Task Attribution V2

shaseley@
2024-01-31

NOTE: THIS DOCUMENT IS SHARED PUBLICLY

Task Attribution, Currently

Task attribution currently works by associating an ID with each *JavaScript task* (insert hand wavy definition) and constructing a task graph, with edges representing a parent/child relationship.

For example, the following snippet

```
JavaScript
setTimeout(() => {
  doSomething();
  setTimeout(() => {
    doSomethingElse();
  });
});
```

gives us a graph that looks something like: [root taskId: 1] → [setTimeout: taskId 2] → [setTimeout: taskId 3].

Internally in Blink, the task attribution API lets you get the task state of the running task, which helps determine causality. For example, if we want to know if a DOM modification happened as the result of a click event, we could:

1. Record each task created during the click (i.e. the task associated with each event handler)
2. Check during DOM modifications if the running task is a descendant of a task recorded in (1)

Microtasks are considered to be continuations of an existing task, and the current task's ID is captured when creating a *promise reaction* (e.g when a promise is `awaited`).

Push-Based Task Attribution

An alternative to *task graph-based attribution* is *push-based attribution*, which works by propagating state to descendant tasks and microtasks.

In the example above, we could determine causality by propagating state through the task graph:

1. Create a *ClickContext* object to associate with click event task
2. Propagate the context from (1) to any tasks (and their descendents) scheduled during the click
3. During DOM modifications, check the propagated state of the running task for the ClickContext from (1)

This has the following benefits:

A. Reduces overhead
    a. We only need to do task attribution/propagation when there's something to propagate (whereas now we always create an object for the task, in case it's needed)
    b. We only need to keep alive what's being propagated, not the whole task graph (see also Task Attribution: Using Task Partitions)
B. Reduces complexity / simplifies code
    a. The blink side of propagation aligns with the V8/promise continuation side (see the section on CPED)
    b. This should simplify the client side: task/ancestry tracking is replaced by reading propagated state
C. This is more closely aligned with AsyncContext, and might be able to leverage AsyncContext in the future (depending on implementation details)
    a. State to propagate == AsyncContext.variable
    b. Set up propagation in click == AsyncContext.run

Implementation

The underlying mechanisms that make task attribution work won't significantly change, specifically:

- CPED is used to store the current task state and propagate it to microtasks
- TaskAttributionTracker::TaskScope is still used to propagate task state

What to Propagate and When?

The underlying mechanism of task attribution (CPED) is shared by two use cases, which differ in the scope of propagation.

- The state associated with `SoftNavigationHeuristics`, which we'll call `SoftNavigationContext`
- The priority and abort signals for `scheduler.yield()`, which are *only propagated to continuations (microtasks)*

Both of these need to be considered since `TaskAttributionTracker` needs to support both. Here's how propagation should work for these:

API	`SoftNavigationContext` to propagate	Signals to propagate	Notes
Anywhere a `SoftNavigationEventScope` is created	The context associated with the scope	None	
`scheduler.postTask()` callback	Current	The task's signals	
`requestIdleCallback()` callback	Current	Fixed background priority signal	
`queueMicrotask()` callback	Current	Current	This is currently broken (the signals aren't propagated)
Other task scopes	Current	None	

Implementation Details

Types

The following is a rough sketch of the data structures for propagation. These are close to what we have now, but refactored to remove the task graph bits:

Class	Purpose	Replaces	Propagated to descendant tasks	Propagated to microtasks
`TaskState`	Interface for propagated task state.	N/A	N/A	N/A
`PropagatedTaskState`	Represents task state that is propagated across both tasks and microtasks. Holds soft navigation state.	`TaskAttributionInfo`	Yes (if exists)	Yes (if exists)
`ContinuationTaskState`	Represents state that is only propagated to microtasks. Holds `PropagatedTaskState` along with scheduling API signals.	`ScriptWrappableTaskState`	No	Yes (if exists)

The current task's associated `TaskState` is one of the following:

A. **Null**: the task did not descend from a task with anything to propagate or JS isn't currently running
B. **A `ContinuationTaskState` object**: the running task is a prioritized task callback or continuation thereof

C. **A `PropagatedTaskState` object**: all other propagation

Note: `TaskAttributionTracker` clients don't need to know this level of detail, since they'll interact with the `TaskState` interface.

(Some details elided for brevity)

```C/C++
// Interface for context that can be propagated to descendant tasks an
// microtasks. This could be either `PropagatedTaskState` or
// `ContinuationPropagatedTaskState`.
class TaskState : public GarbageCollected<TaskState> {
 public:
  ///// Data that propagated to tasks and microtasks /////
  virtual SoftNavigationContext* GetSoftNavigationContext() = 0;

  ///// Data that is only propagated to microtasks /////
  virtual AbortSignal* GetAbortSignal() = 0;
  virtual DOMTaskSignal* GetTaskSignal() = 0;

  // Returns `TaskState` for propagating to descendant tasks.
  virtual PropagatedTaskState* ToPropagatedTaskState() = 0;
};

// Data that is propagated to descendant tasks and microtasks.
class PropagatedTaskState : public TaskState {
 public:
  SoftNavigationContext* GetSoftNavigationContext() override {
    return soft_nav_context_.Get();
  }

  AbortSignal* GetAbortSignal() override { return nullptr; }
  DOMTaskSignal* GetTaskSignal() override { return nullptr; }

 private:
  Member<SoftNavigationContext> soft_nav_context_;
};

// `TaskState` that is only propagated to continuations (microtasks). This is
// composed of `PropagatedTaskState`, which is always propagated to microtasks
// and descenant tasks, and web scheduling API state, which is only propagated
// to microtasks.
class ContinuationPropagatedTaskState : public TaskState {
 public:
  SoftNavigationContext* GetSoftNavigationContext() override {
    return propagated_state_->GetSoftNavigationContext();
```

```
  }

  AbortSignal* GetAbortSignal() override {
   return abort_signal_.Get();
  }

  DOMTaskSignal* GetTaskSignal() override {
    return task_signal_.Get();
  }

  // This is called by the tracker to get the cross-task propagation bits.
  PropagatedTaskState* ToPropagatedTaskState() {
    return propagated_state_.Get();
  }

 private:
  Member<PropagatedTaskState> propagated_state_;
  Member<AbortSignal> abort_signal_;
  Member<DOMTaskSignal> task_signal_;
};
```

APIs

Then, the `TaskAttributionTracker` APIs are updated to interact with the new types:

```
C/C++
// This previously returned the current `TaskAttributionInfo`*; the equivalent
would
// be to return `PropagatedTaskState`, but we need to expose the full `TaskState`
to
// fix queueMicrotask().
TaskState* RunningTask();

// This new method is called anywhere we need to propagate task state to
// descendant tasks. The difference between this and RunningTask() is that
//
// CreateTaskScope() will be updated to take a `PropagatedTaskState`, and
// a new CreateContinuationTaskScope() will be added to take a `TaskState`,
// which will be used by `queueMicrotask()`.
PropagatedTaskState* GetPropagationTaskState();
```

And propagation will work as follows:

1. `TaskState*` is stored in CPED, and it represents the task state of the current running task

    a. **Note**: we previously stored `ScriptWrappableTaskState`, which is similar
2. The current CPED can be retrieved with `TaskAttributionTracker::RunningTask()`
    a. **Note**: this will be either `PropagatedTaskState` or `ContinuationPropagatedTaskState`, depending on where it originated
3. A new variant of `CreateTaskScope()` is added for soft navs, for setting the `SoftNavigationContext`
    a. **Note**: this creates a plain `PropagatedTaskState` object, per the table above
4. For callbacks, `CreateTaskScope()` is updated to take a `PropagatedTaskState`, which is retrieved with `TaskAttributionTracker::GetPropagationTaskState()`
    a. **Note**: the separate type here helps reduce risk of propagating the wrong thing
5. For `queueMicrotask()`, `CreateContinuationTaskScope()` (new) is used to create the task scope
    a. Note: this takes generic `TaskState*`, retrieved with `TaskAttributionTracker::RunningTask()`

Do `TaskScopes` Have to Change?

Mostly, no (aside from passing `PropagatedTaskState` when creating one).

One caveat is that CPED is not restored to its previous state after a microtask runs. This isn't a problem now because we create a `TaskScope` for each v8 callback (microtasks can potentially run between callbacks, e.g. firing multiple events when no JS is on the stack). We'd like this to be more efficient, but we'll likely still need to create a task scope here to ensure that the state is properly restored after a microtask checkpoint. But this can be optimized by:

1. avoiding an allocation (partition alloc) when creating the task scope by making scopes STACK_ALLOCATED (crbug.com/1464579)
2. avoiding setting the state when it doesn't change (get is cheaper than set)
3. avoid setting state when there's nothing to propagate and no previous task scope

How Will This Work With IPCs?

Both same-process `MessagePort.postMessage()` and same document navigations (`popstate` event) propagate the current task state. This is especially tricky because IPCs are involved, which requires special handling to ensure the associated task state remains alive in the renderer while waiting for an IPC. In both cases, this works by some variation of keeping a map of ID → `Member<TaskAttributionInfo>` in the render, and updating entries when needed.

We should be able to keep most of this the same as long as the ID sent via IPC maps to a `SoftNavigationContext` object or object holding one. The least invasive way of doing this is:

1. Store a `TaskAttributionId` on `PropagatedTaskState`
2. Change maps to ID → `Member<PropagatedTaskState>`
3. Make sure duplicates are properly handled
    a. **Note**: MessagePort already handles this; same-document navigation tracking might need to change slightly

Appendix

What is CPED and How Does It Work?

Continuation Preserved Embedder Data (CPED) is the mechanism for propagating task state to microtasks. It works as follows:

1. The current task state is stored in the `v8::Isolate`'s CPED (see `v8::Isolate::SetContinuationPreservedEmbedderData()`)
    a. Part of `TaskAttributionTrackerImpl`'s job is to manage this data
2. When a promise reaction is created in V8 (e.g. `.then()` on a promise is called or a promise is `awaited`), it stores the current CPED on the reaction object
3. When a microtask runs, it sets the `v8::Isolate`'s CPED to the stored CPED in the associated promise reaction
    a. This can then be read with `v8::Isolate::GetContinuationPreservedEmbedderData()`

See scheduler.yield() Inheritance for examples and more information.

How Does Task Attribution Handle IPCs?

Same-Document Navigations

Task Attribution sets the parent task for the the `popstate` event as the task that initiates the navigation as follows:

Initiating navigation

1. Get the current task during navigation API calls
    a. `history.go()`
    b. `navigation.traverseTo()`
2. Register the current task with the `TaskAttributionTracker` via `AddSameDocumentNavigationTask()`, keeping it alive through the navigation
3. Send a navigation IPC to the browser, passing along the current task's ID

Committing same-document navigation

In `DocumentLoader::UpdateForSameDocumentNavigation()`:

1. Get the `TaskAttributionInfo` corresponding to the ID passed the browser IPC
    a. This is retrieved using `TaskAttributionTracker` via `CommitSameDocumentNavigation()`
    b. This step also removes all tracked tasks from previous calls to `AddSameDocumentNavigationTask()` (but later ones, as there could be another navigation pending)
2. Set the `TaskAttributionInfo` in (1) as the parent task for the `popstate` event

Clearing tracked navigation tasks

The set of tracked tasks is also cleared in:

1. `DocumentLoader::CommitNavigation()` (cross-origin navigation)
2. `NavigationApi::InformAboutCanceledNavigation()`

`MessagePort.postMessage()`

Since `MessagePort.postMessage()` is often used as a (same-window) scheduling primitive as opposed to cross-thread/process communication, we propagate context across `MessagePort`s that haven't been transferred. But like navigation, this involves IPCs.

When `port1` sends a message to an untransferred `port2` via `port1.postMessage(...)`:

Sending side

1. Let `task` be the current `TaskAttributionInfo`
2. If we've haven't seen `task`, store a new map entry from `task`'s ID (`task`, counter=0)
3. Increment the map entry's counter for `task`'s ID by 1
4. Send the message along with `task`'s ID

Receiving side

1. Look up the message's task ID in the *sending port's* map and decrement the entry's counter, removing the entry if the counter goes to 0
2. Set the `TaskAttributionInfo` found in (1) as the parent task for the `message` event

Current Task-Attributable APIs

API	Parent capture	Propagate signals?	Notes/Special Handling
`setTimeout()/setInterval()` (with string)	Call time	☐	

setTimeout()/setInterval() (with callback)	Call time	☐	
postMessage() (same-window / not transferred)	Call time	☐	IPC
scheduler.postTask()	Call time	☑	
requestIdleCallback()	Call time	☑	
requestAnimationFrame()	Call time	☐	
queueMicrotask()	Call time	☐	
<script> execution	Construction time (PendingScript construction)	☐	
popstate event (same-document only)	Navigation start	☐	IPC
startViewTransition() (callback function)	Call time	☐	
XMLHttpRequest.send()	Call time		Propagated to events
All other callbacks, when there's no current running task	None	☐	This starts a new task chain when running a callback without JS on the stack, e.g. intersection observers, firing deferred events in a new task, etc.
Promises (e.g. `await fetch()`)	Continuation creation	☑	This uses continuation preserved embedder data (CPED), whereas everything else uses CreateTaskScope.