Blink Scheduler Refactor

{skyostil,alexclarke,rmcilroy}@chromium.org

(Public document)

Tracking bug: http://crbug.com/391005

Patches:

- https://codereview.chromium.org/637303003 (chromium)
- https://codereview.chromium.org/657953004 (chromium)
- https://codereview.chromium.org/664963002 (chromium)
- https://codereview.chromium.org/656463004 (blink)

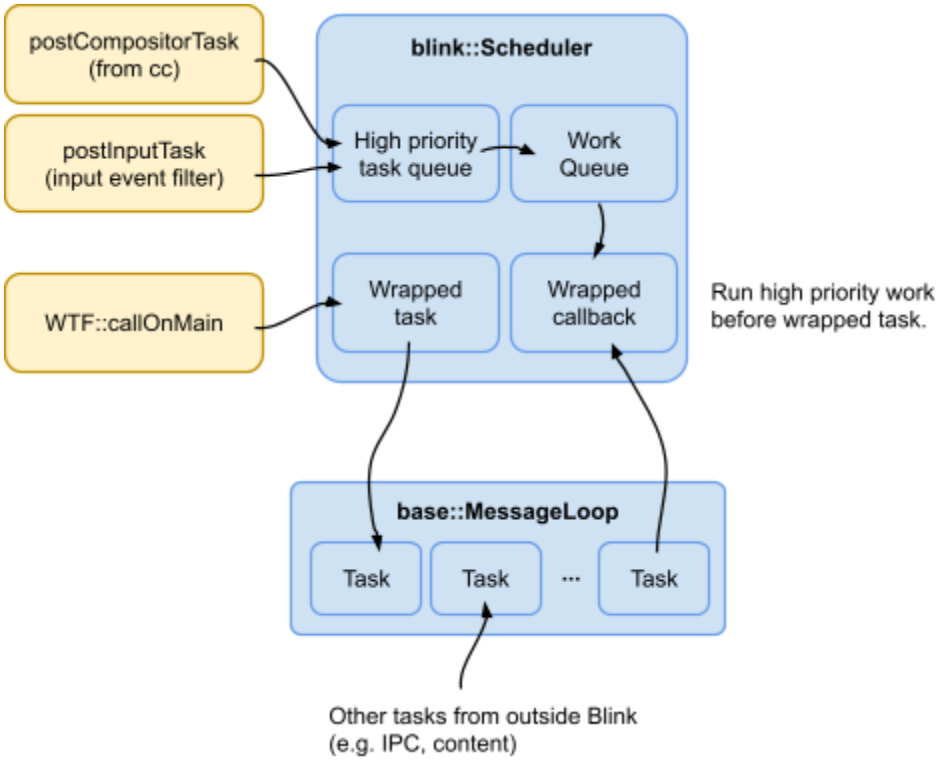
Introduction

The Blink Scheduler is a project to improve the responsiveness of the Blink main thread. One aspect of this problem is making sure urgent tasks such as input handling or compositor tasks are handled promptly. This helps Chrome respond more quickly and regularly to user input. Currently this is done by redirecting these types of tasks from the content layer to a set of task queues implemented inside the Blink Scheduler.

This document proposes a refactoring 1) to simplify the communication between the scheduler and the rest of the system and 2) to separate out the *mechanism* used for task queue management from the *policy* of how the queues are serviced.

Existing scheduling mechanism

The current Blink scheduler implements task prioritization with its own internal task queue. Main thread bound tasks from the renderer compositor and the input event filter are explicitly forwarded to this queue using a proxy interface. The scheduler intercepts all main thread work posted from within Blink (e.g., `WTF::callOnMainThread()`) and arranges itself to run before the actual task that was posted. When the callback is triggered (by `base::MessageLoop`) and the scheduler wants to prioritize compositor and input tasks, it executes the pending tasks from its own queue before letting the Blink-initiated work run.



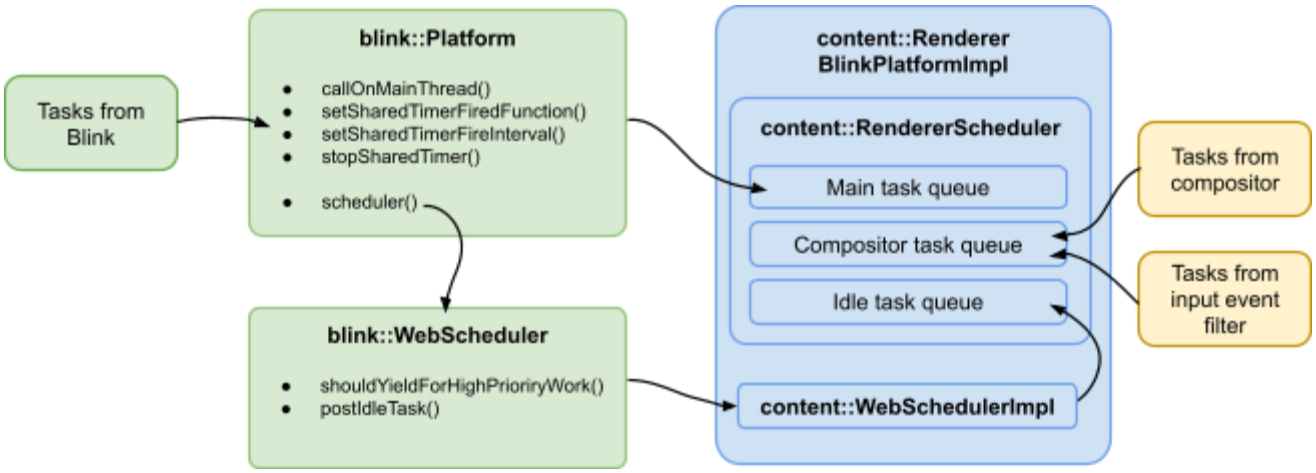
The benefit of this approach is that we can implement the scheduling policy inside Blink using Blink-level concepts such as whether animations are running or the user is actively interacting with the device. However, there are a number of drawbacks:

- Posting tasks to the scheduler from content requires going through several layers and some information is lost in transit (content => blink::WebSchedulerProxy => blink::Scheduler). For example, FROM_HERE tracking, i.e., where a given task was posted from, is only partially supported.
- The internal task queue requires us to reimplement a lot of `base::MessageLoop` inside Blink. There is a lot of subtlety here to avoid problems such as starvation and re-entrancy.
- As tasks are posted from arbitrary threads, the scheduler needs to be very thread-aware. Most other code in Blink does not need to do this, and as a consequence the WTF's threading primitives are weaker than ones in base (e.g., atomics).

The rest of this document describes a proposal for adding a new task queue manager in base with a render scheduler in the content layer to overcome these difficulties.

Renderer scheduler

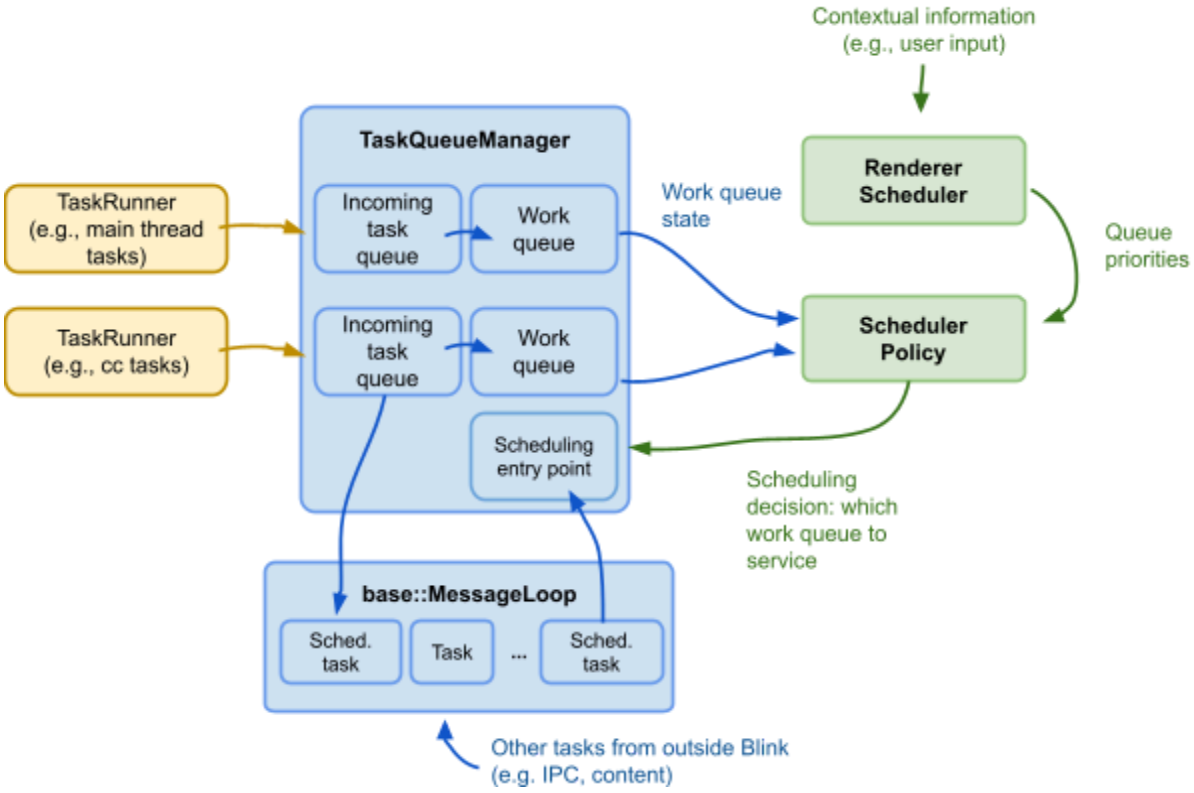
We propose to add a new `content::RendererScheduler` class which implements the same scheduling logic as the current Blink scheduler. The following picture shows how tasks from Blink and the rest of content are routed into this class:



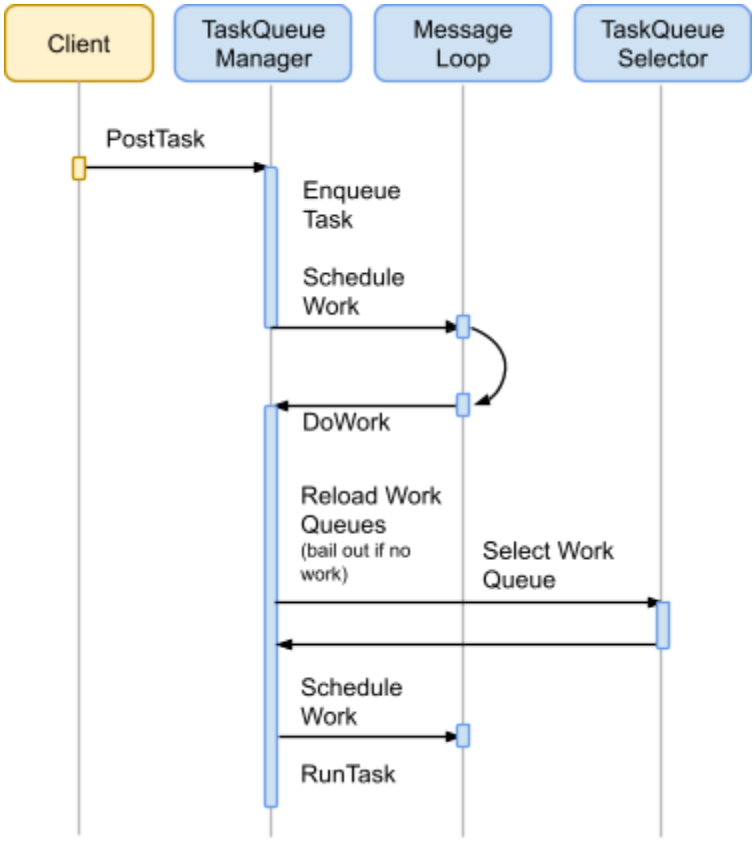
Task Queue Manager

A scheduler is built on a `base::TaskQueueManager` class provides two main pieces of functionality:

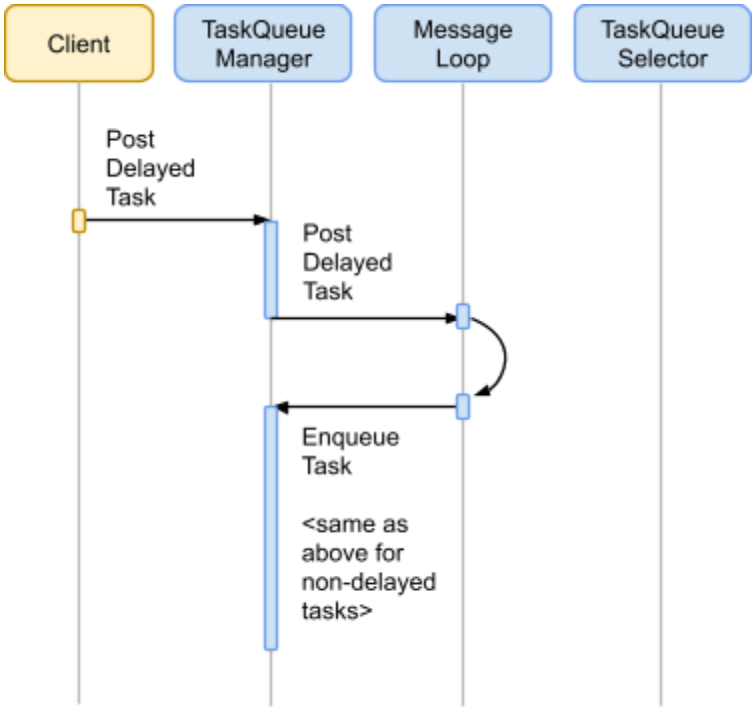
- An arbitrary number of incoming task queues.
- A selector interface which allows an outside agent choose the next task queue to be serviced.



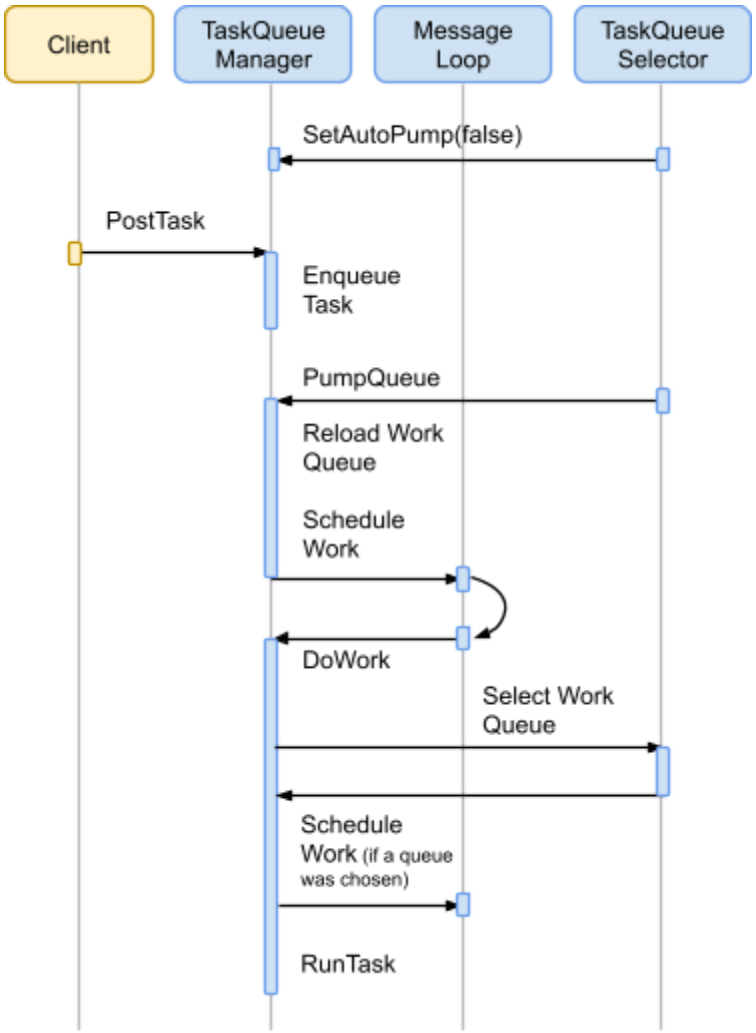
Note that the actual run loop is still owned by `base::MessageLoop`; the task queue manager simply ensures that there is at least one task on the main message loop which ends up calling back into task queue manager when there is work pending. This callback activates the scheduling entry point, where the task queue manager consults the scheduling policy in the renderer scheduler to choose the next work queue to be serviced. This process is illustrated below.



Delayed tasks are handled by posting a delayed wrapped task, which, when run, appends the delayed task into the corresponding incoming task queue. This way the selector sees normal posted tasks and delayed tasks which are due to run in the same way.



The selector can also choose to not run any pending work at all. This is to support idle tasks, which are only allowed to run at specific times. To make this possible, the selector can also disable automatic pumping of a given task queue. In this mode, the selector explicitly controls when tasks get moved from the incoming queue to the work queue and get scheduled for execution. See the idle task design document for more details.



Renderer Task Queue Selector

The `content::RendererTaskQueueSelector` class implements the `TaskQueueSelector`. It performs the prioritization of task queues by selecting the highest priority task queue when queried - with ties decided by the queue with the oldest task.

The `RendererTaskQueueSelector` provides the `RendererScheduler` with the mechanism to to change the priority of task queues or enabled and disabled them, however it does not perform the policy decisions in making these changes - instead defering policy decisions to the `RendererScheduler`.

There are four queue priorities (from highest to lowest priority):

- **CONTROL_PRIORITY**: always executed first and can potentially starve any other queues - intended to be used only for scheduler control tasks which should run before the next task is selected (e.g., updating queue priorities after a policy change)

- **HIGH_PRIORITY**: Run in preference to NORMAL_PRIORITY, but with starvation control to ensure that NORMAL_PRIORITY tasks don't get starved.
- **NORMAL_PRIORITY**: The default priority
- **BEST_EFFORT_PRIORITY**: Only run if there are no tasks in any higher priority task queue. Can be starved.

Renderer Scheduler

The `content::RendererScheduler` class maintains the high-level view of the current scheduling decisions and maintains multiple task queues, which are managed by a `TaskQueueManager.` It also holds a reference to a `RendererSchedulerSelector`, which is used to drive the task queue selection decisions for the `TaskQueueManager`.

Each task queue a particular type of task (e.g., compositor tasks, loading tasks, idle tasks, etc.). Work within a particular task queue is always executed in-order, however the scheduler is free to reorder tasks on different task queues. Each task queue has an associated `base::SingleThreadTaskRunner`, which is the interface used to post tasks to a particular queue. By ensuring task order within a task queue, we limit the scope for reordering bug, since a client has to explicitly choose to post a task to a different queue type for it to be reordered in relation to other tasks.

The `RendererScheduler` has multiple policies, each of which will prioritize different task queues. It listens to signals from the rest of Chrome and moves to a different policy in response to these signals (e.g., moving to `COMPOSITOR_PRIORITY_POLICY` when an user-input event is received).

When updating the policy, the `RendererScheduler` will change the priorities of queues in the `RendererSchedulerSelector`. It can also enable and disable queues, for example, enabling the idle task queue during idle time, and disabling it out-with idle time (see the idle task design doc for more details).

All policy updates happen on the main thread, however some signals which trigger policy changes occur on other threads. These signals will post a policy update task onto a special private control task queue. This control queue has `CONTROL_PRIORITY`, therefore this task will be run before any other task, and the next non-control task will be scheduled based on the updated policy. The control task queue is also employed to post delayed tasks which will restore the policy after a given time period has expired.

The scheduler also provides a `ShouldYieldForHighPriorityWork()` function which returns `true` if there is high priority work waiting and the current task should yield to that work, based on the current policy and whether there are any pending tasks in high priority queues. This function could be called when policy update task is awaiting execution on the control queue (e.g., posted due to a signal from a different thread). Since this policy update could change the decision about whether there is high priority work, we want to react to the policy change during this call

and not wait for the control task to be scheduled after the current task completes. To do this while keeping `ShouldYieldForHighPriorityWork` lightweight, the scheduler maintains an atomic flag which is set whenever a policy change task is posted. The `ShouldYieldForHighPriorityWork` function checks this flag, and performs the policy update inline if it is set.

Reference: Task Queue API

```
class TaskQueueManager {
 public:
  // Create a task queue manager with |task_queue_count| task queues.
  // |main_task_runner| identifies the thread on which where the tasks are
  // eventually run. |scheduler| is used to choose which task queue to service.
  TaskQueueManager(size_t task_queue_count,
                   scoped_refptr<base::SingleThreadTaskRunner> main_task_runner,
                   TaskQueueSelector* selector);
  ~TaskQueueManager();

  // Returns the task runner which targets the queue selected by |queue_index|.
  scoped_refptr<base::SingleThreadTaskRunner> TaskRunnerForQueue(
      size_t queue_index);

  // If |auto_pump| is false, tasks posted to the given incoming queue will not
  // be automatically scheduled for execution or transferred to the work queue.
  void SetQueueAutoPump(size_t queue_index, bool auto_pump);

  // If the work queue is empty, reloads new tasks from the incoming queue.
  // After this, this function ensures that the tasks in the work queue, if any,
  // are scheduled for execution.
  //
  // This function only needs to be called if automatic pumping is disabled
  // for |queue_index|. See |SetQueueAutoPump|. By default automatic pumping
  // is enabled for all queues.
  void PumpQueue(size_t queue_index);

  // Returns true if there are any tasks in either the work or incoming task
  // queue identified by |queue_index|.
  bool PollQueue(size_t queue_index);

  [...]
};

class TaskQueueSelector {
 public:
  virtual ~TaskQueueSelector() {}
```

```
  // Called once to register the work queues to be scheduled. This function is
  // called on the main thread.
  virtual void RegisterWorkQueues(
      const std::vector<const base::TaskQueue*>& work_queues) = 0;

  // Called to choose the work queue from which the next task should be taken
  // and run. Return true if |out_queue_index| indicates the queue to service
  // or false to avoid running any tasks.
  //
  // This function is called on the main thread.
  virtual bool SelectWorkQueueToService(size_t* out_queue_index) = 0;
};
```

Reference: RendererScheduler API

```
class RendererScheduler {
 public:
  virtual ~RendererScheduler();
  static scoped_ptr<RendererScheduler> Create();

  // Returns the default task runner.
  virtual scoped_refptr<base::SingleThreadTaskRunner> DefaultTaskRunner() = 0;

  // Returns the compositor task runner.
  virtual scoped_refptr<base::SingleThreadTaskRunner> CompositorTaskRunner() = 0;

  // Returns the idle task runner. Tasks posted to this runner may be reordered
  // relative to other task types and may be starved for an arbitrarily long
  // time if no idle time is available.
  virtual scoped_refptr<SingleThreadIdleTaskRunner> IdleTaskRunner() = 0;

  // Called to notify about the start of a new frame. Must be called from the
  // main thread.
  virtual void WillBeginFrame(const cc::BeginFrameArgs& args) = 0;

  // Called to notify that a previously begun frame was committed. Must be
  // called from the main thread.
  virtual void DidCommitFrameToCompositor() = 0;

  // Tells the scheduler that the system received an input event. Called by the
  // compositor (impl) thread.
  virtual void DidReceiveInputEventOnCompositorThread(
      blink::WebInputEvent::Type type) = 0;

  // Tells the scheduler that the system is displaying an input animation (e.g.
  // a fling). Called by the compositor (impl) thread.
  virtual void DidAnimateForInputOnCompositorThread() = 0;
```

```
  // Returns true if there is high priority work pending on the main thread
  // and the caller should yield to let the scheduler service that work.
  // Must be called from the main thread.
  virtual bool ShouldYieldForHighPriorityWork() = 0;

  // Shuts down the scheduler by dropping any remaining pending work in the work
  // queues. After this call any work posted to the task runners will be
  // silently dropped.
  virtual void Shutdown() = 0;

  [...]
}
```