Browser UI Thread Scheduler

This Document is Public

Authors: alexclarke@chromium.org
Reviewed by: skyostil@chromium.org, gab@chromium.org, and chrome-eng-review
(sign-in to comment)

One-page overview

Summary

Startup especially on low end Android devices is slower than we'd like. E.g a Huawei gobo 512 device takes over 5s to load https://bbc.co.uk from a cold start with a fast network. We'd like to improve this by introducing a browser UI thread scheduler which will initially prioritize cold starts and later reduce UI thread related jank. A major part of this will be posting Java tasks (wherever possible) through the C++ scheduler.

Platforms

All; focused on Android.

Team

scheduler-dev@chromium.org

Bug

https://bugs.chromium.org/p/chromium/issues/detail?id=872372 (Design review)
https://bugs.chromium.org/p/chromium/issues/detail?id=863341 (Tracking Bug)

Code affected

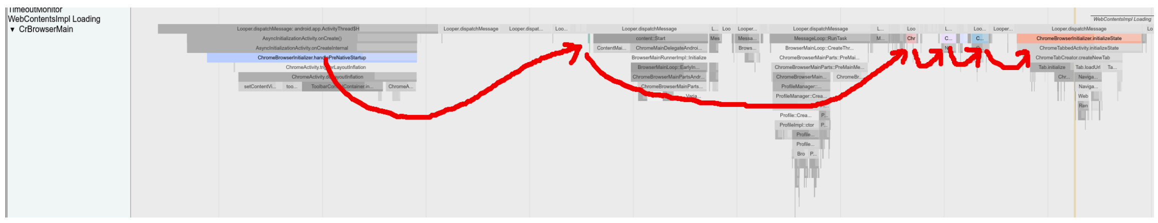
All UI thread task posting in the browser process, including java tasks. NB this is mostly introducing task queue assignment to the various PostTasks, so most "affected" code will remain unchanged.

Design

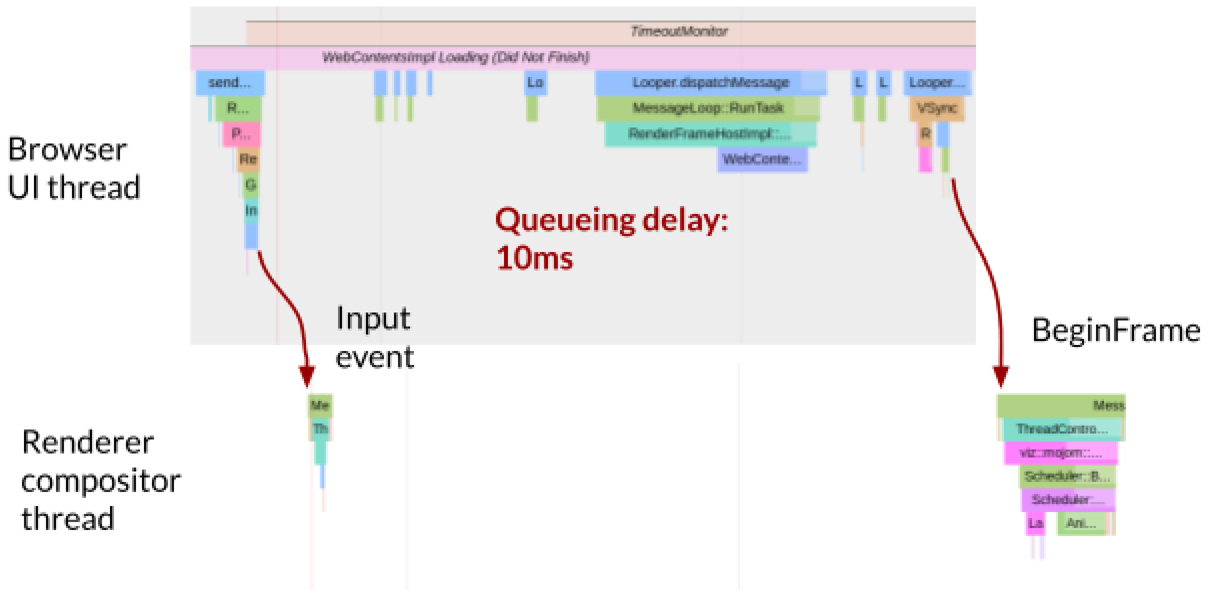
Recently Blink's task queue manager (now called the SequenceManager) has been moved to base/, so it's now possible to use the scheduling primitives it provides outside Blink. The browser UI thread is a prime candidate for this. Previous efforts have moved some things off-thread but not everything can be moved. The UI thread is largely a single unprioritized FIFO, although some tasks are deferred until after startup (see AfterStartupTaskUtils). A great many things are initialized during startup, only some of which are needed urgently to navigate to the target url, so there is an opportunity for scheduling.

On Android, C++ tasks are only half the story because Java also posts UI thread tasks via the Java message loop to the underlying Android looper. Recently the plumbing for C++ tasks changed: They used to go through the java message loop, but are now posted to the underlying native looper directly. Still there is currently no scheduling between Java vs C++ tasks, which causes various problems:

- This trace shows the critical path (on the UI thread) leading up to the initial navigation (mostly java tasks). Because there's no prioritization, these tasks end up being delayed by other work, making the overall operation longer than it needs to be. NB some of the grey tasks are also necessary for navigation, e.g. creating the profile. The UI is also created early on which is important too, however the initial OS loading animation masks (up to a point) short delays in initializing the UI which would be incurred by prioritizing the initial navigation.

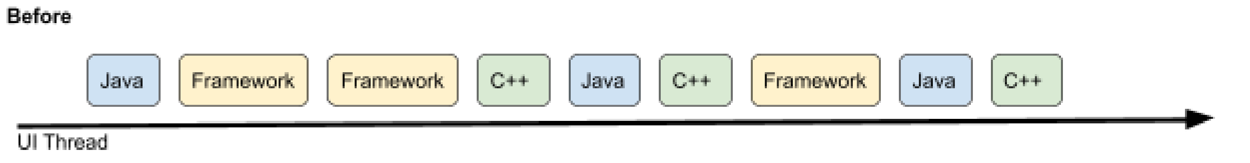

TimeoutMonitor
WebContentsImpl Loading
▾ CrBrowserMain
Looper.dispatchMessage: android.app.ActivityThread$H
ChromeBrowserInitializer.handlePreNativeStartup
content::Start
MessageLoop::RunTask
ChromeBrowserInitializer.initializeState
WebContentsImpl Loading

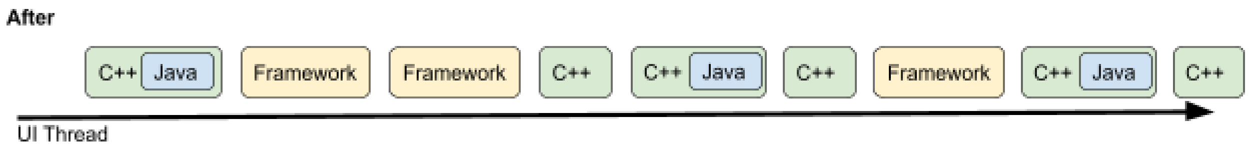
- The following trace of scrolling the site hs.fi shows an example where lack of prioritization causes a frame drop. Since touch events and the vsync signal (which triggers rendering) are delivered by separate Java tasks, it's possible for C++ tasks to run between them. In this case, navigation-related tasks delayed the vsync task by 10 ms, causing the renderer to miss the frame deadline.



Based on these observations a prototype patch was written to investigate the potential for improvement by expediting the chain of tasks leading up to the first navigation. A 10% reduction (measured by taking 10 startup traces) in time to FirstMeaningfulPaint was observed on a Huawai gobo 512. We can't land that patch, but it motivates a proper design.

We have an initial patch (needs splitting up) that introduces the scheduler and posts Java tasks through the C++ scheduler (the ones posted via ThreadUtils.java at least). In subsequent patches, we will have to track down other UI thread tasks and post them via the scheduler.





NB the framework tasks are also java but without OS changes they are (probably) beyond our control. To get an idea of how bad this is, I took a startup trace of bbc.co.uk on a Nokia TA-1060. I observed 68 java tasks, of which 48 went through the scheduler (~65%). Timewise 10173.658ms was spent running java tasks and of that 8066.541ms (~79%) was for tasks under scheduler control.

On all platforms we intend to select and run a single UI thread task at a time before yielding (i.e. the work batch size will be 1).   It's not currently safe to hold the thread for an extended period because on android input and vsync events are delivered via the java framework. In addition, if you peg the thread for too long, the OS will display a message claiming the app became unresponsive. There are similar issues on other platforms too.

Experimentally we have observed many tasks running in the period up from startup to committing the first navigation. Based on this we propose to add a few initial task queues:

Name	**Notes**
BestEffort	Disabled during startup.
Bootstrap	Initially highest priority and then disabled after startup.
Default	Default priority.
GPU	Highest priority.
Loading	High priority.

When we Initially introduce the scheduler, all task queues will have default priority (i.e. it will be a pure FIFO like the current message loop). We will use finch feature flags to run some scheduling policy experiments (see below), to measure the impact of the proposed static priorities.

Startup will be defined the same way AfterStartupTaskUtils does using a WebContentsObserver on most platforms, and via a JNI call on Android. Eventually, we hope to replace AfterStartupTaskUtils. Rather than using randomized delayed tasks to spread out the load, we will use low_priority task queue(s). Tasks posted through AfterStartupTaskUtils onto the UI thread are trivial to refactor, others require more thought. Promises would help with migrating them.

We may also be able to replace content::StartupTaskRunner.

We will focus initially on Android startup performance, in particular for opening a URL. We will try to make sure tasks on the critical path are posted on the appropriate queue. We will do this initially by obtaining a task runner from the scheduler. In C++ we're extending task traits to allow PostTaskWithTraits to route the task to the relevant queue (described in more detail here). We propose to land similar APIs in java.

C++ API

In content/public/browser/browser_task_traits.h

```
 enum TaskType {
   DEFAULT,
   BOOTSTRAP,
   LOADING,
   GPU,

   // This identifier does not represent a task type.  Instead it counts the
   // number of well-known TaskTypes.  Insert new well-known TaskTypes before
   // this identifier.
   TaskType_COUNT
  };

  …
};

// E.g. to post a UI thread GPU task we can do this:
base::PostTaskWithTraits(
     FROM_HERE,
     {BrowserThread::UI, TaskType::GPU},
     base::BindOnce(...));

class CONTENT_EXPORT BrowserUIThreadScheduler {
 public:
  virtual ~BrowserUIThreadScheduler();

  // Sets the MessageLoop's default runner. Must be called before the
  // BrowserThreadImpls are created.
  void TakeOverUIThread();

  // Registers |browser_ui_thread_scheduler| to handle tasks posted on the UI.
  // For tests, prefer content::TestBrowserThreadBundle (ensures isolation).
  static void SetInstance(
      std::unique_ptr<BrowserUIThreadScheduler> browser_ui_thread_scheduler);

  // Retrieve the BrowserUIThreadScheduler set via SetInstance().
  static BrowserUIThreadScheduler* GetInstance();

 protected:
  BrowserUIThreadScheduler();

 private:
  DISALLOW_COPY_AND_ASSIGN(BrowserUIThreadScheduler);
```

```
};
```

Java API

For consistency we think it's a good idea for the Java API to be similar to the C++ APIs, so we intend to introduce the trio of TaskRunners with a subset of the C++API features (delayed tasks are probably not needed in Java code, likewise testing if we're on the current sequence) :

```
/**
 * A task queue that posts Java tasks onto the C++ browser scheduler, if loaded. Otherwise this
 * will be backed by an {@link android.os.Handler} or the java thread pool. The TaskQueue
 * interface provides no guarantee over the order of the thread upon which the task will be
 * executed.
 *
 * Very similar to {@link java.util.concurrent.Executor} but conforms to chromium terminology.
 */
public interface TaskRunner {
    /**
     * Posts a task to run immediately.
     *
     * @param task The task to be run immediately.
     */
    public void postTask(Runnable task);

    /**
     * Instructs the TaskRunner to initialize the native TaskRunner and migrate any tasks over to
     * it.
     */
    abstract void initNativeTaskRunner();
}

/**
 * Tasks posted will be run in order with respect to this sequence, but they may be
 * executed on arbitrary threads.  Ordering between multiple sequences is not
 * guaranteed.
 */
public interface SequencedTaskRunner extends TaskRunner {}

/**
 * Tasks posted will be run in order on a single thread. Ordering between multiple
 * sequences is not guaranteed. This will always be the UI thread.
 */
public interface SingleThreadTaskRunner extends SequencedTaskRunner {}
```

The various C++ postTaskWithTraits APIs let you post tasks on the UI & IO thread. E.g. to post a background priority you'd do something like this:

```
base::PostTaskWithTraits(
    FROM_HERE, {BrowserThread::UI, base::TaskPriority::BEST_EFFORT},
            base::BindOnce(MyTask));
```

We propose to support this from java too. Initial use cases include annotating the tasks that form the initial bootstrap up to navigation, and replacing usages of android.os.MessageQueue.IdleHandler with best effort tasks. We also anticipate AsyncTasks will be posted under the hood via this method.

```
In base:

       /**
        * Keep in sync with base/task/task_traits.h
        */
       public enum TaskPriority {
           /**
            * This will always be equal to the lowest priority available.
            */
           LOWEST(0),

           …

           /**
            * This will always be equal to the highest priority available.
            */
           HIGHEST(2);

           private TaskPriority(int id) {
               this.mId = id;
           }

           int id() {
               return mId;
           }

           private final int mId;
       }

       /**
        * A builder pattern is more consistent with Java style.  NB The only reason C++ uses a
        * bag of traits is because we were able to get the Traits object constructed at
        * compile time.
        */
       public class TaskTraits {
           public TaskTraits() {...}
           public TaskTraits with(TaskPriority taskPriority) {...}
           public TaskTraits boolean mayBlock();
       }

In content:
```

```java
/**
 * Keep in sync with BrowserThread::ID
 */
public enum BrowserThread {
     /**
      * The main thread in the browser.
      */
     UI((byte)0),

     /**
      * This is the thread that processes non-blocking IO.
      */
     IO((byte)1);

    private BrowserThread(byte id) {
        this.mId = id;
    }

    byte id() {
        return mId;
    }

    private final byte mId;
}

/**
 * Keep in sync with content/public/browser/browser_task_traits.h
 */
public enum TaskType {
    DEFAULT((byte)0),
    BOOTSTRAP((byte)1),
    LOADING((byte)2),
    GPU((byte)3);

    private TaskType(byte id) {
        this.mId = id;
    }

    byte id() {
        return mId;
    }

    private final byte mId;

}

public class ContentTaskTraits extends TaskTraits {
    public ContentTaskTraits() {...}
    public ContentTaskTraits with(BrowserThread browserThread) {...}
    public ContentTaskTraits with(TaskType taskType) {...}
}


/**
 * This is intended to replace ThreadUtils.java and it mirrors the C++ scheduler api.
 */
public class TaskScheduler {
    /**
     * @param traits The TaskTraits that describe the desired TaskRunner.
     * @return The TaskRunner for the specified TaskTraits.
     */
    public static TaskRunner createTaskRunner(TaskTraits traits) {  ...  }

    /**
     * @param traits The TaskTraits that describe the desired TaskRunner.
     * @return The TaskRunner for the specified TaskTraits.
     */
    public static SequencedTaskRunner createSequencedTaskRunner(TaskTraits traits) {
        ...
    }

    /**
     * @param traits The TaskTraits that describe the desired TaskRunner.
     * @return The TaskRunner for the specified TaskTraits.
     */
    public static SingleThreadTaskRunner createSingleThreadTaskRunner(
        TaskTraits traits) {
        ...
    }

    public static void postTask(TaskTraits traits, Runnable task) {
        postDelayedTask(traits, task, 0);
    }

    public static void postDelayedTask(
        TaskTraits traits, Runnable task, long delayMs)  {
        ...
    }

    ...

}
```

Example usage:

```
import org.chromium.base.task.TaskScheduler;
import org.chromium.base.task.TaskTraits;
import org.chromium.content_public.browser.BrowserThread;
import org.chromium.content_public.browser.ContentTraits;
import org.chromium.content_public.browser.TaskType;

TaskScheduler.postTask(new ContentTraits()
                           .with(BrowserThread.UI)
                           .with(TaskType.BOOTSTRAP),
                       new Runnable() {
     @Override
     public void run() {
         mNativeInitializationController.firstDrawComplete();
     }
 });
```

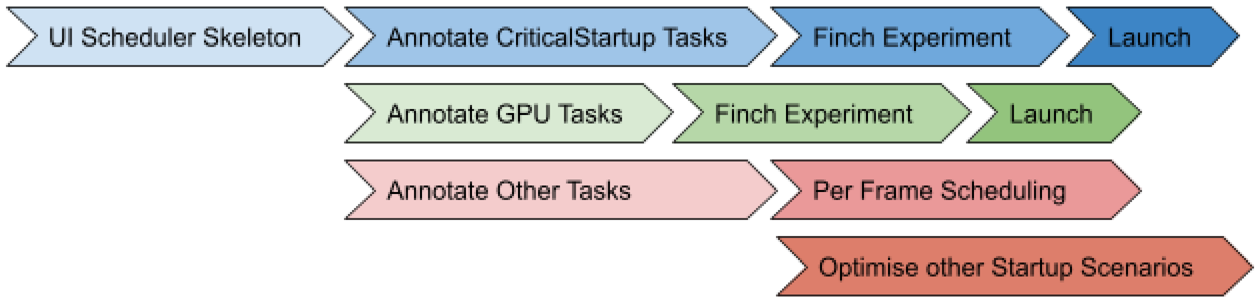
NB it is possible to post java tasks before the C++ library has been loaded so we need lightweight java scheduling in the interim built on top of the UI thread Handler, which transitions posted tasks over to the C++ scheduler when available.

Annotating net/ UI tasks

Ideally all net/ tasks posted on the UI thread would be annotated with BrowserThread::UI & BrowserThread::LOADING. This is a little tricky because of layering, net/ code shouldn't depend on content/.

An example is NetworkChangeNotifierDelegateAndroid::AddObserver, it's vital the PostTask done by the base::ObserverListThreadSafe gets marked with the right traits or the ordering assumptions of ChromeBrowserInitializer.initNetworkChangeNotifier and the subsequent pre-connect will break (leading to loading regression because the renderer will get erroneously killed and re-created).   In this instance it's simple to plumb a TaskRunner in from the net::NetworkChangeNotifier::Create() call in BrowserMainLoop::PostMainMessageLoopStart.

Implementation Timeline



Metrics

Success metrics

Prioritizing critical path startup tasks should result in a reduction in Startup.Android.Cold.* times UMA, particularly for lower end devices.

Prioritizing GPU tasks should result in improvements to the 99th percentile of Event.Latency.ScrollBegin.Touch.TimeToScrollUpdateSwapBegin2 and Event.Latency.ScrollUpdate.Touch.TimeToScrollUpdateSwapBegin2 on android.

Regression metrics

We will watch carefully to ensure there are no memory or performance regressions associated with the Finch trials.

Experiments

The default scheduling policy will be a pure FIFO (i.e. no change). We plan to use Finch to run two experiments with a view to making these policies permanent:

- Critical path startup prioritization
- GPU task prioritization

There's a caveat to running an experiment for the critical path startup prioritization, some of the very early java stages happen before finch is loaded. The work around is to write the value to shared pref and use it on next launch (see this).

TODO: Link names for the experiments when we've set them up

Rollout plan

Standard experiment-controlled rollout.

TODO: Link names for the experiments when we've set them up

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

Startup Scenarios
On android there are a number of startup scenarios, each with a different critical path.

- Opening new tab page
- Opening a URL
- Chrome custom tab
- Opening an installed PWA
- Handling a push notification
- Background downloading

To account for this, we will plumb through to the scheduler a StartupScenario enum, which will tell it what's going on and it will use that to manipulate task queue priorities accordingly. For non-Android platforms, we can inspect the result of DetermineStartupTabs in StartupBrowserCreatorImpl::DetermineURLsAndLaunch to determine the startup scenario. On Android, there doesn't appear to be a convenient central location, so we'll add some ad-hoc instrumentation in the various Activity classes e.g. in CustomTabActivity.   Intents can be sent to an already running Chrome (e.g. opening a custom tab) which means we need to be able to set the StartupScenario dynamically.

Java AsyncTasks
Because it will be possible to post java tasks through to the base::TaskScheduler, arguably we don't need the java threadpool to hang around after native libraries have been loaded. This represents an opportunity to reduce the number of threads and potentially save some memory as a consequence.

Scheduling system events

On ChromeOS & Linux system events such as input come in via libevent which is invoked by the message pump to polls various file descriptors (via epoll). Ideally while we have high priority work, we'd like to keep running tasks and not poll until something important (input or vsync) comes in. This likely entails modifying libevent to add a mode where it only runs high priority events.

There is a similar problem on android for framework tasks which among other things, deliver input and vsync events. We will consider landing changes to the Android OS to let us know when there are high priority events pending, so we can safely hold the thread for longer. Alternatively, we might consider timing-based metrics to infer this, because the vsync signal comes in at a regular interval and we suspect input does too (i.e. it's safe to hold the thread until shortly before we expect to receive a vsync).

On windows the WNDCLASSEX object specifies the WindowImpl::WndProc callback which handles events delivered by the system message loop (e.g. WM_MOUSEMOVE). We may be able to use the win32 PeakMessage API to determine if there's any high priority events pending, and yield accordingly.

Misc

We intend to introduce policies to optimize start up for the other start up scenarios where that makes sense.

Long term we'd like to minimise the number of tasks posted to the default queue, although experience with blink suggests this may be difficult.