The Future of TaskRunnerHandles

Status: chromium-dev review complete, conclusion below.
*This document is **PUBLIC**.*
(sign in to comment -- request comment access if your account isn't in the broad set already allowed)
CL, BUG
Author: gab@chromium.org
Last Updated: 2016-07-13

Conclusion

This conclusion overrides this document and every other side-discussion.

Conclusion from chromium-dev thread:

Quoting myself (gab@) from thread:
"""
Keep ThreadTaskRunnerHandle and SequencedTaskRunnerHandle (forget doc proposal). Ban implicit creation of sequences. Force any caller that uses a component that cares to have a SequencedTaskRunnerHandle available to create a sequence token (in our canonical example, the poster of LoadSTHsFromDisk would have to not post directly to the BlockingPool but rather to a sequence on it).

That solves my immediate problem and puts the burden on the occasional odd caller which I think is reasonable.

While we're on this topic: do we think PostTaskAndReply should only be supported on SequencedTaskRunners? I'm happy to make this change (it "should" be a simple replacement per the impl actually only working from SingleThreadTaskRunners in practice and existing usage thus always coming from one today in theory -- though some (hopefully not many :-S!) callsites may have a generic TaskRunner which happens to be a more specific one).

And I would of course also add support for PostTaskAndReply to work from sequences in practice (have PostTaskAndReplyRelay use SequencedTaskRunnerHandle instead of ThreadTaskRunnerHandle) -- another prereq for the TaskScheduler.
”””

brettw@:
“This all sounds right to me. We can always relax the rules later if we find we're too limited.”

Objective

Come up with semantics for TaskRunnerHandles that cover existing *intended* use cases and are easily supportable by non single-threaded runners (e.g. upcoming base/task_scheduler).

Background

TaskRunnerHandles are global getters intended to get a TaskRunner to execute tasks (now or later) in the same *context* as the current task.

In Chromium, most TaskRunners are backed by MessageLoops (and are in fact SingleThreadTaskRunners) and a few are backed by the SequencedWorkerPool (and are most often SequencedTaskRunners -- unless referring to the pool itself which is a plain TaskRunner).

scoped_refptr<SingleThreadTaskRunner> ThreadTaskRunnerHandle::Get(); has been around for a while and worked fine to refer back to the current MessageLoop.

ThreadTaskRunnerHandle was recently deemed over-specific for components that desired to be equally callable from a SequencedWorkerPool and along came **scoped_refptr<SequencedTaskRunner> SequencedTaskRunnerHandle::Get();** as many components didn’t care about thread-affinity per se but merely about *same context* and a SequencedTaskRunner in the current *context* would work just as well.

These are used today to:

- **Use case #1.1:** Store TaskRunner to reply to caller in its *context* later (caller provides Callback for reply but not TaskRunner which is derived from TaskRunnerHandle and saved for later).
    - e.g. SafeJsonParserImpl::Start()

- **Use case #1.2:** Synchronously post back to caller from current *context*.
    - (identified use case was argued to instead be a case of 1.4 and I'm not finding another one easily, doesn't affect the premise of this doc though)
- **Use case #1.3:** Asynchronous error handling from *context* it's registered on.
    - e.g. BrowserX11ErrorHandler()
- **Use case #1.4:** Queue task at end of *current sequence*, e.g.:
    - To make sure any pre-queued initialization completes: HttpServer::HttpServer()
    - To prevent re-entrancy: CommandBufferProxyImpl::DisconnectChannelInFreshCallStack()
    - To let the stack unwind and pending tasks complete on *current sequence* before running this task (a common net/ idiom): FuzzedSocket::Read()
- **Use case #1.5:** Split long tasks by re-enqueuing remaining work on *current sequence* after processing an item, yielding to queued work on the *same sequence*.
    - e.g. HttpCache::OnBackendCreated()
- **Use case #1.6:** Hand *current sequence* to sub-component for it to communicate with owner on it.
    - e.g. new CrossThreadNetworkEventObserver()

This can also be used to:

- **Use case #2:** Use the *current sequence/thread* when creating a component which takes a (Sequenced|SingleThread)TaskRunner as input (to keep the owner's code generic: either to support it running in multiple contexts or simply to avoid taking a dependency on which context that happens to be in practice).

And lastly to:

- **Use case #3:** Get a reference to the main(UI)/IO runners from components that know by contract which *context* they're invoked on.
    - e.g. SimpleBackendImpl::Init()

The Problem

SequencedTaskRunnerHandle was a great step towards more generic TaskRunnerHandles that let the callers specify the semantics they need (the majority of use cases presented above don't actually need the specificity of SingleThreadTaskRunner's API).

But note that **use cases 1.1, 1.2, and 1.3**'s semantics can still go down a notch from depending on SequencedTaskRunner and instead would be happy with merely a TaskRunner that represents the *current context* (context == MessageLoop or sequence on SequencedWorkerPool/TaskScheduler or merely a SequencedWorkerPool if task isn't on a sequence or merely TaskTraits of a one off task on TaskScheduler).

The reason we need to go one-level down from SequencedTaskRunnerHandle is that today it supports being called even from a one-off parallel task by implicitly creating a new sequence with the current task at its head (and some components in Chromium actually depend on this behavior, i.e. SafeJsonParserImpl::Start() will hit DCHECK in SequencedTaskRunnerHandle::Get() on browser startup if this ability is removed today).

Supporting returning a SequencedTaskRunner from the context of a one-off task in TaskScheduler would be ugly and doesn't feel right per this not being the caller's intent (i.e. use cases #1.1-1.2 above which don't actually want/need a sequence).
It feels wrong to bind the current one-off task's TaskTraits to a sequence when, again, that's not what the caller wants.

The Proposal

Essentially the proposal is to introduce **TaskRunnerHandle::Get()** for callers that merely want the current context without caring for sequencing/etc. It would inherit SequencedTaskRunnerHandle::Get() (which already inherits ThreadTaskRunnerHandle::Get()) and current impls would thus work as-is while providing extra flexibility for those that care to use it.

But instead of adding yet another header I propose we merge them all in a single API (deprecating and redirecting existing ones : ~~ThreadTaskRunnerHandle / SequencedTaskRunnerHandle~~ => TaskRunnerHandle). Having a single API helps developers know that multiple choices are available and pick the right one from the side-by-side documentation (as opposed to people copying ThreadTaskRunnerHandle usage all over the place without requiring thread-affinity say).

Note: this proposal is not about a mass refactor. Each use case will be attended one-by-one over time as the over-specifity of its current TaskRunnerHandle usage gets in the way. This proposal is about (1) making the API more flexible to allow changing problematic use cases as they are identified and (2) changing the recommandation for new code to encode the semantics it needs from the get go (we will add PRESUBMITs that the old API is no longer used to force these choices going forward).

API

Full CL @ https://codereview.chromium.org/2042383004/

```
// tl;dr; use:
//   - GetSingleThreaded()
//         for tasks bound to thread-affine state requiring sequencing on a
//         single-thread (should be fairly rare, most state only requires the
//         implicit memory safety of a sequence)
```

```
//   - GetSequenced()
//         for tasks requiring sequencing/thread-safety (e.g. involving thread
//         unsafe state -- the sequence implicitly guarantees memory barriers).
//  - GetAny()
//         for anything else (e.g. calling a caller provided callback -- it's  the
//                            one responsible for memory safety)
class BASE_EXPORT TaskRunnerHandle {
 public:
  // Returns a TaskRunner which will run tasks in the same context as the
  // current task. That context could, for example, be: a MessageLoop, a
  // sequence on a thread pool, or merely the current set of TaskTraits on a
  // non-sequenced TaskScheduler task (in which case the posted task may run in
  // parallel before the current task completes).
  static scoped_refptr<TaskRunner> GetAny();

  // Returns a SequencedTaskRunner associated with the current sequence. This
  // must only be called if HasSequencedTaskScope(). Prefer Get() to this when
  // sequencing (thread-safety) is not required. TODO(gab): Migrate all
  // callsites using SequencedTaskRunnerHandle::Get() to TaskRunnerHandle::Get()
  // and remove support for getting a SequencedTaskRunner from a non-sequenced
  // task in SequencedWorkerPool which currently breaks the semantics intended here.
  static scoped_refptr<SequencedTaskRunner> GetSequenced();

  // Returns a SingleThreadTaskRunner associated with the current single-
  // threaded sequence. This must only be called if
  // HasSingleThreadedTaskScope(). Only use this when thread-affinity is
  // required, prefer Get() or GetSequenced() otherwise.
  static scoped_refptr<SingleThreadTaskRunner> GetSingleThreaded();

  // Returns true if the current task is running as part of any TaskScope.
  static bool HasTaskScope();

  // Returns true if the current task is running as part of a SequencedTaskScope
  // or a SingleThreadTaskScope. Note: This currently also supports
  // SequencedWorkerPool's threads although they don't explicitly have a
  // SequencedTaskScope but this support will be going away with the migration
  // to the TaskScheduler.
  static bool HasSequencedTaskScope();

  // Returns true if the current task is running as part of a
  // SingleThreadTaskScope.
  static bool HasSingleThreadTaskScope();

  // Each task should run in one and only one TaskScope (TaskScopes are held on the
  // stack in the scope of a task and no TaskScope can stack within another
  // TaskScope). The more specific Scopes enable the more generic getters but
  // not vice-versa (e.g. SequencedTaskScope enables Get() and GetSequenced()
  // but not GetSingleThreaded()).
```

```
// A TaskScope which, throughout its lifetime, identifies a default context
// (thread/sequence/etc.) that tasks can be posted to on the thread it lives on.
class TaskScope {
 public:
  TaskScope(scoped_refptr<TaskRunner> task_runner);
  ~TaskScope();

 private:
  friend class TaskRunnerHandle;

  // Immutable TaskRunner state.
  const scoped_refptr<TaskRunner> task_runner_;

  DISALLOW_COPY_AND_ASSIGN(TaskScope);
};

// A SequencedTaskScope which, throughout its lifetime, identifies the
// sequenced context (SequencedTaskRunner) in which tasks on the thread it
// lives on run in.
class SequencedTaskScope {
 public:
  SequencedTaskScope(
      scoped_refptr<SequencedTaskRunner> sequenced_task_runner);
  ~SequencedTaskScope();

 private:
  friend class TaskRunnerHandle;

  // Immutable SequencedTaskRunner state.
  const scoped_refptr<SequencedTaskRunner> sequenced_task_runner_;

  DISALLOW_COPY_AND_ASSIGN(SequencedTaskScope);
};

// A SingleThreadedTaskScope which, throughout its lifetime, identifies the
// single- threaded context (SingleThreadedTaskRunner) in which tasks on the
// thread it lives on run in.
class SingleThreadTaskScope {
 public:
  SingleThreadTaskScope(
      scoped_refptr<SingleThreadTaskRunner> single_thread_task_runner);
  ~SingleThreadTaskScope();

 private:
  friend class TaskRunnerHandle;

  // Immutable SingleThreadTaskRunner state.
```

```
    const scoped_refptr<SingleThreadTaskRunner> single_thread_task_runner_;

    DISALLOW_COPY_AND_ASSIGN(SingleThreadTaskScope);
  };

 private:
  DISALLOW_IMPLICIT_CONSTRUCTORS(TaskRunnerHandle);
};
```

This also comes with the nice property that callers using `TaskRunnerHandle::GetSequenced()` now verify the assumption of running on a sequence (DCHECK in `TaskRunnerHandle::GetSequenced()` instead of previously potentially creating a sequence unintentionally).

Work Estimates

The only “hard” part is agreeing on the API we want for the future. Then implementing it is easy (CL1, CL2) and migrating existing call sites also is (i.e. need to migrate a handful callers of SequencedTaskRunnerHandle::Get() (CL3); callers of ThreadTaskRunnerHandle::Get() will be handled one-by-one when/if affected by the TaskScheduler migration).

Caveats

Use case #2 is broken when done from a one-off task (no longer implicitly creating a sequence). I don’t think this use case actually happens in practice today and I also don’t think this is a theoretical problem as components that take a TaskRunner as input (for their own internal work, not those that are part of a system sharing a given TaskRunner -- e.g. ui::Compositor) should be converted to getting their own TaskRunner in the TaskScheduler world as base::CreateTaskRunnerWithTraits() will finally let a component get a TaskRunner that does exactly what it wants instead of requesting one as a parameter. The only thing we’ll probably want to add is something like
`static TaskTraits TaskTraits::TaskTraitsWithCurrentPriority();`
which would return a new TaskTraits with the priority bit initialized to the current context’s -- all other traits are about the type of work the component wants to do, not what the caller was doing.

Alternatives

1. Keep existing APIs (modulo implicit sequence creation) and workaround necessary use cases

AFAICT (DCHECKs in live build when feature removed), the only practical use case of implicitly creating a sequence from a one-off task today is SafeJsonParserImpl::Start() when it's called from STHSetComponentInstallerTraits::LoadSTHsFromDisk() which runs as a one-off task in the BlockingPool. The fact that this is a caller though is irrelevant to SafeJsonParserImpl's design and it would be just as correct for it to post back to the caller's TaskRunner instead of SequencedTaskRunner.

An alternative to the above proposal would be to still prevent implicit sequence creation and either (1) force callers to call from a sequence or (2) have APIs that require this take a TaskRunner as input instead of using SequencedTaskRunnerHandle.

Given the requirement comes from the caller not the API though I think it's weird to have the API change because of this and would much rather have the API (i.e. SafeJsonParserImpl) encode its requirements (i.e. a mere TaskRunnerHandle::GetAny()) than have either the caller create an uncessary sequence or have the API take an extra parameter (APIs would have to change after the fact as such callers arise which just feels wrong).

2. Get rid of *TaskRunnerHandle::Get*() APIs altogether and expose static *TaskRunner::Get() methods instead

The idea of static *TaskRunner::Get() was brought up recently in other contexts. Instead of merging all APIs under one we could get rid of them altogether and have each TaskRunner type expose a static Get() call which would work the same.

My vote is still on the TaskRunnerHandle proposal above as it documents all choices in one header instead of having each static Get() call refer to the other ones cross-headers.