Browser I/O Scheduler

PUBLIC

Status: v1 (approved by Chrome Eng Review)

*Update: Implemented (API @ base/task_scheduler/post_task.h). **The scope has evolved beyond I/O in browser process and ~~will soon be~~ is now available in all process types.** The ultimate goal is for all work to go through it for it to have full information when making scheduling decisions.*

Tracker bug for implementation: https://crbug.com/553459

Open to all@chromium.org or request access from any other account and we'll grant it immediately (only not public public because from experience that results in many unintentionally anonymous comments)

Authors: gab@chromium.org, robliao@chromium.org, fdoray@chromium.org, scheduler-dev

Last Updated: 2016/01/26

- Introduction
- Background
- Objective
  - Objectives
  - Key Results
- Overview
- Detailed Design
  - API Overview
  - Migrating Chrome
    - Phase 1: Creating the Task Scheduler
    - Phase 2: Migrate Existing Threads to the new Task Scheduler.
    - Phase 3: Deprecate old PostTask APIs
    - Phase 4: Remove old PostTask APIs
  - Proposed TaskScheduler Implementation
    - Embedder Choice
    - Overall Component View
      - Task
      - Sequence
      - Priority Queue (and SequenceSortKey)
    - Locking Visual Language and Notes
    - Scheduling Algorithm
    - Posting A Task
    - Thread Creation and Assignment

Single Threaded Execution Mode
Delayed Tasks
Shutdown
Tracing
Future Work
Caveats
Security Considerations
Privacy Considerations
Testing Plan
Work Estimates
Document History
Appendix: Sequence Diagram

Introduction

Today, asynchronous programming in Chrome requires developers to understand (or randomly pick) a number of different task posting frameworks implemented in different layers of the codebase. It's not always obvious what the correct choice is for any individual task, nor is it easy to search the codebase for good patterns to follow. In this document, we propose a framework to simplify asynchronous programming in Chrome and enable scheduling optimizations that make better use of available hardware (e.g., number of computing cores, spinning vs. solid state disks, etc).

Background

As discussed internally and approved by Chrome Eng Review in the internal version of this doc, here's a proposed implementation for a browser I/O scheduler initially aimed at converging all but the UI/IO threads in the browser process.

Objective

Objectives

- Simplify asynchronous task management in Chromium (initial focus on I/O in browser process)
- Provide a single API to post tasks in the browser process to replace all but the UI/IO threads.
- Reuse existing Chromium PostTask semantics in the API.

Key Results

- An initial task_scheduler API and implementation with documentation and tests.
- Developer documentation of the migration flow.
- Deprecation and removal of many existing PostTask APIs.
- Posters can provide the right semantics for their tasks so that they can be scheduled appropriately (I/O or not; high or low priority; etc.).
- Posters can easily guarantee the COM (Windows' Component Object Model) context of their task and correctly manage the lifetime of that context (today not all dedicated threads do this correctly, restricting the execution options of code requiring COM).
- Centralized task management allows for enhanced instrumentation and metrics on task flows (ref. tracing, RAIL, perf insights, etc.).

Overview

We propose introducing a `base/task_scheduler/` API and implementation that would be usable for any process that initializes this component.

The API is in `base` so that `components/` et al. can post tasks without having to be handed a `TaskRunner`. Handing off strongly typed `TaskRunners` doesn't make much sense in a world of traits as the types only specify partial traits. It's preferable to have the full context in the component which can request all the traits it needs rather than ask for them from its creator.

The main differences between the task_scheduler API and existing APIs are:

1. `TaskTraits` → Ability to specify what a task (or group of task via `CreateTaskRunnerWithTraits()`) is/are.
2. Traits are hints to the `TaskScheduler` who's to decide where tasks execute in practice instead of having the caller provide this implicitly by his PostTask API choice.
3. Threads and pools are an implementation detail and are neither exposed nor even mentioned by the API.
4. Full knowledge of pending tasks → Enables smart scheduling.

Detailed Design

API Overview

FYI, code below is formatted with the *Code Pretty* Google Doc Add-On.

```
// base/task_scheduler/post_task.h
#include "base/task_scheduler/task_traits.h"
```

```
namespace base {
// The functions below forward posted tasks to the task scheduler.
// A TaskScheduler must have been initialized via
// base::TaskScheduler::Initialize() before these are valid.
//
// To post a simple one-off task:
//     base::PostTask(FROM_HERE, base::Bind(...));
//
// To post a high priority one-off task to respond to a user interaction:
//     base::PostTaskWithTraits(
//         FROM_HERE,
//         base::TaskTraits().WithPriority(base::TaskPriority::USER_BLOCKING),
//         base::Bind(...));
//
// To post tasks that must run in sequence:
//     scoped_refptr<base::TaskRunner> task_runner = CreateTaskRunnerWithTraits(
//         base::TaskTraits(), base::ExecutionMode::SEQUENCED);
//     task_runner.PostTask(FROM_HERE, base::Bind(...));
//     task_runner.PostTask(FROM_HERE, base::Bind(...));
//
// To post file I/O tasks that must run in sequence and can be skipped on
// shutdown:
//     scoped_refptr<base::TaskRunner> task_runner =
//         CreateTaskRunnerWithTraits(
//             base::TaskTraits().WithFileIO().WithShutdownBehavior(
//                 TaskShutdownBehavior::SKIP_ON_SHUTDOWN),
//             base::ExecutionMode::SEQUENCED);
//     task_runner.PostTask(FROM_HERE, base::Bind(...));
//     task_runner.PostTask(FROM_HERE, base::Bind(...));
//
//
// The default TaskTraits apply to tasks that:
//     (1) don't need to do I/O,
//     (2) don't affect user interaction and/or visible elements, and
//     (3) can either block shutdown or be skipped on shutdown
//         (barring current TaskScheduler default).
// If those loose requirements are sufficient for your task, use
// PostTask[AndReply], otherwise override these with explicit traits via
// PostTaskWithTraits[AndReply].

// Posts |task| to the TaskScheduler. Calling this is equivalent to
// calling PostTaskWithTraits with plain TaskTraits.
BASE_EXPORT void PostTask(const tracked_objects::Location& from_here,
                          const Closure& task);

// Posts |task| to the TaskScheduler and posts |reply| on the
// caller's execution context (i.e. same Sequence or MessageLoop)
// when |task| completes. Calling this is equivalent to calling
// PostTaskWithTraitsAndReply with plain TaskTraits.
BASE_EXPORT void PostTaskAndReply(const tracked_objects::Location& from_here,
                                   const Closure& task,
                                   const Closure& reply);

// Posts |task| with specific |traits| to the TaskScheduler.
BASE_EXPORT void PostTaskWithTraits(const tracked_objects::Location& from_here,
                                    TaskTraits traits,
                                    const Closure& task);

// Posts |task| with specific |traits| to the TaskScheduler and posts
// |reply| on the caller's execution context (i.e. same Sequence or
// MessageLoop).
BASE_EXPORT void PostTaskWithTraitsAndReply(
    const tracked_objects::Location& from_here,
    TaskTraits traits,
    const Closure& task,
    const Closure& reply);

// Returns a TaskRunner whose PostTask invocations will result in scheduling
// tasks using |traits| which will be executed according to |execution_mode|.
BASE_EXPORT scoped_refptr<TaskRunner> CreateTaskRunnerWithTraits(
    TaskTraits traits, ExecutionMode execution_mode);

// Returns a TaskRunner akin to one created through CreateTaskRunnerWithTraits()
// except that it will inherit |parent_task_runner|'s sequence instead of
// creating its own. Useful for posting tasks with different traits to the same
// sequence. |parent_task_runner| must be a SEQUENCED TaskRunner returned from
// CreateTaskRunnerWithTraits are supported.
BASE_EXPORT scoped_refptr<TaskRunner> CreateChildTaskRunnerWithTraits(
    TaskRunner* parent_task_runner, TaskTraits traits);

}  // namespace base


// base/task_scheduler/task_scheduler.h
#include "base/task_scheduler/task_traits.h"

namespace base {

class BASE_EXPORT TaskScheduler {
 public:
  // Posts |task| with specific |traits|.
  // For one off tasks that don't require a TaskRunner.
  virtual void PostTaskWithTraits(
      const tracked_objects::Location& from_here,
      TaskTraits traits,
      const Closure& task) = 0;

  // Returns a TaskRunner whose PostTask invocations will result in scheduling
  // tasks using |traits| which will be executed according to |execution_mode|.
  virtual scoped_refptr<TaskRunner> CreateTaskRunnerWithTraits(
      TaskTraits traits, ExecutionMode execution_mode) = 0;

  // Performs a synchronous cleanup of threads and state in the TaskScheduler.
  virtual void Shutdown() = 0;

  // Registers |task_scheduler| to handle tasks posted through the
  // post_task.h API for this process. Must be called on the main thread
  // before creating any other threads. Note that this replaces any preiovusly
  // instantiated task scheduler.
  static void SetInstance(scoped_ptr<TaskScheduler> task_scheduler);

  // Retrieve the TaskScheduler set via SetInstance(). This should be used very
  // rarely; most users of TaskScheduler should use the post_task.h API.
  static TaskScheduler* GetInstance();

  // Initializes the default task scheduler for this process.
  static void InitializeDefaultTaskScheduler();
};

}  // namespace base

// base/task_scheduler/task_traits.h
namespace base {

enum class TaskPriority : char {
  // This task affects UI immediately after a user interaction.
  // Example: Generating data to show in the UI immediately after a click.
  USER_BLOCKING = 2,
  // This task affects UI or responsiveness of future user interactions. It is
  // not an immediate response to a user interaction.
  // Examples:
  // - Updating the UI to reflect that a long task was completed.
  // - Loading data that might be shown in the UI after a future user
  //   interaction.
  USER_VISIBLE = 1,
  // Everything else (user won't notice if this takes an arbitrarily long time
  // to complete).
  BACKGROUND = 0,
};

// (Same as in today's SequencedWorkerPool.)
enum class TaskShutdownBehavior {
  // Tasks posted with this mode which have not started executing before
  // shutdown is initiated will never run. Tasks with this mode running
  // at shutdown will be ignored (the worker thread will not be joined).
  //
  // TODO: Copy usage examples from sequenced_worker_pool.h here.
  CONTINUE_ON_SHUTDOWN,

  // Tasks posted with this mode that have not started executing at
  // shutdown will never run. However, any task that has already begun
  // executing when shutdown is invoked will be allowed to continue and
  // will block shutdown until completion.
  SKIP_ON_SHUTDOWN,

  // Tasks posted with this mode before shutdown is complete will block
  // shutdown until they're executed.
  // Note: Tasks with BACKGROUND priority that block shutdown will be promoted
  // to USER_VISIBLE priority during shutdown.
  //
  // Generally, this should be used only to save critical user data.
  BLOCK_SHUTDOWN,
};

// TaskTraits holds metadata about a task and prevents the combinatorial
// explosion of PostTaskWith*() call sites.
// Usage example:
//     base::PostTaskWithTraits(
//         FROM_HERE, base::TaskTraits().WithFileIO().WithPriority(USER_BLOCKING),
//         base::Bind(...));
struct BASE_EXPORT TaskTraits {
  // Constructs a default TaskTraits for tasks with
  //     (1) no I/O,
  //     (2) same priority as current context or user visible priority, and
  //     (3) may block shutdown or be skipped on shutdown.
  // Tasks that require stricter guarantees should highlight those by requesting
  // explicit traits below.
  //
  // NOTE FOR THIS DOC: We would really like the default to be SKIP_ON_SHUTDOWN
  // but SequencedWorkerPool uses BLOCK_SHUTDOWN as a default and it may be hard
  // to to figure out during the migration which of its callers relies on it
  // implicitly versus which ones don't care. For now we loosely define our API
  // in order to prevent anyone else from depending on this.
  TaskTraits();
  ~TaskTraits();

  // Allows tasks with these traits to do file I/O.
  TaskTraits& WithFileIO();

  // Applies a priority to tasks with these traits.
  TaskTraits& WithPriority(TaskPriority priority);

  // Applies a shutdown behavior to tasks with these traits.
  TaskTraits& WithShutdownBehavior(TaskShutdownBehavior behavior);

 private:

  (...)
};
```

```
enum class ExecutionMode {
  // Can execute multiple tasks at a time.
  // High priority tasks will execute first under contention.
  PARALLEL,

  // Executes one task at a time. Tasks are guaranteed to run in posting order.
  // The sequence's priority will be that of its pending task with the highest
  // priority.
  SEQUENCED,

  // Executes one task at a time on a single thread. Tasks are guaranteed to run
  // in posting order.
  SINGLE_THREADED,

#if defined(OS_WIN)
  // Executes one task at a time on a single thread that has initialized the COM
  // library with the single-threaded apartment (STA) concurrency model. Tasks
  // are guaranteed to run in posting order and the assigned thread will also
  // process COM messages so long as the associated TaskRunner is kept alive.
  SINGLE_THREADED_COM_STA,
#endif  // defined(OS_WIN)
};

}  // namespace base
```

Migrating Chrome

Phase 1: Creating the Task Scheduler

This involves creating the above TaskScheduler API with a full suite of unit tests. At this point, the scheduler will not be hooked up to the Chrome production code.

Phase 2: Migrate Existing Threads to the new Task Scheduler.

Once the scheduler basic functionality is ready to go, the existing PostTask APIs (ignoring IO and UI threads at first) can call directly to the new task_scheduler API to dispatch tasks as such:

Old APIs	**TaskTraits mapping** (USER_VISIBLE priority is default)	**ExecutionMode**

`BrowserThread::PostTask` → `DB`	`WithFileIO().WithShutdownBehavior(`
	`BLOCK_SHUTDOWN)`
	(need to confirm that BACKGROUND priority is appropriate for this)
	(ideally no BLOCK_SHUTDOWN but this thread and others below are either joined or are in pools which default to
		SINGLE_THREADED
		(ideally SEQUENCED but can't assume no thread affinity requirements initially)

	BLOCK_SHUTDOWN task semantics and will thus need to continue to do so until tasks are split off and manage their own specific semantics)	
`FILE`	`WithFileIO().WithShutdownBehavior( BLOCK_SHUTDOWN)`	SINGLE_THREADED_COM_STA
`CACHE`	`WithFileIO(). WithPriority(USER_VISIBLE). WithShutdownBehavior( BLOCK_SHUTDOWN)`	SINGLE_THREADED
`FILE_USER_BLOCKING`	`WithFileIO(). WithPriority(USER_BLOCKING). WithShutdownBehavior( BLOCK_SHUTDOWN)`	SINGLE_THREADED
`PROCESS_LAUNCHER`	`WithPriority(USER_VISIBLE). WithShutdownBehavior( BLOCK_SHUTDOWN)`	SINGLE_THREADED
`BrowserThread ::PostBlockingPoolTask ::PostBlockingPoolTaskAndReply`	`WithFileIO().WithShutdownBehavior( BLOCK_SHUTDOWN)`	PARALLEL
`BrowserThread ::PostBlockingPoolSequencedTask ::GetBlockingPool`	`WithFileIO().WithShutdownBehavior( BLOCK_SHUTDOWN or custom if caller specifies a ShutdownBehavior)`	SEQUENCED 1 sequence per sequence token. A map of sequence tokens to task runners would exist at this API level.
base::WorkerPool	Migrate calls (27) on a case by case basis.	-
History Thread	`WithFileIO(). WithPriority(USER_BLOCKING). WithShutdownBehavior(BLOCK_SHUTDOWN)`	SEQUENCED (technically SINGLE_THREADED is exact mapping but SEQUENCED is likely fine and is required to get USER_VISIBLE/USER_BLOCKING mix)
Chrome_PasswordStore_Thread	`WithFileIO(). WithPriority(USER_VISIBLE). WithShutdownBehaviour( BLOCK_SHUTDOWN) (note: check with owners for proper`	TBD

	`priority and shutdown semantics)`	
IndexedDB and 2 LevelDB threads	`WithFileIO().WithPriority(TBD)` `Will most likely end up having multiple TaskRunners funneling into a single Sequence (ref. CreateChildTaskRunnerWithTraits) and plumb priorities from context aware callers.`	TBD
NetworkConfigWatcher	Discuss with OWNERS, likely BACKGROUND work.	TBD
V8's ProxyResolver threads	Assuming `.WithPriority(USER_VISIBLE)` but will confirm with OWNERS	TBD
SimpleCache Pool	`WithFileIO(). WithPriority(USER_VISIBLE). WithShutdownBehavior( BLOCK_SHUTDOWN)` `(also some USER_BLOCKING tasks in here for foreground tab navigations, may need dynamic priorities, keep this migration last)`	PARALLEL (e.g. 1 Sequence per token, but couldn't find any Sequences on SimpleCache pool in code search)
ServiceWorkers (browser side)	WithFileIO().WithProperty("WebContents", web_contents) `(still under design, but we could add a concept of generic "property" to task traits which external voters could interpret, inducing the concept of a dynamic priority -- that way a ServiceWorker which doesn't know if it's foreground or background could simply say which tab it's associated to and let the voter decide its priority at post time and even dynamically adjust it on tab switch)`	
COM threads (see Thread::init_com_with_mta(false)) -- could eventually also support the MTA)	-	SINGLE_THREADED_COM_STA 1 TaskRunner per thread
`BrowserThread ::PostAfterStartupTask`	Made irrelevant (the scheduler will take care of delaying background work as required). The API will still work during the migration phase and be removed once every TaskRunner passed to it is backed by the new API.	-

Note: the many BLOCK_SHUTDOWN requirements simply highlight today's reality which sadly blocks on WAY too many things on shutdown… This could be a problem given all the BACKGROUND work will land on a single thread (ref. Thread Creation and Assignment) which will thus reduce parallelism on shutdown. To alleviate that, tasks with the BLOCK_SHUTDOWN trait will be promoted to the USER_VISIBLE priority during shutdown (as-is in fact appropriate per them now blocking an observable event). Also worth experimenting with would be to make TaskScheduler default to SKIP_ON_SHUTDOWN and only mark truly blocking tasks as blocking, alleviating this self-imposed situation further.

At this stage, redirection to the new API will be controlled by Finch, allowing us to see changes in behavior in the field.

Phase 3: Deprecate old PostTask APIs

Move callers of the old APIs above to using the task_scheduler API directly (breaking down the implicit sequence represented by each thread into multiple ones as appropriate). Adding PRESUBMIT checks as we close down on each individual APIs' removal to ease the removal of its final bits.

Update the Chromium documentation to encourage developers to use the new API. Link. Link.

Send an email to chromium-dev@ to describe the new task_scheduler API and ask developers to stop using the old APIs for new code. Also, encourage developers to migrate existing code to calling the new API directly.

NOTE: This step will require plumbing of some traits (e.g. priority) between generic task posting sites (e.g. IndexedDB) and context aware callers in that stack.

At this stage, usage of the new API is no longer controlled by Finch.

Phase 4: Remove old PostTask APIs

The following APIs will be removed:

- `content::BrowserThread:`
    - `DB, FILE, FILE_USER_BLOCKING, PROCESS_LAUNCHER, CACHE`
    - `BlockingPool`
    - `PostAfterStartupTask`
- `base::WorkerPool`
- `base::SequencedWorkerPool` as other smaller pools are migrated as well.

Proposed TaskScheduler Implementation

Embedder Choice

The default task scheduler will live in base/. Possible extension points for this task scheduler for non-browser uses will be considered but won't be implemented for v1.

The arguments for base/ or content/browser are as follows:

1. Have the implementation live in base/:
   Pros:
   a. Projects that depend on base/ (outside of Chromium) still have a “worker pool” after WorkerPool and SequencedWorkerPool are phased out.

   Cons:
   b. base/ has no public/ API so the PostTask API and its impl would be mixed.
   c. Instead of TaskScheduler::SetInstance() we would have TaskScheduler::Instantiate(...) which would take in parameters as required to inject process-specific requirements and delegates from the instantiator (initially no parameters as the original TaskScheduler implementation would only be aimed at the browser process -- which also makes it weird that it would live in base/).
2. Have the implementation live in content/browser/:
   Pros:
   a. Implementation is hidden in a non-public (at DEPS level) section of the codebase.
   b. Completely different implementations can be provided for tests and other Chromium processes.
   c. While a common base will likely be shared between this scheduler and future ones for other processes, the common code could live in content/common/ and be shared with content/renderer/, content/browser/, and content/gpu/ without requiring delegates as in 1.c.

   Cons:
   d. No more easy to use “worker pool” API in base/ after WorkerPool and SequenceWorkerPool are phased out for projects that use base/ outside of Chromium.

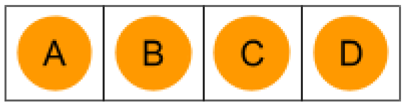
Having an immediately usable API specified in base/ means that the implementation should live in base/.

Overall Component View

Task

A task is a unit of work that needs to be executed. Inside the scheduler, a task is an object that has

- a closure (`base::Closure`),
- traits (`base::TaskTraits`), and
- a post timestamp (`base::TimeTicks`).



```
int num_background_;
int num_user_visible_;
int num_user_blocking_;
```

A is the next task to execute in this sequence.

Sequence

Tasks are contained in sequences. The order in which tasks are added to a given sequence determines the order in which they will be executed. Sequences maintain per-priority task counters.

Sequences are ref-counted-thread-safe.
A sequence is created when a `SEQUENCED`, `SINGLE_THREADED` or `SINGLE_THREADED_COM_STA` TaskRunner is created. The TaskRunner keeps a reference to its sequence. A sequence is also created when a task is posted with the `PARALLEL` execution strategy. Such sequences always contain exactly 1 task.

In addition to the references kept by TaskRunners, a priority queue (described below) keeps references to the sequences it contains and a worker threads keeps a reference to the sequence it is currently running. That means that a sequence created by a `SEQUENCED, SINGLE_THREADED` or `SINGLE_THREADED_COM_STA` TaskRunner is automatically deleted when it is empty (and thus isn't in a priority queue) and its TaskRunner no longer exists. A sequence created for a `PARALLEL` task is deleted when its only task has been executed.

Priority Queue (and SequenceSortKey)

Sequences that are not running and not empty live in a priority queue. See Thread Creation and Assignment for a description of the different priority queues that live in the scheduler.

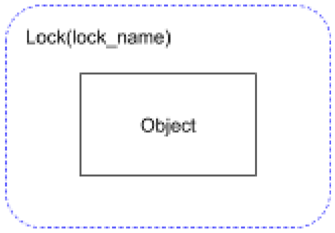
Inside a priority queue are
`std::pair<Sequence, SequenceSortKey>`
which are sorted according to the `SequenceSortKey`'s priority first and timestamp second.

```
struct SequenceSortKey {
  // Priority of the Sequence at the time it
  // was posted.
  TaskPriority sequence_queue_priority;

  // Creation timestamp of the first Task in
  // the Sequence.
  TimeTicks first_timestamp;
}
```

Note: The reason to use `SequenceSortKey` instead of the Sequence itself is that the priority of the Sequence may change after being posted (ref. Posting A Task) yet the sort key can't change while it's in the queue. It also avoids having to repeatedly lock Sequences when inserting a new Sequence and having to compare sort keys.

Locking Visual Language and Notes



Represents an object that is synchronized by a lock called |lock_name|.

Notes about locking:

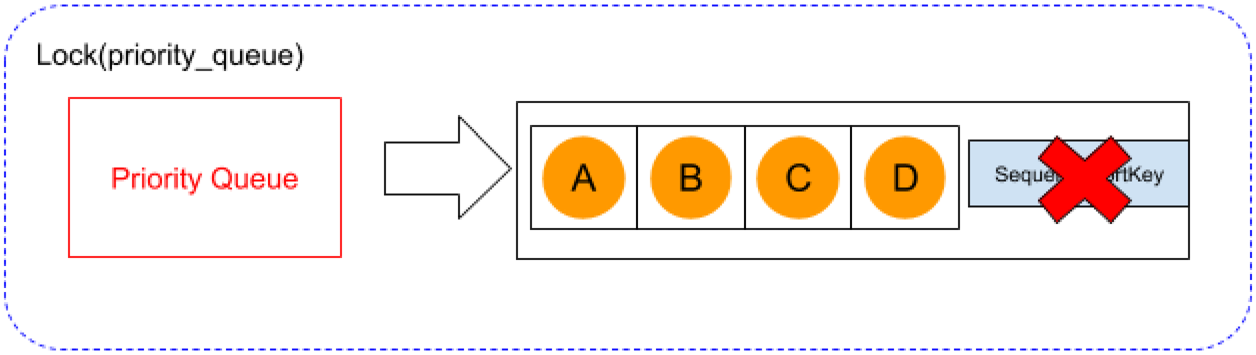
- **Priority inversion**: in order to prevent priority inversion now that Chrome will have threads at different priorities (e.g. high priority thread posts task to background queue while the background thread owns the queue's lock), we propose the following:

    - On POSIX: set the PTHREAD_PRIO_INHERIT protocol on pthread_mutexes in lock_impl_posix.cc.
    - On Windows: rely on the kernel scheduler's solution (random bumping of ready threads) which should be sufficient given our algorithms below only ever hold the lock for a short operation (insert/peek/pop/etc.) on a shared data structure.
- **Deadlocks**: a lot of care was taken in writing the algorithms below to ensure no two locks are ever held at once by a single thread, removing deadlock possibilities.

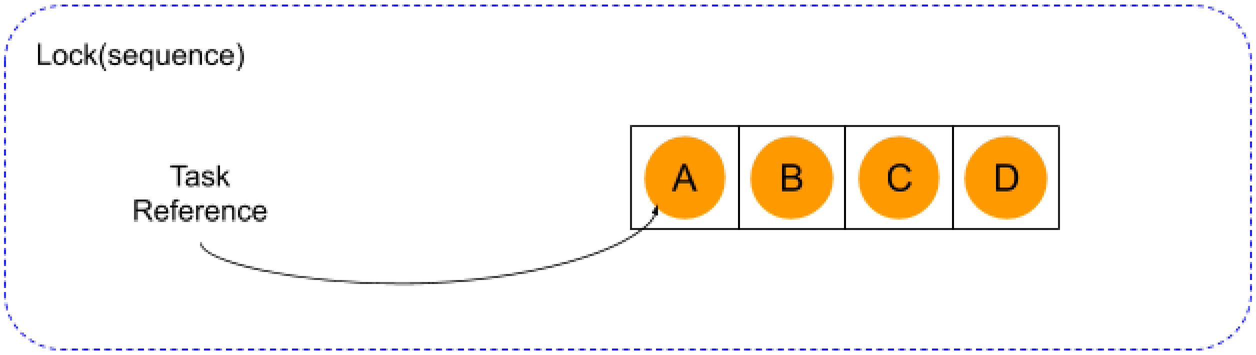
Scheduling Algorithm

Each thread owned by the scheduler runs the scheduling algorithm described below in a loop. For now, we'll assume that the whole system contains a single priority queue that is being processed by multiple threads.

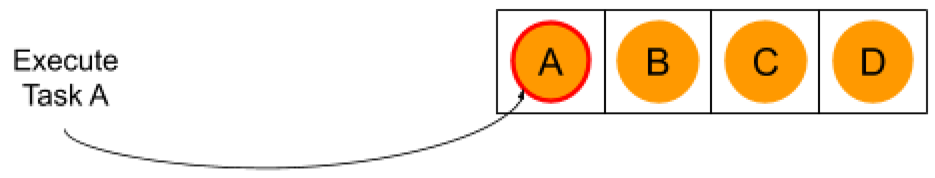
1. Lock the Priority Queue to pop its top Sequence (ref. Priority Queue for sorting order -- note: the associated SequenceSortKey is discarded in this step).



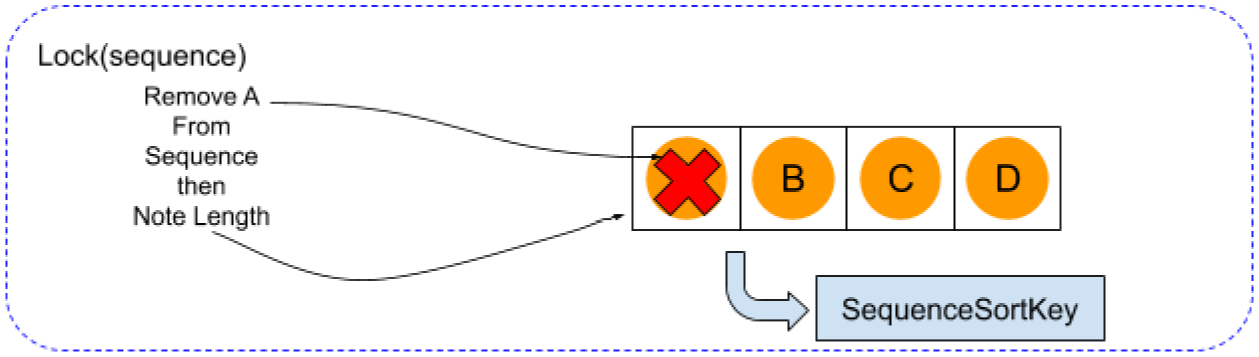
2. Lock the Sequence to peek the next Task to execute. Note: the Task is not removed from the Sequence yet, to prevent Add operations from re-inserting the sequence in a Priority Queue (ref. Posting A Task).



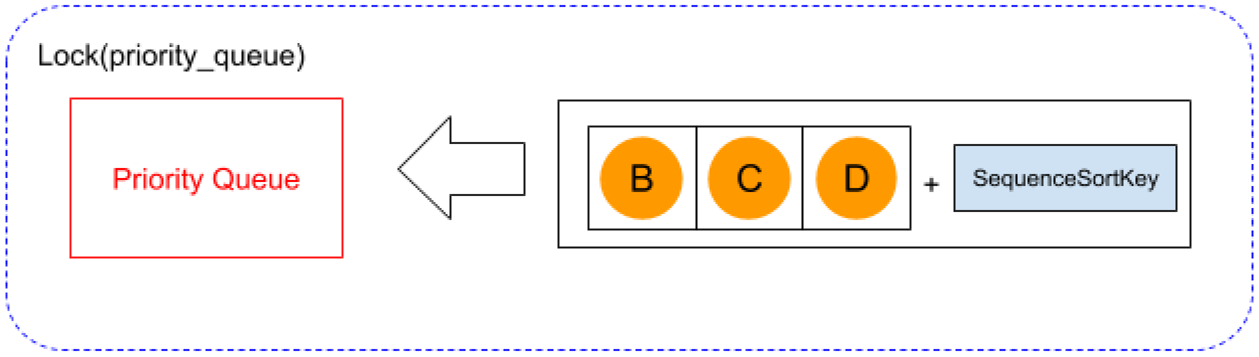
3. Release the sequence lock and execute the task. Note: new tasks can be added to the end of the Sequence's queue in the meantime (ref. Posting A Task).



4. Lock the Sequence. Remove the Task that was just executed. Read the length of the Sequence `[SequenceLength]`. Update the per-priority task counters and get the new SequenceSortKey if `[SequenceLength]` != 0. Release the lock.



5. If `[SequenceLength]` != 0: lock the PriorityQueue and insert the sequence into it. Note: the Sequence's lock doesn't need to be held as it's inserted as a pointer + sort key (it's okay if tasks are being added to it in parallel).

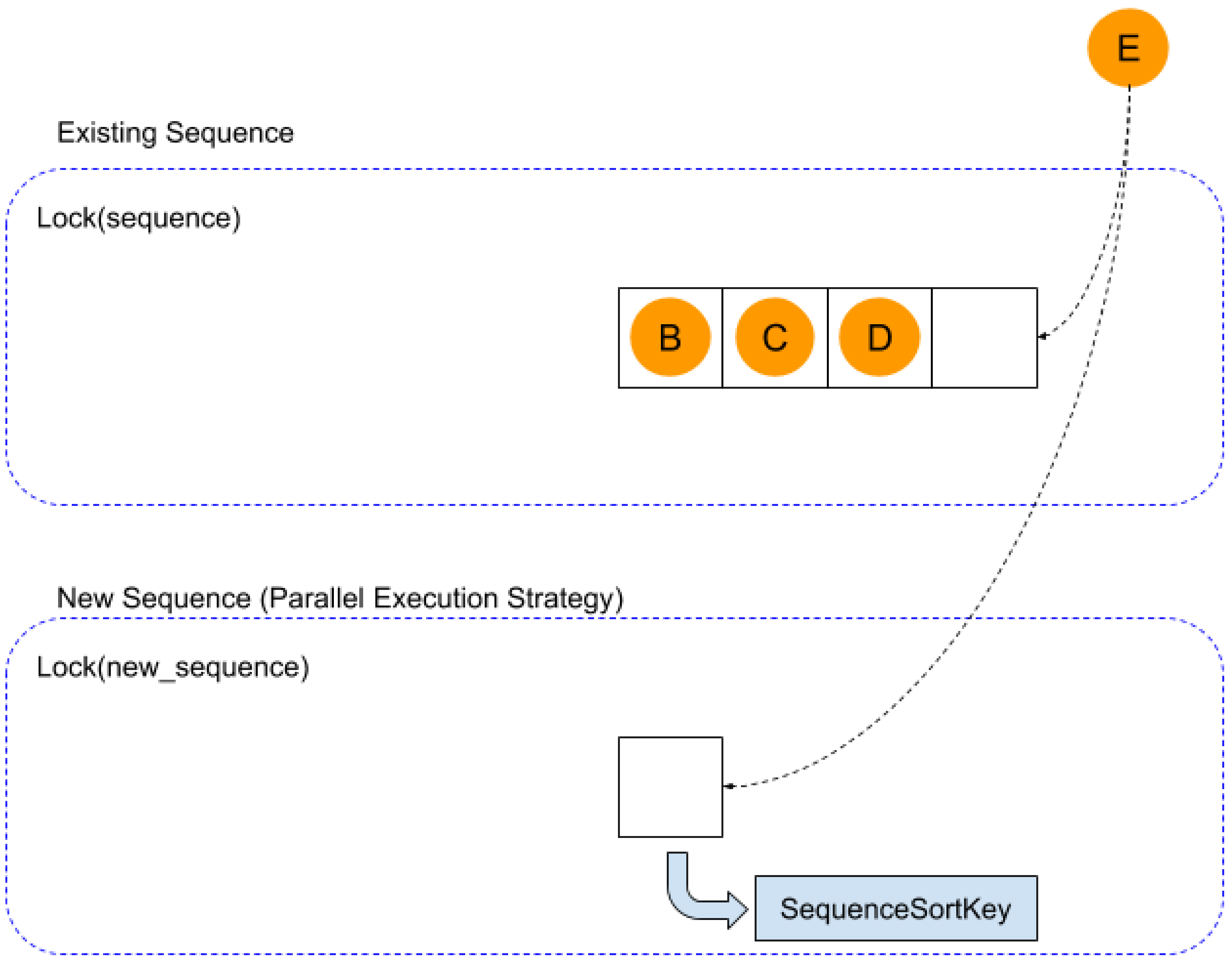


Posting A Task

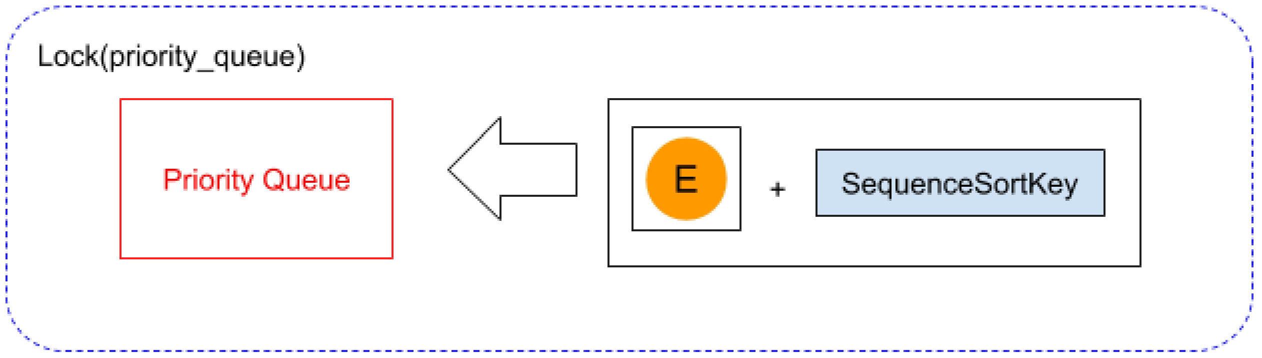
When a task is posted to the scheduler, via the PostTask* functions or via a TaskRunner, the following steps are performed:

1. Create a Task object, annotated with a creation time.

2. Get or create a Sequence.
    a. If the task is posted via a `SEQUENCED`, `SINGLE_THREADED` or `SINGLE_THREADED_STA` TaskRunner, the TaskRunner has a reference to the Sequence to use.
    b. Otherwise, create a new Sequence.
3. Add the Task to the Sequence.
    a. Lock the Sequence.
    b. Read the length of the Sequence `[SequenceLength]`.
    c. Add the Task to the Sequence.
    d. Increment the Sequence’s priority counter corresponding to the inserted task.
    Note: If this step increases the priority and the Sequence is already scheduled, it will only be rescheduled with its new priority by the Scheduling algorithm on a worker thread *after* it runs the currently scheduled task for this Sequence.
    We could support immediate priority bump, but it would require more synchronization and complexity. Instead we will add metrics to measure how often this happens and how much a task is delayed by this to assess whether this is required in practice.
    e. If `[SequenceLength]` == 0: obtain the Sequence’s SequenceSortKey.
    f. Release the lock.



4. If `[SequenceLength]` == 0, determine in which Priority Queue the Sequence should be inserted. Lock that Priority Queue and insert the Sequence into it.



Note: The Sequence thus enters the Priority Queue either (1) as part of the Add step if the Add happened on an empty Sequence or (2) at the end of the Scheduling step if the Sequence is not empty after running the picked up Task.

Thread Creation and Assignment

The above algorithm describes how tasks from a single Priority Queue are scheduled on a given worker thread, but we have yet to discuss the policy for bringing up the worker threads and assigning them to Priority Queues.

The simplest design would be to have a single priority queue for I/O work and another one for non-I/O work. I/O worker threads would feed from the I/O queue and vice-versa for the non-I/O threads.

This however brings the potential issue of having all threads busy on `BACKGROUND` work when higher priority work comes in and also causes unnecessary contention for system resources for low priority tasks.

To avoid this, we propose having a single thread processing `BACKGROUND` work at `ThreadPriority::BACKGROUND`. The pool of worker threads would then only be responsible for its queue's `USER_VISIBLE`/`USER_BLOCKING` work. This would also avoid complete starvation of `BACKGROUND` work as OS' typically promise some progress on background priority threads.

Note: We don't think there will be enough `USER_BLOCKING` work to cause starvation of `USER_VISIBLE` work, but we will measure this and assess whether a more advanced design is required here.

Ramping up: A new thread would be added to the pool of worker threads in the Posting A Task step if these conditions are met:

1. Max number of threads not reached (max based on system spec -- to be experimented with).
2. Only one idle thread (i.e. this task will likely cause all threads to become busy).

Assigning work: Threads will pickup work with a LIFO (last-in-first-out) policy in order to improve cache locality. A thread-safe stack will be used for idle threads to be signaled when posting a task.

Ramping down: The LIFO policy has the nice property that, if there isn't enough work for all threads, the same threads will remain idle instead of randomly taking turns on processing tasks that come in. As such, the ramp down policy will simply be to have a thread exit if it didn't get work over the last X (X=30?) seconds as long as it's not the only idle thread remaining.

Notes:

- Try to keep one idle thread ready to do work around as much as possible.
    - Conversely, never wind down the last thread.

- Need to measure how much thread churn this causes in practice and adjust constants above accordingly.
- "Max number of threads" could be dynamic and allow for low-power states where throughput is intentionally lowered.

Overall this looks like:

	I/O	**No I/O**
`BACKGROUND`	1 Dedicated thread	1 Dedicated thread
`USER_VISIBLE/USER_BLOCKING`	1 Dynamic LIFO thread pool	1 Dynamic LIFO thread pool

Single Threaded Execution Mode

Handling Single Threaded strategies is an addition to the Task Scheduler architecture described above. It involves the following changes:

- Each thread also has its own dedicated Priority Queue.
- When a `SINGLE_THREADED` or `SINGLE_THREADED_COM_STA` TaskRunner is created, a thread is assigned to it. When Posting A Task, the TaskRunner will insert its sequence in the Priority Queue of its assigned thread rather than in a global Priority Queue.
- When a thread is looking for work, it will check both its own Priority Queue and the global Priority Queue. It will choose the Sequence with the highest priority from the top of these 2 Priority Queues.

Both the `SINGLE_THREADED` and `SINGLE_THREADED_COM_STA` operate similarly. The main difference with `SINGLE_THREADED_COM_STA` is the following:

- COM initialization on thread upon executing the first task from a COM TaskRunner.
- COM uninitialization on thread upon executing the last possible task from a COM TaskRunner (the TaskRunner has to be destroyed to trigger this).
- Interleaved with processing regular tasks, the thread will pump Windows messages addressed to it.

Delayed Tasks

It is possible to post delayed tasks to the scheduler through the `PostDelayedTask` method of `TaskRunner`. Delays are honored for `BACKGROUND` tasks only. The delay is ignored for `USER_VISIBLE` and `USER_BLOCKING` tasks (which is valid under the TaskRunner::PostDelayedTask API).

Note: most PostDelayedTask() today are simply delayed per lack of a better way to declare "background work" and most should converge to BACKGROUND tasks without delays. On a *similar note, none of the threads we are originally going to map to USER_VISIBLE/USER_BLOCKING have PostDelayedTask calls associated with them today.*

Each `BACKGROUND` worker thread has a heap of delayed tasks. A delayed task has:

- base::Closure
- base::TaskTraits
- Planned execution time (TimeTicks::Now() + delay).
  - Used as the sort key by the heap.
- Sequence in which the task should be added once it becomes ripe for execution.

When a delayed task is posted:

1. Lock the heap of delayed tasks of the appropriate `BACKGROUND` worker thread (with/without file I/O). Add the delayed task to it. Release the lock.
2. Wake up the appropriate `BACKGROUND` worker thread (with/without file I/O) if both:
   a. The worker thread is already idle.
   b. The new delayed task lands on top of the heap (i.e. is either first or is scheduled earlier than all others).

To support delayed tasks, `BACKGROUND` worker threads will perform these steps at the beginning (before step 1) of *each iteration* of the Scheduling Algorithm:

1. Lock the heap of delayed tasks. Pop all tasks that are ripe for execution. Release the lock.
2. Post all popped tasks, by running the Posting a Task algorithm (at step 2, use the Sequence that was associated with the task in the heap).

Also, when all priority queues of a `BACKGROUND` worker thread are empty, it won't wait for work indefinitely. Instead, it will:

1. Lock the heap of delayed tasks to get the next planned execution time |Wake Up Time|.
2. Wait for work with a timeout |Wake Up Time| - Now().

Notes:

- If two `BACKGROUND` tasks T1 and T2 are posted to the same `SEQUENCED` or `SINGLE_THREADED` task runner one after the other, T1 will run before T2 if its delay is smaller than or equal to T2's delay. This is the only ordering guarantee for delayed tasks.
- In a follow-up version of the API, we plan to support delays for `USER_VISIBLE` and `USER_BLOCKING` tasks. However, we don't think it is required for the browser threads that we initially propose to replace; plus supporting a multi-threaded delayed work queue adds unnecessary complexity at this point.

Shutdown

Shutdown will be handled in a similar way to what SequencedWorkerPool does today.

When a `BLOCK_SHUTDOWN` task is *posted* or when a `SKIP_ON_SHUTDOWN` task is *scheduled*, a counter of tasks blocking shutdown will be incremented. The counter will be decremented when a `BLOCK_SHUTDOWN` or `SKIP_ON_SHUTDOWN` task completes its execution.

On shutdown, the browser calls a `Shutdown` method on the task_scheduler API. The `Shutdown` method of `TaskScheduler` sets a shutdown flag, visible by all threads. It then blocks until the counter of tasks blocking shutdown reaches zero (a condition variable is used to avoid a busy loop).

Threads check the value of the shutdown flag before getting new work. When a thread sees that the shutdown flag has been set, it enters a “shutdown” mode. In “shutdown” mode, a thread continues to run the scheduling algorithm, but it drops tasks that don’t have the `BLOCK_SHUTDOWN` shutdown behavior. A thread in “shutdown” mode exits when all its priority queues are empty.

Notes:

- When the `TaskScheduler::Shutdown` method exits, there might still be some threads alive in the scheduler (running `CONTINUE_ON_SHUTDOWN` tasks, or about to exit).
- The `TaskScheduler` is never destroyed. That ensures that threads can access its members until the very end of the process’ life.

Tracing

We will keep existing top-level chrome://tracing working in v1 and will look to iteratively improve it based on the extra state available in this API.

Future Work

There are several platform-specific mechanisms to manage a thread pool and schedule work in an optimal fashion with respect to available resources. For example, Windows has a thread pool API which could be made aware of IO completion ports and Mac has Grand Central Dispatch.

These could replace the “Thread Creation and Assignment” section of this doc, managing threads and work dispatch policies through kernel-aware mechanisms that can’t be matched in user land, while still allowing us to use our scheduling algorithm to decide which work goes first.

It will certainly be interesting to investigate the benefits of these mechanisms in the future. However, for the initial version of the project, we will only use the cross-platform algorithm described above which is already a strict improvement over the existing model.

Caveats

While our goal is to merge many existing APIs into one, we are aware that, until that is complete, our proposal merely means n+1 APIs. We are looking to mitigate this by (1) basing our API off of TaskRunner for familiarity, (2) providing clear migration documentation to developers, and (3) keeping the migration under cover as long as possible by merely redirecting the old APIs until we are fully ready to port things over. And of course (4), get through the finish line by actually deprecating and removing the APIs being replaced :-).

Security Considerations

This change is not expected to have any security implications.

Privacy Considerations

This change is not expected to have any privacy implications.

Testing Plan

Unit tests for the task_scheduler API that are 100% deterministic.

A `TestTaskScheduler` to be used in unit tests that need to have a overridable TaskScheduler in the scope of their test (essentially replacing today's `TestBrowserThreadBundle`). This task scheduler will run tasks in FIFO order on a single thread and will provide a way to explicitly block until all queued work has been processed.

Initial redirection of old APIs to task_scheduler API to be controlled via Finch and impact to be studied thoroughly on early channels.

Work Estimates

Phases:

1. Q1 '16: Write v1 of task_scheduler API + documentation + tests
2. Q1-Q2 '16: Redirect old APIs to the task_scheduler API under the hood.
3. Q2-Q3 '16: Migrate away from old APIs and deprecate them.
4. Q2-Q3 '16: Iterate and improve task_scheduler API implementation.

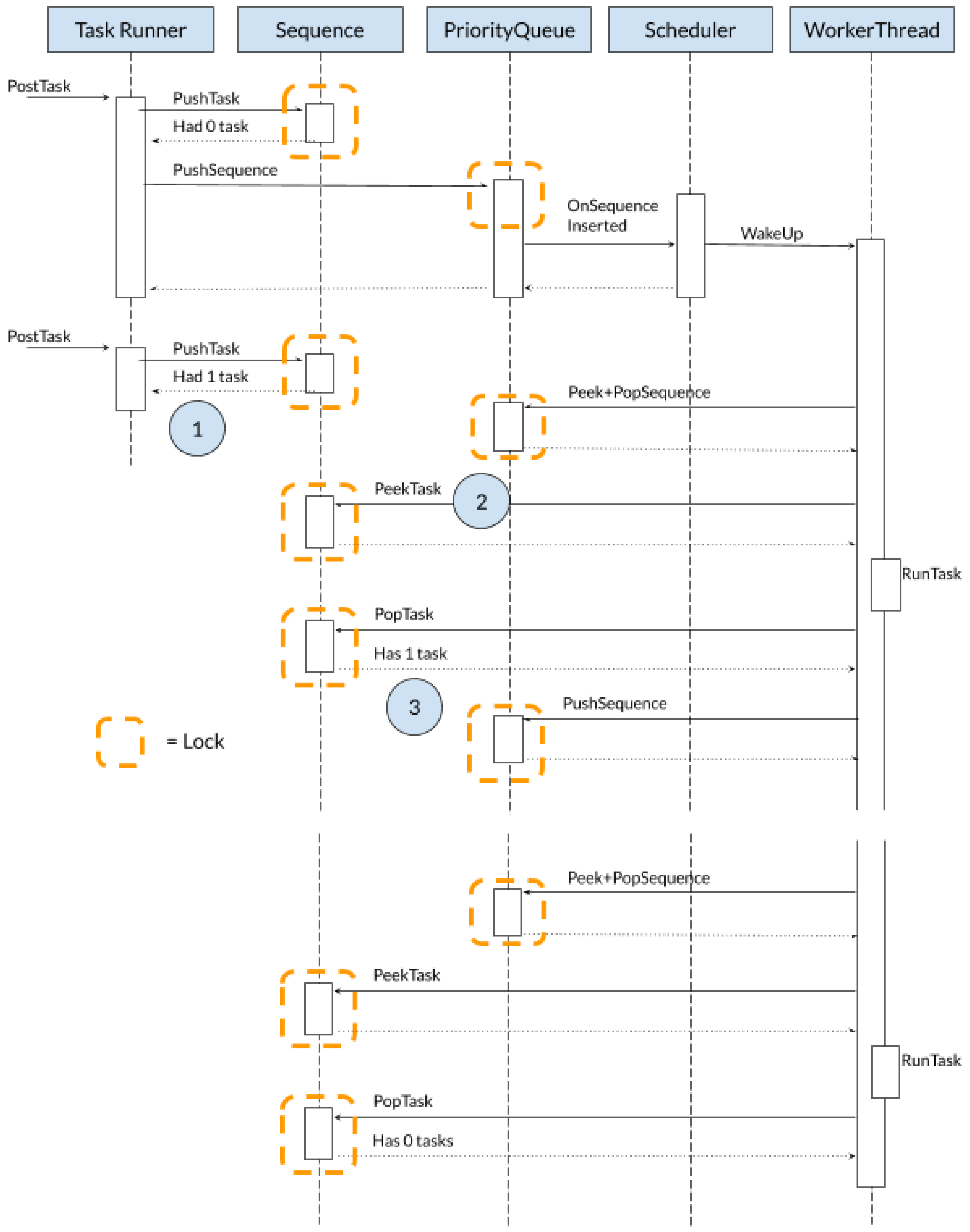
Document History

Whenever you add a significant new revision to the document, add a new line to the table below.

@reviewers: please add your name below via Docs' "suggestion tool".

Date	Author	Description	Reviewed by	Signed off by
2015/11/18	gab@, robliao@, fdoray@	Created doc after API proposal was given go ahead at eng-review.	tansell@, alexclarke@, skyostil@, grt@, robertshield@	
2016/01		Initial proposal sent to Chrome Eng Review	Chrome Eng Review	
2016/02/05		Adjusted after round 1 Chrome Eng Review		Chrome Eng Review

Appendix: Sequence Diagram



Notes:

1. Since the sequence already contained 1 task prior to the push, it doesn't need to be reinserted a the priority queue. Indeed, a non-empty sequence is either:
    - Already in a priority queue, *or*
    - Being executed by a worker thread. The worker thread will reinsert the sequence in a priority queue if it is non-empty after popping the task that is being executed.
2. Peek the task to run, but do not pop it immediately. That way, if a new task is posted to the sequence, it won't be reinserted in a priority queue (the sequence will be reinserted in a priority queue by the worker thread once it completes the execution of the task).
3. The sequence needs to be reinserted in the priority queue because it still contains 1 task after the pop.