Cooperative scheduling in Javascript

alexclarke@, rmcilroy@, skyostil@

Problem: It's difficult to perform Javascript work without introducing delays to other high priority tasks such as input processing and compositing. The main issue is that Javascript has limited knowledge about the priority and timing of the other work that shares the same thread. This document proposes a cooperative scheduling API which gives Javascript that information.

Proposed API

```
doWork = function(deadline) {
      while (performance.now() + expectedWorkUnitTime < deadline) {
            // Do work
      }
      if (hasMoreWork()) {
            requestIdleFrame(doWork);
      }
}

requestIdleFrame(doWork);
```

Ideally the work unit granularity should be at least on the order of milliseconds to reduce the overhead from checking the current time.

requestIdleFrame behaviour

The purpose of the requestEd Frame is to enable web pages t
o perform background work during the time when the ç Jank and dropped frames the web http://home.nextel.com.br/ should ensure that their requestIdleFrame callback completes by this deadline, however this is not enforced by the browser (i.e., developer can to shoot themselves in the foot by ignoring the deadline).

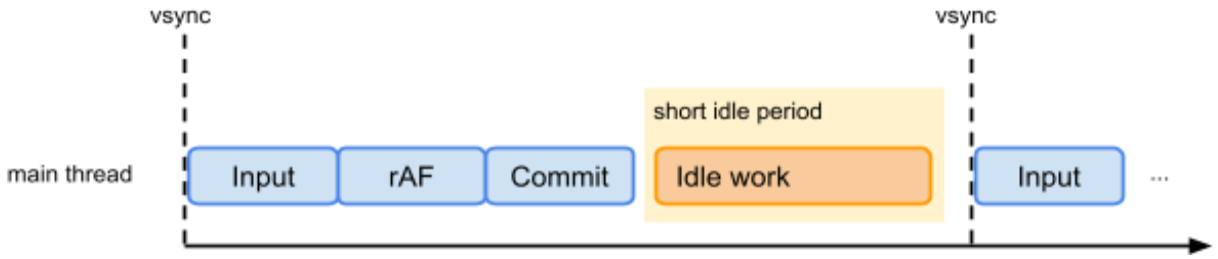
One important subtlety of requestIdleFrame is that the callback will not [Long idle time scenario]

---

happen on the current frame, but will only be called back on a subsequent frame. This is an important aspect of requestIdleFrame since a typical scenario will involve the callback checking whether it can do any work within the deadline and if not, re-requesting a requestIdleFrame and returning immediately (as in the example above). By preventing execution of the requestIdleFrame callback until the next frame, we avoid a *storm of reposts* during the remainder of a deadline (more details in the Blink Idle Task Scheduling design doc).

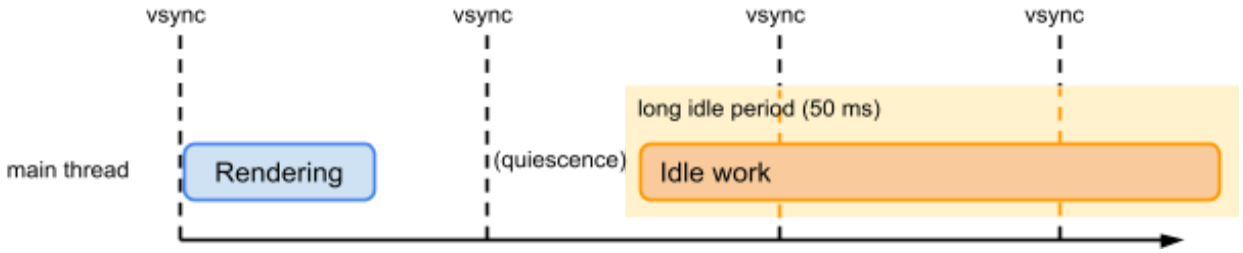
This is similar to a requestAnimationFrame, however the difference is that the requestIdleFrame callback will occur after a subsequent frame has been committed[1], not before. Unlike a requestForAnimation, there is no guarantee that a requestIdleFrame callback will be called on the next frame (or any frame) – if the browser determines it does not have any idle time available to service the requestIdleFrame callback before drawing the next frame. Similarly, while multiple requestIdleFrames can be posted in one frame, there is no guarantee that all of these callbacks will be serviced in a single frame – if the first one uses up its deadline then the subsequent callbacks will not be called until a later idle time.

Short idle time scenario



After input processing, rendering and compositing for a frame has been completed, the browser main thread often becomes idle until the next frame begins. To make productive use of this time, the browser will schedule execution of a requestIdleFrame callback when the idle period begins. The callback is given a deadline time – determined by the browser – by which it should return. As long as the callback respects this deadline, rendering and input processing remains smooth. If the callback returns before this deadline then the browser will continue running any previously scheduled requestIdleFrame callbacks until the deadline is used up or no more remain.

Long idle time scenario



The browser can detect when the system has become quiescent and give the requestIdleFrame callback more time to finish. The 50 ms time period – as defined by Midnight Train – ensures

[1] or after the start of a subsequent idle period for long idle time scenarios where no frames are being committed

that the browser can still respond to new user input with a reasonable delay. If no input or rendering requests occur during this time, then the browser will schedule another 50ms idle period subsequently until no requestIdleFrames callbacks remain registered. More details of the mechanism used to schedule longer idle times is available in the Long Idle Tasks design doc).

Advantage over setImmediate et al

Another way of implementing cooperative multitasking between JS and the browser, is to split all the JS work up into a sequence of short tasks and use setImmediate (or one of the polyfills e.g. post message) to post these tasks onto the browser's run loop. The problem with this is JS has no idea how long it's allowed to run for, so it relies on the browser to interleave other work between each callback. This means there's extra overhead from switching repeatedly between JS and C++ and because the browser has no idea how long a setImmediate task is likely to take, it will blindly run them just before the input deadline leading to janks.

Open issues

- Should these tasks be guaranteed to run or not?
    - No, because 1) the caller can trivially determine if a task was run or not (e.g., using a timer or polling in rAF), and 2) it's not clear what deadline should be given to the callback if we call it without any available idle time.
- Should we throttle the tasks?
    - No for foreground tabs, but yes for background tabs. This matches the behavior of setTimeout, requestAnimationFrame, etc.
- Should we use the performance.now() or Date timebase? What does rAF do?
    - rAF uses performance.now() as of Chrome 21, so let's do the same.
- Can we use the FrameTiming API to polyfill on other browsers?
- How should we prioritize between internal idle tasks (e.g., V8's idle GC tasks) and requestIdleFrame callbacks?

Alternative APIs considered

1. Returning the deadline in the requestAnimationFrame callback. The problem is we don't know ahead of time how long the commit is going to take.
2. var desiredDuration = 0.05;
   requestIdleFrame(desiredDuration, function(deadline) { … });
3. Having a query (e.g., shouldYieldForHighPriorityWork()) that tells Javascript whether it should return control back to the main loop. The main disadvantage is that this leads to (potentially expensive) polling. Also, Javascript still needs a way to know when the idle time starts.

Related Documents

- Blink Idle Task Scheduling
- Long Idle Tasks
- Midnight Train