Simplifying Task Management in Chromium

(copied from internal doc)
Status: Went to eng-review, adopted modulo details on approach -- concrete implementation plan to follow.
Authors: gab@chromium.org, robliao@chromium.org, scheduler-dev
Last Updated: 2015/11/06

Introduction

Today, asynchronous programming in Chrome requires developers to understand (or randomly pick one of) 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 future scheduling optimizations that make better use of available hardware (e.g., number of computing cores, spinning vs. solid state disks, etc).

Background

Following the initial Browser Process Threading Model Cleanup and Re-Design proposal at the end of April to bring down the number of threads: we sat down, digested the feedback, and thought about it some more.

The problem after all is not so much the number of threads per se, but rather the code complexity and lack of flexibility brought by all the dedicated named threads.

Today, Chrome provides many interfaces to post tasks:

- On various dedicated named threads
    - UI/CrBrowserMain, DB, FILE, FILE_USER_BLOCKING, PROCESS_LAUNCHER, CACHE, IO
- On a SequencedWorkerPool (e.g. BlockingPool, CachePool)
- On the WorkerPool (Different from the SequencedWorkerPool!)
- Through PostAfterStartupTask APIs.
- ...and more!

One must choose a specific thread or pool to dispatch work. This means considering the execution semantics of each place:

- Is the throughput of this thread sufficient for my task?
- Is File IO permitted on this thread?
- Are COM calls permitted on this thread?
- etc.

And often no specific thread provides quite the right semantic for a given task:

- High priority I/O? Single queue on FILE_USER_BLOCKING thread…
- High priority DB operation? Same queue as unrelated low priority operations.
- Background work? PostDelayedTask(), PostAfterStartupTask()

Dedicated threads and pools are simple and made sense initially but we believe they no longer scale at Chromium's size today.

What one really wants is to be able to specify the characteristics of what one's task is (e.g., user interaction result depends on it or it's some low priority file I/O). This proposal attempts to abstract away the process of deciding where and when tasks are executed while providing opportunities for eventual performance and power management improvements.

Objectives and Key Results

Objectives

- Simplify asynchronous task management in Chromium
- Provide a *task executor* API which allows for a flexible and evolving implementation of the *task executor*'s implementation(s) while simplifying the developer's involvement in posting tasks (no knowledge of the underlying system required).
- The *task executor*'s API is an evolution of Chromium's current PostTask model and feels natural for Chromium developers.
- The *task executor*'s API allows an in-place migration to the API while keeping the same execution logic as before, leaving logic tweaks as an eventual implementation detail refinement.
- Centralized task management allows for instrumentation and metrics on task flows (ref. tracing, RAIL, perf insights, etc.).

Key Results

- An initial *task executor* implementation which maintains current execution logic is created and tested.
- Chromium is migrated to post tasks through that new *task executor* (still maintaining the current execution flow).
- A second implementation of the *task executor* can refine the execution flow as an implementation detail (without having to touch any of the previously migrated Chromium code).
- 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.).
- The API provides a clear guide for developers to choose the right semantics for their tasks.
- Posters don't have to specify where their task should run, instead they only provide the task's traits and where/when the task runs becomes an implementation detail of the *task executor*.
- 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 right, restricting the execution options of code requiring COM).

- Analyses like RAIL can be run on the tasks to highlight task dependencies and task execution lengths allowing us to see the critical paths and draw additional perf conclusions of the system without manual instrumentation of the tasks.

Overview

We propose introducing an *Executor* interface in Chromium (name inspired from google3).

This interface will be inspired from the existing PostTask APIs in Chromium today and provide semantics that allow all existing callers to migrate to the new implementation.

Similar prior art has proved itself already in various Apple applications that utilize Apple’s Grand Central Dispatch API (libdispatch).

This proposal was written in collaboration with scheduler-dev.

It would likely be implemented by something akin to an enhanced version of today’s SequencedWorkerPool, but before we delve into the implementation, let’s first agree on an API.

The core idea is that the consumer of the Executor API **does not have to** do the work of finding **where** (e.g. previously: dedicated named thread/pool) and **when** (e.g. previously: implicitly encoded by the posting sequence of unrelated tasks on the same thread or even sometimes explicitly as arbitrary delays for low priority tasks) a task should be executed. **Instead**, the user simply specifies **what** their tasks’ traits are and optionally orders them for execution by the *Executor*. In other words, the user takes care of the semantics, the *Executor* takes care of the execution pattern (threads/pools/etc. become implementation details).

Detailed Design

Executor API

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

```
// An Executor is an interface to post asynchronous tasks. Tasks may be marked with TaskTraits
// to hint the ExecutionManager receiving the task about when and where it should run.
// Executors may be marked with ExecutorTraits which then become default traits for tasks
// posted through them.
// Tasks posted through the Executor are guaranteed to run unless shutdown occurs, in which
// case a task's ShutdownBehaviour trait will be honoured.
class Executor : public RefCountedThreadSafe<Executor> {
 public:
  // Handles posting tasks with this Executor's default traits.
  virtual void PostTask(const tracked_objects::Location& from_here,
                        const base::Closure& task) = 0;

  // Handles posting tasks that have specific |traits|.
  // These |traits| will be merged with the underlying Executor's traits.
  virtual void PostTaskWithTraits(const tracked_objects::Location& from_here,
                                  TaskTraits traits,
                                  const base::Closure& task) = 0;

  // Handles posting tasks with this Executor's default traits.
  // On |task|'s completion, |reply| will be scheduled on the current Executor.
  virtual void PostTaskAndReply(const tracked_objects::Location& from_here,
                                const base::Closure& task,
                                const base::Closure& reply) = 0;

  // Handles posting tasks that have specific |traits|.
  // These |traits| will be merged with the underlying Executor's traits.
  // On |task|'s completion, |reply| will be scheduled on the current Executor.
  virtual void PostTaskAndReplyWithTraits(const tracked_objects::Location& from_here,
                                          TaskTraits traits,
                                          const base::Closure& task,
                                          const base::Closure& reply) = 0;

  // Target for all the static Executor:: getters below.
  static void SetExecutionManager(const scoped_refptr<ExecutionManager>& manager);

  // Returns an Executor with specific |traits|.
  // NOTE for the purpose of this doc: the idea in practice is that multiple Executor
  // interfaces would be handed off with specific traits, but the implementation
  // of ExecutionManager would have all tasks posted through those funneled into the
  // same backend which still gets full knowledge of all posted tasks (i.e., each
  // Executor is not its own execution environment, it merely defines different
  // traits for tasks posted through it).
  static scoped_refptr<Executor> GetForTraits(ExecutorTraits traits);

  // Returns an Executor with default traits for one-off tasks.
  static scoped_refptr<Executor> global();

  // Returns an Executor that executes all tasks on the main thread.
  static scoped_refptr<Executor> main();

  // Returns the Executor executing the task making this call (or null if none).
  static scoped_refptr<Executor> current();
};

// TaskTraits hold metadata about a task and prevent the combinatorial explosion
// of PostTaskWith*() call sites.
// Usage example:
//     PostTaskWithTraits(FROM_HERE, TaskTraits().WithFileIO().WithPriority(INTERACTIVE),
//                        base::Bind(...));
```

```
struct Executor::TaskTraits {
  enum class Priority {
    // The user is actively waiting on this.
    INTERACTIVE,
    // Contributes to visible UI or potential future user interactions.
    DISPLAY,
    // Everything else.
    BACKGROUND,
  }

  // Same as in today's SequencedWorkerPool, detailed comment to be copied from there.
  // Posting BLOCK_SHUTDOWN tasks in the late stages of shutdown is not supported
  // (note for this doc: we could support it but that would imply adding a bool return value
  // to PostTask and we haven’t seen proof that this use case alone warrants a custom return
  // type; instead we believe that implementations should NOTREACHED() if a BLOCK_SHUTDOWN
  // task is posted after the ExecutionManager was shutdown as we consider this to be an
  // implementation bug).
  enum class ShutdownBehavior {
    CONTINUE_ON_SHUTDOWN,
    SKIP_ON_SHUTDOWN,
    BLOCK_SHUTDOWN,
  }

  // Forces tasks with these traits to run under |context|.
  TaskTraits& WithContext(scoped_ptr<ThreadContext> context);

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

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

  // Applies a delay to tasks with these traits.
  TaskTraits& WithDelay(base::TimeDelta delay);

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

 private:
  (...)
};

// ExecutorTraits provide the TaskTraits to be inherited by tasks posted through the
// associated Executor plus some Executor specific traits around execution semantics of
// tasks posted through the associated Executor.
struct Executor::ExecutorTraits : public Executor::TaskTraits {
  enum class Strategy {
    // Can execute multiple tasks at a time.
    // High priority tasks will execute first under contention.
    PARALLEL,
    // Executes one task at a time (no order guaranteed).
    // High priority tasks may jump the queue.
    SERIAL,
    // Executes one task at a time (posting order guaranteed).
    // High priority tasks will spill their priority to tasks ahead of them in the queue.
    SEQUENCED,
  };

  // Builds an ExecutorTraits with an explicit |strategy|.
  ExecutorTraits(Strategy strategy);

  // An explicit token for serializing tasks posted by this Executor.
  // This should not be required in most execution contexts but can be useful for
  // operations at opposite end of the codebase that need to be serialized
  // (e.g. iterating over desktop shortcuts) but for which sharing a global Executor
  // object isn't practical.
  ExecutorTraits& WithSerialToken(const std::string& serial_token);
};

// ThreadContext tells the Executor to set up a dedicated thread for this |thread_key| type.
// If a thread of this |thread_key| type already exists, the Executor may decide to either
// create a new one or share it with the other instance. An empty |thread_key| indicates that
// any thread can be selected as this Executor's dedicated thread.
// InitializeThreadContext/UninitializeThreadContext() will be called once per ThreadContext
// on their dedicated thread. The dedicated thread's lifetime is guaranteed to be at least as
// long as the ThreadContext associated to it.
class ThreadContext {
 public:
  virtual void InitializeThreadContext() = 0;
  virtual void UninitializeThreadContext() = 0;

 protected:
  ThreadContext(StringPiece thread_key);
  ~ThreadContext() {
  }
}

// A ThreadContext which initializes its threads in a COM STA, guaranteeing that tasks
// posted to the Executor containing this ThreadContext will run on the same COM initialized
// thread. Also makes sure to pump Windows messages posted to its thread for COM.
// The COM STA is released when this object is destroyed.
class StaThreadContext : public ThreadContext {
 public:
  StaThreadContext() : ThreadContext("STA") {}

  void InitializeThreadContext() override;  // CoInit COM
  void UninitializeThreadContext() override;  // CoUninit COM
}

// A ThreadContext that represents a single thread, guaranteeing that tasks posted to the
// Executor containing this ThreadContext will run on the same thread.
class SingleThreadContext : public ThreadContext {
 public:
  StaThreadContext() : ThreadContext("") {}

  void InitializeThreadContext() override; // Do nothing
  void UninitializeThreadContext() override;  // Do nothing
}

class ExecutionManager {
  // Returns an Executor with specific |traits|.
  virtual scoped_refptr<Executor> GetForTraits(ExecutorTraits traits) = 0;

  // Returns an Executor with default traits for one-off tasks.
  virtual scoped_refptr<Executor> global() = 0;

  // Returns an Executor that executes all tasks on the main thread.
  virtual scoped_refptr<Executor> main() = 0;

  // Returns the Executor executing the task making this call (or null if none).
  virtual scoped_refptr<Executor> current() = 0;
};
```

What about strongly typed Executors?

TaskRunners today have strongly typed interfaces enforcing properties at compile time (e.g., SequencedTaskRunner/SingleThreadTaskRunner).

The Executor will not support strongly typed interfaces, and here is why this is okay (and actually even a good thing):

- **Today** well-designed classes often **receive a TaskRunner** abstraction as a constructor parameter. This is nice because it allows the class to **not worry about where it runs besides assuming properties bound to the strongly typed interface**.
- But **with Executor**, a core component of the design is that **the user doesn't need to specify where its tasks are to run**, but **merely how they are to run**. Therefore instead of requiring being handed off an Executor with the proper traits in its constructor, it should just get one itself :-).
- This can even be seen as a code complexity win over the previous model because the class' impl now has all the information a reader needs to understand its task flow in a single file.

In the heavily componentized world we're going towards, this will remove the need for dependency injection of TaskRunners everywhere.

Migration Plan

1. **Introduce:** Add the API and an ExecutionManager implementation which initially has a dedicated thread per sequence.
2. **Behind the scenes**: Redirect all tasks posted to dedicated threads to dedicated sequences (Executors) for these threads (and update tests as well).
3. **Evolve:** Support the execution of multiple sequences on a restricted set of threads in the ExecutionManager's implementation.
4. **Over time**: Refactor independent chunks from the dedicated sequences and move them to their own independent execution context (Executor).
5. **Eventually**: Deprecate and remove unused dedicated threads and associated infrastructure.
6. **Long term**: Experiment with more advanced scheduling techniques in the ExecutionManager's impl.

How do existing BrowserThreads map to the Executor?

Initially tasks posted to each thread will simply be redirected to that “thread”'s dedicated single threaded sequence on the Executor. Posting to BrowserThreads will be deprecated and over time we will remove them by redirecting everybody directly to the Executor (and thus off of their implicit sequencing with other mostly unrelated tasks).

Priorities will be required to map today's dedicated threads into the Executor. Here is the outline of what this may look like for BrowserThreads:

	WithFileIO()	**Normal**
Interactive **Display**	PROCESS_LAUNCHER FILE_USER_BLOCKING CACHE? History CachePool	UI: Extracted non UI thread affine tasks executed for **foreground** items
Background	FILE DB BlockingPool WorkerPool	UI: Extracted non UI thread affine tasks executed for **background** items

Enabled Use Cases

- Dynamically adapt to the task execution environment:
  - To achieve the best throughput vs resource utilization balance possible (things like CPU and battery/energy utilization optimization are only possible now that all tasks are known by a single entity).
- Better power management:
  - Can control when low priority tasks execute and avoid power intensive bursts in idle periods.
- --single-thread mode for the browser process:
  - The embedder of the ExecutionManager could decide to only use one thread for all tasks == easier live debugging in some scenarios.
- Deeper task analysis (ref. perf insights -- nduca@ et al.):
  - Ability to track groups of tasks with the same traits.
  - Ability to track each task's origin (i.e. the task that posted it itself rather than just FROM_HERE).
- Different priorities for tasks in the same series:
  - For example JsonPrefStore today runs off of a single SequencedTaskRunner which means prefs read/writes are made from the same execution context (BlockingPool) while their only common requirement is to be serialized (not executed in parallel).
    - One would like to have reads on the FILE_USER_BLOCKING thread and writes on the BlockingPool but this isn't possible without some extra self-managed synchronisation in JsonPrefStore. And even then a single thread for "high priority" reads could still result in high priority task contention while low priority tasks being run in parallel (this is where *Executor* can help :-)).
- Ability to participate in ARM's upcoming BIG.little's "Application guided capacity allocation (QoS)" which lets applications specify performance requirements of each threads (rather than using kernel heuristics which often get things backward) (reference).
  - Having a single place aware of all tasks is key to being able to dynamically hint the kernel.

Code Samples Utilizing the API

```
using Executor::TaskTraits;
using Executor::TaskTraits::Priority;
```

Posting generic background tasks

(from chrome_browser_main.cc:1501)

```
  if (NetworkProfileBubble::ShouldCheckNetworkProfile(profile_)) {
~~    BrowserThread::PostTask(~~
~~        BrowserThread::FILE, FROM_HERE,~~
~~        base::Bind(&NetworkProfileBubble::CheckNetworkProfile,~~
~~                   profile_->GetPath()));~~
    Executor::global()->PostTaskWithTraits(
        FROM_HERE, TaskTraits().WithFileIO(),
        base::Bind(&NetworkProfileBubble::CheckNetworkProfile,
                   profile_->GetPath()));
  }
```

(from pepper_flash_settings_manager.cc:445)

```
~~  BrowserThread::PostBlockingPoolTask(FROM_HERE,~~
~~      base::Bind(&Core::DeauthorizeContentLicensesOnBlockingPool, this,~~
~~                 request_id, browser_context_path_));~~
  Executor::global()->PostTaskWithTraits(
      FROM_HERE, TaskTraits().WithFileIO(),
      base::Bind(&Core::DeauthorizeContentLicensesOnBlockingPool, this,
                 request_id, browser_context_path_));
```

(from render_message_filter.cc:609)

```
  // Dispatch to worker pool, so we do not block the IO thread.
~~  if (!base::WorkerPool::PostTask(~~
~~          FROM_HERE,~~
~~          base::Bind(&RenderMessageFilter::OnKeygenOnWorkerThread,~~
~~                     this,~~
~~                     base::Passed(&keygen_handler),~~
~~                     reply_msg),~~
~~          true)) { (...)~~
  Executor::global()->PostTaskWithTraits(
        FROM_HERE, TaskTraits().WithPriority(Priority::BACKGROUND),
        base::Bind(&RenderMessageFilter::OnKeygenOnWorkerThread,
                   this,
                   base::Passed(&keygen_handler),
                   reply_msg));
```

… and in fact since BACKGROUND is the default priority level, this could simply be:

```
   Executor::global()->PostTask(
        FROM_HERE,
        base::Bind(&RenderMessageFilter::OnKeygenOnWorkerThread,
                   this,
                   base::Passed(&keygen_handler),
                   reply_msg));
```

Main difference: Same call for all ad-hoc background tasks, no longer need to pick amongst many similar entry points for background work.

Google Update and Windows COM

```
int BrowserMainLoop::CreateThreads() {
```

```
[...]
     case BrowserThread::FILE:
        TRACE_EVENT_BEGIN1("startup",
            "BrowserMainLoop::CreateThreads:start",
            "Thread", "BrowserThread::FILE");
        thread_to_start = &file_thread_;
~~#if defined(OS_WIN)~~
~~        // On Windows, the FILE thread needs to be have a UI message loop~~
~~        // which pumps messages in such a way that Google Update can~~
~~        // communicate back to us.~~
~~        options = ui_message_loop_options;~~
~~#else~~
        options = io_message_loop_options;
~~#endif~~
[...]
}

void VersionUpdaterWin::BeginUpdateCheckOn~~File~~BackgroundThread(
    bool install_update_if_possible) {
  BeginUpdateCheck(~~content::BrowserThread::GetMessageLoopProxyForThread(~~
~~                       content::BrowserThread::FILE),~~
                   g_browser_process->GetApplicationLocale(),
                   install_update_if_possible, owner_widget_,
                   weak_factory_.GetWeakPtr());
}

void BeginUpdateCheck(
    ~~const scoped_refptr<base::SingleThreadTaskRunner>& task_runner,~~
    const std::string& locale,
    bool install_update_if_possible,
    gfx::AcceleratedWidget elevation_window,
    const base::WeakPtr<UpdateCheckDelegate>& delegate) {
  UpdateCheckDriver::RunUpdateCheck(~~task_runner, ~~locale,
                                    install_update_if_possible,
                                    elevation_window, delegate);
}

// Set by constructor with
// Executor::GetForTraits(
//     new ExecutorTraits(Strategy::SERIAL).WithContext(new StaThreadContext));
scoped_refptr<Executor> UpdateCheckDriver::sta_executor_;

scoped_refptr<Executor> UpdateCheckDriver::sta_executor() {
  return sta_executor_;
}

// static
void UpdateCheckDriver::RunUpdateCheck(
~~    const scoped_refptr<base::SingleThreadTaskRunner>& task_runner,~~
    const std::string& locale,
    bool install_update_if_possible,
    gfx::AcceleratedWidget elevation_window,
    const base::WeakPtr<UpdateCheckDelegate>& delegate) {
```

```
  // The driver is owned by itself, and will self-destruct when its work is
  // done.
  UpdateCheckDriver* driver =
      new UpdateCheckDriver(~~task_runner, ~~locale, install_update_if_possible,
                            elevation_window, delegate);
  ~~task_runner->PostTask(FROM_HERE,~~
  driver.sta_executor()->PostTask(FROM_HERE, base::Bind(&UpdateCheckDriver::BeginUpdateCheck,
                                   base::Unretained(driver)));
}
```

Main difference: In Windows, the FILE thread initializes COM just for Google Update (though more call sites have likely grown to depend on this and the FILE sequence will likely need to run in an STAThreadContext until it is fully deprecated). In this version, Google Update can manage its own COM context instead of relying on a far away initialization in CreateThreads.

Interface using SequencedTaskRunner

Inspired from JsonPrefStore.

```
using Executor::ExecutorTraits;
using Executor::ExecutorTraits::Strategy;
JsonPrefStore::JsonPrefStore(
    const base::FilePath& pref_filename,
    const base::FilePath& pref_alternate_filename,
    ~~const scoped_refptr<base::SequencedTaskRunner>& sequenced_task_runner,~~
    scoped_ptr<PrefFilter> pref_filter)
    : ~~sequenced_task_runner_(sequenced_task_runner),~~
      executor_(Executor::GetForTraits(ExecutorTraits(Strategy::SEQUENCED).WithFileIO())),
      [...] {
  DCHECK(!path_.empty());
}

void JsonPrefStore::ReadPrefsAsync(ReadErrorDelegate* error_delegate) {
  DCHECK(CalledOnValidThread());

  initialized_ = false;
  error_delegate_.reset(error_delegate);

  // Weakly binds the read task so that it doesn't kick in during shutdown.
~~  base::PostTaskAndReplyWithResult(~~
~~      sequenced_task_runner_.get(),~~
~~      FROM_HERE,~~
~~      base::Bind(&ReadPrefsFromDisk, path_, alternate_path_),~~
~~      base::Bind(&JsonPrefStore::OnFileRead, AsWeakPtr()));~~
  // Note: An Executor-friendly version of task_runner_util.h will be required.
  base::PostTaskAndReplyWithResultWithTraits(
      executor_.get(), FROM_HERE, TaskTraits().WithPriority(INTERACTIVE),
      base::Bind(&ReadPrefsFromDisk, path_, alternative_path_),
      base::Bind(&JsonPrefStore::OnFileRead, AsWeakPtr()));
}
```

Main difference:

1. The creator of JsonPrefStore no longer needs to provide it an execution context and the JsonPrefStore no longer needs to rely on that context allowing I/O. The JsonPrefStore simply requests an execution context with the semantics it needs without otherwise needing to depend on where the tasks will actually run.
2. The read can be given an INTERACTIVE priority while the default priority for writes remains BACKGROUND. This wasn't possible before as sequences were bound to a specific thread (or pool of threads with the same properties).

Non Goals For v1

The core goal for v1 is to be able to replicate the task management of the browser process as we have it today but under a single umbrella (which means as long as tasks are all funnelled through the Executor, it's okay if the implementation still uses a dedicated thread per sequence at first).

It is however a non-goal for v1 to expose advanced traits/etc. that would ultimately make for better scheduling but aren't required for the initial migration nor until we have a good enough backing implementation that can start making use of such signals.

We are on the flip side doing everything we can to keep the API as flexible as possible to such future additions and would welcome feedback highlighting lack of flexibility in that regards.

Explicit non-goals for v1 are thus (could also be seen as potential features for v2):

- Non-browser-process use cases
    - While we would love for all of Chromium to eventually post tasks through the Executor, v1 is focusing on the browser process' use case.
- Converge all scheduling systems (e.g. network) in Chromium to this system.
    - If we can evolve this system to handle the other cases, great, but it's not a goal for v1.
- Dynamic priority (generic traits/voters)
    - We would eventually like to be able to attach generic traits which are foreign to an implementation in base/ (e.g. associated WebContent) but whose priority could be interpreted by registered voters (e.g. foreground vs background tab) -- brain dump of what this could look like, the afore-proposed Executor interface is ready to support this as extra TaskTraits, but initially won't in v1 for simplicity.
- Explicit Cross-process Coordination
    - While the tasks themselves may coordinate with other processes, for v1 Executor coordination of these other processes is out of scope.

- Time range instead of time delay
    - tansell@ suggested having a min/max range of when a task should run instead of only a min (delay) will help scheduling timer-based tasks. This is not required for v1, but something to keep in mind.
- Promises
    - We have been tinkering with the idea of PostTask returning a Promise object allowing chaining of follow-up tasks, but it isn't a requirement for v1.

Caveats

During the transitional stage, both the current task infrastructure and Executor system will coexist. This means there will temporarily be yet another location to post tasks. This state should only be temporary as all call sites are being migrated into the Executor system. After that point, we can start decommissioning the old infrastructure.

Benefits

The main benefit we are going after is **reducing code complexity**. Giving developers a single place to post all tasks to with the ability to provide the right semantics for the task to run where and when it really should.

Having a trait-based task API will make it easier to perform **analyses that track task causality**. This is a key enabler for RAIL analysis that requires identifying all the tasks that resulted from a given user interaction. At the moment, the absence of meta-information on PostTask makes it hard to distinguish tasks that are required to update the frame (i.e. highlighting a button when the user clicks on it) from tasks that are merely nice-to-have and can happen later.

Having a centralized location for tasks will allow an increase in the amount of parallelism. From our experience looking at startup performance (too little parallel critical work contending with too much parallel background work was identified as a core issue; e.g. appendix A) we think this will **help startup performance as well as potentially reduce jank post startup**.

It may even provide an opportunity to experiment with **smarter power management** by controlling background tasks throughput and thus avoiding unnecessary surges while idle.

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

One thing to consider is how test frameworks that run multiple tests in sequence in the same execution context (e.g. unit_tests) cope with the Executor. Since the proposed Executor's implementation would likely be embedded by content/, unit_tests requiring an Executor would embed a test ExecutionManager instead (akin to TestBrowserThreadBundle today) which would self-clean as part of each test fixture's destruction.

An advantage of controlling the embedded ExecutionManager is that tests could be configured to be single-threaded (already true today with TestBrowserThreadBundle, but the Executor's interface spans a larger portion of the codebase).
It could even allow Chrome to a have a single-threaded mode if that's deemed useful for debugging.

Document History

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

If you just reviewed this document, please add your Chromium ID to the latest row via Google Docs' *Suggesting mode*.

Date	Author	Description	Reviewed by	Signed off by
Aug. 2015	gab, robliao	Initial draft		
Aug. 27 2015	beaudoin	Review, added benefit for RAIL.	beaudoin, grt, robertshield, asvitkine, skyostil, alexclarke, tansell, brianderson	
September	gab,	Forked and simplified document.	grt, skyostil,	

22, 2015	robliao		alexclarke, dcheng, avi	
October 6, 2015	gab, robliao	Presented at Chrome Scheduling Summit (go/chromesched2015)	go/chromesched2015-attendees	
October 19, 2015	gab, robliao	First official revision sent to eng-review.	go/chrome-eng-review	
November 5, 2015	gab,robliao	Concept (modulo integration approach) approved at eng-review, proceeding to writing implementation plan.		

Appendix

Appendix A: EarlyInit Startup Experiment

The EarlyInitStartup experiment moved the extensions API initialization codepath off of CrBrowserMain (on the FILE_USER_BLOCKING thread). This particular work runs around 170ms at the 75% percentile (gathered from Extensions.FeatureProviderStaticInitTime).

Below is a table for Windows at the 75% percentile, showing gains in the order of 170ms.

This is great and highlights the problem today that the main thread races to I/O and isn't making efficient use of system resources to solve the critical path. We would like to force more things to be loaded before the critical path needs them, but queuing all of these back-to-back on the FILE_USER_BLOCKING thread while other threads are happily contending with non-critical work doesn't make much sense. This proposal gets rid of this problem entirely by stripping the notion of stating **where** tasks should run.

CANARY-W 1 Day - 75% Percentile - Startup.FirstWebContents.NonEmptyPaint
Green indicates better result

Date	**EarlyInitStartup_Control**	**EarlyInitStartup**
M44 Tue 5/12	17138+/- 544	🟩 16976+/- 506
M44 Wed 5/13	17859+/- 627	🟩 17598+/- 630
M44 Thu 5/14	🟩 17611+/- 484	18000+/- 539

M44 Fri 5/15	18071+/- 554	17701+/- 759
M44 Sat 5/16	17977+/- 741	17472+/- 863
M45 Sun 5/17	13821+/- 530	13712+/- 611
M45 Mon 5/18	16166+/- 484	15862+/- 472
M45 Tue 5/19	16088+/- 561	16344+/- 577
M45 Wed 5/20	16089+/- 502	16472+/- 628
M45 Thu 5/21	16768+/- 746	16459+/- 432
M45 Fri 5/22	16769+/- 719	16463+/- 450
M45 Sat 5/23	16977+/- 444	16714+/- 442

Appendix B: Sample of Chrome Browser Process on Mac with 74 threads

After regular use for a few days with 2 active profiles… This proposal removes the need for many of these dedicated threads and thus avoids this bloated thread count situation.

AudioThread
BrowserBlockingWorker1/33539
BrowserBlockingWorker2/33795
BrowserBlockingWorker3/59139
BrowserWatchdog
CachePoolWorker1/112131
CachePoolWorker2/130819
CachePoolWorker3/137091
CachePoolWorker4/136843
CachePoolWorker5/137267
Chrome_CacheThread
Chrome_DBThread
Chrome_FileThread
Chrome_FileUserBlockingThread
Chrome_HistoryThread
Chrome_HistoryThread
Chrome_HistoryThread
Chrome_HistoryThread
Chrome_HistoryThread
Chrome_IOThread

```
Chrome_PasswordStore_Thread
Chrome_PasswordStore_Thread
Chrome_PasswordStore_Thread
Chrome_PasswordStore_Thread
Chrome_ProcessLauncherThread
Chrome_SyncThread
Chrome_SyncThread
Chrome_SyncThread
com.apple.CFSocket.private
CompositorTileWorker1/44035
CrShutdownDetector
DispatchQueue_10: com.apple.root.default-qos  (serial)
DispatchQueue_1: com.apple.main-thread  (serial)
DispatchQueue_2: com.apple.libdispatch-manager  (serial)
DnsConfigService
handle-watcher-thread
IndexedDB
LevelDBEnv
LevelDBEnv.IDB
NetworkConfigWatcher
NetworkConfigWatcher
NetworkConfigWatcher
NetworkConfigWatcher
NetworkConfigWatcher
NetworkConfigWatcher
NetworkConfigWatcher
NetworkConfigWatcher
org.libusb.device-hotplug
Platform Key ThreadWorker1/169195
PowerSaveBlocker
Proxy Resolver
Proxy Resolver
Proxy Resolver
Proxy Resolver
Proxy Resolver
Proxy Resolver
Proxy Resolver
Service Discovery Thread
SimpleCacheWorker1/106527
SimpleCacheWorker2/132107
SimpleCacheWorker3/7167
SimpleCacheWorker4/116795
SimpleCacheWorker5/106311
```

UsbEventHandler/167171
WorkerPool/119939
WorkerPool/125243
WorkerPool/129827
WorkerPool/160139
WorkerPool/177539
WorkerPool/179151
WorkerPool/183463
WorkerPool/189199
WorkerPool/195695
WorkerPool/215175

Appendix C: Existing TaskRunner implementations.

For the record, here's a list of today's core TaskRunner implementations used in the browser process (full list here):

1. MessageLoopTaskRunner
    a. Many instances: one per dedicated named threads.
2. DeferredSequencedTaskRunner
3. WorkerPoolTaskRunner
4. SequencedWorkerPool
5. SequencedWorkPoolTaskRunner
6. SequencedWorkPoolSequencedTaskRunner
7. TestSimpleTaskRunner
8. AfterStartupTask
    a. Not a TaskRunner per se, but uses one and is also superseded by Executor.

All of these can be deprecated in favor of the Executor.

Appendix D: Early Perf Insights results pointing fingers at async I/O contention

An early bulk comparison of slow vs fast startup from traces (obtained from users via Slow Reports) shows that slow startups have a much higher relative time spent in asynchronous tasks (especially I/O) then fast startups do. The key part here is "*relative time spent*", i.e. it's not just that those tasks are linearly taking more time, they take a bigger portion of the overall startup (i.e. their presence correlates with slow startups). The Executor will let us have fine grain control over when these tasks get to run, allowing a better throughput of the critical path on startup.

Full report here.