panicker@google.com
in collaboration with jasonjmiller@google.com, domenic@google.com, surma@google.com
Aug 27, 2018
Status: draft

Motivation

A strategy for improving *guarantees* of user responsiveness is to free up the main thread by moving script work off main thread. (See: Off Main Thread Roadmap.) At the same time, improved scheduling *on* the main thread also helps responsiveness.

The purpose of this doc is to explore a scheduling API backed by a thread-pool that supports main thread and off main thread scheduling.

To promote moving script off-main-thread we need a thread-pool based API, as it is not reasonable for web developers to reason about cost of spinning up threads (one per dedicated worker) and utilizing them effectively.
A Scheduling API enables us to:

- provide more ergonomic way to offload work from main-thread (compared to dedicated worker)
- maintain control over management of threads and decisions on scheduling of tasks on them
- encourage developers to think carefully about how to structure their app and which work to move off main-thread
- fill gaps in scheduling work **on** the main thread
- enable indicating semantic priority to help the system with scheduling

The proposal is inspired from **iOS' GCD aka Grand Central Dispatch**, as it is a proven model that addresses the above goals. Lessons we learned from iOS and GCD are captured in this doc.
A centerpiece of GCD design is abstracting away threads and having developers think of posting async tasks which may (or may not) run on different thread. While the web has good primitives for async (async / await, promises), it is important for an effective Scheduling API to support async and sync work, both on and off the main thread.

Use-cases | Example Apps

- On main thread: smart pen app: As the user draws on a canvas, gestures, letters & words are recognized. Enqueues constant stream of "input" tasks competing with a constant stream of "render" tasks (recognizer results) and some intermittent "background" tasks (recognition kick-off and possibly reanalysis as drawing context increases).

- Off main thread: Search app updates results as-you-type: enqueues tasks off main thread to fetch results, post-processes them to generate result set. When a final (high confidence) result set is obtained for a search, it enqueues a main thread tasks to display results.

Requirements

- support for posting work on and off the main thread
- support for “concurrent execution” of posted tasks: tasks start in the order they were posted (so that no task gets indefinitely postponed), but may run concurrently (i.e. may not finish in the order they were posted)
- support for “serial execution” i.e. guarantees of tasks executing serially one after another.
- enable the system (i.e., not the web developer) to maintain control over thread management:
    - creating / removing threads
    - sizing / resizing of thread-pool
    - decision on whether posted tasks will run off or on main thread (when off main thread is requested)
    - ability to penalize / lower the priority of queues with misbehaving tasks
    - encourage cooperative scheduling
- API must require developer to explicitly specify thread / queue:
    - for posting results back (otherwise thread hops at arbitrary times can add latency)
    - for current and any subsequent tasks
- semantic priority to aid scheduling

Non-goals

- fully replace existing scheduling APIs like rIC, rAF, setTimeout, setInterval.

NOTE: We may build a higher level JS library as a “canonical scheduler” for the app that in turn uses rIC, rAF, setTimeout, this proposed API etc.

API sketch

The following API sketch is heavily inspired by iOS’ GCD Dispatch Queues; it also takes inspiration from Justin Fagnani’s queue scheduler API.

Semantic priority for queue

Semantic priority i.e. `enum TaskQueuePriority` can be one of these:

- `"user-blocking"`
- `"user-visible" (similar priority as rAF)`

- `"default"`
- `"background" (similar priority as rIC)`

NOTE: These match up with GCD and somewhat match our own internal TaskTraits.

Serial & Concurrent Task queues

Serial task queue: Tasks are guaranteed to start and finish in the order submitted, i.e. a task does not start until the previous task has completed.

Concurrent task queue: Tasks are guaranteed to start in the order they were posted (so that no task gets indefinitely postponed), but may run concurrently i.e. may not finish in the order they were posted.

User defined Serial Task queues

User can define their own serial task queues on “scheduler”, which is on the window object:

```
myQueue = new TaskQueue(‘myCustomQueue’, "input");

const result = await myQueue.postTask(task, <list of args for task>);
```

where `task` is a function.
`args` will be structured cloned, by default.

NOTE: We need new syntax here to prevent double-parsing of the function -- both on & off main thread. We may need syntax for capturing, for example see JS blocks.

See this section on considerations for specifying where & when results are posted.

`scheduler.getTaskQueue(‘myCustomQueue’)` returns the previously created queue.

TODO: Add syntax for using modules.

Global Task queues

A set of global (default) concurrent task queues will be made available to post work on to the main thread. There will be a global queue for each priority level.
In addition, a default global serial task queue will be available for posting on the main thread.

`TaskQueue.default("input")` returns the global concurrent task queue for posting to main thread.

TODO: API considerations

Returning Results from Tasks

Default behavior of posting result back to main thread at arbitrary times is problematic -- as it could result in latency, and unexpected user experience.

Posting subsequent tasks

Default posting of subsequent tasks to the same / current thread is problematic -- instead this should be explicitly specified.

Handling cancellation & changing priority of posted tasks

TODO: can the queue priority be later updated, after queue creation?

Yielding to support cooperative scheduling

TODO: scheduler.yield()

Other Considerations

Access to Shared Data in a task queue

On the web, we have structured cloning, and no shared memory yet (lack of shared memory also takes away a class of thread safety concerns).
How could tasks within a task-queue share data without shared memory access?
This could be accomplished in a couple ways:
- with one isolate and one JS context per thread: a task queue could be pinned to a thread and can access the (shared) JS context across tasks; later the isolate could be migrated to another thread as needed
- with one isolate and multiple JS contexts -- one context per task-queue: a task queue would be pinned to a thread and have its own (not shared) context, but them it's hard to move it across threads.
- waiting for something like Typed Object for data sharing.

NOTE: GCD queues allow associating custom context data and have "finalizer" for custom clean up..

Scope, Isolate & JS Context

Each workerpool thread will have a scope, and an isolate.
See above comment on task queue to JS context mapping.

TODO: what (extent of) problems can arise from access of global state (and overhead of breaking dependence - if needed); would this be exacerbated from wide usage from script, including external libraries and 3P content?

Background: Worklet is backed by a (or N) global-scope (opposite of dedicated worker which is not available globally) which are spun up / down (in the case of worklet, this is done proactively to break dependence on global state.

Code importing & Module map sharing

Threads in the threadpool would share a module map (similar to worklets).
While all posted tasks (running off thread) adding up to a single global module map seems odd, putting something in the module map generally does not interfere with other code. It just makes the module cached for future imports.
There are exceptions when you use side-effecting modules, like polyfills. But those are rare, and at least in the polyfill case sharing them should be OK. So global module map for the threadpool is likely OK.
Note: worklets don't support dynamic import() because worklets require all worklet global scopes import the same modules in order.

Mitigating multiple thread hops + latency

Ideally the API will not make it easy for developers to create multiple thread hops -- which can stagger across frames and become a major source of user latency. This is a known issue with Android's AsyncTask for instance, as it makes it too easy to post to main thread (default behavior) too often and at arbitrary times.
Ideas:

- [panicker] developer explicitly indicates where (which thread / queue) the results are posted on
- [skyostil] return finished results once per frame
- [ojan] If we add the ability to deliver tasks batched in some way (e.g. once per frame), we could make that optional for worker threads and required for the main thread as a way of forcing app developers to avoid many round trips for UI work.
- consider something like Framework (TikTok) Android's use of sequential executor for task queues on UI: add task queues and flush before yielding UI thread ?

Capabilities

We likely need to support most existing capabilities in worker, as well consider:

- which ones should not be supported (any blocking / sync work will become a bottleneck)
- what additions are needed.

Restrictions

We may need to add certain restrictions from the get-go, as it will be nearly impossible to add them later.
Maybe rules like:

- having an upper bound on thread hops based on priority level and system knowledge.
- autonomy to NOT run task off thread
- any sticks for penalizing misbehaving / long tasks?

Performance

Having quantifiable data on performance, at least for best-case scenarios, is critical.
We are starting to look at benchmarking here.

FAQ

Why not just build a JS library instead of platform primitive?

There are gaps in the platform that make a purely JS polyfill quite lacking:

Off Main Thread Gaps:

1. System controlled thread management: developers (both 1P and 3P) should be able to carve out and schedule work off main thread without doing their own thread management, instead they should use a system managed threadpool that can coordinate shared data for task queues. Also every app component, included library, 3P embed or Ad shouldn't create and manage its own threads, which is the default behavior with using worker API.
2. Handling task cancellation: a JS library would have to spin entire worker up / down to cancel posted tasks, while a platform primitive could do this efficiently by relaying messages to the worker.
3. Double parsing of functions in posted tasks
4. Ergonomics of posting tasks

NOTE: Something like JS blocks would tackle #3 and #4 but not #1 and #2.

On Main Thread Gaps:

- Prioritizing against browser internal work and network fetching
- Lack of task priority i.e. ability to schedule script chunks at different priorities
- Support for cooperative scheduling via yield() (TODO: clarify relationship with shouldYield()): "await yield" in JS causes a microtask to be queued
    - workarounds used today include: postmessage after each rAF,