoothness telemetry measurement:

- `queueing_durations`, which measures the delay between posting and executing a BeginMainFrame task.
- `mean_input_event_latency`, which is an end-to-end measure of input event processing time.
- `input_event_latency_discrepancy`, which measures how regular the response to input is.
- `first_gesture_scroll_update_latency`, to track latency during early phases of page loading, when the system is being maximally stressed.
- `jank`, which measures overall rendering smoothness.
- page_cycler.top_10_mobile: `cold_times` and `warm_times`, which track effects on page load time.
- tab_switching.top_10: `idle_wakeups_total` and `application_energy_consumption_mwh` for tracking energy metrics.

These metrics will be used with existing page sets (key_mobile_sites, key_silk_cases) as well as a new page set of pages with known poor scheduling behavior.

Additionally the PLT.PT_RequestToFinish histogram (as well as the rest of PLT.PT_*) should give an indication of the scheduler's impact on page load times. Event.Latency.TouchToScrollUpdateSwap should also give an indication of how scroll update delays are being affected.

Phase 1: Scheduling 101

As a first step, we will introduce bare-bones Scheduler which hooks all the necessary components together. It implements a static scheduling policy which:

1. Considers the input queue as the highest priority.
2. BeginFrame task queue as the second highest.

3. Everything else below these two categories.

This scheduler also implements timer yielding so that timer tasks are deferred if higher priority tasks are pending.

Patches

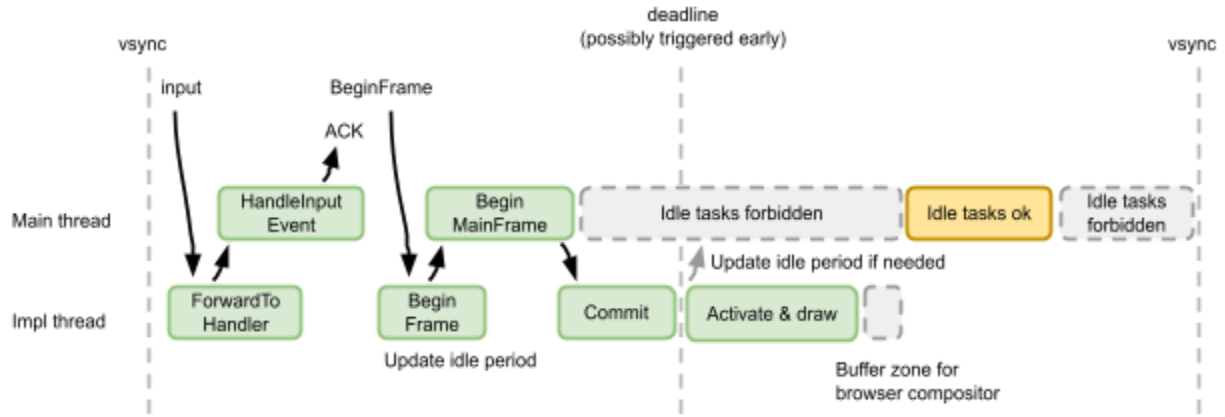
- ~~Add scheduler in Blink: https://codereview.chromium.org/364873002/~~
- ~~Hook up the Blink scheduler into the compositor (WIP): https://codereview.chromium.org/363383002/~~

Phase 2: Advanced scheduling

Phase 2 builds on the groundwork laid in phase 1 and introduces more sophisticated scheduling algorithms. This section describes parallel subtasks for these improvements.

Schedule idle work

Saturating the available CPU cores with work is a large contributor to missed deadlines, even on fairly powerful mobile devices. The scheduler has an opportunity to mitigate this by avoiding doing non-critical idle work while code on the graphics pipeline critical path is running.



To enable this, the compositor gives the scheduler a time period during which idle work is allowed. Generally this period starts soon after the renderer compositor has swapped a new frame and ends before the next vsync interval to avoid delaying out any new input at the start of the subsequent frame.

The idle time period is also given an expiration time, after which the scheduler is allowed to run idle tasks again. This is to let main thread idle work progress when the compositor itself is idle. The compositor is free to adjust the idle period at any time according to its own scheduling decisions. For instance, if the BeginFrame deadline is triggered early, the start of the idle period can be moved to start sooner.
In the compositor, the idle interval itself is derived from the BeginFrame deadline and frame interval. We will also explore disabling idle work entirely during scrolling and touch interactions.

To avoid completely starving idle work, the scheduler may choose to violate the idle period if needed.

An open issue is whether the idle period concept should be extended to other threads in the system. This is predicated on being able to detect whether the host device has limited parallelism capabilities in the first place.

Make timers first class

In the first phase, the `WebCore::ThreadTimers` class is still responsible for managing and running timer callbacks. This is a problem because the timer mechanism is heavily (ab)used by various parts of Blink – often as a replacement for postTask(). These tasks have no relative priority and are posted without regard for the larger system state.

The goal of this subtask is to audit the users of timers in Blink and migrate them to either postTask or more sophisticated scheduling primitives if possible. To support this the scheduler will also be enhanced to support delayed tasks and we'll build support for in wtf for cancellable functions.

The DOMTimers will need special handling since they require mutable deadlines, timer alignment and are reentrant as well as cancelable.

Give deadline hints for v8

When we are executing v8 code in response to critical events such as input or BeginFrame, v8 can do a better job if it is made aware of the timing constraints, i.e., the deadline for the upcoming frame. This signal could be used, for instance, to enlarge the heap to avoid a garbage collection or to bypass expensive compiler optimizations.

Break down monolithic main thread tasks

Even if we do perfect scheduling, there are still long running tasks on the main thread that will push out other urgent work. We should investigate breaking these down into smaller iterative tasks which so that the scheduler can service time critical tasks sooner.

Gather real world data about scheduling

While highly repeatable, the synthetic gestures used by telemetry benchmarks don't do a very good job at mimicking a real user. To get a real world perspective we should create a custom version of the browser (for internal use only) that automatically records tracing data and submits it for analysis.

Apply slop (coalescing) to timers

Timer coalescing can be used to save power on modern hardware by allowing longer CPU naps. http://en.wikipedia.org/wiki/Timer_coalescing. Additionally window.setTimeout and many

other core::Timer uses don't need to be exact and could be taught to slop to avoid RAF or save power.

Once Tasks are treated as Tasks in Blink instead of one-shot 0 ms timers, the idea of slopping these tasks is more straightforward, and we leave timers which actually need to execute at a given time as Timers.

- Bug: crbug.com/356804

Provide developers APIs for all long-tasks they might want to do async (parsing, layout, rendering, etc.)

At the limit, eventually the main thread will only be for JS. However even then current JS APIs encourage synchronous execution of possibly long tasks on the main thread. We've been successful at deprecating synchronous XHR, but we need to go further and provide authors asynchronous ways to parse large sections of HTML, layout arbitrarily complicated sections of DOM, etc.

Proposal: Review our current offerings and make sure that we're providing developers sufficient async APIs. For example, there is no way to insert nodes incrementally and async (like the parser does during load) or to have a callback when layout is completed of a section of DOM you would like to display, etc.

Resources and Related Work

- Bug label: SlowBecauseOfBlinkScheduler.
- eseidel@'s original Blink Scheduling document.
- Prioritized task queue design from bashi@.
- Making Blink aware of frame times from tansell@.
- Compositor driven garbage collection from ernstm@.

Old, unorganized and incoherent notes

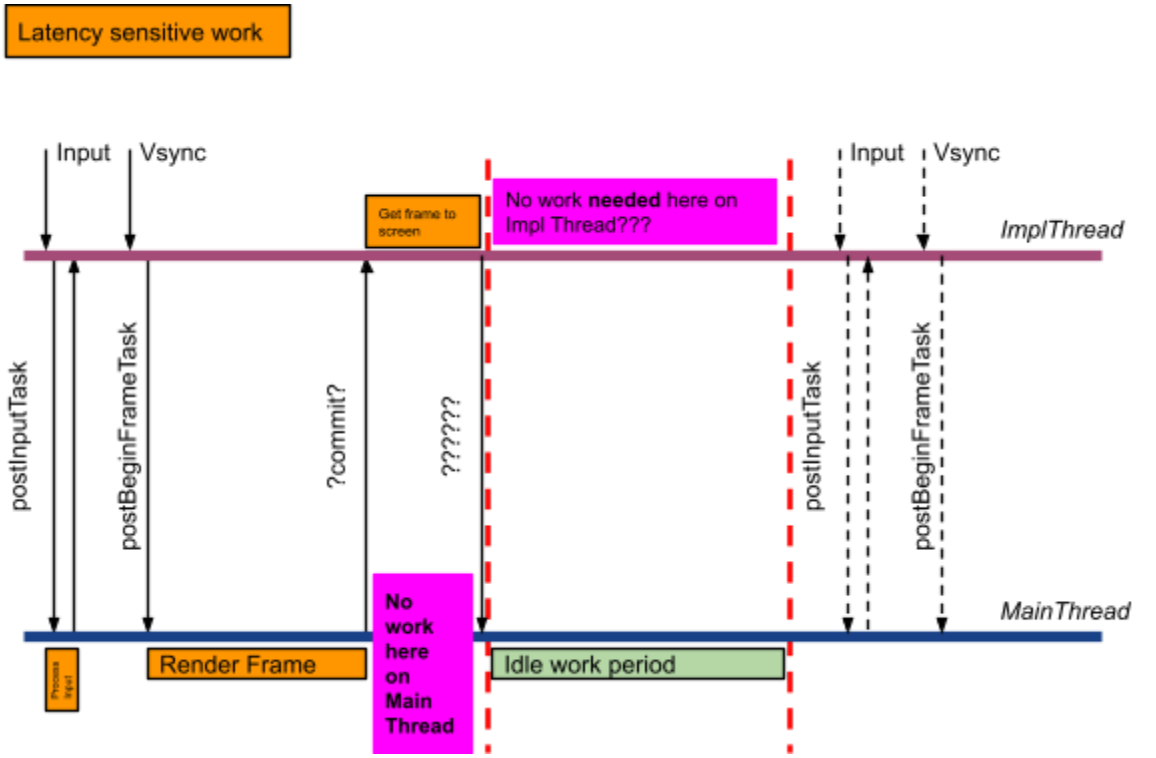
- Add support for idle and quiet periods
    - “Now is a good time to do idle work.”
    - “Let go of the CPU -- someone else needs to do high priority work.”
    - These can change dynamically, so we probably need a separate function to update them.
    - Need to avoid starving other work.
- Take input state into account
    - Should this live in the compositor or blink? Some input doesn’t flow through the compositor.
    - Can we encode this as idle/quiet periods instead?
- Remove v8 idle task in favor of v8 posting incremental gc runs

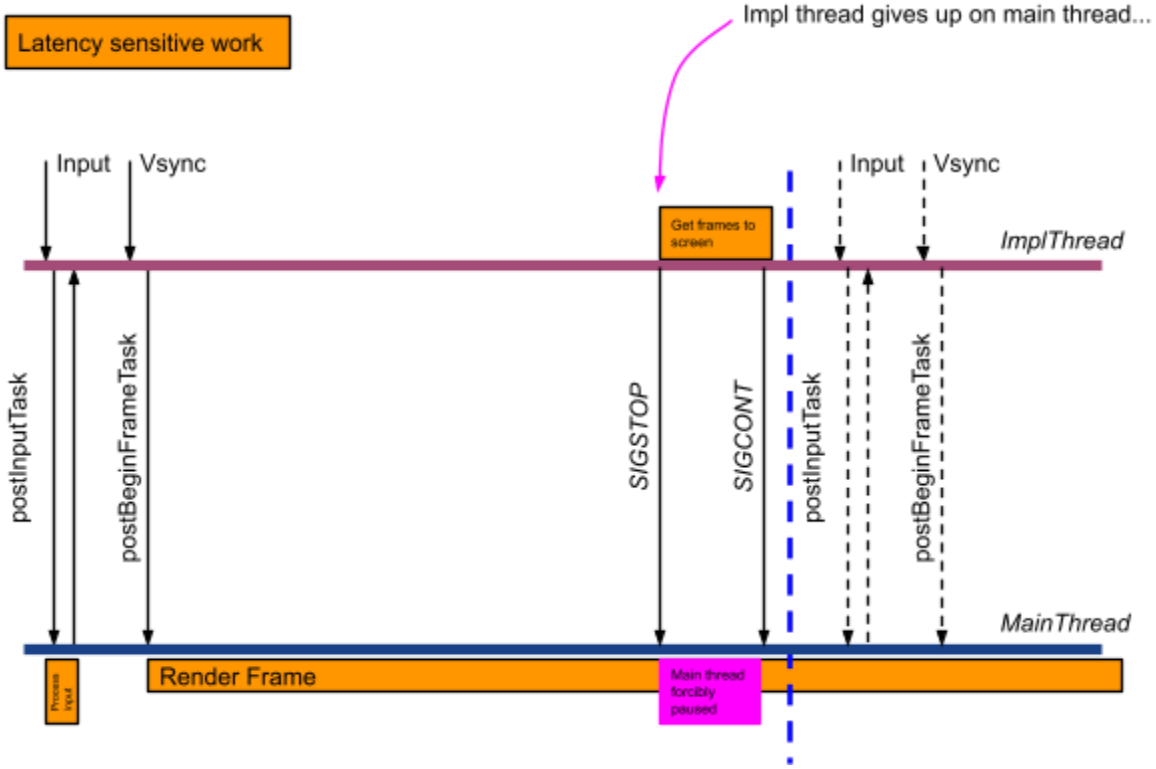
- Inputs
    - Tasks from outside Blink using PostTask
        - Input -- InputEventFilter::ForwardToHandler
        - BeginMainFrame -- ThreadProxy::ScheduledActionSendBeginMainFrame via Proxy::MainThreadTaskRunner()
        - IPC -- ChannelProxy::OnMessageReceivedNoFilter
            - Network request results (ResourceDispatcher::OnRequestComplete)
            - Problem with WebFonts: OnRequestComplete() should have higher priority than callOnMainThread() tasks. Just in this case or in general?
        - Miscellaneous
            - ThreadProxy::DidSwapBuffersCompleteOnImplThread
    - Tasks from within Blink using PostTask
        - Platform::callOnMainThread() used by HTML parser thread and others
        - Document::postTask() used for web workers and logging?
        - Platform::current()->currentThread()->postTask() mainly used in tests
    - Tasks from within Blink using SharedTimers
        - Can be suspended while sending IPC (? -- RenderThreadImp::Send)
        - Can also be cancelled
        - Dispatched by ThreadTimers::sharedTimerFiredInternal
            - Uses single underlying timer to coordinate a set of pending timers. Sleeps until the next timer should fire.
            - No re-entrant firing, except when overridden (by whom?)
            - Up to 50ms per batch
            - Some timers may be repeating
        - Notable users

        - DOMTimer (4 ms min interval, clamped to same if >= 5 levels deep)
        - v8
        - XMLHttpRequest
        - ImageLoader
        - EventSender::dispatchEventSoon
        - ScriptRunner::notifyScriptReady
- Currently mainly first-come-first-serve. Only shared timers yield after running for 50 ms

Scheduler requirements

- Allow prioritizing certain messages over others
    - For instance: input > BeginFrame > network loads > everything else
    - However, the prioritization will not be static! Consider load time vs. scroll time.
- Let iterative tasks know about the idle interval so they know when to stop.
    - Expected time until the next BeginFrame
- Annotate tasks based on their purpose
    - Initial classes: INPUT, RESOURCE_COMPLETION, UNSPECIFIED (lowest priority)
    - BeginFrame gets special treatment because we need to inspect its parameters
    - Separate queue for each task class. Ordering guaranteed within single queue but not between classes.
- Thread-safe entrypoints for adding tasks and indicating other events which influence scheduling decisions


Latency sensitive work
Input
Vsync
Get frame to screen
No work **needed** here on Impl Thread???
Input
Vsync
ImplThread
postInputTask
postBeginFrameTask
?commit?
??????
postInputTask
postBeginFrameTask
MainThread
No work here on Main Thread
Render Frame
Idle work period


Latency sensitive work
Impl thread gives up on main thread...
Input
Vsync
Get frames to screen
Input
Vsync
ImplThread
postInputTask
postBeginFrameTask
SIGSTOP
SIGCONT
postInputTask
postBeginFrameTask
MainThread
Render Frame
Main thread forcibly paused

Assorted (probably overly-complex) thoughts

1. Priority - There is an implicit priority (the order in which tasks actually get executed) which feels like it is an emergent (calculated) value that depends on the attributes of the task, queue and scheduler state. Is there any use in actually 'surfacing' this value, so we can take a snap-shot of all the queues & tasks and produce a what-will-happen-in-the-next-n-frames list? Can we use this meta-level schedule view to see problems ahead? i.e. If we do all this GC this frame we'll push the input events out by 2 frames, so lets do 1 input event now, even though GC is important.
   1a. Meta-priority rules. Do we have implicit rules about task execution that we should make note of i.e. we always execute one input task and one timer task per frame regardless of other task priorities.

2. Order - We currently assume that task-queues are FIFO for execution order. Should we allow this to be explicitly specified by task-sequences (e.g. "AddOrderedTask" Vs "AddIsolatedTask"), so we are able to shuffle non-order dependent tasks around [see point 6 too]?

3. Cancellation/Redundancy detection - Some new tasks may make old tasks redundant, i.e. layout->load font->layout, might be nice to know this and as soon as the 2nd layout gets queued we cancel the first layout. It might be better to cancel old Input tasks if there are many already queued. What clues can the caller give us about this? e.g. Pass a task type ID (Layout=37) and specify somewhere that only the most recent type 37 task needs executing.

4. Handles. Do we need to pass a handle back to the calling functions so they can change the contents of the task once its been posted (i.e. cancel it, or alter its delay).

5. Expected Duration + scalar. If we could estimate execution duration it would enable us to select tasks to fit in the available time, and avoid executing tasks that would tip us over into the next frame. A possible way to implement this would be for tasks to pass through an ID for the type of task (e.g. layout, GC, input etc) and a scalar value that represents the complexity of the task (i.e. the size of the DOM for a layout, number of allocated blocks for GC). The scheduler could, over time, learn the average duration per scalar value (e.g. layout =0.01ms per DOM element) for each type of task and use this to produce a guesstimate.

6. Task group IDs. To group tasks that have been split (i.e. a monolith V8 task that has been sharded into smaller tasks), so we can guarantee FIFO order (etc?), it might be

useful to either allow tasks to pass through a common group ID, or have something like StartV8Group()/StopV8Group() that can be used to bracket submission of related execution-order dependent tasks.

7. Starvation strategies/Aging/Late execution. We need to figure out how each queue will deal with starvation. We also need to think about how lateness of execution should affect a task's importance - GC gets more urgent as it gets older/later, but input tasks (past a certain threshold), become ignorable and out-of-date.

General implementation ruminations

There are 3 different objects that are interacting in the scheduler, each knowing little (nothing?) about the level above:

1. The tasks themselves: These have knowledge of what code they need to execute and (potentially) other details - how long an execution will take, how important they are, if they have any urgent deadlines, if they have been delayed.
2. The queues: They know how many tasks they contain, when they were last serviced, if they are being starved, the execution order of the tasks.
3. The Scheduler itself: It knows how many queues it has, their relative importance (i.e. an idle queue is less important than a user input queue) and system wide knowledge (i.e. we are in an 'idle' state, deadlines are approaching, page-load is underway etc.).

If feels as if 1 (tasks) and 3 (scheduler) should have no knowledge of each other. The queues feel like a mediator between the them.

Addition of new queues: We are starting with 3 or 4 queues and will be scaling up and adding more complexity over time. We should be able to add new queues to the scheduler with the minimum of code modification. It feels like we should be able to 'inject' extra queues or new'up queues when the scheduler is created and add them into a list-of-queues.

I (picksi@) would really like to attempt to hide a queue's specific knowledge about itself from the scheduler - i.e. V8 garbage collection getting steadily more urgent as it isn't serviced should not be specifically known about in the scheduler, rather this is a property of the queue (one could argue a property of the task?), which is passed to the scheduler in a common (V8 GC independent) way. We could then re-use this mechanism for other queues as needed.

Thoughts on Timers

All the startOneShot(0, …) timers can likely be refactored to postTask() however the timer needs to get cancelled if the object goes out of scope. The usual way of doing that in chromium would be via base::Bind and weak pointers, Blink has it's own reference counting system

Instead I propose we add new classes:

OneShotTask
RepeatingTask

The shortest DOMTimer delay is 1ms, although 4ms is more common. Under the hood we might be able to round all timers to the nearest ms.
With the exception of DOMTimer and V8GCForContextDispose::notifyIdleSooner delayed timers are typically scheduled more than 1/60th of a second in advance, with many timers being scheduled to execute in over 1s.

Histogram of timer intervals while loading http://theverge.com

```
0.000000s = 545 times (postMessage in ../../third_party/WebKit/Source/core/frame/LocalDOMWindow.cpp)
0.001000s = 8 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.004000s = 569 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.010000s = 2 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.020000s = 2 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.025000s = 7 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.048000s = 1 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.050000s = 7 times (dispatchProgressEvent in
../../third_party/WebKit/Source/core/xml/XMLHttpRequestProgressEventThrottle.cpp)
0.091000s = 1 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.097000s = 1 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.098000s = 1 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.099000s = 2 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.100000s = 186 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.125000s = 1 times (notifyIdleSooner in
../../third_party/WebKit/Source/bindings/v8/V8GCForContextDispose.cpp)
0.200000s = 2 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.250000s = 31 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.300000s = 2 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.351000s = 1 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.500000s = 12 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)
0.800000s = 4 times (notifyContextDisposed in
../../third_party/WebKit/Source/bindings/v8/V8GCForContextDispose.cpp)
>= 0.999000s = 115 times (DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp)

Caller frequency
(didAssociateFormControl in ../../third_party/WebKit/Source/core/dom/Document.cpp) 7 times
(didLoadAllScriptBlockingResources in ../../third_party/WebKit/Source/core/dom/Document.cpp) 24 times
(finishedParsing in ../../third_party/WebKit/Source/core/dom/Document.cpp) 56 times
(checkLoadEventSoon in ../../third_party/WebKit/Source/core/dom/Document.cpp) 19 times
(notifyScriptReady in ../../third_party/WebKit/Source/core/dom/ScriptRunner.cpp) 42 times
(enqueueEvent in ../../third_party/WebKit/Source/core/events/GenericEventQueue.cpp) 1 times
(dispatchEventSoon in ../../third_party/WebKit/Source/core/events/EventSender.h) 7 times
(dispatchEventSoon in ../../third_party/WebKit/Source/core/events/EventSender.h) 6 times
(dispatchEventSoon in ../../third_party/WebKit/Source/core/events/EventSender.h) 22 times
(runAsync in ../../third_party/WebKit/Source/platform/AsyncMethodRunner.h) 10 times
(addFontToBeginLoading in ../../third_party/WebKit/Source/core/css/FontLoader.cpp) 3 times
```

(add in ../../third_party/WebKit/Source/core/css/resolver/MatchedPropertiesCache.cpp) 3 times
(beginLoadIfNeeded in ../../third_party/WebKit/Source/core/fetch/FontResource.cpp) 9 times
(allClientsRemoved in ../../third_party/WebKit/Source/core/fetch/Resource.cpp) 13 times
(scheduleDocumentResourcesGC in ../../third_party/WebKit/Source/core/fetch/ResourceFetcher.cpp) 73 times
(requestLoadStarted in ../../third_party/WebKit/Source/core/fetch/ResourceFetcher.cpp) 15 times
(DOMTimer in ../../third_party/WebKit/Source/core/frame/DOMTimer.cpp) 870 times
(scrollPositionChanged in ../../third_party/WebKit/Source/core/frame/FrameView.cpp) 2 times
(postMessage in ../../third_party/WebKit/Source/core/frame/LocalDOMWindow.cpp) 221 times
(startCheckCompleteTimer in ../../third_party/WebKit/Source/core/loader/FrameLoader.cpp) 7 times
(updatedHasPendingEvent in ../../third_party/WebKit/Source/core/loader/ImageLoader.cpp) 70 times
(startTimer in ../../third_party/WebKit/Source/core/loader/NavigationScheduler.cpp) 1 times
(PingLoader in ../../third_party/WebKit/Source/core/loader/PingLoader.cpp) 2 times
(dispatchProgressEvent in
../../third_party/WebKit/Source/core/xml/XMLHttpRequestProgressEventThrottle.cpp) 6 times
(notifyContextDisposed in ../../third_party/WebKit/Source/bindings/v8/V8GCForContextDispose.cpp) 4 times
(notifyIdleSooner in ../../third_party/WebKit/Source/bindings/v8/V8GCForContextDispose.cpp) 6 times