Optimize long tasks

- What is a task?
- What is the main thread?
- Task management strategies
  - Manually defer code execution
  - Use async/await to create yield points
  - Yield only when necessary
- Gaps in current APIs
  - A dedicated scheduler API
  - Built-in yield with continuation via scheduler.yield
- Conclusion

If you read lots of stuff about web performance, the advice for keeping your JavaScript apps fast tends to involve some of these tidbits:

- "Don't block the main thread."
- "Break up your long tasks."

What does any of that mean? Shipping *less* JavaScript is good, but does that automatically equate to snappier user interfaces throughout the page lifecycle? Maybe, but maybe not.

To get your head around why it's important to optimize tasks in JavaScript, you need to understand the role of tasks and how the browser handles them—and that starts with understanding what a task is.

What is a task?

A *task* is any discrete piece of work that the browser does. Tasks involve work such as rendering, parsing HTML and CSS, running the JavaScript code you write, and other things you may not have direct control over. Of all of this, the JavaScript you write and deploy to the web is a major source of tasks.

{% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/uDfxNSJzEDGD2jy1fU45.png", alt="A screenshot of a task as depicted in the performance profliler of Chrome's DevTools. The task is at the top of a stack, with a click event handler, a function call, and more items beneath it. The task also includes some rendering work on the right-hand side.", width="800", height="338" %}
A depiction of a task kicked off by a `click` event handler in the performance profiler in Chrome DevTools.

Tasks impact performance in a couple of ways. For example, when a browser downloads a JavaScript file during startup, it queues tasks to parse and compile that JavaScript so it can be executed. Later on in the page lifecycle, tasks are kicked off when your JavaScript does work such as driving interactions through event handlers, JavaScript-driven animations, and background activity such as analytics collection. All of this stuff—with the exception of web workers and similar APIs—happens on the main thread.

What is the main thread?

The *main thread* is where most tasks are run in the browser. It's called the main thread for a reason: it's the one thread where nearly all the JavaScript you write does its work.

The main thread can only process one task at a time. When tasks stretch beyond a certain point—50 milliseconds to be exact—they're classified as *long tasks*. If the user is attempting to interact with the page while a long task runs—or if an important rendering update needs to happen—the browser will be delayed in handling that work. This results in interaction or rendering latency.

{% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/ZQtmFaSwfopJ3mEn7whr.png", alt="A long task in the performance profiler of Chrome's DevTools. The blocking portion of the task (greater than 50 milliseconds) is depicted with a pattern of red diagonal stripes.", width="800", height="171" %}
A long task as depicted in Chrome's performance profiler. Long tasks are indicated by a red triangle in the corner of the task, with the blocking portion of the task filled in with a pattern of diagonal red stripes.

You need to *break up* tasks. This means taking a single long task and dividing it into smaller tasks that take less time to run individually.

{% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/8Bhl9Ilki4tM0aC1nfn8.png", alt="A single long task versus the same task broken up into shorter task. The long task is one large rectangle, whereas the chunked task is five smaller boxes which are collectively the same width as the long task.", width="800", height="242" %}
A visualization of a single long task versus that same task broken up into five shorter tasks.

This matters because when tasks are broken up, the browser has more opportunities to respond to higher-priority work—and that includes user interactions.

{% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/0yV0ynwW7FujIwvCbCxQ.png", alt="A depiction of how breaking up a task can facilitate a user interaction. At the top, a long task blocks an event handler from running until the task is finished. At the bottom, the chunked up task permits the event handler to run sooner than it otherwise would have.", width="800", height="448" %}
A visualization of what happens to interactions when tasks are too long and the browser can't respond quickly enough to interactions, versus when longer tasks are broken up into smaller tasks.

At the top of the preceding figure, an event handler queued up by a user interaction had to wait for a single long task before it could run, This delays the interaction from taking place. At the bottom, the event handler has a chance to run sooner. Because the event handler had an opportunity to run in between smaller tasks, it runs sooner than if it had to wait for a long task to finish. In the top example, the user might have noticed lag; in the bottom, the interaction might have felt *instant*.

The problem, though, is that the advice of "break up your long tasks" and "don't block the main thread" isn't specific enough unless you already know *how* to do those things. That's what this guide will explain.

Task management strategies

A common bit of advice in software architecture is to break up your work into smaller functions. This gives you the benefits of better code readability, and project maintainability. This also makes tests easier to write.

```
function saveSettings () {
  validateForm();
  showSpinner();
  saveToDatabase();
  updateUI();
  sendAnalytics();
}
```

In this example, there's a function named `saveSettings()` that calls five functions within it to do the work, such as validating a form, showing a spinner, sending data, and so on. Conceptually, this is well architected. If you need to debug one of these functions, you can traverse the project tree to figure out what each function does.

The problem, however, is that JavaScript doesn't run each of these functions as separate tasks because they are being executed within the `saveSettings()` function. **This means that all five functions run as a single task.**

{% Aside 'important' %} JavaScript works this way because it uses the run-to-completion model of task execution. This means that each task will run until it finishes, regardless of how long it blocks the main thread. {% endAside %}

{% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/1gENo7PnUwPhbTJmQKmX.png", alt="The saveSettings function as depicted in Chrome's performance profiler. While the top-level function calls five other functions, all the work takes place in one long task that blocks the main thread.", width="800", height="301" %}
A single function `saveSettings()` that calls five functions. The work is run as part of one long monolithic task.

In the best case scenario, even just one of those functions can contribute 50 milliseconds or more to the total length of the task. In the worst case, more of those tasks can run quite a bit longer—especially on resource-constrained devices. What follows is a set of strategies you can use to break up and prioritize tasks.

Manually defer code execution

One method developers have used to break up tasks into smaller ones involves `setTimeout()`. With this technique, you pass the function to `setTimeout()`. This postpones execution of the callback into a separate task, even if you specify a timeout of `0`.

```
function saveSettings () {
  // Do critical work that is user-visible:
  validateForm();
  showSpinner();
  updateUI();

  // Defer work that isn't user-visible to a separate task:
  setTimeout(() => {
    saveToDatabase();
    sendAnalytics();
  }, 0);
}
```

This works well if you have a series of functions that need to run sequentially, but your code may not always be organized this way. For example, you could have a large amount of data that needs to be processed in a loop, and that task could take a very long time if you have millions of items.

```
function processData () {
  for (const item of largeDataArray) {
    // Process the individual item here.
  }
}
```

Using `setTimeout()` here is problematic, because the ergonomics of it make it difficult to implement, and the entire array of data could take a very long time to process, even if each item can be processed very quickly. It all adds up, and `setTimeout()` isn't the right tool for the job—at least not when used this way.

In addition to `setTimeout()`, there are a few other APIs that allow you to defer code execution to a subsequent task. One involves using `postMessage()` for faster timeouts. You can also break up work using `requestIdleCallback()`—but beware!—`requestIdleCallback()` schedules tasks at the lowest possible priority, and only during browser idle time. When the main thread is congested, tasks scheduled with `requestIdleCallback()` may never get to run.

Use `async`/`await` to create yield points

A phrase you'll see throughout the rest of this guide is "yield to the main thread"—but what does that mean? Why should you do it? When should you do it?

{% Aside 'important' %} When you *yield* to the main thread, you're giving it an opportunity to handle more important tasks than the ones that are currently queued up. Ideally, you should yield to the main thread whenever you have some crucial user-facing work that needs to execute sooner than if you didn't yield. **Yielding to the main thread creates opportunities for critical work to run sooner.** {% endAside %}

When tasks are broken up, other tasks can be prioritized better by the browser's internal prioritization scheme. One way to yield to the main thread involves using a combination of a `Promise` that resolves with a call to `setTimeout()`:

```
function yieldToMain () {
  return new Promise(resolve => {
    setTimeout(resolve, 0);
  });
}
```

{% Aside 'caution' %} While this code example returns a `Promise` that resolves after a call to `setTimeout()`, it's not the `Promise` that is responsible for running the rest of the code in a new task, it's the `setTimeout()` call. Promise callbacks run as microtasks rather than tasks, and therefore don't yield to the main thread. {% endAside %}

In the `saveSettings()` function, you can yield to the main thread after each bit of work if you `await` the `yieldToMain()` function after each function call:

```
async function saveSettings () {
  // Create an array of functions to run:
  const tasks = [
    validateForm,
    showSpinner,
    saveToDatabase,
    updateUI,
    sendAnalytics
  ]

  // Loop over the tasks:
  while (tasks.length > 0) {
    // Shift the first task off the tasks array:
    const task = tasks.shift();

    // Run the task:
    task();

    // Yield to the main thread:
    await yieldToMain();
  }
}
```

{% Aside 'important' %} You don't have to yield after *every* function call. For example, if you run two functions that result in critical updates to the user interface, you probably don't want to yield in between them. If you can, let that work run first, *then* consider yielding in between functions that do less critical or background work that the user doesn't see. {% endAside %}

The result is that the once-monolithic task is now broken up into separate tasks.

> {% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/mzrFeqjk00zUlKK0xzI8.png", alt="The same saveSettings function depicted in Chrome's performance profiler, only with yielding. The result is the once-monolithic task is now broken up into five separate tasks—one for each function.", width="800", height="350" %}
> The `saveSettings()` function now executes its child functions as separate tasks.

The benefit of using a promise-based approach to yielding rather than manual use of `setTimeout()` is better ergonomics. Yield points become declarative, and therefore easier to write, read, and understand.

Yield only when necessary

What if you have a bunch of tasks, but you only want to yield if the user attempts to interact with the page? That's the kind of thing that `isInputPending()` was made for.

`isInputPending()` is a function you can run at any time to determine if the user is attempting to interact with a page element: a call to `isInputPending()` will return `true`. It returns `false` otherwise.

Say you have a queue of tasks you need to run, but you don't want to get in the way of any inputs. This code—which uses both `isInputPending()` and our custom `yieldToMain()` function—ensures that an input won't be delayed while the user is trying to interact with the page:

```
async function saveSettings () {
  // A task queue of functions
  const tasks = [
    validateForm,
    showSpinner,
    saveToDatabase,
    updateUI,
    sendAnalytics
  ];

  while (tasks.length > 0) {
    // Yield to a pending user input:
    if (navigator.scheduling.isInputPending()) {
      // There's a pending user input. Yield here:
      await yieldToMain();
    } else {
      // Shift the task out of the queue:
      const task = tasks.shift();

      // Run the task:
      task();
    }
  }
}
```

While `saveSettings()` runs, it will loop over the tasks in the queue. If `isInputPending()` returns `true` during the loop, `saveSettings()` will call `yieldToMain()` so the user input can be handled. Otherwise, it will shift the next task off the front of the queue and run it continuously. It will do this

until no more tasks are left.

> {% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/Oz00JZNrYH4dNE3pmYss.png", alt="A depiction of the saveSettings function running in Chrome's performance profiler. The resulting task blocks the main thread until isInputPending returns true, at which point, the task yields to the main thread.", width="800", height="260" %}
> `saveSettings()` runs a task queue for five tasks, but the user has clicked to open a menu while the second work item was running. `isInputPending()` yields to the main thread to handle the interaction, and resume running the rest of the tasks.

{% Aside %} `isInputPending()` may not always return `true` immediately after user input. This is because it takes time for the operating system to tell the browser that the interaction occurred. This means that other code may have already started executing (as you can see with the `saveToDatabase()` function in the above screenshot). Even if you use `isInputPending()` it's still crucial that you limit the amount of work you do in each function. {% endAside %}

Using `isInputPending()` in combination with a yielding mechanism is a great way to get the browser to stop whatever tasks it's processing so that it can respond to critical user-facing interactions. That can help improve your page's ability to respond to the user in many situations when a lot of tasks are in flight.

Another way to use `isInputPending()`—particularly if you're concerned about providing a fallback for browsers that don't support it—is to use a time-based approach in conjunction with the optional chaining operator:

```
async function saveSettings () {
  // A task queue of functions
  const tasks = [
    validateForm,
    showSpinner,
    saveToDatabase,
    updateUI,
    sendAnalytics
  ];

  let deadline = performance.now() + 50;

  while (tasks.length > 0) {
    // Optional chaining operator used here helps to avoid
    // errors in browsers that don't support `isInputPending`:
    if (navigator.scheduling?.isInputPending() || performance.now() >= deadline) {
      // There's a pending user input, or the
      // deadline has been reached. Yield here:
```

```
      await yieldToMain();

      // Extend the deadline:
      deadline = performance.now() + 50;

      // Stop the execution of the current loop and
      // move onto the next iteration:
      continue;
    }

    // Shift the task out of the queue:
    const task = tasks.shift();

    // Run the task:
    task();
  }
}
```

With this approach, you get a fallback for browsers that don't support `isInputPending()` by using a time-based approach that uses (and adjusts) a deadline so that work will be broken up where necessary, whether by yielding to user input, or by a certain point in time.

Gaps in current APIs

The APIs mentioned so far can help you break up tasks, but they have a significant downside: when you yield to the main thread by deferring code to run in a subsequent task, that code gets added to the very end of the task queue.

If you control all the code on your page, it's possible to create your own scheduler with the ability to prioritize tasks, but third-party scripts won't use your scheduler. In effect, you're not really able to *prioritize* work in such environments. You can only chunk it up, or explicitly yield to user interactions.

Fortunately, there is a dedicated scheduler API that is currently in development that addresses these problems.

A dedicated scheduler API

The scheduler API currently offers the `postTask()` function which, at the time of writing, is available in Chromium browsers, and in Firefox behind a flag. `postTask()` allows for finer-grained scheduling

of tasks, and is one way to help the browser prioritize work so that low priority tasks yield to the main thread. `postTask()` uses promises, and accepts a `priority` setting.

The `postTask()` API has three priorities you can use:

- `'background'` for the lowest priority tasks.
- `'user-visible'` for medium priority tasks. This is the default if no `priority` is set.
- `'user-blocking'` for critical tasks that need to run at high priority.

Take the following code as an example, where the `postTask()` API is used to run three tasks at the highest possible priority, and the remaining two tasks at the lowest possible priority.

```
function saveSettings () {
  // Validate the form at high priority
  scheduler.postTask(validateForm, {priority: 'user-blocking'});

  // Show the spinner at high priority:
  scheduler.postTask(showSpinner, {priority: 'user-blocking'});

  // Update the database in the background:
  scheduler.postTask(saveToDatabase, {priority: 'background'});

  // Update the user interface at high priority:
  scheduler.postTask(updateUI, {priority: 'user-blocking'});

  // Send analytics data in the background:
  scheduler.postTask(sendAnalytics, {priority: 'background'});
};
```

Here, the priority of tasks is scheduled in such a way that browser-prioritized tasks—such as user interactions—can work their way in.

> {% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/J6Ec735EReBFCTj1CuVf.png", alt="The saveSettings function as depicted in Chrome's performance profiler, but using postTask. postTask splits up each function saveSettings runs, and prioritizes them such that a user interaction has a chance to run without being blocked.", width="800", height="342" %}
> When `saveSettings()` is run, the function schedules the individual functions using `postTask()`. The critical user-facing work is scheduled at high priority, while work the user doesn't know about is scheduled to run in the background. This allows for user interactions to execute more quickly, as the work is both broken up *and* prioritized appropriately.

This is a simplistic example of how `postTask()` can be used. It's possible to instantiate different `TaskController` objects that can share priorities between tasks, including the ability to change priorities for different `TaskController` instances as needed.

{% Aside 'important' %} `postTask()` is not supported in all browsers. You can use feature detection to see if it's available, or consider using a polyfill. {% endAside %}

Built-in yield with continuation via `scheduler.yield`

{% Aside 'important' %} For a more detailed explanation on `scheduler.yield`, read this post about its origin trial, as well as its explainer. {% endAside %}

One proposed part of the scheduler API is `scheduler.yield`, an API specifically designed for yielding to the main thread in the browser which is currently available to try as an origin trial. Its use resembles the `yieldToMain()` function demonstrated earlier in this article:

```
async function saveSettings () {
  // Create an array of functions to run:
  const tasks = [
    validateForm,
    showSpinner,
    saveToDatabase,
    updateUI,
    sendAnalytics
  ]

  // Loop over the tasks:
  while (tasks.length > 0) {
    // Shift the first task off the tasks array:
    const task = tasks.shift();

    // Run the task:
    task();

    // Yield to the main thread with the scheduler
    // API's own yielding mechanism:
    await scheduler.yield();
  }
```

```
}
```

You'll note that the code above is largely familiar, but instead of using `yieldToMain()`, you call and `await scheduler.yield()` instead.

> {% Img
> src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/5q9WG0lmBOwlqGYK16hd.png",
> alt="Three diagrams depicting tasks without yielding, yielding, and with yielding and continuation. Without yielding, there are long tasks. With yielding, there are more tasks that are shorter, but may be interrupted by other unrelated tasks. With yielding and continuation, there are more tasks that are shorter, but their order of execution is preserved.", width="800", height="337" %}
> A visualization of task execution without yielding, with yielding, and with yielding and continuation. When `scheduler.yield()` is used, task execution picks up where it left off even after the yield point.

The benefit of `scheduler.yield()` is continuation, which means that if you yield in the middle of a set of tasks, the other scheduled tasks will continue in the same order after the yield point. **This avoids code from third-party scripts from usurping the order of your code's execution.**

Conclusion

Managing tasks is challenging, but doing so helps your page respond more quickly to user interactions. There's no one single piece of advice for managing and prioritizing tasks. Rather, it's a number of different techniques. To reiterate, these are the main things you'll want to consider when managing tasks:

- Yield to the main thread for critical, user-facing tasks.
- Use `isInputPending()` to yield to the main thread when the user is trying to interact with the page.
- Prioritize tasks with `postTask()`.
- Finally, **do as little work as possible in your functions.**

With one or more of these tools, you should be able to structure the work in your application so that it prioritizes the user's needs, while ensuring that less critical work still gets done. That's going to create a better user experience which is more responsive and more enjoyable to use.

Special thanks to Philip Walton for his technical vetting of this article.

Hero image sourced from Unsplash, courtesy of Amirali Mirhashemian.

□□□ #2
□ □□□ □□□ 2026-10-31 22:07:34 UTC
□ □□□ □□□ 2026-10-31 22:07:35 UTC