Task Scheduler - Message Loop Integration and Migration

This Document is Public

Authors: robliao@chromium.org

One-page overview

Summary

The MessageLoop will begin integration with base/task_scheduler so that the scheduler will be aware of most, if not all, tasks for scheduling decisions. This allows prioritizing BrowserThread UI work above less important work, among other scheduling scenarios.

Platforms

All

Team

scheduler-dev@chromium.org

Bug

http://crbug.com/749310

Code affected

All code that deals with a MessageLoop on threads backed by base::Thread, BrowserThread, and likely more.

Design and Implementation

Goals

- The Task Scheduler is aware of the tasks for all threads and the status of all threads.
- The UI thread maintains high priority.
- Tasks run on threads not owned by the task scheduler have task traits (they are prioritized according to their TaskPriority and honor their TaskShutdownBehavior).

Message Loop Integration and Migration Example

Before	**Redirection**	**After**

`main_message_loop_.reset(`
`  new base::MessageLoopForUI);`
 	
`// or`
 	
`MessageLoop message_loop;`
 	
`RunLoop().run()`
	`main_message_loop_.reset(`
	`  new base::MessageLoopForUI({`
	`  TaskPriority::`
	`  USER_BLOCKING}));`
	
	`// or`
	
	`MessageLoop message_loop(`
	`  {TaskPriority::`
	`  USER_BLOCKING,`
	`  MayBlock()});`
	
	`RunLoop().run()`
		`scheduler_loop_.reset(`
		`  new base::internal::SchedulerLoop(`
		`  UI,`
		`  {TaskPriority::`
		`  USER_BLOCKING});`
		
		`// or`
		
		`base::internal::SchedulerLoop`
		`  scheduler_loop(`
		`  IO,`
		`  {TaskPriority::`
		`  USER_BLOCKING,`
		`  MayBlock()});`
		
		`RunLoop().run()`

Chromium today.	The MessageLoop delegates execution of tasks to the Task Scheduler when redirection is enabled. Otherwise, the MessageLoop behaves like it does today. MessageLoops that do not specify a trait will default to TaskPriority::USER_VISIBLE.	The MessageLoop no longer exists. SchedulerLoop provides similar services.

Redirection Design

Because MessageLoop has many references in Chromium, the first step here is to provide an easy switch between the current MessageLoop behavior and the TaskScheduler redirection.

Threads like the UI/Main thread are not owned by the TaskScheduler. As a result some additional coordination is needed for the TaskScheduler to be aware of the tasks running on those threads.

Today, the MessageLoop executes tasks for the UI/MainThread and base::Thread among others.

Abstracting Tasks from the MessageLoop

The Task Scheduler receives notifications about tasks through its Task Tracker. The goal for the redirection is to notify the Task Tracker about incoming tasks and have the Task Tracker execute these tasks.

One way to handle this redirection is to redistribute the responsibilities of the task pipeline between MessageLoop and IncomingTaskQueue. This will provide a clean boundary to switch between the existing implementation and the Task Scheduler redirection.

Before	**Refactored**	**Redirected**
IncomingTaskQueue ● State ○ Incoming queue ● Action ○ Wakes the MessageLoop upon receiving a pending task.	**IncomingTaskQueue** ● State ○ Incoming queue ○ Pending Task Queue ○ Delayed Task Queue ○ Deferred Task Queue ● Action ○ Wakes the MessageLoop upon receiving a pending task. ○ Directly executes the tasks	**SchedulerIncomingTaskQueue** ● State ○ Pending Task Sequence* ○ High frequency delayed task sequence ○ Deferred Task Sequence* ● Action ○ Wakes the MessageLoop upon receiving a pending task. ○ TaskTracker executes the tasks Long delayed tasks are handled centrally by the Task Scheduler DelayedTaskManager. Shorter delayed tasks for animations will continue to be handled directly by the task queue itself to avoid thread hopping for animations. *Sequences are task queues in the Task Scheduler.
MessageLoop ● Policy ○ When to execute tasks ○ When to delay tasks ○ When to defer tasks ○ When to cancel tasks ● State ○ High Resolution Timer Tracking ○ Pending Task	**MessageLoop** ● Policy ○ When to execute tasks ○ When to delay tasks ○ When to defer tasks ○ When to cancel tasks ● State ○ High Resolution Timer Tracking	**MessageLoop** ● Policy ○ When to execute tasks ○ When to delay tasks ○ When to defer tasks ○ When to cancel tasks ● State ○ High Resolution Timer Tracking

Queue ○ Delayed Task Queue ○ Deferred Task Queue ● Action ○ Directly executes the tasks	● Action ○ Calls methods on IncomingTaskQueue to enforce policy	● Action ○ Calls methods on IncomingTaskQueue to enforce policy

IncomingTaskQueue Changes

The IncomingTaskQueue interface will be expanded and modified to accommodate this new boundary:

```
class BASE_EXPORT IncomingTaskQueue : public RefCountedThreadSafe<IncomingTaskQueue> {
 public:
  // Appends a task to the incoming queue. Posting of all tasks is routed through
  // AddToIncomingQueue() or TryAddToIncomingQueue() to make sure that posting
  // task is properly synchronized between different threads.
  //
  // Returns true if the task was successfully added to the queue, otherwise
  // returns false. In all cases, the ownership of |task| is transferred to the
  // called method.
  bool AddToIncomingQueue(const tracked_objects::Location& from_here,
                          OnceClosure task,
                          TimeDelta delay,
                          bool nestable);

  // Returns true if the message loop is "idle". Provided for testing.
  bool IsIdleForTesting();
  // IsIdleForTesting only checks the incoming queue. IsEmpty() supersedes this by checking all queues.

  // Loads tasks from the |incoming_queue_| into |*work_queue|. Must be called
  // from the thread that is running the loop. Returns the number of tasks that
  // require high resolution timers.
  int ReloadWorkQueue(TaskQueue* work_queue);
  // Since the IncomingTaskQueue is managing execution, ReloadWorkQueue can be made private.

  // Disconnects |this| from the parent message loop.
  void WillDestroyCurrentMessageLoop();

  // This should be called when the message loop becomes ready for
  // scheduling work.
  void StartScheduling();

  // Runs |pending_task|.
  void RunTask(PendingTask* pending_task);

  ReadAndRemoveOnlyQueue& triage_tasks() { return triage_tasks_; }

  Queue& delayed_tasks() { return delayed_tasks_; }

  Queue& deferred_tasks() { return deferred_tasks_; }

  HasPendingHighResolutionTasks()

  // Provides a read and remove only view into a task queue.
  class ReadAndRemoveOnlyQueue {
   public:
    ReadAndRemoveOnlyQueue() = default;
    virtual ~ReadAndRemoveOnlyQueue() = default;
```

```
    // Returns the next task. HasTasks() is assumed to be true.
    virtual const PendingTask& Peek() = 0;

    // Removes and returns the next task. HasTasks() is assumed to be true.
    virtual PendingTask Pop() = 0;

    // Whether this queue has tasks.
    virtual bool HasTasks() = 0;

    // Removes all tasks.
    virtual void Clear() = 0;

   private:
    DISALLOW_COPY_AND_ASSIGN(ReadAndRemoveOnlyQueue);
  };

  // Provides a read-write task queue.
  class Queue : public ReadAndRemoveOnlyQueue {
   public:
    Queue() = default;
    ~Queue() override = default;

    // Adds the task to the end of the queue.
    virtual void Push(PendingTask pending_task) = 0;

   private:
    DISALLOW_COPY_AND_ASSIGN(Queue);
  };
};
```

The Scheduler Task Queue will simply implement the same methods to accomplish the redirection.

Fitting Redirection Into the MessageLoop

The following MessageLoop components are of interest to the Task Scheduler integration:

Component and Responsibility	Redirection Plan
Incoming Task Queue Holds tasks and wakes the MessageLoop to process new tasks.	Use SchedulerIncomingTaskQueue.
MessageLoopTaskRunner A public interface for the IncomingTaskQueue.	Move to a SchedulerIncomingTaskQueue backed SingleThreadTaskRunner.
MessagePump An abstraction that actually implements the loop to process work from tasks or optionally from the OS.	No change. This will abstract away the OS details from the task scheduler and will remain in the final design. When MessageLoop is replaced, the MessagePump delegate will naturally change to accommodate SchedulerLoop.

The MessageLoop implements the MessagePump::Delegate interface that performs different actions on tasks at various phases:

Phase	Redirection Plan

DoWork Retrieves tasks from the IncomingWorkQueue and when possible, runs a single task. The following cases cause the MessageLoop to defer execution of a task: • Delay - These are reposted to a delayed work queue. • Non-nestability - Tasks can be marked as non-nestable in a nested environment are pushed to a deferred non nestable work queue.	The SchedulerIncomingTaskQueue will also expose a method to run a single task. Delayed tasks will be posted directly to the task scheduler DelayedTaskManager that reposts all tasks from a single thread for execution. The scheduler managed Task Queue will provide a way to defer these tasks for later execution.
DoDelayedWork Checks the top of the delayed work queue and runs the task around the requested delay time.	Short Delays: Similar behavior today to allow these tasks to be posted and processed without a thread hop. Long Delays: Redirect to the DelayedTaskManager. Long delays are less urgent and having one place handle policy about delayed tasks is better than multiple places.
DoIdleWork • Runs non-nestable tasks when the environment is no longer nested.	The SchedulerTaskQueue will expose a method to run a deferred single non-nestable task.

Additional MessageLoop Considerations

Task Nesting

Task Nesting is the main scenario that currently is unsupported by the TaskScheduler as many items are set in TLS and are presumed to only be set before a task and unset after a task (see ScopedSetSequenceTokenForCurrentThread for example). With task nesting, these will need to support storing the previous state.

Task Observers

With redirection, TaskObservers will continue to work like they do today.

These are currently used for scenarios like hang detection (media/gpu/avda_codec_allocator.cc), testing, and in Blink.

Setting a Custom Task Runner

Setting a custom task runner could bypass redirection at it causes tasks to bypass the IncomingTaskQueue. This is used primarily for testing and provides a way for unit tests to verify tasks are posted in response to various calls.

There is one use in production in the Blink Scheduler for redirecting MessageLoop bound tasks to queues, but the Message Loop Task runner remains exposed in SchedulerTqmDelegateImpl::PostDelayedTask, SchedulerTqmDelegateImpl::PostNonNestableDelayedTask, and SchedulerTqmDelegateImpl::RunsTasksInCurrentSequence

Destruction Observation

With redirection, destruction observation will continue to work like it did previously. https://cs.chromium.org/search/?q=WillDestroyCurrentMessageLoop%5C(%5C)+-file:unittests&sq=package:chromium&type=cs

SchedulerLoop Design

The 3 phase MessagePump::Delegate system doesn't make as much sense with the TaskScheduler, especially since the TaskScheduler can handle delayed tasks centrally.

This will be designed in detail once the redirection works well and the actual interaction between the MessagePump and the TaskScheduler can be assessed.

From a high level, uses of Task Observers and custom task runners will likely need to be reconsidered.

Implementation Plan

Phase 0 - Cleanup and Refactor

RunLoop - MessageLoop Separation and Cleanup - http://crbug.com/703346
MessageLoop - MessagePump Cleanup and Refactor - http://crbug.com/749312

During Phase 0, MessageLoop will stop managing its own set of tasks and delegate all task operations to the IncomingTaskQueue.

Phase 1 - Redirection (http://crbug.com/749310)

The TaskScheduler will have drop-in replacement for the IncomingTaskQueue as well as handle scenarios encountered by the MessageLoop like nesting.

Phase 2 - Direct Task Scheduler Binding (crbug TBA)

Once redirection is stable, this work aims to replace the MessageLoop with a drop-in replacement that provides similar services with more direct integration to the Task Scheduler.

Rollout plan

The code will be committed in accordance with the phases above.

Core principle considerations

Everything we do should be aligned with and consider Chrome's core principles. If there are any specific stability concerns, be sure to address them with appropriate experiments.

Speed

This migration will use the existing core speed metrics to ensure the same performance guarantees are maintained.

Security

There should not be an impact to security from this migration.

Privacy considerations

There should not be an impact to privacy from this migration.

Testing plan

No special testing considerations are needed here. Unit tests will test the work and it will be pretty obvious if things are broken.