Starting a thread should not be janky (PUBLIC)

kinuko@ Mar 2015 (Last updated: April 14, 2015)

Tracking bug: https://crbug.com/465458
Patches: 1 2 3
Supporting document that may help understanding: Anatomy of Chromium MessageLoop

TL;DR: This document discusses making base::Thread::Start() return without waiting for the thread to start in order to remove jank on the IO and UI thread especially during startup. There are several possible approaches, and this document discusses each.

Background:

Chrome (and Blink) uses base::Thread class for creating and starting a new thread, but it is suspected that Start() method of base::Thread is causing jank on IO thread and UI thread [1][2]. On the jankiness dashboard it is identified 11th jankiest action on Windows[1]. The biggest reason this contributes to jank is base::Thread::Start() blocks the calling thread until the newly created thread actually starts running and finishes initialization. We've been fixing these one by one (e.g. http://crbug.com/454983) but it is easy to introduce new jank in the current architecture. The current design is fine when the system is not overloaded, but could cause jank otherwise, especially during the chrome startup time where Chrome creates 40+ threads in browser process and make the system heavily loaded.

Solution:

There are a few possible solutions proposed by several folks (see https://crbug.com/465458 for more details), but essentially what (we think) we want to do is: make base::Thread::Start() return without waiting for the thread to get started. To make this not break the callers, Start method should set up a usable MessageLoop (or TaskRunner at least) that allows callers to start posting a task, regardless of whether the thread has actually started or not.

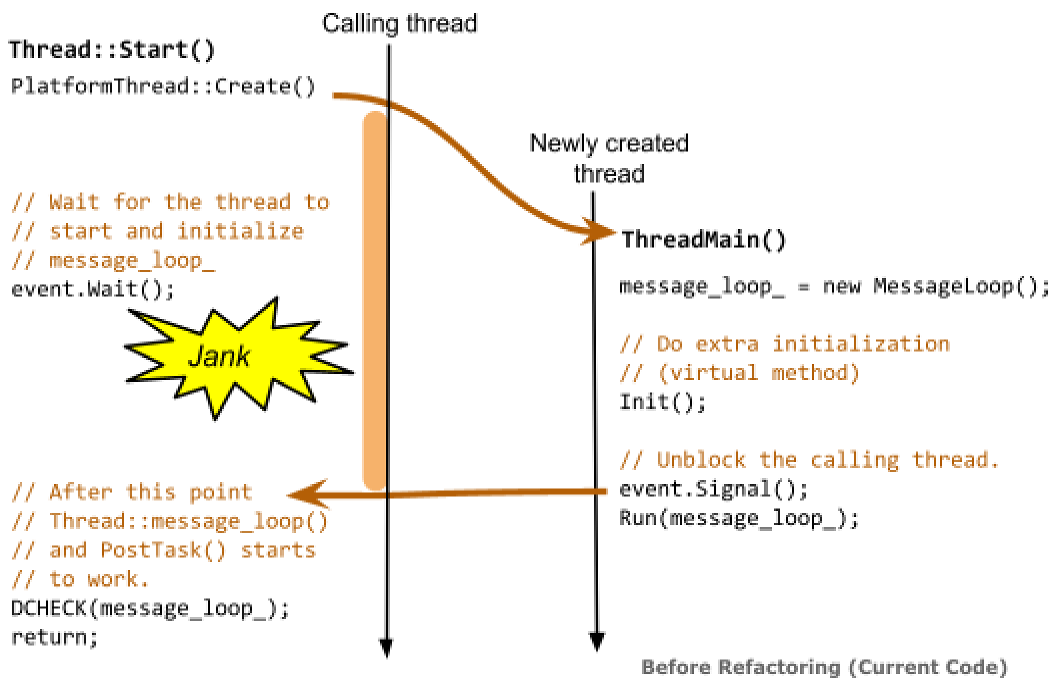
Another idea that was discussed is to move the thread creation part to a background thread if the callers care about the jank, but this solution is not scalable and new code could easily introduce jank. There's also an ongoing longer-term plan to move many of the threads to a thread pool, which should also drastically reduce the thread creation cost on the main thread.

[1] As of Apr 14, 2015. To be more specific: 17th jankiest on UI@Browser, 32th jankiest on IO@Browser, 6th jankiest on Main@Renderer, and 9th jankiest on IO@Renderer.

Current Architecture:

In order to understand what approaches are available to address this we need to know how Thread and MessageLoop are handling newly posted tasks in the current architecture.

base::Thread works closely with MessageLoop. Notably it creates a new MessageLoop on the newly created thread and calls MessageLoop::Run() to start running tasks. base::Thread also exposes Thread::message_loop() getter so that new tasks can be posted to the message loop via MessageLoop::PostTask(). The following figure summarizes what the current code is doing curing the thread startup:



```
Thread::Start()
PlatformThread::Create()

// Wait for the thread to
// start and initialize
// message_loop_
event.Wait();

// After this point
// Thread::message_loop()
// and PostTask() starts
// to work.
DCHECK(message_loop_);
return;
```

```
ThreadMain()
message_loop_ = new MessageLoop();

// Do extra initialization
// (virtual method)
Init();

// Unblock the calling thread.
event.Signal();
Run(message_loop_);
```

The overview of how MessageLoop's PostTask and Run methods work is explained in this document. In short, MessageLoop's PostTask and Run methods are tightly coupled with a few related classes, namely: MessagePump, MessageLoopProxyImpl and IncomingTaskQueue. IncomingTaskQueue is a thread-safe task queue and it provides implementation details for MessageLoop to queue up incoming tasks. All tasks posted to the MessageLoop are routed to IncomingTaskQueue via MessageLoopProxyImpl (which provides TaskRunner interface for the MessageLoop). MessageLoop itself has its own local work queue, which is not thread-safe and periodically reloaded from IncomingTaskQueue during MessageLoop::Run. In other words, MessageLoop expects the TaskRunner implementation (i.e. MessageLoopProxyImpl) to eventually queue the tasks to IncomingTaskQueue.

Possible Approaches:

There are at least three different approaches to make Start return immediately:

- **Approach 1:** Refactor MessageLoop so that it can be created on a different thread from the thread where the message loop eventually runs. Thread::Start() method can then create a message loop before creating a thread and return immediately. The message loop should be able to be lazily initialized on the newly created thread. The message loop should start accepting PostTask's right after its creation, but will start running those tasks only after it's initialized on the new thread.

    - Prototype patch: https://codereview.chromium.org/1011683002/
    - **Pros:** Existing consumer of Thread class should just work.
    - **Cons**: This adds new state to MessageLoop class after the loop is created and before it is initialized. How consumers can interact with the message loop during this state is explained here.
    - **Necessary changes:**
        - MessageLoop's initialization logic needs to be changed. This also involves changes in IncomingTaskQueue and MessageLoopProxyImpl.

- **Approach 2:** Create a mock TaskRunner in Thread::Start() and provide it as the primary post-tasking interface for customers until the new thread gets started. The mock TaskRunner is responsible for queueing tasks until it gets the real task runner. When the new thread is started we re-post all tasks to the real one. Customers of base::Thread should use base::Thread::task_runner() (which returns the mock task runner before thread starts, and real one after that) to post tasks. This requires no changes in MessageLoop but has multiple subtle consequences, which may not be really problematic but I listed them as 'cons' below.

    - Prototype patch: https://codereview.chromium.org/1086663002/
    - **Pros:** No need to modify MessageLoop class.
    - **Cons**: We need extra locks in each PostTask. We could add this lock only for the customers that have obtained the task runner before the thread starts up, but doing so requires another lock in Thread::task_runner().
    - **Cons**: PostTask semantics will be slightly changed for the tasks that are posted before the thread starts. For example: if we re-post all tasks naively the tasks that are posted with 'X' delay will be executed after 'X' + 'Y' if the thread has started after 'Y' since the task is posted. If we take the 'Y' delay into account when reposting it will then affect how the IncomingTaskQueue marks each task as 'requires high-resolution timing'. Similarly TaskAnnotator will record the 'DidQueueTask' timing not when the task is posted initially, but when the task is reposted.

    - **Cons**?: After switching the task runner the task runner equality comparison using == wouldn't work. (Should be minor as the caller should use RunsTasksOnCurrentThread() instead)
    - **Necessary changes:**
        - We need to change all consumers of base::Thread::message_loop() and base::Thread::message_loop_proxy() to use Thread::task_runner() and TaskRunner interface.

- **Approach 3:** Create a TaskRunner that talks to IncomingTaskQueue in Thread::Start() and provide it as the primary post-tasking interface for customers. The TaskRunner can be used for PostTask's regardless of whether the new thread has started or not. Thread passes the IncomingTaskQueue to the MessageLoop when it is created on the new thread. Customers of base::Thread should use base::Thread::task_runner() (which returns the task runner) to post tasks.

    - Prototype patch**:** https://codereview.chromium.org/1058603004/
    - **Pros:** The change required to MessageLoop is smaller than approach 1, and we can preserve existing PostTask semantics as we reuse the same IncomingTaskQueue code.
    - **Cons**: This makes the relationship between Thread and MessageLoop a bit tighter as it lets the thread class deal with IncomingTaskQueue.
    - **Necessary changes:**
        - IncomingTaskQueue's initialization sequence needs to be changed so that it can be constructed without/before MessageLoop.
        - We need to change all consumers of base::Thread::message_loop() and base::Thread::message_loop_proxy() to use Thread::task_runner() and TaskRunner interface.

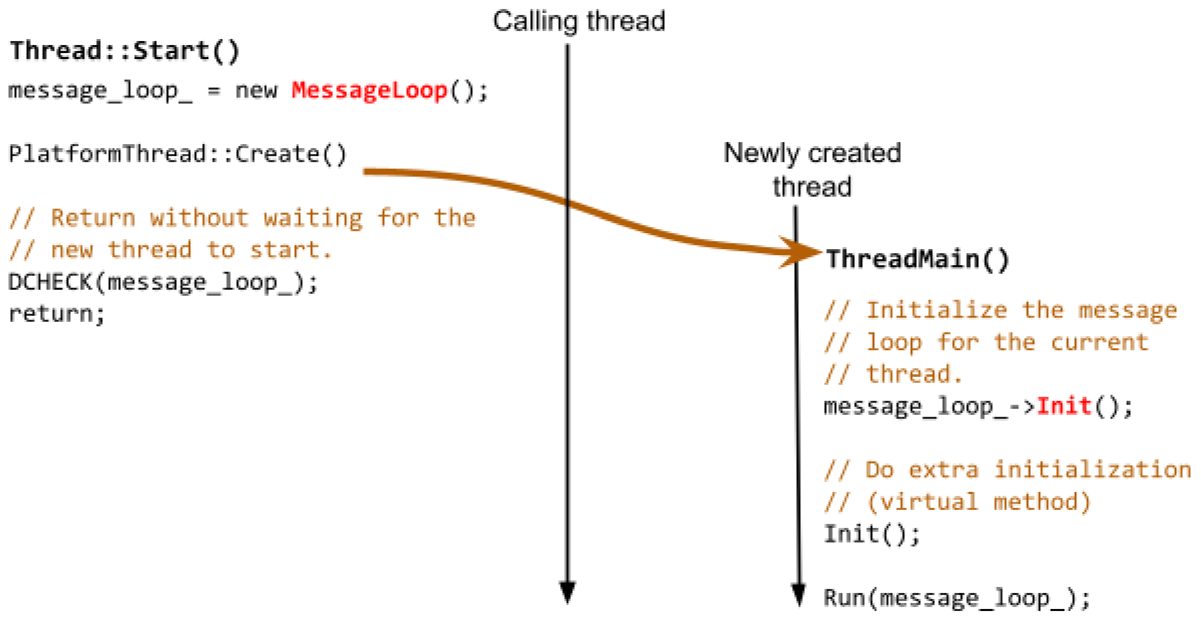
Design Details:

Approach 1 (Refactor MessageLoop initialization logic):

This approach basically splits the MessageLoop initialization into two-phases:

1. first to create an 'inactive' loop without a pump (but just be able to accept new tasks),
2. and then to bind the loop to the newly created thread once the thread actually started.

The following figure shows how base::Thread::Start (or StartWithOptions) would look like after refactoring with approach 1:



```
Thread::Start()
message_loop_ = new MessageLoop();

PlatformThread::Create()

// Return without waiting for the
// new thread to start.
DCHECK(message_loop_);
return;
```

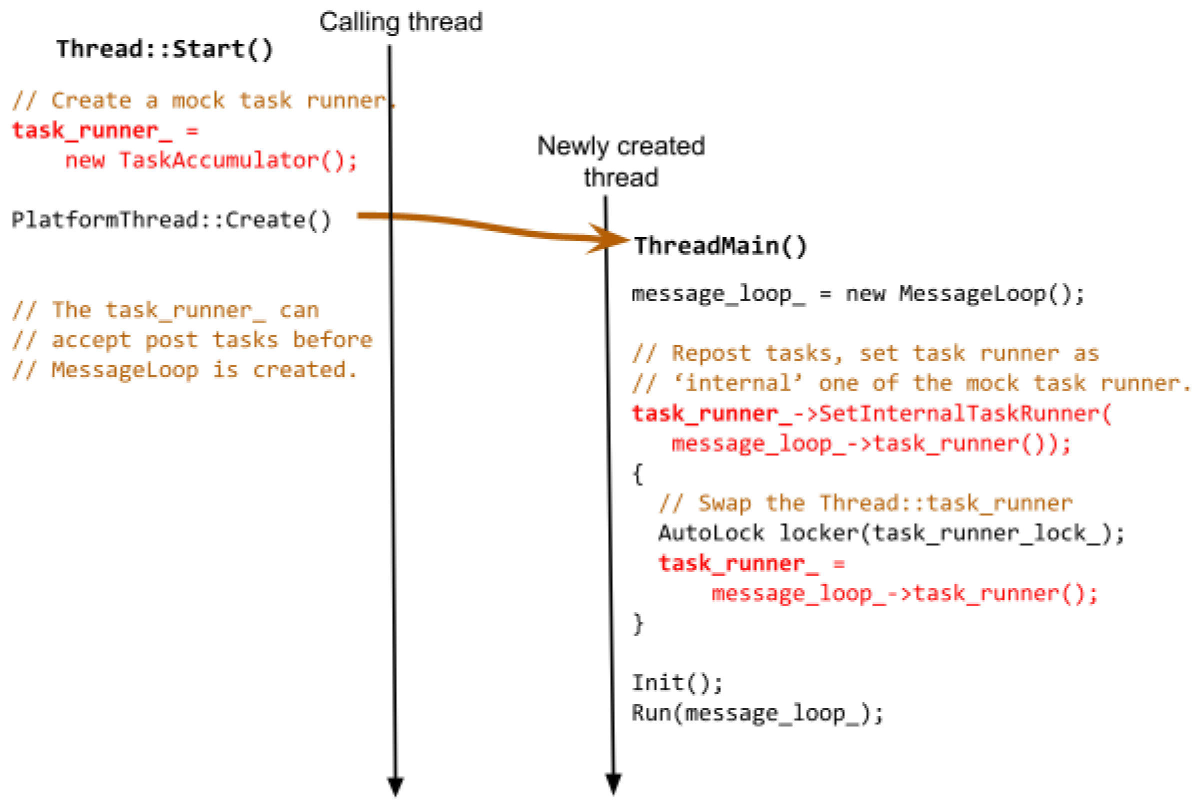
```
ThreadMain()
// Initialize the message
// loop for the current
// thread.
message_loop_->Init();

// Do extra initialization
// (virtual method)
Init();

Run(message_loop_);
```

The change itself is rather straightforward, we stop calling Init() in the constructor, but instead let the customer call it on the new thread. The new constructor creates a MessageLoop which has no pump yet, and we make it valid to instantiate a new MessageLoop on a different thread from the final ‘current’ thread.

MessageLoop will have a new ‘unbound’ state before Init() is called, where it has no associated thread or pump. However it is possible to make all public methods that can be called on a different thread from the final ‘current’ one callable right after its construction. Namely, PostTask methods family and ScheduleWork are the only methods that can be called on any threads, and we can make the former available right after the MessageLoop construction, and can make the latter private that can be called only from IncomingTaskQueue (whose design is tightly coupled with MessageLoop anyway).

Approach 2 (Have Mock TaskRunner Accumulate Tasks):

The following figure shows how base::Thread::Start (or StartWithOptions) would look like after refactoring with approach 2:



```
Thread::Start()

// Create a mock task runner.
task_runner_ =
    new TaskAccumulator();

PlatformThread::Create()

// The task_runner_ can
// accept post tasks before
// MessageLoop is created.
```

```
ThreadMain()
message_loop_ = new MessageLoop();

// Repost tasks, set task runner as
// ‘internal’ one of the mock task runner.
task_runner_->SetInternalTaskRunner(
   message_loop_->task_runner());
{
  // Swap the Thread::task_runner
  AutoLock locker(task_runner_lock_);
  task_runner_ =
      message_loop_->task_runner();
}

Init();
Run(message_loop_);
```

The ‘mock’ task runner would look like following (in pseudo code):

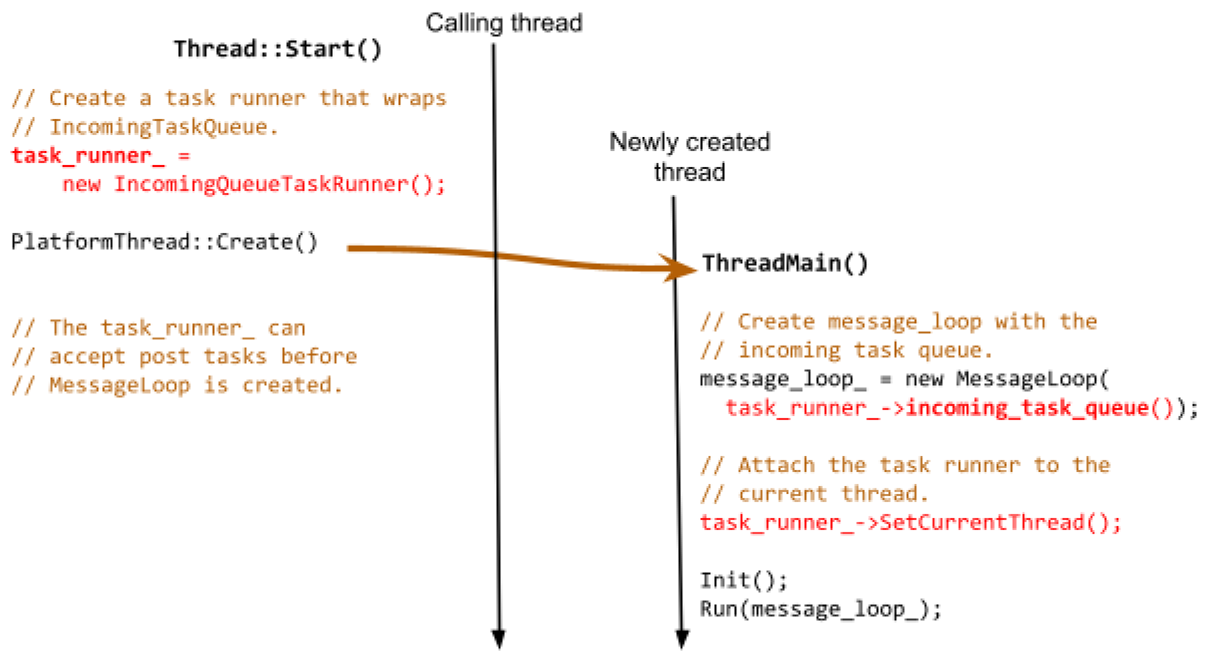
```
class TaskAccumulator : public TaskRunner {
 public:
   // Sets the given task_runner as the internal one and
   // repost all queued tasks to the internal one.
   void SetInternalTaskRunner(task_runner) {
      AutoLocker locker(lock_);
      internal_task_runner_ = task_runner;
      while (!task_queue_.empty()) {
        internal_task_runner_->PostTask(task.from_here, task.task, task.delay);
        task_queue_.pop();
      }
   }

   // Forwards the task to the internal task runner if it’s already there,
   // just accumulates it in the queue otherwise.
   bool PostTask(from_here, task, delay) {
     AutoLocker locker(lock_);
```

```
      if (internal_task_runner_)
        return internal_task_runner_->PostTask(from_here, task, delay);
      return task_queue_.push(Task(from_here, task, delay));
    }

 private:
    Lock lock_;
    scoped_refptr<TaskRunner> internal_task_runner_;
    std::queue<Task> task_queue_;
};
```

Approach 3 (Pre-create IncomingTaskQueue and Pass it to MessageLoop):

This approach is a bit similar to approach 2, but we create a thin TaskRunner that wraps IncomingTaskQueue, and pass the IncomingTaskQueue to the MessageLoop when it is created upon thread startup. No locking is required as the TaskRunner and MessageLoop will share the IncomingTaskQueue, which is a thread-safe ref-counted class that has a thread-safe task queue.

The following figure shows how base::Thread::Start (or StartWithOptions) would look like after refactoring with approach 3:



```
Thread::Start()

// Create a task runner that wraps
// IncomingTaskQueue.
task_runner_ =
    new IncomingQueueTaskRunner();

PlatformThread::Create()

// The task_runner_ can
// accept post tasks before
// MessageLoop is created.
```

```
ThreadMain()

// Create message_loop with the
// incoming task queue.
message_loop_ = new MessageLoop(
  task_runner_->incoming_task_queue());

// Attach the task runner to the
// current thread.
task_runner_->SetCurrentThread();

Init();
Run(message_loop_);
```

The task runner that wraps IncomingTaskQueue would look like following (in pseudo code):

```
class IncomingTaskQueueTaskRunner : public TaskRunner {
 public:
    bool PostTask(from_here, task, delay) {
```

```
    return incoming_task_queue_->AddToIncomingQueue(from_here, task, delay);
  }

  scoped_refptr<IncomingTaskQueue> incoming_task_queue() {
    return incoming_task_queue_;
  }

 private:
  scoped_refptr<IncomingTaskQueue> incoming_task_queue_;
};
```

Common Changes that are Necessary Regardless of Approaches

Implementation Note for POSIX Systems

Currently PlatformThread::Create implementation for POSIX also blocks until the newly created thread starts, and Thread::Start() relies on PlatformThread::Create() internally, so only changing Thread::Start() does not really make starting a thread wait-free. Changing the PlatformThread::Create non-blocking for POSIX requires a bit more work, and this document does not discuss the details for the change. (You can find WIP patch for this here)

Making Thread::thread_id() not racy

(Note that this issue needs to be addressed regardless of which approach we take)
As noted in the solution section, if we change Thread::Start() to return immediately before a new thread actually starts, Thread::thread_id() value will become racy since it will be initialized lazily on the new thread when it starts to run. This means that accessing thread_id() could return kInvalidThreadId undeterministically depending on the timing. A possible solution to work around this issue is platform-dependent:

On Windows: We have a non-blocking API to get the thread ID without waiting for the thread to start, e.g. GetThreadId() or via CreateThread(), so we can simply use these APIs instead of GetCurrentThreadId() which needs to be called on the newly created thread. In the proposing patch I changed PlatformThread::Create() to include the new thread ID in the PlatformThreadHandle that PlatformThread::Create returns.

On POSIX: As noted in the implementation note for POSIX section, currently PlatformThread::Create() blocks until the new thread starts, and this is done mainly to return a valid thread ID when PlatformThread::Create() returns. This is not ideal for our main purpose (i.e. make starting a thread not janky) but for the time being this helps us make thread_id() not racy, since we can simply use the value returned by PlatformThread::Create(). I plan to utilize this fact to make the outer Thread::Start() non-blocking initially, then plan to work on the POSIX PlatformThread::Create() issue separately.

The question here is how we make thread_id() not racy for POSIX after we change PlatformThread::Create() non-blocking. One approach is probably to change all platform-dependent code that uses kernel task ID to use thread handle, though it seems not possible without changing our sandbox policy. Another approach will be to make thread_id() blocking if (and only if) it is called before the new thread actually starts. This is probably a bit controversial, since we might be just moving jank from Thread::Start() to Thread::thread_id() if thread_id() is accessed right after Thread::Start() is called, but at least in my local measurement this does not seem to be the case. With a quick patch that changes thread_id() blocking, all thread_id() (including Chrome startup time) still returns almost instantly (0.01 ~ 0.03 msec). In order to make sure that this will not introduce another jank we should also probably keep watching the jankiness value on the UMA profiler.

Deprecating IOThread::InitAsync()

IOThread::InitAsync() is separated from IOThread::Init() in order to reduce jank, as IOThread::Init (which overrides Thread::Init) runs synchronously on the newly created thread before Thread::Start() is unblocked and returned on the calling thread in the current code base. IOThread::InitAsync() is posted asynchronously from Init(), and is expected to be called as the first task on the new thread.

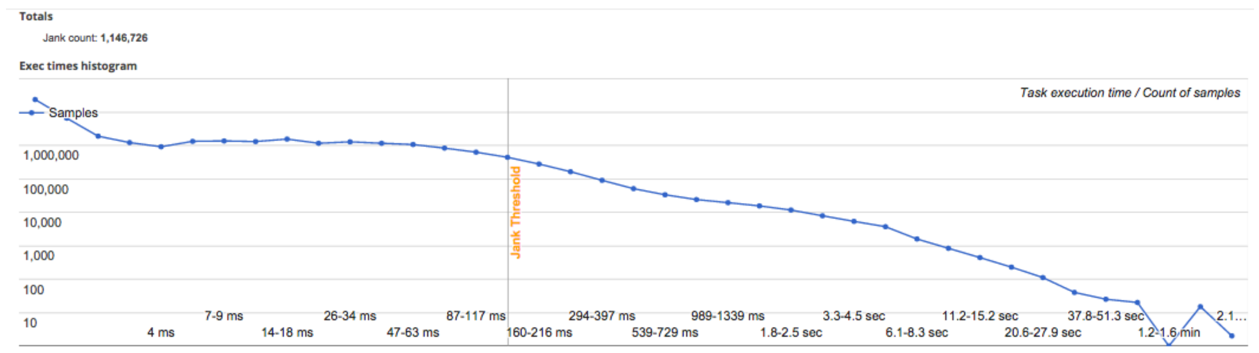
However after MessageLoop refactoring this no longer holds: IOThread::Init() runs asynchronously after Thread::Start() returns, and IOThread::InitAsync() no longer runs as the first task of the new thread. This change seems to breaks bunch of browser tests on Windows, therefore IOThread::InitAsync() needs to be merged back to IOThread::Init().

Introducing Thread::StartAndWaitForTesting()

There are several tests that assume some threads are already running when they run test code. For example, ThreadIdNameManagerTest assumes that the target threads are already running when it calls ThreadIdNameManager::GetName() methods right after Thread::Start(), but this no longer holds true after this change. For these tests we also add a new method called **StartAndWaitForTesting()**, which behaves mostly same as what Thread::Start() used to do, i.e. it doesn’t return until the thread starts running.

Performance

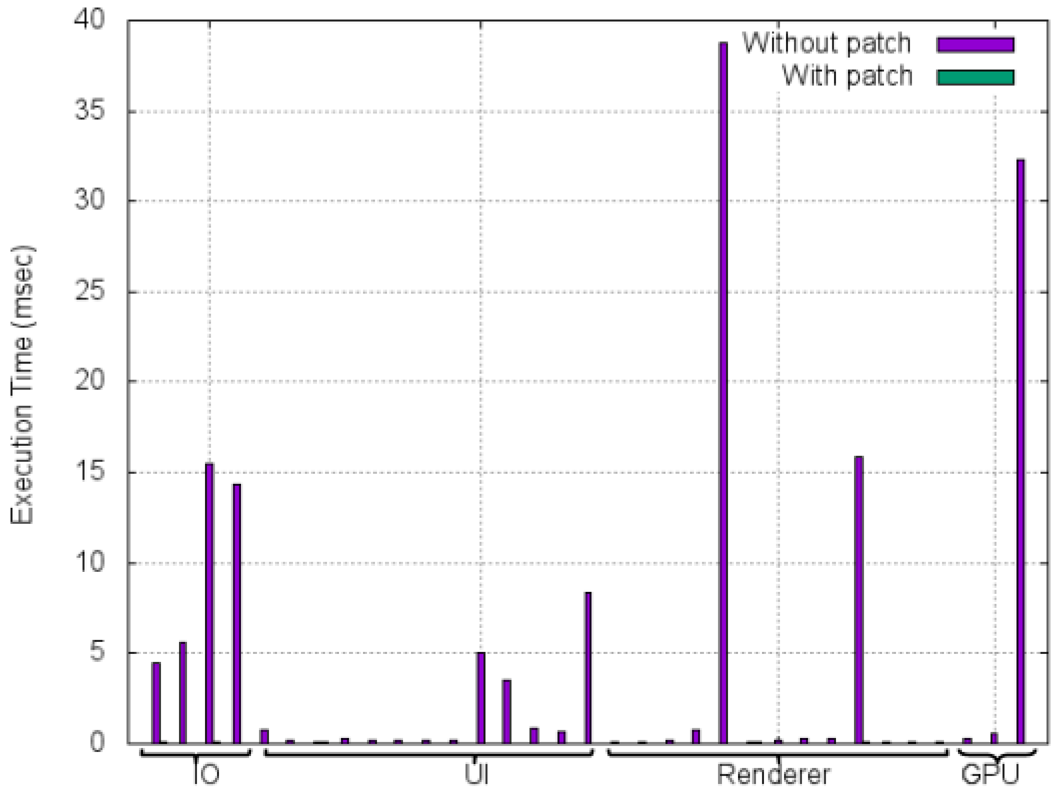
Per the jankiness dashboard using UMA profiler currently thread startup overhead is janky mostly on Windows, and it is ranked around 10th to 15th jankiest action. The following figure shows execution time histograms for 7 days UMA profiler data. As it shows most of the execution finishes less than a few msec (typically ~1ms), while small number of executions take longer than 1 min.



Execution Time Histograms for 7 days UMA profile data (-Apr 14 2015) collected using ScopedTracker

In a very limited preliminary testing on my local Linux machine (with --trace-startup and chrome://tracing with 5 open tabs), average duration time of Thread::StartWithOptions() for the first 5 secs of Chrome startup has become roughly 1/4 compared with normal Chrome without the patch. On Windows with 2 CPU BIOS settings the effect was more obvious, the average duration time has become roughly 1/130 compared with normal Chrome without the patch.

The following graph shows each execution time of tThread::StartWithOptions() for the first 10 seconds of Chrome startup only for showing the new tab page. During the 10 seconds I could observe that Chrome creates 33 threads, 4 on IO thread, 13 on UI thread, 13 on RendererMain (on Renderer process) and 3 on GPU process. As is shown most execution times were less than 1 msec, but 5 of 33 took longer than 10 msec without the patch, and two of them took longer than 30 msec. With patch all execution times were less than 0.05 msec (which are barely in the graph).



I plan to collect more UMA data after landing this change (assuming that I can) to see the real effect on jankiness. I also plan to measure the duration until the new thread starts (currently it happens during the thread startup but after this change it'll likely be delayed when there's CPU contention).

Links:

Tracking bug: https://crbug.com/465458
Patches: Approach 1 Approach 2 Approach 3