W3C Working Draft

Web Locks API

W3C First Public Working Draft, 5 January 2023

▼ **More details about this document**

This version:
https://www.w3.org/TR/2023/WD-web-locks-20230105/

Latest published version:
https://www.w3.org/TR/web-locks/

Editor's Draft:
https://w3c.github.io/web-locks/

History:
https://www.w3.org/standards/history/web-locks

Test Suite:
https://github.com/web-platform-tests/wpt/tree/master/web-locks

Feedback:
GitHub
Inline In Spec

Editors:
Joshua Bell (Google Inc.)
Kagami Rosylight (Mozilla)

Abstract

This document defines a web platform API that allows script to asynchronously acquire a lock over a resource, hold it while work is performed, then release it. While held, no other script in the origin can acquire a lock over the same resource. This allows contexts (windows, workers) within a web application to coordinate the usage of resources.

Status of this document

This section describes the status of this document at the time of its publication. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This document was published by the Web Applications Working Group as a Working Draft. This document is intended to become a W3C Recommendation.

This document was published by the Web Applications Working Group as a First Public Working Draft using the Recommendation track. Feedback and comments on this specification are welcome. Please use GitHub issues Historical discussions can be found in the public-webapps@w3.org archives.

Publication as a First Public Working Draft does not imply endorsement by W3C and its Members. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 2 November 2021 W3C Process Document.

Table of Contents

1. Introduction

This section is non-normative.

A lock request is made by script for a particular resource name and mode. A scheduling algorithm looks at the state of current and previous requests, and eventually grants a lock request. A lock is a granted request; it has a resource name and mode. It is represented as an object returned to script. As long as the lock is held it may prevent other lock requests from being granted (depending on the name and mode). A lock can be released by script, at which point it may allow other lock requests to be granted.

The API provides optional functionality that may be used as needed, including:

- returning values from the asynchronous task,
- shared and exclusive lock modes,
- conditional acquisition,
- diagnostics to query the state of locks, and
- an escape hatch to protect against deadlocks.

Cooperative coordination takes place within the scope of agents sharing a storage bucket; this may span multiple agent clusters.

> Agents roughly correspond to windows (tabs), iframes, and workers. Agent clusters correspond to independent processes in some user agent implementations.

1.1. Usage Overview

The API is used as follows:

1. The lock is requested.
2. Work is done while holding the lock in an asynchronous task.
3. The lock is automatically released when the task completes.

EXAMPLE 1 ¶

A basic example of the API usage is as follows:

```
navigator.locks.request('my_resource', async lock => {
   // The lock has been acquired.
   await do_something();
   await do_something_else();
   // Now the lock will be released.
});
```

Within an asynchronous function, the request itself can be awaited:

```
// Before requesting the lock.
await navigator.locks.request('my_resource', async lock => {
   // The lock has been acquired.
   await do_something();
   // Now the lock will be released.
});
// After the lock has been released
```

§ 1.2. Motivating Use Cases

A web-based document editor stores state in memory for fast access and persists changes (as a series of records) to a storage API such as the Indexed Database API for resiliency and offline use, and to a server for cross-device use. When the same document is opened for editing in two tabs the work must be coordinated across tabs, such as allowing only one tab to make changes to or synchronize the document at a time. This requires the tabs to coordinate on which will be actively making changes (and synchronizing the in-memory state with the storage API), knowing when the active tab goes away (navigated, closed, crashed) so that another tab can become active.

In a data synchronization service, a "primary tab" is designated. This tab is the only one that should be performing some operations (e.g. network sync, cleaning up queued data, etc). It holds a lock and never releases it. Other tabs can attempt to acquire the lock, and such attempts will be queued. If the "primary tab" crashes or is closed then one of the other tabs will get the lock and become the new primary.

The Indexed Database API defines a transaction model allowing shared read and exclusive write access across multiple named storage partitions within an origin. Exposing this concept as a primitive allows any Web Platform activity to be scheduled based on resource availability, for example allowing transactions to be composed for other storage types (such as Caches [Service-Workers]), across storage types, even across non-storage APIs (e.g. network fetches).

§ 2. Concepts

For the purposes of this specification:

- Separate user profiles within a browser are considered separate user agents.
- Every private mode browsing session is considered a separate user agent.

A user agent has a ***lock task queue*** which is the result of starting a new parallel queue.

The task source for steps enqueued below is the ***web locks tasks source***.

§ 2.1. Resources Names

A ***resource name*** is a JavaScript string chosen by the web application to represent an abstract resource.

A resource name has no external meaning beyond the scheduling algorithm, but is global across agents sharing a storage bucket. Web applications are free to use any resource naming scheme.

EXAMPLE 2 ¶

To mimic transaction locking over named stores within a named database in [IndexedDB-2], a script might compose resource names as:

```
encodeURIComponent(db_name) + '/' + encodeURIComponent(store_name)
```

Resource names starting with U+002D HYPHEN-MINUS (-) are reserved; requesting these will cause an exception.

§ 2.2. Lock Managers

A ***lock manager*** encapsulates the state of locks and lock requests. Each storage bucket includes one lock manager through an associated storage bottle for the Web Locks API.

> Note: Pages and workers (agents) sharing a storage bucket opened in the same user agent share a lock manager even if they are in unrelated browsing contexts.

To ***obtain a lock manager***, given an environment settings object *environment*, run these steps:

1. Let *map* be the result of obtaining a local storage bottle map given *environment* and "`web-locks`".
2. If *map* is failure, then return failure.
3. Let *bottle* be *map*'s associated storage bottle.
4. Return *bottle*'s associated lock manager.

> ISSUE 1 Refine the integration with [Storage] here, including how to get the lock manager properly from the given environment. ¶

§ 2.3. Modes and Scheduling

A ***mode*** is either "`exclusive`" or "`shared`". Modes can be used to model the common readers-writer lock pattern. If an "`exclusive`" lock is held, then no other locks with that name can be granted. If a "`shared`" lock is held, other "`shared`" locks with that name can be granted — but not any "`exclusive`" locks. The default mode in the API is "`exclusive`".

Additional properties may influence scheduling, such as timeouts, fairness, and so on.

§ 2.4. Locks

A ***lock*** represents exclusive access to a shared resource.

A lock has an ***agent*** which is an agent.

A lock has a ***clientId*** which is an opaque string.

A lock has a ***manager*** which is a lock manager.

A lock has a ***name*** which is a resource name.

A lock has a ***mode*** which is one of "`exclusive`" or "`shared`".

A lock has a ***waiting promise*** which is a Promise.

A lock has a ***released promise*** which is a Promise.

There are two promises associated with a lock's lifecycle:

- A promise provided either implicitly or explicitly by the callback when the lock is granted which determines how long the lock is held. When this promise settles, the lock is released. This is known as the lock's waiting promise.
- A promise returned by `LockManager`'s `request()` method that settles when the lock is released or the request is aborted. This is known as the lock's released promise.

```
const p1 = navigator.locks.request('resource', lock => {
  const p2 = new Promise(r => {
    // Logic to use lock and resolve promise...
  });
  return p2;
});
```

In the above example, `p1` is the released promise and `p2` is the waiting promise. Note that in most code the callback would be implemented as an `async` function and the returned promise would be implicit, as in the following example:

```
const p1 = navigator.locks.request('resource', async lock => {
  // Logic to use lock...
});
```

The waiting promise is not named in the above code, but is still present as the return value from the anonymous `async` callback. Further note that if the callback is not `async` and returns a non-promise, the return value is wrapped in a promise that is immediately resolved; the lock will be released in an upcoming microtask, and the released promise will also resolve in a subsequent microtask.

Each lock manager has a ***held lock set*** which is a set of locks.

When lock *lock*'s waiting promise settles (fulfills or rejects), enqueue the following steps on the lock task queue:

1. Release the lock *lock*.
2. Resolve *lock*'s released promise with *lock*'s waiting promise.

§ 2.5. Lock Requests

A ***lock request*** represents a pending request for a lock.

A lock request is a struct with items ***agent***, ***clientId***, ***manager***, ***name***, ***mode***, ***callback***, ***promise***, and ***signal***.

A ***lock request queue*** is a queue of lock requests.

Each lock manager has a ***lock request queue map***, which is a map of resource names to lock request queues.

To ***get the lock request queue*** from lock request queue map *queueMap* from resource name *name*, run these steps:

1. If *queueMap*[*name*] does not exist, set *queueMap*[*name*] to a new empty lock request queue.
2. Return *queueMap*[*name*].

A lock request *request* is said to be ***grantable*** if the following steps return true:

1. Let *manager* be *request*'s manager.
2. Let *queueMap* be *manager*'s lock request queue map.
3. Let *name* be *request*'s name.
4. Let *queue* be the result of getting the lock request queue from *queueMap* for *name*.
5. Let *held* be *manager*'s held lock set
6. Let *mode* be *request*'s mode
7. If *queue* is not empty and *request* is not the first item in *queue*, then return false.
8. If *mode* is "`exclusive`", then return true if no lock in *held* has name equal to *name*, and false otherwise.
9. Otherwise, *mode* is "`shared`"; return true if no lock in *held* has mode "`exclusive`" and has name equal to *name*, and false otherwise.

§ 2.6. Termination of Locks

Whenever the unloading document cleanup steps run with a document, terminate remaining locks and requests with its agent.

When an agent terminates, terminate remaining locks and requests with the agent.

> ISSUE 2 This is currently only for workers and is vaguely defined, since there is no normative way to run steps on worker termination. ¶

To ***terminate remaining locks and requests*** with *agent*, enqueue the following steps on the lock task queue:

1. For each lock request *request* with agent equal to *agent*:
    1. Abort the request *request*.
2. For each lock *lock* with agent equal to *agent*:
    1. Release the lock *lock*.

§ 3. API

§ 3.1. Navigator Mixins

```
[SecureContext]
interface mixin NavigatorLocks {
  readonly attribute LockManager locks;
};
Navigator includes NavigatorLocks;
WorkerNavigator includes NavigatorLocks;
```

Each environment settings object has a `LockManager` object.

The ***`locks`*** getter’s steps are to return this's relevant settings object's `LockManager` object.

§ 3.2. `LockManager` class

✔ MDN

```
[SecureContext, Exposed=(Window,Worker)]
interface LockManager {
  Promise<any> request(DOMString name,
                       LockGrantedCallback callback);
  Promise<any> request(DOMString name,
                       LockOptions options,
                       LockGrantedCallback callback);

  Promise<LockManagerSnapshot> query();
};

callback LockGrantedCallback = Promise<any> (Lock? lock);

enum LockMode { "shared", "exclusive" };

dictionary LockOptions {
  LockMode mode = "exclusive";
  boolean ifAvailable = false;
  boolean steal = false;
  AbortSignal signal;
};

dictionary LockManagerSnapshot {
  sequence<LockInfo> held;
  sequence<LockInfo> pending;
};

dictionary LockInfo {
  DOMString name;
  LockMode mode;
  DOMString clientId;
};
```

A `LockManager` instance allows script to make lock requests and query the state of the lock manager.

§ 3.2.1. The `request()` method

✔ MDN

FOR WEB DEVELOPERS (NON-NORMATIVE)

***promise* = navigator . locks . request(*name*, *callback*)**

***promise* = navigator . locks . request(*name*, *options*, *callback*)**

The `request()` method is called to request a lock.

The *name* (initial argument) is a resource name string.

The *callback* (final argument) is a callback function invoked with the `Lock` when granted. This is specified by script, and is usually an `async` function. The lock is held until the callback function completes. If a non-async callback function is passed in, then it is automatically wrapped in a promise that resolves immediately, so the lock is only held for the duration of the synchronous callback.

The returned *promise* resolves (or rejects) with the result of the callback after the lock is released, or rejects if the request is aborted.

Example:

```
try {
  const result = await navigator.locks.request('resource', async lock =>
    // The lock is held here.
    await do_something();
    await do_something_else();
    return "ok";
    // The lock will be released now.
  });
  // |result| has the return value of the callback.
} catch (ex) {
  // if the callback threw, it will be caught here.
}
```

The lock will be released when the callback exits for any reason — either when the code returns, or if it throws.

An *options* dictionary can be specified as a second argument; the *callback* argument is always last.

options . mode

The mode option can be "exclusive" (the default if not specified) or "shared". Multiple tabs/workers can hold a lock for the same resource in "shared" mode, but only one tab/worker can hold a lock for the resource in "exclusive" mode.

The most common use for this is to allow multiple readers to access a resource simultaneously but prevent changes. Once reader locks are released a single exclusive writer can acquire the lock to make changes, followed by another exclusive writer or more shared readers.

```
await navigator.locks.request('resource', {mode: 'shared'}, async lock =
  // Lock is held here. Other contexts might also hold the lock in share
  // but no other contexts will hold the lock in exclusive mode.
});
```

options . ifAvailable

If the ifAvailable option is `true`, then the lock is only granted if it can be without additional waiting. Note that this is still not *synchronous*; in many user agents this will require cross-process communication to see if the lock can be granted. If the lock cannot be granted, the callback is invoked with `null`. (Since this is expected, the request is *not* rejected.)

```
await navigator.locks.request('resource', {ifAvailable: true}, async loc
  if (!lock) {
    // Didn't get it. Maybe take appropriate action.
    return;
  }
  // Lock is held here.
});
```

options . signal

The signal option can be set to an AbortSignal. This allows aborting a lock request, for example if the request is not granted in a timely manner:

```
const controller = new AbortController();
setTimeout(() => controller.abort(), 200); // Wait at most 200ms.

try {
  await navigator.locks.request(
    'resource', {signal: controller.signal}, async lock => {
      // Lock is held here.
  });
  // Done with lock here.
} catch (ex) {
  // |ex| will be a DOMException with error name "AbortError" if timer t
}
```

If an abort is signalled before the lock is granted, then the request promise will reject with an AbortError. Once the lock has been granted, the signal is ignored.

options . steal

If the steal option is `true`, then any held locks for the resource will be released (and the released promise of such locks will resolve with AbortError), and the request will be granted, preempting any queued requests for it.

If a web application detects an unrecoverable state — for example, some coordination point like a Service Worker determines that a tab holding a lock is no longer responding — then it can "steal" a lock using this option.

Use the steal option with caution. When used, code previously holding a lock will now be executing without guarantees that it is the sole context with access to the resource. Similarly, the code that used the option has no guarantees that other contexts will not still be executing as if they have access to the abstract resource. It is intended for use by web applications that need to attempt recovery in the face of application and/or user-agent defects, where behavior is already unpredictable.

The ***request(*name, callback*)*** and ***request(*name, options, callback*)*** method steps are:

1. If *options* was not passed, then let *options* be a new `LockOptions` dictionary with default members.
2. Let *environment* be this's relevant settings object.
3. If *environment*'s relevant global object's associated Document is not fully active, then return a promise rejected with a "`InvalidStateError`" `DOMException`.
4. Let *manager* be the result of obtaining a lock manager given *environment*. If that returned failure, then return a promise rejected with a "`SecurityError`" `DOMException`.
5. If *name* starts with U+002D HYPHEN-MINUS (-), then return a promise rejected with a "`NotSupportedError`" `DOMException`.
6. If both *options*["`steal`"] and *options*["`ifAvailable`"] are true, then return a promise rejected with a "`NotSupportedError`" `DOMException`.
7. If *options*["`steal`"] is true and *options*["`mode`"] is not "`exclusive`", then return a promise rejected with a "`NotSupportedError`" `DOMException`.
8. If *options*["`signal`"] exists, and either of *options*["`steal`"] or *options*["`ifAvailable`"] is true, then return a promise rejected with a "`NotSupportedError`" `DOMException`.
9. If *options*["`signal`"] exists and is aborted, then return a promise rejected with *options*["`signal`"]'s abort reason.
10. Let *promise* be a new promise.
11. Request a lock with *promise*, the current agent, *environment*'s id, *manager*, *callback*, *name*, *options*["`mode`"], *options*["`ifAvailable`"], *options*["`steal`"], and *options*["`signal`"].
12. Return *promise*.

§ 3.2.2. The `query()` method

✔ MDN

FOR WEB DEVELOPERS (NON-NORMATIVE)

`state = await navigator . locks . query()`

The `query()` method can be used to produce a snapshot of the lock manager state for an origin, which allows a web application to introspect its usage of locks, for logging or debugging purposes.

The returned promise resolves to *state*, a plain-old-data structure (i.e. JSON-like data) with this form:

```
{
  held: [
    { name: "resource1", mode: "exclusive",
      clientId: "8b1e730c-7405-47db-9265-6ee7c73ac153" },
    { name: "resource2", mode: "shared",
      clientId: "8b1e730c-7405-47db-9265-6ee7c73ac153" },
    { name: "resource2", mode: "shared",
      clientId: "fad203a5-1f31-472b-a7f7-a3236a1f6d3b" },
  ],
  pending: [
    { name: "resource1", mode: "exclusive",
      clientId: "fad203a5-1f31-472b-a7f7-a3236a1f6d3b" },
    { name: "resource1", mode: "exclusive",
      clientId: "d341a5d0-1d8d-4224-be10-704d1ef92a15" },
  ]
}
```

The `clientId` field corresponds to a unique context (frame or worker), and is the same value returned by `Client`'s `id` attribute.

This data is just a *snapshot* of the lock manager state at some point in time. By the time the data is returned to script, the actual lock state might have changed.

The ***`query()`*** method steps are:

1. Let *environment* be this's relevant settings object.

2. If *environment*'s relevant global object's associated Document is not fully active, then return a promise rejected with a "`InvalidStateError`" `DOMException`.
3. Let *manager* be the result of obtaining a lock manager given *environment*. If that returned failure, then return a promise rejected with a "`SecurityError`" `DOMException`.
4. Let *promise* be a new promise.
5. Enqueue the steps to snapshot the lock state for *manager* with *promise* to the lock task queue.
6. Return *promise*.

3.3. `Lock` class


```
[SecureContext, Exposed=(Window,Worker)]
interface Lock {
  readonly attribute DOMString name;
  readonly attribute LockMode mode;
};
```

A `Lock` object has an associated lock.

The ***name*** getter's steps are to return the associated lock's name.

The ***mode*** getter's steps are to return the associated lock's mode.

4. Algorithms

4.1. Request a lock

To ***request a lock*** with *promise*, *agent*, *clientId*, *manager*, *callback*, *name*, *mode*, *ifAvailable*, *steal*, and *signal*:

1. Let *request* be a new lock request (*agent*, *clientId*, *manager*, *name*, *mode*, *callback*, *promise*, *signal*).
2. If *signal* is present, then add the algorithm signal to abort the request *request* with *signal* to *signal*.
3. Enqueue the following steps to the lock task queue:

    1. Let *queueMap* be *manager*'s lock request queue map.

2. Let *queue* be the result of getting the lock request queue from *queueMap* for *name*.
3. Let *held* be *manager*'s held lock set.
4. If *steal* is true, then run these steps:
    1. For each *lock* of *held*:
        1. If *lock*'s name is *name*, then run these steps:
            1. Remove lock from *held*.
            2. Reject *lock*'s released promise with an "`AbortError`" `DOMException`.
    2. Prepend *request* in *queue*.
5. Otherwise, run these steps:
    1. If *ifAvailable* is true and *request* is not grantable, then enqueue the following steps on *callback*'s relevant settings object's responsible event loop:
        1. Let *r* be the result of invoking *callback* with `null` as the only argument.
        2. Resolve *promise* with *r* and abort these steps.
    2. Enqueue *request* in *queue*.
6. Process the lock request queue *queue*.

4. Return *request*.

§ 4.2. Release a lock

To ***release the lock*** *lock*:

1. Assert: these steps are running on the lock task queue.
2. Let *manager* be *lock*'s manager.
3. Let *queueMap* be *manager*'s lock request queue map.
4. Let *name* be *lock*'s resource name.
5. Let *queue* be the result of getting the lock request queue from *queueMap* for *name*.
6. Remove lock from the *manager*'s held lock set.
7. Process the lock request queue *queue*.

§ 4.3. Abort a request

To ***abort the request*** *request*:

1. Assert: these steps are running on the lock task queue.
2. Let *manager* be *request*'s manager.
3. Let *name* be *request*'s name.
4. Let *queueMap* be *manager*'s lock request queue map.
5. Let *queue* be the result of getting the lock request queue from *queueMap* for *name*.
6. Remove *request* from *queue*.
7. Process the lock request queue *queue*.

To ***signal to abort the request*** *request* with *signal*:

1. Enqueue the steps to abort the request *request* to the lock task queue.
2. Reject *request*'s promise with *signal*'s abort reason.

§ 4.4. Process a lock request queue for a given resource name

To ***process the lock request queue*** *queue*:

1. Assert: these steps are running on the lock task queue.
2. For each *request* of *queue*:

    1. If *request* is not grantable, then return.

        > Note: Only the first item in a queue is grantable. Therefore, if something is not grantable then all the following items are automatically not grantable.

    2. Remove *request* from *queue*.
    3. Let *agent* be *request*'s agent.
    4. Let *manager* be *request*'s manager.
    5. Let *clientId* be *request*'s clientId.
    6. Let *name* be *request*'s name.
    7. Let *mode* be *request*'s mode.
    8. Let *callback* be *request*'s callback.

9. Let *p* be *request*'s promise.
10. Let *signal* be *request*'s signal.
11. Let *waiting* be a new promise.
12. Let *lock* be a new lock with agent *agent*, clientId *clientId*, manager *manager*, mode *mode*, name *name*, released promise *p*, and waiting promise *waiting*.
13. Append *lock* to *manager*'s held lock set.
14. Enqueue the following steps on *callback*'s relevant settings object's responsible event loop:
    1. If *signal* is present, then run these steps:
        1. If *signal* is aborted, then run these steps:
            1. Enqueue the following step to the lock task queue:
                1. Release the lock *lock*.
            2. Return.
        2. Remove the algorithm signal to abort the request *request* from *signal*.
    2. Let *r* be the result of invoking *callback* with a new `Lock` object associated with *lock* as the only argument.
    3. Resolve *waiting* with *r*.

§ 4.5. Snapshot the lock state

To ***snapshot the lock state*** for *manager* with *promise*:

1. Assert: these steps are running on the lock task queue.
2. Let *pending* be a new list.
3. For each *queue* of *manager*'s lock request queue map's values:
    1. For each *request* of *queue*:
        1. Append «["name" → *request*'s name, "mode" → *request*'s mode, "clientId" → *request*'s clientId]» to *pending*.
4. Let *held* be a new list.
5. For each *lock* of *manager*'s held lock set:

1. Append «["name" → *lock*'s name, "mode" → *lock*'s mode, "clientId" → *lock*'s clientId]» to *held*.

6. Resolve *promise* with «["held" → *held*, "pending" → *pending*]».

For any given resource, the snapshot of the pending lock requests will return the requests in the order in which they were made; however, no guarantees are made with respect to the relative ordering of requests across different resources. For example, if pending lock requests A1 and A2 are made against resource A in that order, and pending lock requests B1 and B2 are made against resource B in that order, than both «A1, A2, B1, B2» and «A1, B1, A2, B2» would be possible orderings for a snapshot's pending list.

No ordering guarantees exist for the snapshot of the held lock state.

§ 5. Usage Considerations

This section is non-normative.

§ 5.1. Deadlocks

Deadlocks are a concept in concurrent computing, and deadlocks scoped to a particular lock manager can be introduced by this API.

EXAMPLE 3 ¶

An example of how deadlocks can be encountered through the use of this API is as follows.

Script 1:

```
navigator.locks.request('A', async a => {
  await navigator.locks.request('B', async b => {
    // do stuff with A and B
  });
});
```

Script 2:

```
navigator.locks.request('B', async b => {
  await navigator.locks.request('A', async a => {
    // do stuff with A and B
  });
});
```

If script 1 and script 2 run close to the same time, there is a chance that script 1 will hold lock A and script 2 will hold lock B and neither can make further progress - a deadlock. This will not affect the user agent as a whole, pause the tab, or affect other script in the origin, but this particular functionality will be blocked.

Preventing deadlocks requires care. One approach is to always acquire multiple locks in a strict order.

EXAMPLE 4 ¶

A helper function such as the following could be used to request multiple locks in a consistent order.

```
async function requestMultiple(resources, callback) {
  const sortedResources = [...resources];
  sortedResources.sort(); // Always request in the same order.

  async function requestNext(locks) {
    return await navigator.locks.request(sortedResources.shift(), async
      // Now holding this lock, plus all previously requested locks.
      locks.push(lock);

      // Recursively request the next lock in order if needed.
      if (sortedResources.length > 0)
        return await requestNext(locks);

      // Otherwise, run the callback.
      return await callback(locks);

      // All locks will be released when the callback returns (or throws
    });
  }
  return await requestNext([]);
}
```

In practice, the use of multiple locks is rarely as straightforward — libraries and other utilities can often unintentionally obfuscate their use.

§ 6. Security and Privacy Considerations

§ 6.1. Lock Scope

The definition of a lock manager's scope is important as it defines a privacy boundary. Locks can be used as an ephemeral state retention mechanism and, like storage APIs, can be used as a communication mechanism, and must be no more privileged than storage facilities. User agents that impose finer granularity on one of these services must impose it on others; for example, a user agent

that exposes different storage partitions to a top-level page (first-party) and a cross-origin iframe (third-party) in the same origin for privacy reasons must similarly partition locking.

This also provides reasonable expectations for web application authors; if a lock is acquired over a storage resource, all same-origin browsing contexts must observe the same state.

6.2. Private Browsing

Every private mode browsing session is considered a separate user agent for the purposes of this API. That is, locks requested/held outside such a session have no affect on requested/held inside such a session, and vice versa. This prevents a website from determining that a session is "incognito" while also not allowing a communication mechanism between such sessions.

6.3. Implementation Risks

Implementations must ensure that locks do not span origins. Failure to do so would provide a side-channel for communication between script running in two origins, or allow one script in one origin to disrupt the behavior of another (e.g. denying service).

6.4. Checklist

The W3C TAG has developed a Self-Review Questionnaire: Security and Privacy for editors of specifications to informatively answer. Revisiting the questions here:

- The specification does not deal with personally identifiable information, or high-value data.
- No new state for an origin that persists across browsing sessions is introduced.
- No new persistent, cross-origin state is exposed to the web.
- No new data is exposed to an origin that it doesn't currently have access to (e.g. via polling [IndexedDB-2].)
- No new script execution/loading mechanisms are enabled.
- This specification does not allow an origin access to any of the following:
    - The user's location.
    - Sensors on a user's device.
    - Aspects of a user's local computing environment.

    - Access to other devices.
    - Any measure of control over a user agent's native UI.
- No temporary identifiers to the web are exposed to the web. All resource names are provided by the web application itself.
- Behavior in first-party and third-party contexts is distinguished in a user agent if storage is distinguished. See § 6.1 Lock Scope.
- Behavior in the context of a user agent's "incognito" mode is described in § 6.2 Private Browsing.
- No data is persisted to a user's local device by this API.
- This API does not allow downgrading default security characteristics.

§ 7. Acknowledgements

Many thanks to Alex Russell, Andreas Butler, Anne van Kesteren, Boris Zbarsky, Chris Messina, Darin Fisher, Domenic Denicola, Gus Caplan, Harald Alvestrand, Jake Archibald, Kagami Sascha Rosylight, L. David Baron, Luciano Pacheco, Marcos Caceres, Ralph Chelala, Raymond Toy, Ryan Fioravanti, and Victor Costan for helping craft this proposal.

Special thanks to Tab Atkins, Jr. for creating and maintaining Bikeshed, the specification authoring tool used to create this document, and for his general authoring advice.

§ Conformance

§ Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 terminology. The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in the normative parts of this document are to be interpreted as described in RFC 2119. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. [RFC2119]

Examples in this specification are introduced with the words “for example” or are set apart from the normative text with `class="example"`, like this:

EXAMPLE 5 ¶

This is an example of an informative example.

Informative notes begin with the word “Note” and are set apart from the normative text with `class="note"`, like this:

Note, this is an informative note.

§ Conformant Algorithms

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the key word ("must", "should", "may", etc) used in introducing the algorithm.

Conformance requirements phrased as algorithms or specific steps can be implemented in any manner, so long as the end result is equivalent. In particular, the algorithms defined in this specification are intended to be easy to understand and are not intended to be performant. Implementers are encouraged to optimize.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

[CSS21] defines the following terms:

- user agent

[DOM] defines the following terms:

- AbortSignal
- abort reason
- aborted
- add
- document
- remove

[ECMA262] defines the following terms:

- agent

[HTML] defines the following terms:

- Navigator
- WorkerNavigator
- agent cluster
- associated document
- browsing context
- enqueue steps
- enqueue the following steps
- environment settings object
- fully active
- id
- relevant global object
- relevant settings object
- responsible event loop
- starting a new parallel queue
- task source
- unloading document cleanup steps

[INFRA] defines the following terms:

- append (for list)
- append (for set)
- assert
- enqueue
- exist
- for each (for list)
- for each (for map)
- is empty
- item (for list)
- item (for struct)
- javascript string
- list
- map
- prepend
- queue
- remove
- set
- set (for map)
- struct
- values

[Service-Workers] defines the following terms:

- Client
- id

[Storage] defines the following terms:

- obtain a local storage bottle map
- storage bottle
- storage bucket

[WEBIDL] defines the following terms:

- AbortError
- DOMException
- DOMString
- Exposed
- InvalidStateError
- NotSupportedError
- Promise
- SecureContext
- SecurityError
- a new promise
- a promise rejected with
- any
- boolean
- callback function
- invoke
- reject
- resolve
- sequence
- this

§ References

§ Normative References

[CSS21]
Bert Bos; et al. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. 7 June 2011. REC. URL: https://www.w3.org/TR/CSS21/

[DOM]
Anne van Kesteren. *DOM Standard*. Living Standard. URL: https://dom.spec.whatwg.org/

[HTML]
Anne van Kesteren; et al. *HTML Standard*. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]
Anne van Kesteren; Domenic Denicola. *Infra Standard*. Living Standard. URL: https://infra.spec.whatwg.org/

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://datatracker.ietf.org/doc/html/rfc2119

[Storage]
Anne van Kesteren. *Storage Standard*. Living Standard. URL: https://storage.spec.whatwg.org/

[WEBIDL]
Edgar Chen; Timothy Gu. *Web IDL Standard*. Living Standard. URL: https://webidl.spec.whatwg.org/

§ Informative References

[IndexedDB-2]
Ali Alabbas; Joshua Bell. *Indexed Database API 2.0*. 30 January 2018. REC. URL: https://www.w3.org/TR/IndexedDB-2/

[Service-Workers]
Jake Archibald; Marijn Kruisselbrink. *Service Workers*. 12 July 2022. CR. URL: https://www.w3.org/TR/service-workers/

§ IDL Index

```
[SecureContext]
interface mixin NavigatorLocks {
  readonly attribute LockManager locks;
};
Navigator includes NavigatorLocks;
WorkerNavigator includes NavigatorLocks;

[SecureContext, Exposed=(Window,Worker)]
interface LockManager {
  Promise<any> request(DOMString name,
                       LockGrantedCallback callback);
  Promise<any> request(DOMString name,
                       LockOptions options,
                       LockGrantedCallback callback);

  Promise<LockManagerSnapshot> query();
};
```

```
callback LockGrantedCallback = Promise<any> (Lock? lock);

enum LockMode { "shared", "exclusive" };

dictionary LockOptions {
  LockMode mode = "exclusive";
  boolean ifAvailable = false;
  boolean steal = false;
  AbortSignal signal;
};

dictionary LockManagerSnapshot {
  sequence<LockInfo> held;
  sequence<LockInfo> pending;
};

dictionary LockInfo {
  DOMString name;
  LockMode mode;
  DOMString clientId;
};

[SecureContext, Exposed=(Window,Worker)]
interface Lock {
  readonly attribute DOMString name;
  readonly attribute LockMode mode;
};
```

§ Issues Index

ISSUE 1 Refine the integration with [Storage] here, including how to get the lock manager properly from the given environment.

ISSUE 2 This is currently only for workers and is vaguely defined, since there is no normative way to run steps on worker termination.