# Cooperative Scheduling in Blink

haraken@chromium.org
2017 Jun 5
**Status: PUBLIC**

**Summary:** This document proposes to introduce cooperative scheduling to Blink's architecture. The cooperative scheduling will massively help improve the responsiveness and schedulability of the rendering engine in many ways. The cooperative scheduling will enable us to 1) yield long-running JavaScript in third-party iframes, 2) host multiple workers on one physical thread and 3) host many tabs within a limited number of OS processes without causing big janks.



# Motivation

Cooperative scheduling is a well-established task scheduling technology to improve the responsiveness of the system. Introducing the cooperative scheduling to Blink will massively help improve the responsiveness of the rendering engine, which matters for users :D

There are several key use cases for the cooperative scheduling in Blink.

## [Use case 1] Yielding long-running JavaScript on third-party iframes

Third-party iframes (mostly Ads) can cause a big jank on a top-level frame. Third-party iframes may run synchronous document.write, which blocks the main thread execution until the script resource is fetched from the network and completely loaded into the page. Third-party iframes may run JavaScript that takes a couple of seconds. Third-party iframes may load many cross-origin iframes; e.g., theverge.com loads 70 different origins (not including sub-origins). The top-level frame has to be blocked while the third-party iframes are doing work. The top-level frame cannot handle scrolling, touch events or any other user interactions.

The cooperative scheduling will remove the jank by **yielding a long-running task of the third-party iframe and cooperatively scheduling a task of the top-level frame**. This will have a high user impact.

(Note: This problem can be solved by Top Document Isolation as well. If you're interested in the relationship between TDI and the cooperative scheduling, see the Discussion section.)

## [Use case 2] Hosting multiple workers on one physical thread

Modern websites rely more and more on worker support; e.g., offline functionality and push messaging requires usage of a ServiceWorker. At the same time, Blink developers are working on new features that make use of workers, e.g. AudioWorklet, PaintWorklet. Blink has a plan to support nested workers in HTML5.

We also need a way to offload more work from ServiceWorker. ServiceWorker needs to stay responsive to not block the foreground app, so it cannot do any heavy work itself such as compiling and executing JavaScript. A natural way to address this would be to support nested workers for ServiceWorker so that ServiceWorker can offload heavy work to the nested workers.

All of these efforts will increase # of workers in the system. The problem is, however, that if we create one thread and one V8 isolate for each worker, it will add 20 MB per worker to the memory footprint of the website. This constant overhead makes it hard to create many workers in the system.

The cooperative scheduling will solve this problem by enabling Blink to **host multiple workers on one physical thread and one V8 isolate**. The workers can be yielded and cooperatively scheduled with each other on the same physical thread.

## [Use case 3] Hosting many tabs within a limited number of OS processes

Mozilla has introduced the cooperative scheduling in the Quantum project. Let me quote a couple of sentences from Bill's blog:

> *Unfortunately, increasing the number of content processes also increases memory usage. In some preliminary experiments we've done, it looks unlikely that we'll be able to increase the number of content processes beyond 8 in the near future; any more processes will increase memory usage unacceptably. Eight content processes are certainly better than one, but many users keep more than 8 tabs open. Mozilla developers are working to reduce the memory overhead of a content process, but we will never be able to reduce the overhead to zero. So we're exploring some new ideas, such as cooperative scheduling for tabs and tab freezing.*
>
> *As an addendum, I would be remiss if I didn't point out that Opera pioneered a cooperatively scheduled architecture. They did all the good stuff first.*

As quoted above, the cooperative scheduling enables us to **host any number of tabs within a limited number of OS processes while avoiding big janks**. This is because the cooperative scheduling can yield a long-running task of one tab and cooperatively schedule a task of another tab.

Some people say that Chrome creates too many processes. Creating more processes is good for security and performance but not good for memory. The good balance point varies depending on the resource constraints of a given system. What matters here is to **make Chrome work with any number of processes without causing big janks and thus make the whole system adaptable to a broad spectrum of devices from high-end to low-end**. Here the cooperative scheduling helps.

# Design

This section explains how the cooperative scheduling works for [Use case 1]; i.e., how to yield long-running JavaScript of third-party iframes. It's easy to extend the design for [Use case 2] and [Use case 3] as discussed below.

## Overview

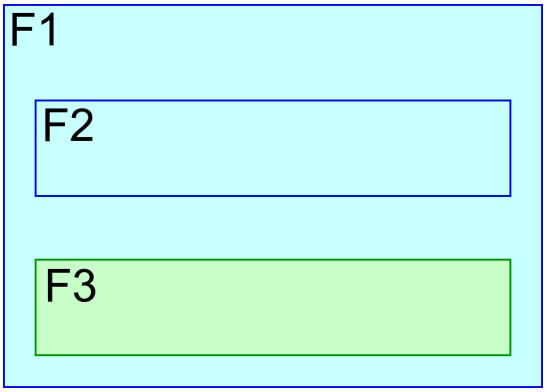


Fig.1 A web page that has three frames

As illustrated in Fig.1, imagine a web page that has a top-level frame (F1) and two iframes (F2, F3). F1 and F2 are from origin A. F3 is from origin B (e.g., F3 is Ads).

Almost all tasks in Blink are associated with a frame and posted to a per-frame scheduler. Let FS1, FS2 and FS3 denote the per-frame scheduler of F1, F2 and F3, respectively. A task of FS1 is guaranteed to touch only things of origin A (i.e., the task cannot touch things of origin B).

On the other hand, there are some tasks that cannot be associated with any frame for some reason. These tasks are posted to a default scheduler (DS). Tasks of the default scheduler may touch things of all origins.

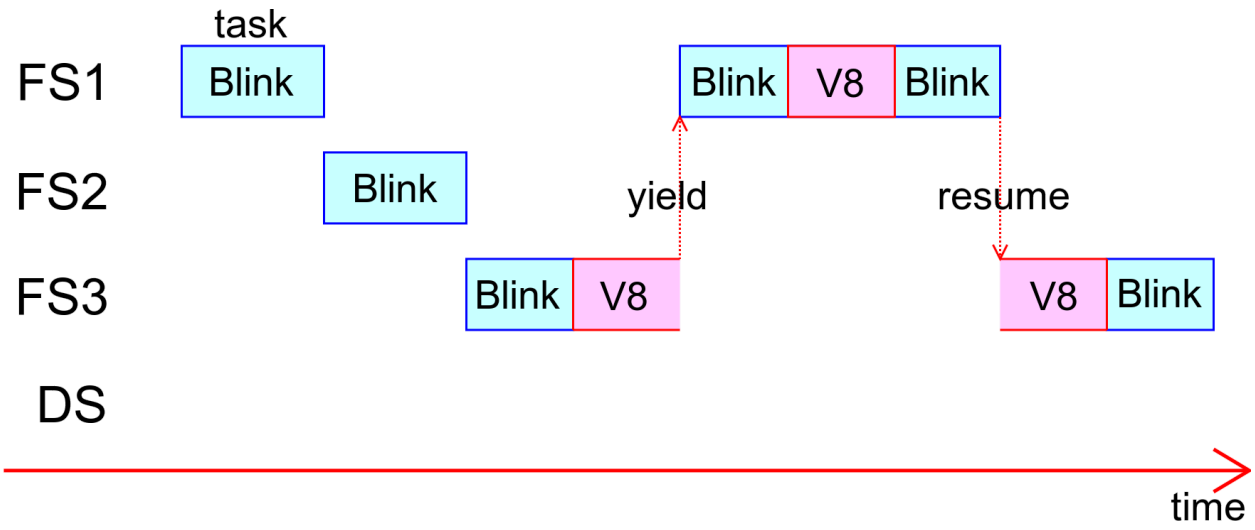


Fig.2 How the cooperative scheduling works

The cooperative scheduling works as follows:

- At a system level, all tasks in FS1, FS2, FS3 and DS run on the same physical thread (=the main thread) sequentially.
- When a task of FS3 started running a long-running JavaScript, we yield the task as follows:
    1. Yield the task of FS3 at a ***safe point*** (explained below).
    2. Cooperatively schedule a task of FS1.
    3. When the task of FS1 finished, resume the task of FS3 from the safe point.

If the task of FS1 is long-running, the task may also be yielded between step 2 and 3.

## Task scheduling

There is an important constraint about the task scheduling order. For example, we are not allowed to yield a task of FS1 and then schedule a task of FS2. To understand the constraint, consider the following example:

```
var foo = document.getElementById("foo");
var id = foo.id;
…;  // Imagine that this script execution is yielded here and a script that modifies foo.id is scheduled
assert(id === foo.id);  // Then this invariant will break.
```

We should not schedule a task from one origin while another task from that origin is yielded. In other words, we have to schedule tasks with the following constraint:

- **A task X can be scheduled only after all tasks from the same origin have been completed.**

Specifically, this means the followings:

- When we yield a task of FS1, we are not allowed to schedule a task of FS1, FS2 or DS.
- When we yield a task of FS2, we are not allowed to schedule a task of FS1, FS2 or DS.
- When we yield a task of FS3, we are not allowed to schedule a task of FS3 or DS.
- When we yield a task of DS, we are not allowed to schedule any task.

This means that **the cooperative scheduling is a technology to yield cross-origin tasks**. It is NOT a technology to yield same-origin tasks (it is not allowed for correctness). For example, it's not possible to yield a task of the top-level frame and then schedule another task of the top-level frame.

## Safe points

A safe point is a place where Blink may run any cross-origin task.

A good news is that there are already many places we can insert the safe point easily. For example, we can insert the safe point to places where V8 calls back Blink (e.g., DOM attribute getters/setters, DOM operations etc). This is because V8 is already implemented with an assumption that Blink may do anything (may invoke a cross-origin task via a nested message loop).

However, it's not guaranteed that V8 calls back Blink periodically. So we need to change V8 so that V8 calls back Blink periodically (e.g., every 16 ms) and enter the safe point.

## No big change is needed to V8

Another interesting fact is that **the cooperative scheduling won't require any substantial change to V8**. V8 is already implemented in a re-entrant manner, which supports the following execution sequence:

> V8 => Blink => V8 => Blink => V8 => …

Since V8 is already implemented with an assumption that Blink may do anything (may invoke JavaScript that touches cross-origin things via a nested message loop), the cooperative scheduling will "just work" from the V8 perspective.

V8 execution relies on v8::Isolate, which is stored on a thread-local storage. This will also "just work". From the V8 perspective, the cooperative scheduling is not that different from a case where multiple tabs are hosted in one renderer process (where one v8::Isolate is shared among multiple main frames from different origins).

## Global variables and thread-local storage

The current Blink code base is heavily using global variables and thread-local storages. This will cause a problem with the cooperative scheduling if the following scenario happens (Fig.3):

1. A task of FS3 stores a global state on a global variable.
2. The task of FS3 is yielded.
3. A task of FS1 is scheduled. The task manipulates the global state. The task finishes.
4. The task of FS3 is resumed. The task ends up with seeing the manipulated global state.

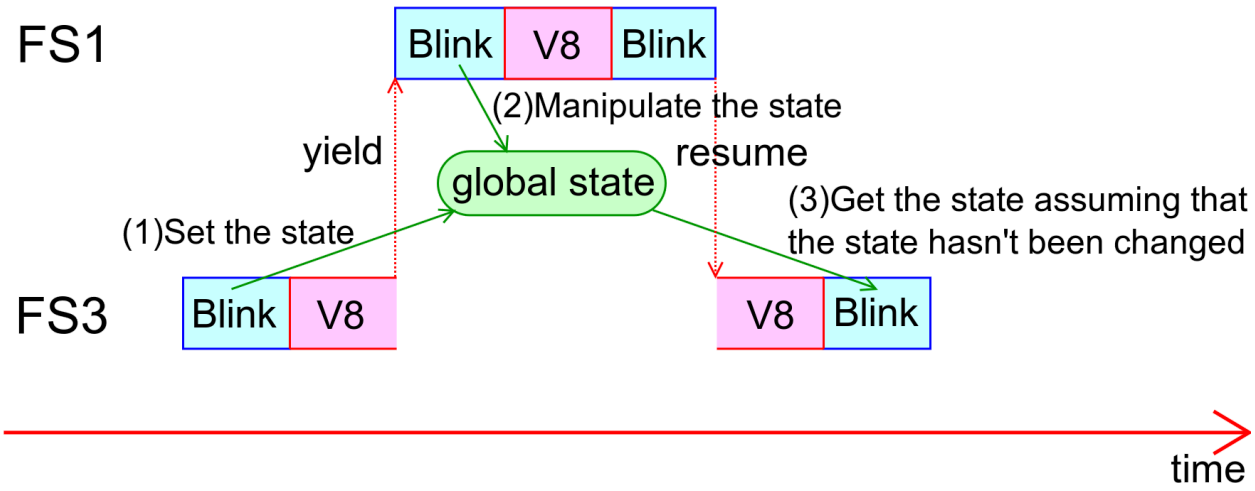


Fig.3 A problematic scenario where two cross-origin tasks share one global state

This scenario can happen if we have the following code like this:

```
void taskRun() {  // A task of FS3
  State state = …;
  setGlobalState(state);
  invokeV8();  // The V8 execution may be yielded. A task of FS1 may
run and manipulate the global state.
  assert(state == getGlobalState());  // Then this invariant will
break...
}
```

However, if this happens, it is already a security bug before the cooperative scheduling being introduced. Remember that when Blink calls V8, Blink must assume that V8 may do anything (e.g., V8 may call back Blink and run a cross-origin task via a nested message loop). It is already wrong to write code that assumes that invokeV8() won't mutate the global state. Blink should not have such code in the first place.

For example, the AtomicString table (ThreadSpecific<>) will "just work" because all tasks that use the AtomicString table run on the same physical thread. Most global variables (DEFINE_STATIC_LOCAL) are used just to cache data per renderer process, and they will "just work" as well. Remember that this situation is the same as a case where multiple tabs (from cross origins) are hosted in one renderer process.

For the same reason, we don't need to worry about machine stacks. We don't need to switch machine stacks when yielding a task and scheduling another task from a different origin.

## Synchronization between cross-origin iframes

You may wonder if in terms of correctness it's really okay to yield a task of FS3 and cooperatively schedule a task of FS1. What happens if the task of FS1 triggers a layout which resizes the frame F3 while the task of FS3 is being yielded? When should we run the layout for the frame F3? How should operations between cross-origin iframes be synchronized?

The spec is saying that cross-origin iframes may work concurrently. This is why OOPIF can host cross-origin iframes in different processes. The fact that OOPIF is working means that we already have an implementation that allows cross-origin iframes to work concurrently (with some asynchronous communications between the cross-origin iframes via a browser process). **Assuming that OOPIF is working, the cooperative scheduling should just work** from the perspective of synchronizing operations between cross-origin iframes.

## For [Use case 2] and [Use case 3]

Until now we have looked at how the cooperative scheduling works for [Use case 1]. It's not that hard to generalize the design for [Use case 2] and [Use case 3].

Regarding [Use case 2], the key fact is that workers can run concurrently. This means that we just need to yield one worker and cooperatively schedule another worker on one physical thread (just like we yield cross-origin iframes in [Use case 1]).

Regarding [Use case 3], this is essentially the same as [Use case 1]. We can just host many tabs in one renderer process and run the cooperative scheduling for all frames in those tabs.

## Summary

In summary, there are a couple of key requirements to make the cooperative scheduling work:

- **OOPIF is working (i.e., cross-origin iframes can run concurrently)**
- **V8 is implemented with an assumption that when V8 calls back Blink, Blink may run a cross-origin task**
- **Blink is implemented with an assumption that when Blink calls back V8, V8 may run a cross-origin task**

These requirements should already be guaranteed in the current code base. (If the second or third requirement is not guaranteed, it's a serious security bug already before talking about the cooperative scheduling.)

Assuming that the requirements are guaranteed, the only substantial changes needed to make the cooperative scheduling work would be:

- **Introducing the safe points.**
- **Introducing the task scheduler that schedules tasks with the ordering constraint.**

That's it! Of course the devils will be in the details :) However, at least in theory it won't be unrealistic to make the cooperative scheduling work.

# Discussion

## Cooperative scheduling vs. TDI vs. OOPIF

What are the differences between the cooperative scheduling, TDI (Top Document Isolation) and OOPIF (Out-Of-Process Iframe)? This is a very good question.

A short explanation would be:

- **OOPIF is a technology to improve security**. OOPIF creates new processes for cross-origin iframes and isolates memory address spaces between the iframes.
- **TDI is a technology to improve responsiveness**. TDI creates a new dedicated process and puts all third-party iframes (of all tabs) in the TDI process. This enables the top-level frame to run on a different process from the third-party iframes and thus improves the responsiveness of the top-level frame.
- **The cooperative scheduling is a technology to improve responsiveness in a more lightweight manner than TDI**. TDI can be heavy because it needs to create a new process. Also **the cooperative scheduling is a more general solution to improve responsiveness than TDI**. Whereas TDI can just isolate performance between the top-level frame and third-party iframes, the cooperative scheduling enables more fine-grained scheduling such as [Use case 1], [Use case 2] and [Use case 3].

TDI is already close to shipping, so we should focus on shipping TDI first. Also the fact that TDI (or OOPIF) is working is a requirement to make the cooperative scheduling work (as described above). From my viewpoint, the cooperative scheduling is an optimized version of TDI in a sense that it is a more lightweight and powerful solution to improve the responsiveness of the rendering engine. I propose investing on the cooperative scheduling after we ship TDI.

On the other hand, OOPIF is a different thing. OOPIF is a technology to improve security.

A longer explanation is described in this document.

## Won't the cooperative scheduling open new security holes?

In theory, no. As mentioned above, the cooperative scheduling won't open any new security holes as long as the following conditions are guaranteed:

- V8 is implemented with an assumption that when V8 calls back Blink, Blink may run a cross-origin task
- Blink is implemented with an assumption that when Blink calls back V8, V8 may run a cross-origin task

If the conditions are not guaranteed, it's already a security bug in the current code base. That said, it's possible that the cooperative scheduling exposes existing security bugs. We should probably implement some verification to detect cases where the conditions are violated.