Wakeup-based throttling

altimin@chromium.org

March 7, 2017

PUBLIC

TL;DR: This document discusses generalizing the background throttling mechanism, which will enable us to throttle internal non-javascript timer (and, possibly, non-timer) tasks as well.

Motivation

Since version 11 (Chrome 11), Chrome has limited timers in background tabs to running once a second. That means that chained calls, even if they are lightweight, take 1 second * #calls in the chain.

```
function k() {
  doStuff();
}

function g() {
  doLightweightStuff();
  doStuff(k);
}

function f() {
  doOtherLightweightStuff();
  doStuff(g);
}

f()
```

In the example above, the code will take 2 seconds and three wakeups to execute in a background tab, even when `doStuff()` calls are lightweight (e.g. take <1 ms). This poses a special problem when trying to throttle internal Chrome timers, where a pattern of posting a continuation task is a common one.

Wakeup-based throttling

The proposed solution is wakeup-based throttling. Under the existing mechanism before the existing timers are allowed to run, all new timers are blocked from executing until explicitly allowed. Under the new scheme, new timer tasks are allowed to run for a small period of time (~10ms) and are blocked only after this small period of time. This guarantees that when the process was woken up, it has a small window of time to run all tasks freely in any pattern.

A backgrounded application will receive a wakeup budget of 1 wakeup per second, which closely matches the previous policy of aligning timers to 1Hz.

Regression concerns

This change allows javascript timers to run more frequently in background, which can increase power usage. However, we believe that this has an advantage which justifies the risk: this new mechanism can be applied to new types of tasks (internal Chrome timers, non-timer javascript like websocket messages, etc). Making one mechanism for all javascript tasks will enable us to communicate what we are doing more clearly to the web developers.

Furthermore, CPU usage increase should be small: by the fact that wakeup window is small (10ms), CPU usage increase should be limited to 1%. Coupled with time budget-based throttling, this ensures that background tab CPU usage is limited to ~1%.

However, to measure the exact influence of this change on the Chrome resource usage, we will make this change Finch-controlled and perform an experiment-based rollout.

Implementation details

Existing implementation relies on rounding up run time of a call to TaskQueueThrottler::PumpThrottledTasks. It is proposed to extend a TimeBudgetPool mechanism used for time budget-based throttling to a generic BudgetPool throttling mechanism and derive two classes (WakeupBudgetPool and TimeBudgetPool) which will implement different throttling mechanisms.

Current TimeBudgetPool class signature:

```
class TimeBudgetPool {
  void SetTimeBudgetRecoveryRate(TimeTicks now, double cpu_percentage);

  void AddQueue(TimeTicks now, TaskQueue* queue);
  void RemoveQueue(TimeTicks now, TaskQueue* queue);

  void RecordTaskRunTime(TimeTicks start, TimeTicks end);

  void GrantAdditionalBudget(TimeTicks now, TimeDelta additional_budget);

  const char* Name() const;

  void SetReportingCallback(
        base::Callback<void(base::TimeDelta)> reporting_callback);


  void Close();
```

```
  bool HasEnoughBudgetToRun(base::TimeTicks now);
  base::TimeTicks GetNextAllowedRunTime();
};
```

After proposed changes it will look like this:

```
class BudgetPool {
  const char* Name() const = 0;

  void AddQueue(TimeTicks now, TaskQueue* queue) = 0;
  void RemoveQueue(TimeTicks now, TaskQueue* queue) = 0;

  void EnableThrottling(TimeTicks now) = 0;
  void DisableThrottling(TimeTicks now) = 0;
  bool IsThrottlingEnabled() = 0;

  void RecordTaskRunTime(TimeTicks start, TimeTicks end);
  // NEW: Notifies BudgetPool about new tasks.
  void OnNextQueueWakeupChanged(TaskQueue* queue) = 0;

  void Close() = 0;

  bool HasEnoughBudgetToRun(base::TimeTicks now) = 0;
  base::TimeTicks GetNextAllowedRunTime() = 0;
};

class TimeBudgetPool : BudgetPool {
  // Budget pool implementation.

  void SetTimeBudgetRecoveryRate(TimeTicks now, double cpu_percentage);
  void SetReportingCallback(
         base::Callback<void(base::TimeDelta)> reporting_callback);
  void GrantAdditionalBudget(TimeTicks now, TimeDelta additional_budget);
};

class WakeupBudgetPool : BudgetPool {
  // Budget Pool implementation.

  void SetWakeupWindowLength(TimeDelta length);
  void SetWakeupRate(double wakeup_per_second);
};
```

While TimeBudgetPool blocks queue from running new tasks (and schedules a call to PumpThrottledTasks) immediately inside RecordTaskRunTime when time budget is exhausted, WakeupBudgetPool will block queue inside RecordTaskRunTime when there are no tasks scheduled to run in the current wakeup window or current time is outside of the wakeup window. Also queue will be temporarily unblocked when OnNextQueueWakeupChanged notifies about a new task that can run inside current wakeup window.