Threading and Tasks in Chrome

Contents

- Overview
  - Nomenclature
- Core Concepts
- Threading Lexicon
  - Threads
  - Tasks
  - Prefer Sequences to Physical Threads
- Posting a Parallel Task
  - Direct Posting to the Thread Pool
  - Posting via a TaskRunner
- Posting a Sequenced Task
  - Posting to a New Sequence
  - Posting to the Current (Virtual) Thread
- Using Sequences Instead of Locks
- Posting Multiple Tasks to the Same Thread
  - Posting to the Main Thread or to the IO Thread in the Browser Process
  - Posting to the Main Thread in a Renderer Process
  - Posting to a Custom SingleThreadTaskRunner
  - Posting to the Current Thread
- Posting Tasks to a COM Single-Thread Apartment (STA) Thread (Windows)
- Annotating Tasks with TaskTraits
- Keeping the Browser Responsive
- Posting a Task with a Delay
  - Posting a One-Off Task with a Delay
  - Posting a Repeating Task with a Delay
- Cancelling a Task
  - Using base::WeakPtr
  - Using base::CancelableTaskTracker
- Posting a Job to run in parallel

    - Adding additional work to a running job
- Testing
- Using ThreadPool in a New Process
- TaskRunner ownership (encourage no dependency injection)
- FAQ
- Internals
    - SequenceManager
    - MessagePump
    - RunLoop
    - Task Reentrancy
- APIs for general use
    - SingleThreadTaskExecutor and TaskEnvironment
- MessageLoop and MessageLoopCurrent

Note: See Threading and Tasks FAQ for more examples.

Overview

Chrome has a multi-process architecture and each process is heavily multi-threaded. In this document we will go over the basic threading system shared by each process. The main goal is to keep the main thread (a.k.a. "UI" thread in the browser process) and IO thread (each process's thread for receiving IPC) responsive. This means offloading any blocking I/O or other expensive operations to other threads. Our approach is to use message passing as the way of communicating between threads. We discourage locking and thread-safe objects. Instead, objects live on only one (often virtual -- we'll get to that later!) thread and we pass messages between those threads for communication. Absent external requirements about latency or workload, Chrome attempts to be a highly concurrent, but not necessarily parallel, system.

A basic intro to the way Chromium does concurrency (especially Sequences) can be found here.

This documentation assumes familiarity with computer science threading concepts.

Nomenclature

Core Concepts

- **Task**: A unit of work to be processed. Effectively a function pointer with optionally associated state. In Chrome this is `base::OnceCallback` and `base::RepeatingCallback` created via `base::BindOnce` and `base::BindRepeating`, respectively. (documentation).
- **Task queue**: A queue of tasks to be processed.
- **Physical thread**: An operating system provided thread (e.g. pthread on POSIX or CreateThread() on Windows). The Chrome cross-platform abstraction is `base::PlatformThread`. You should pretty much never use this directly.

- **`base::Thread`** : A physical thread forever processing messages from a dedicated task queue until Quit(). You should pretty much never be creating your own `base::Thread` 's.
- **Thread pool**: A pool of physical threads with a shared task queue. In Chrome, this is `base::ThreadPoolInstance` . There's exactly one instance per Chrome process, it serves tasks posted through `base/task/thread_pool.h` and as such you should rarely need to use the `base::ThreadPoolInstance` API directly (more on posting tasks later).
- **Sequence** or **Virtual thread**: A chrome-managed thread of execution. Like a physical thread, only one task can run on a given sequence / virtual thread at any given moment and each task sees the side-effects of the preceding tasks. Tasks are executed sequentially but may hop physical threads between each one.
- **Task runner**: An interface through which tasks can be posted. In Chrome this is `base::TaskRunner` .
- **Sequenced task runner**: A task runner which guarantees that tasks posted to it will run sequentially, in posted order. Each such task is guaranteed to see the side-effects of the task preceding it. Tasks posted to a sequenced task runner are typically processed by a single thread (virtual or physical). In Chrome this is `base::SequencedTaskRunner` which is-a `base::TaskRunner` .
- **Single-thread task runner**: A sequenced task runner which guarantees that all tasks will be processed by the same physical thread. In Chrome this is `base::SingleThreadTaskRunner` which is-a `base::SequencedTaskRunner` . We prefer sequences to threads whenever possible.

Threading Lexicon

Note to the reader: the following terms are an attempt to bridge the gap between common threading nomenclature and the way we use them in Chrome. It might be a bit heavy if you're just getting started. Should this be hard to parse, consider skipping to the more detailed sections below and referring back to this as necessary.

- **Thread-unsafe**: The vast majority of types in Chrome are thread-unsafe (by design). Access to such types/methods must be externally synchronized. Typically thread-unsafe types require that all tasks accessing their state be posted to the same `base::SequencedTaskRunner` and they verify this in debug builds with a `SEQUENCE_CHECKER` member. Locks are also an option to synchronize access but in Chrome we strongly prefer sequences to locks.
- **Thread-affine**: Such types/methods need to be always accessed from the same physical thread (i.e. from the same `base::SingleThreadTaskRunner`) and typically have a `THREAD_CHECKER` member to verify that they are. Short of using a third-party API or having a leaf dependency which is thread-affine: there's pretty much no reason for a type to be thread-affine in Chrome. Note that `base::SingleThreadTaskRunner` is-a `base::SequencedTaskRunner` so thread-affine is a subset of thread-unsafe. Thread-affine is also sometimes referred to as **thread-hostile**.
- **Thread-safe**: Such types/methods can be safely accessed in parallel.

- **Thread-compatible**: Such types provide safe parallel access to const methods but require synchronization for non-const (or mixed const/non-const access). Chrome doesn't expose reader-writer locks; as such, the only use case for this is objects (typically globals) which are initialized once in a thread-safe manner (either in the single-threaded phase of startup or lazily through a thread-safe static-local-initialization paradigm a la `base::NoDestructor`) and forever after immutable.
- **Immutable**: A subset of thread-compatible types which cannot be modified after construction.
- **Sequence-friendly**: Such types/methods are thread-unsafe types which support being invoked from a `base::SequencedTaskRunner`. Ideally this would be the case for all thread-unsafe types but legacy code sometimes has overzealous checks that enforce thread-affinity in mere thread-unsafe scenarios. See Prefer Sequences to Threads below for more details.

Threads

Every Chrome process has

- a main thread
  - in the browser process (BrowserThread::UI): updates the UI
  - in renderer processes (Blink main thread): runs most of Blink
- an IO thread
  - in all processes: all IPC messages arrive on this thread. The application logic to handle the message may be in a different thread (i.e., the IO thread may route the message to a Mojo interface which is bound to a different thread).
  - more generally most async I/O happens on this thread (e.g., through base::FileDescriptorWatcher).
  - in the browser process: this is called BrowserThread::IO.
- a few more special-purpose threads
- and a pool of general-purpose threads

Most threads have a loop that gets tasks from a queue and runs them (the queue may be shared between multiple threads).

Tasks

A task is a `base::OnceClosure` added to a queue for asynchronous execution.

A `base::OnceClosure` stores a function pointer and arguments. It has a `Run()` method that invokes the function pointer using the bound arguments. It is created using `base::BindOnce`. (ref. Callback<> and Bind() documentation).

```
void TaskA() {}
void TaskB(int v) {}
```

```
auto task_a = base::BindOnce(&TaskA);
auto task_b = base::BindOnce(&TaskB, 42);
```

A group of tasks can be executed in one of the following ways:

- Parallel: No task execution ordering, possibly all at once on any thread
- Sequenced: Tasks executed in posting order, one at a time on any thread.
- Single Threaded: Tasks executed in posting order, one at a time on a single thread.
    - COM Single Threaded: A variant of single threaded with COM initialized.

Prefer Sequences to Physical Threads

Sequenced execution (on virtual threads) is strongly preferred to single-threaded execution (on physical threads). Except for types/methods bound to the main thread (UI) or IO threads: thread-safety is better achieved via `base::SequencedTaskRunner` than through managing your own physical threads (ref. Posting a Sequenced Task below).

All APIs which are exposed for "current physical thread" have an equivalent for "current sequence" (mapping).

If you find yourself writing a sequence-friendly type and it fails thread-affinity checks (e.g., `THREAD_CHECKER`) in a leaf dependency: consider making that dependency sequence-friendly as well. Most core APIs in Chrome are sequence-friendly, but some legacy types may still over-zealously use ThreadChecker/SingleThreadTaskRunner when they could instead rely on the "current sequence" and no longer be thread-affine.

Posting a Parallel Task

Direct Posting to the Thread Pool

A task that can run on any thread and doesn't have ordering or mutual exclusion requirements with other tasks should be posted using one of the `base::ThreadPool::PostTask*()` functions defined in `base/task/thread_pool.h`.

```
base::ThreadPool::PostTask(FROM_HERE, base::BindOnce(&Task));
```

This posts tasks with default traits.

The `base::ThreadPool::PostTask*()` functions allow the caller to provide additional details about the task via TaskTraits (ref. Annotating Tasks with TaskTraits).

```
base::ThreadPool::PostTask(
    FROM_HERE, {base::TaskPriority::BEST_EFFORT, MayBlock()},
```

```
    base::BindOnce(&Task));
```

Posting via a TaskRunner

A parallel `base::TaskRunner` is an alternative to calling `base::ThreadPool::PostTask*()` directly. This is mainly useful when it isn't known in advance whether tasks will be posted in parallel, in sequence, or to a single-thread (ref. Posting a Sequenced Task, Posting Multiple Tasks to the Same Thread). Since `base::TaskRunner` is the base class of `base::SequencedTaskRunner` and `base::SingleThreadTaskRunner`, a `scoped_refptr<TaskRunner>` member can hold a `base::TaskRunner`, a `base::SequencedTaskRunner` or a `base::SingleThreadTaskRunner`.

```
class A {
 public:
  A() = default;

  void PostSomething() {
    task_runner_->PostTask(FROM_HERE, base::BindOnce(&A, &DoSomething));
  }

  void DoSomething() {
  }

 private:
  scoped_refptr<base::TaskRunner> task_runner_ =
      base::ThreadPool::CreateTaskRunner({base::TaskPriority::USER_VISIBLE})
};
```

Unless a test needs to control precisely how tasks are executed, it is preferred to call `base::ThreadPool::PostTask*()` directly (ref. Testing for less invasive ways of controlling tasks in tests).

Posting a Sequenced Task

A sequence is a set of tasks that run one at a time in posting order (not necessarily on the same thread). To post tasks as part of a sequence, use a `base::SequencedTaskRunner`.

Posting to a New Sequence

A `base::SequencedTaskRunner` can be created by `base::ThreadPool::CreateSequencedTaskRunner()`.

```
scoped_refptr<SequencedTaskRunner> sequenced_task_runner =
    base::ThreadPool::CreateSequencedTaskRunner(...);

// TaskB runs after TaskA completes.
sequenced_task_runner->PostTask(FROM_HERE, base::BindOnce(&TaskA));
sequenced_task_runner->PostTask(FROM_HERE, base::BindOnce(&TaskB));
```

Posting to the Current (Virtual) Thread

The preferred way of posting to the current (virtual) thread is via `base::SequencedTaskRunner::GetCurrentDefault()`.

```
// The task will run on the current (virtual) thread's default task queue.
base::SequencedTaskRunner::GetCurrentDefault()->PostTask(
    FROM_HERE, base::BindOnce(&Task);
```

Note that `SequencedTaskRunner::GetCurrentDefault()` returns the default queue for the current virtual thread. On threads with multiple task queues (e.g. BrowserThread::UI) this can be a different queue than the one the current task belongs to. The “current” task runner is intentionally not exposed via a static getter. Either you know it already and can post to it directly or you don't and the only sensible destination is the default queue. See https://bit.ly/3JvCLsX for detailed discussion.

Using Sequences Instead of Locks

Usage of locks is discouraged in Chrome. Sequences inherently provide thread-safety. Prefer classes that are always accessed from the same sequence to managing your own thread-safety with locks.

Thread-safe but not thread-affine; how so? Tasks posted to the same sequence will run in sequential order. After a sequenced task completes, the next task may be picked up by a different worker thread, but that task is guaranteed to see any side-effects caused by the previous one(s) on its sequence.

```
class A {
 public:
  A() {
    // Do not require accesses to be on the creation sequence.
    DETACH_FROM_SEQUENCE(sequence_checker_);
  }

  void AddValue(int v) {
    // Check that all accesses are on the same sequence.
    DCHECK_CALLED_ON_VALID_SEQUENCE(sequence_checker_);
```

```
    values_.push_back(v);
}

 private:
  SEQUENCE_CHECKER(sequence_checker_);

  // No lock required, because all accesses are on the
  // same sequence.
  std::vector<int> values_;
};

A a;
scoped_refptr<SequencedTaskRunner> task_runner_for_a = ...;
task_runner_for_a->PostTask(FROM_HERE,
                        base::BindOnce(&A::AddValue, base::Unretained(&a), 42)
task_runner_for_a->PostTask(FROM_HERE,
                        base::BindOnce(&A::AddValue, base::Unretained(&a), 27)

// Access from a different sequence causes a DCHECK failure.
scoped_refptr<SequencedTaskRunner> other_task_runner = ...;
other_task_runner->PostTask(FROM_HERE,
                            base::BindOnce(&A::AddValue, base::Unretained(&a
```

Locks should only be used to swap in a shared data structure that can be accessed on multiple threads. If one thread updates it based on expensive computation or through disk access, then that slow work should be done without holding the lock. Only when the result is available should the lock be used to swap in the new data. An example of this is in PluginList::LoadPlugins (`content/browser/plugin_list.cc` . If you must use locks, here are some best practices and pitfalls to avoid.

In order to write non-blocking code, many APIs in Chrome are asynchronous. Usually this means that they either need to be executed on a particular thread/sequence and will return results via a custom delegate interface, or they take a `base::OnceCallback<>` (or `base::RepeatingCallback<>`) object that is called when the requested operation is completed. Executing work on a specific thread/sequence is covered in the PostTask sections above.

Posting Multiple Tasks to the Same Thread

If multiple tasks need to run on the same thread, post them to a `base::SingleThreadTaskRunner` . All tasks posted to the same `base::SingleThreadTaskRunner` run on the same thread in posting order.

Posting to the Main Thread or to the IO Thread in the Browser Process

To post tasks to the main thread or to the IO thread, use `content::GetUIThreadTaskRunner({})` or `content::GetIOThreadTaskRunner({})` from `content/public/browser/browser_thread.h`

You may provide additional BrowserTaskTraits as a parameter to those methods though this is generally still uncommon in BrowserThreads and should be reserved for advanced use cases.

There's an ongoing migration (task APIs v3) away from the previous base-API-with-traits which you may still find throughout the codebase (it's equivalent):

```
base::PostTask(FROM_HERE, {content::BrowserThread::UI}, ...);

base::CreateSingleThreadTaskRunner({content::BrowserThread::IO})
    ->PostTask(FROM_HERE, ...);
```

Note: For the duration of the migration, you'll unfortunately need to continue manually including `content/public/browser/browser_task_traits.h` . to use the browser_thread.h API.

The main thread and the IO thread are already super busy. Therefore, prefer posting to a general purpose thread when possible (ref. Posting a Parallel Task, Posting a Sequenced task). Good reasons to post to the main thread are to update the UI or access objects that are bound to it (e.g. `Profile`). A good reason to post to the IO thread is to access the internals of components that are bound to it (e.g. IPCs, network). Note: It is not necessary to have an explicit post task to the IO thread to send/receive an IPC or send/receive data on the network.

Posting to the Main Thread in a Renderer Process

TODO(blink-dev)

Posting to a Custom SingleThreadTaskRunner

If multiple tasks need to run on the same thread and that thread doesn't have to be the main thread or the IO thread, post them to a `base::SingleThreadTaskRunner` created by `base::Threadpool::CreateSingleThreadTaskRunner` .

```
scoped_refptr<SingleThreadTaskRunner> single_thread_task_runner =
    base::Threadpool::CreateSingleThreadTaskRunner(...);

// TaskB runs after TaskA completes. Both tasks run on the same thread.
single_thread_task_runner->PostTask(FROM_HERE, base::BindOnce(&TaskA));
single_thread_task_runner->PostTask(FROM_HERE, base::BindOnce(&TaskB));
```

Remember that we prefer sequences to physical threads and that this thus should rarely be necessary.

Posting to the Current Thread

IMPORTANT: To post a task that needs mutual exclusion with the current sequence of tasks but doesn't absolutely need to run on the current physical thread, use `base::SequencedTaskRunner::GetCurrentDefault()` instead of `base::SingleThreadTaskRunner::GetCurrentDefault()` (ref. Posting to the Current Sequence). That will better document the requirements of the posted task and will avoid unnecessarily making your API physical thread-affine. In a single-thread task, `base::SequencedTaskRunner::GetCurrentDefault()` is equivalent to `base::SingleThreadTaskRunner::GetCurrentDefault()`.

If you must post a task to the current physical thread nonetheless, use `base::SingleThreadTaskRunner::CurrentDefaultHandle`.

```
// The task will run on the current thread in the future.
base::SingleThreadTaskRunner::GetCurrentDefault()->PostTask(
    FROM_HERE, base::BindOnce(&Task));
```

Posting Tasks to a COM Single-Thread Apartment (STA) Thread (Windows)

Tasks that need to run on a COM Single-Thread Apartment (STA) thread must be posted to a `base::SingleThreadTaskRunner` returned by `base::ThreadPool::CreateCOMSTATaskRunner()`. As mentioned in Posting Multiple Tasks to the Same Thread, all tasks posted to the same `base::SingleThreadTaskRunner` run on the same thread in posting order.

```
// Task(A|B|C)UsingCOMSTA will run on the same COM STA thread.

void TaskAUsingCOMSTA() {
  // [ This runs on a COM STA thread. ]

  // Make COM STA calls.
  // ...

  // Post another task to the current COM STA thread.
  base::SingleThreadTaskRunner::GetCurrentDefault()->PostTask(
      FROM_HERE, base::BindOnce(&TaskCUsingCOMSTA));
```

```
}
void TaskBUsingCOMSTA() { }
void TaskCUsingCOMSTA() { }

auto com_sta_task_runner = base::ThreadPool::CreateCOMSTATaskRunner(...);
com_sta_task_runner->PostTask(FROM_HERE, base::BindOnce(&TaskAUsingCOMSTA));
com_sta_task_runner->PostTask(FROM_HERE, base::BindOnce(&TaskBUsingCOMSTA));
```

Annotating Tasks with TaskTraits

`base::TaskTraits` encapsulate information about a task that helps the thread pool make better scheduling decisions.

Methods that take `base::TaskTraits` can be be passed `{}` when default traits are sufficient. Default traits are appropriate for tasks that:

- Don't block (ref. MayBlock and WithBaseSyncPrimitives);
- Pertain to user-blocking activity; (explicitly or implicitly by having an ordering dependency with a component that does)
- Can either block shutdown or be skipped on shutdown (thread pool is free to choose a fitting default). Tasks that don't match this description must be posted with explicit TaskTraits.

`base/task/task_traits.h` provides exhaustive documentation of available traits. The content layer also provides additional traits in `content/public/browser/browser_task_traits.h` to facilitate posting a task onto a BrowserThread.

Below are some examples of how to specify `base::TaskTraits` .

```
// This task has no explicit TaskTraits. It cannot block. Its priority is
// USER_BLOCKING. It will either block shutdown or be skipped on shutdown.
base::ThreadPool::PostTask(FROM_HERE, base::BindOnce(...));

// This task has the highest priority. The thread pool will schedule it befo
// USER_VISIBLE and BEST_EFFORT tasks.
base::ThreadPool::PostTask(
    FROM_HERE, {base::TaskPriority::USER_BLOCKING},
    base::BindOnce(...));

// This task has the lowest priority and is allowed to block (e.g. it
// can read a file from disk).
base::ThreadPool::PostTask(
    FROM_HERE, {base::TaskPriority::BEST_EFFORT, base::MayBlock()},
    base::BindOnce(...));
```

```
// This task blocks shutdown. The process won't exit before its
// execution is complete.
base::ThreadPool::PostTask(
    FROM_HERE, {base::TaskShutdownBehavior::BLOCK_SHUTDOWN},
    base::BindOnce(...));
```

Keeping the Browser Responsive

Do not perform expensive work on the main thread, the IO thread or any sequence that is expected to run tasks with a low latency. Instead, perform expensive work asynchronously using `base::ThreadPool::PostTaskAndReply*()` or `base::SequencedTaskRunner::PostTaskAndReply()`. Note that asynchronous/overlapped I/O on the IO thread are fine.

Example: Running the code below on the main thread will prevent the browser from responding to user input for a long time.

```
// GetHistoryItemsFromDisk() may block for a long time.
// AddHistoryItemsToOmniboxDropDown() updates the UI and therefore must
// be called on the main thread.
AddHistoryItemsToOmniboxDropdown(GetHistoryItemsFromDisk("keyword"));
```

The code below solves the problem by scheduling a call to `GetHistoryItemsFromDisk()` in a thread pool followed by a call to `AddHistoryItemsToOmniboxDropdown()` on the origin sequence (the main thread in this case). The return value of the first call is automatically provided as argument to the second call.

```
base::ThreadPool::PostTaskAndReplyWithResult(
    FROM_HERE, {base::MayBlock()},
    base::BindOnce(&GetHistoryItemsFromDisk, "keyword"),
    base::BindOnce(&AddHistoryItemsToOmniboxDropdown));
```

Posting a Task with a Delay

Posting a One-Off Task with a Delay

To post a task that must run once after a delay expires, use `base::ThreadPool::PostDelayedTask*()` or `base::TaskRunner::PostDelayedTask()`.

```
base::ThreadPool::PostDelayedTask(
  FROM_HERE, {base::TaskPriority::BEST_EFFORT}, base::BindOnce(&Task),
  base::Hours(1));

scoped_refptr<base::SequencedTaskRunner> task_runner =
    base::ThreadPool::CreateSequencedTaskRunner(
        {base::TaskPriority::BEST_EFFORT});
task_runner->PostDelayedTask(
    FROM_HERE, base::BindOnce(&Task), base::Hours(1));
```

NOTE: A task that has a 1-hour delay probably doesn't have to run right away when its delay expires. Specify `base::TaskPriority::BEST_EFFORT` to prevent it from slowing down the browser when its delay expires.

Posting a Repeating Task with a Delay

To post a task that must run at regular intervals, use `base::RepeatingTimer`.

```
class A {
 public:
  ~A() {
    // The timer is stopped automatically when it is deleted.
  }
  void StartDoingStuff() {
    timer_.Start(FROM_HERE, Seconds(1),
                 this, &A::DoStuff);
  }
  void StopDoingStuff() {
    timer_.Stop();
  }
 private:
  void DoStuff() {
    // This method is called every second on the sequence that invoked
    // StartDoingStuff().
  }
  base::RepeatingTimer timer_;
};
```

Cancelling a Task

Using base::WeakPtr

`base::WeakPtr` can be used to ensure that any callback bound to an object is canceled when that object is destroyed.

```
int Compute() { … }

class A {
 public:
  void ComputeAndStore() {
    // Schedule a call to Compute() in a thread pool followed by
    // a call to A::Store() on the current sequence. The call to
    // A::Store() is canceled when |weak_ptr_factory_| is destroyed.
    // (guarantees that |this| will not be used-after-free).
    base::ThreadPool::PostTaskAndReplyWithResult(
        FROM_HERE, base::BindOnce(&Compute),
        base::BindOnce(&A::Store, weak_ptr_factory_.GetWeakPtr()));
  }

 private:
  void Store(int value) { value_ = value; }

  int value_;
  base::WeakPtrFactory<A> weak_ptr_factory_{this};
};
```

Note: `WeakPtr` is not thread-safe: `~WeakPtrFactory()` and `Store()` (bound to a `WeakPtr`) must all run on the same sequence.

Using base::CancelableTaskTracker

`base::CancelableTaskTracker` allows cancellation to happen on a different sequence than the one on which tasks run. Keep in mind that `CancelableTaskTracker` cannot cancel tasks that have already started to run.

```
auto task_runner = base::ThreadPool::CreateTaskRunner({});
base::CancelableTaskTracker cancelable_task_tracker;
cancelable_task_tracker.PostTask(task_runner.get(), FROM_HERE,
                                 base::DoNothing());
// Cancels Task(), only if it hasn't already started running.
cancelable_task_tracker.TryCancelAll();
```

Posting a Job to run in parallel

The `base::PostJob` is a power user API to be able to schedule a single base::RepeatingCallback worker task and request that ThreadPool workers invoke it in parallel. This avoids degenerate cases:

- Calling `PostTask()` for each work item, causing significant overhead.
- Fixed number of `PostTask()` calls that split the work and might run for a long time. This is problematic when many components post "num cores" tasks and all expect to use all the cores. In these cases, the scheduler lacks context to be fair to multiple same-priority requests and/or ability to request lower priority work to yield when high priority work comes in.

See `base/task/job_perftest.cc` for a complete example.

```
// A canonical implementation of |worker_task|.
void WorkerTask(base::JobDelegate* job_delegate) {
  while (!job_delegate->ShouldYield()) {
    auto work_item = TakeWorkItem(); // Smallest unit of work.
    if (!work_item)
      return:
    ProcessWork(work_item);
  }
}

// Returns the latest thread-safe number of incomplete work items.
void NumIncompleteWorkItems(size_t worker_count) {
  // NumIncompleteWorkItems() may use |worker_count| if it needs to account
  // local work lists, which is easier than doing its own accounting, keepin
  // mind that the actual number of items may be racily overestimated and th
  // WorkerTask() may be called when there's no available work.
  return GlobalQueueSize() + worker_count;
}

base::PostJob(FROM_HERE, {},
              base::BindRepeating(&WorkerTask),
              base::BindRepeating(&NumIncompleteWorkItems));
```

By doing as much work as possible in a loop when invoked, the worker task avoids scheduling overhead. Meanwhile `base::JobDelegate::ShouldYield()` is periodically invoked to conditionally exit and let the scheduler prioritize other work. This yield-semantic allows, for example, a user-visible job to use all cores but get out of the way when a user-blocking task comes in.

Adding additional work to a running job

When new work items are added and the API user wants additional threads to invoke the worker task in parallel, `JobHandle/JobDelegate::NotifyConcurrencyIncrease()` *must* be invoked shortly after max concurrency increases.

Testing

For more details see Testing Components Which Post Tasks.

To test code that uses `base::SingleThreadTaskRunner::CurrentDefaultHandle`, `base::SequencedTaskRunner::CurrentDefaultHandle` or a function in `base/task/thread_pool.h`, instantiate a `base::test::TaskEnvironment` for the scope of the test. If you need BrowserThreads, use `content::BrowserTaskEnvironment` instead of `base::test::TaskEnvironment`.

Tests can run the `base::test::TaskEnvironment`'s message pump using a `base::RunLoop`, which can be made to run until `Quit()` (explicitly or via `RunLoop::QuitClosure()`), or to `RunUntilIdle()` ready-to-run tasks and immediately return.

TaskEnvironment configures RunLoop::Run() to GTEST_FAIL() if it hasn't been explicitly quit after TestTimeouts::action_timeout(). This is preferable to having the test hang if the code under test fails to trigger the RunLoop to quit. The timeout can be overridden with base::test::ScopedRunLoopTimeout.

```
class MyTest : public testing::Test {
 public:
  // ...
 protected:
   base::test::TaskEnvironment task_environment_;
};

TEST_F(MyTest, FirstTest) {
  base::SingleThreadTaskRunner::GetCurrentDefault()->PostTask(FROM_HERE, bas
  base::SequencedTaskRunner::GetCurrentDefault()->PostTask(FROM_HERE,
                                                           base::BindOnce(&B));
  base::SingleThreadTaskRunner::GetCurrentDefault()->PostDelayedTask(
      FROM_HERE, base::BindOnce(&C), base::TimeDelta::Max());

  // This runs the (SingleThread|Sequenced)TaskRunner::CurrentDefaultHandle
  // Delayed tasks are not added to the queue until they are ripe for execut
  // Prefer explicit exit conditions to RunUntilIdle when possible:
  // bit.ly/run-until-idle-with-care2.
  base::RunLoop().RunUntilIdle();
```

```
  // A and B have been executed. C is not ripe for execution yet.

  base::RunLoop run_loop;
  base::SingleThreadTaskRunner::GetCurrentDefault()->PostTask(FROM_HERE, base
  base::SingleThreadTaskRunner::GetCurrentDefault()->PostTask(FROM_HERE, run_
  base::SingleThreadTaskRunner::GetCurrentDefault()->PostTask(FROM_HERE, base

  // This runs the (SingleThread|Sequenced)TaskRunner::CurrentDefaultHandle
  // invoked.
  run_loop.Run();
  // D and run_loop.QuitClosure() have been executed. E is still in the queue

  // Tasks posted to thread pool run asynchronously as they are posted.
  base::ThreadPool::PostTask(FROM_HERE, {}, base::BindOnce(&F));
  auto task_runner =
      base::ThreadPool::CreateSequencedTaskRunner({});
  task_runner->PostTask(FROM_HERE, base::BindOnce(&G));

  // To block until all tasks posted to thread pool are done running:
  base::ThreadPoolInstance::Get()->FlushForTesting();
  // F and G have been executed.

  base::ThreadPool::PostTaskAndReplyWithResult(
      FROM_HERE, {}, base::BindOnce(&H), base::BindOnce(&I));

  // This runs the (SingleThread|Sequenced)TaskRunner::CurrentDefaultHandle
  // (SingleThread|Sequenced)TaskRunner::CurrentDefaultHandle queue and the
  // empty. Prefer explicit exit conditions to RunUntilIdle when possible:
  // bit.ly/run-until-idle-with-care2.
  task_environment_.RunUntilIdle();
  // E, H, I have been executed.
}
```

Using ThreadPool in a New Process

ThreadPoolInstance needs to be initialized in a process before the functions in `base/task/thread_pool.h` can be used. Initialization of ThreadPoolInstance in the Chrome browser process and child processes (renderer, GPU, utility) has already been taken care of. To use ThreadPoolInstance in another process, initialize ThreadPoolInstance early in the main function:

```
// This initializes and starts ThreadPoolInstance with default params.
base::ThreadPoolInstance::CreateAndStartWithDefaultParams("process_name");
// The base/task/thread_pool.h API can now be used with base::ThreadPool tra
// Tasks will be scheduled as they are posted.

// This initializes ThreadPoolInstance.
base::ThreadPoolInstance::Create("process_name");
// The base/task/thread_pool.h API can now be used with base::ThreadPool tra
// threads will be created and no tasks will be scheduled until after Start(
// called.
base::ThreadPoolInstance::Get()->Start(params);
// ThreadPool can now create threads and schedule tasks.
```

And shutdown ThreadPoolInstance late in the main function:

```
base::ThreadPoolInstance::Get()->Shutdown();
// Tasks posted with TaskShutdownBehavior::BLOCK_SHUTDOWN and
// tasks posted with TaskShutdownBehavior::SKIP_ON_SHUTDOWN that
// have started to run before the Shutdown() call have now completed their
// execution. Tasks posted with
// TaskShutdownBehavior::CONTINUE_ON_SHUTDOWN may still be
// running.
```

TaskRunner ownership (encourage no dependency injection)

TaskRunners shouldn't be passed through several components. Instead, the component that uses a TaskRunner should be the one that creates it.

See this example of a refactoring where a TaskRunner was passed through a lot of components only to be used in an eventual leaf. The leaf can and should now obtain its TaskRunner directly from `base/task/thread_pool.h`.

As mentioned above, `base::test::TaskEnvironment` allows unit tests to control tasks posted from underlying TaskRunners. In rare cases where a test needs to more precisely control task ordering: dependency injection of TaskRunners can be useful. For such cases the preferred approach is the following:

```
class Foo {
 public:

  // Overrides |background_task_runner_| in tests.
```

```
  void SetBackgroundTaskRunnerForTesting(
      scoped_refptr<base::SequencedTaskRunner> background_task_runner) {
    background_task_runner_ = std::move(background_task_runner);
  }

 private:
  scoped_refptr<base::SequencedTaskRunner> background_task_runner_ =
      base::ThreadPool::CreateSequencedTaskRunner(
          {base::MayBlock(), base::TaskPriority::BEST_EFFORT});
}
```

Note that this still allows removing all layers of plumbing between //chrome and that component since unit tests will use the leaf layer directly.

FAQ

See Threading and Tasks FAQ for more examples.

Internals

SequenceManager

SequenceManager manages TaskQueues which have different properties (e.g. priority, common task type) multiplexing all posted tasks into a single backing sequence. This will usually be a MessagePump. Depending on the type of message pump used other events such as UI messages may be processed as well. On Windows APC calls (as time permits) and signals sent to a registered set of HANDLEs may also be processed.

MessagePump

MessagePumps are responsible for processing native messages as well as for giving cycles to their delegate (SequenceManager) periodically. MessagePumps take care to mixing delegate callbacks with native message processing so neither type of event starves the other of cycles.

There are different MessagePumpTypes, most common are:

- DEFAULT: Supports tasks and timers only
- UI: Supports native UI events (e.g. Windows messages)
- IO: Supports asynchronous IO (not file I/O!)
- CUSTOM: User provided implementation of MessagePump interface

RunLoop

RunLoop is a helper class to run the RunLoop::Delegate associated with the current thread (usually a SequenceManager). Create a RunLoop on the stack and call Run/Quit to run a nested RunLoop but please avoid nested loops in production code!

Task Reentrancy

SequenceManager has task reentrancy protection. This means that if a task is being processed, a second task cannot start until the first task is finished. Reentrancy can happen when processing a task, and an inner message pump is created. That inner pump then processes native messages which could implicitly start an inner task. Inner message pumps are created with dialogs (DialogBox), common dialogs (GetOpenFileName), OLE functions (DoDragDrop), printer functions (StartDoc) and *many* others.

```
Sample workaround when inner task processing is needed:
  HRESULT hr;
  {
    CurrentThread::ScopedAllowApplicationTasksInNativeNestedLoop allow;
    hr = DoDragDrop(...); // Implicitly runs a modal message loop.
  }
  // Process |hr| (the result returned by DoDragDrop()).
```

Please be SURE your task is reentrant (nestable) and all global variables are stable and accessible before before using CurrentThread::ScopedAllowApplicationTasksInNativeNestedLoop.

APIs for general use

User code should hardly ever need to access SequenceManager APIs directly as these are meant for code that deals with scheduling. Instead you should use the following:

- base::RunLoop: Drive the SequenceManager from the thread it's bound to.
- base::Thread/SequencedTaskRunner::CurrentDefaultHandle: Post back to the SequenceManager TaskQueues from a task running on it.
- SequenceLocalStorageSlot : Bind external state to a sequence.
- base::CurrentThread : Proxy to a subset of Task related APIs bound to the current thread
- Embedders may provide their own static accessors to post tasks on specific loops (e.g. content::BrowserThreads).

SingleThreadTaskExecutor and TaskEnvironment

Instead of having to deal with SequenceManager and TaskQueues code that needs a simple task posting environment (one default task queue) can use a SingleThreadTaskExecutor.

Unit tests can use TaskEnvironment which is highly configurable.

MessageLoop and MessageLoopCurrent

You might come across references to MessageLoop or MessageLoopCurrent in the code or documentation. These classes no longer exist and we are in the process or getting rid of all references to them. `base::MessageLoopCurrent` was replaced by `base::CurrentThread` and the drop in replacements for `base::MessageLoop` are `base::SingleThreadTaskExecutor` and `base::Test::TaskEnvironment`.