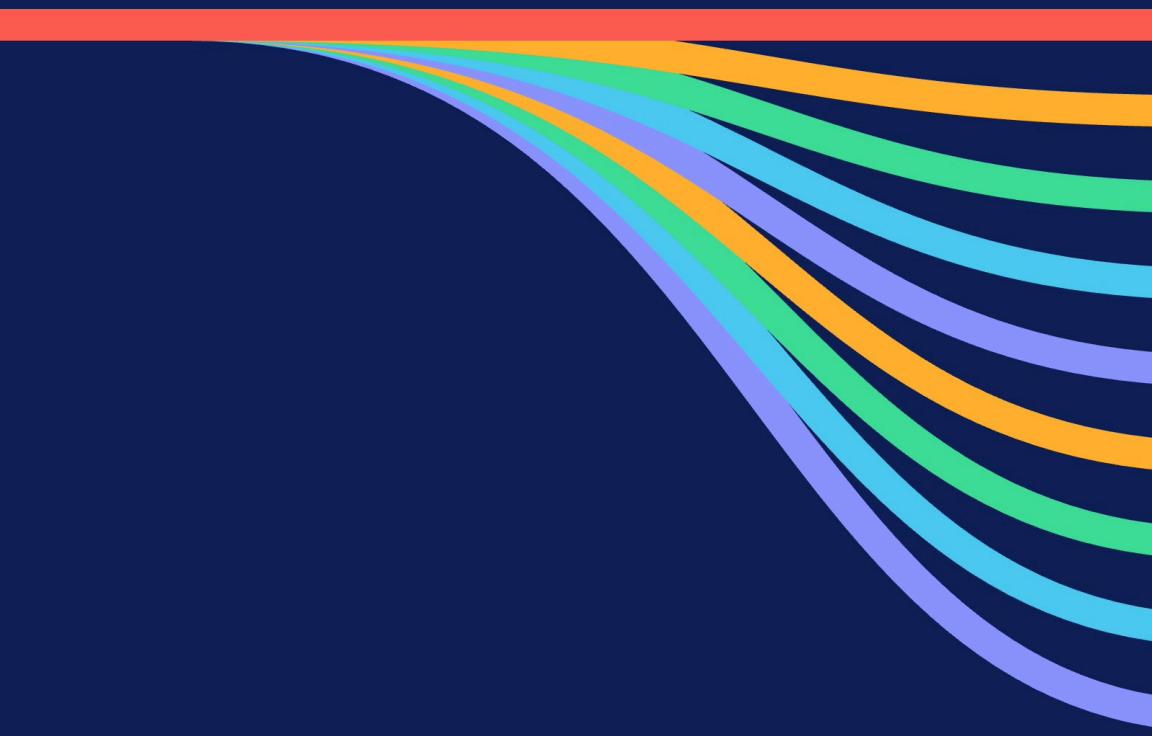
JAVASCRIPT



Beyond the Main Thread

Workers, Shared Memory and Scheduling for **JavaScript** in the Browser

Sandeep Kumar Patel

Beyond the Main Thread

Workers, Shared Memory and Scheduling for JavaScript in the Browser

Sandeep Kumar Patel

Copyright

Beyond the Main Thread: Workers, Shared Memory and Scheduling for JavaScript in the Browser

Copyright © 2026 Sandeep Kumar Patel. All rights reserved.

No part of this book may be reproduced or transmitted in any form without the written permission of the author, except for short quotations in reviews and articles.

The demo code that goes with this book is released under the MIT License, and the license text is in the code download. Every effort was made to check the facts and measurements in this book, but the author accepts no liability for damage caused by using the information in it. Browser features change, so check current documentation before you depend on one.

Chrome, WebAssembly and other product names are the trademarks of their owners. They are used here only to identify the technologies the book describes.

First edition, 2026.

Contents

Preface **1**

Who this book is for 1

What you'll need 1

How the book is organized 2

Conventions 2

A note on browsers 2

1 The Main Thread Problem **3**

1.1 Technical requirements 3

1.2 One thread, many jobs 3

1.3 Three ways out 5

1.4 A map of the tools 6

1.5 What the numbers will look like 7

1.6 A word on the demos 8

1.7 Summary 9

1.8 Check your understanding 9

2 Dedicated Workers **10**

2.1 Technical requirements 10

2.2 The idea 10

2.3 How it works 12

2.4 Where it is used 16

2.5 Building the demo 17

2.6 Pitfalls and browser support 23

2.7 Summary 24

2.8 Check your understanding 24

Preface

A page works fine in development, then a user with a slower machine or a bigger file makes it freeze. The fixes are scattered across a dozen browser features, written up in a dozen places, and few say what a feature costs or when it goes wrong.

This book puts them side by side. It covers the browser's tools for work that doesn't belong on the main thread: dedicated, shared and service workers, `OffscreenCanvas`, shared memory with `Atomics`, WebAssembly threads, the Scheduler API, the Origin Private File System, and Observables. For each one, we explain the idea, show where it's used, build a demo you can run, and measure what it does.

Who this book is for

You're a working web developer. You're comfortable with JavaScript, including promises and `async` and `await`, and you've built pages that run in a browser. You don't need to have used workers or shared memory before. If you have, the measurements may still tell you something new.

You don't need to know C, Rust or WebAssembly. Chapter 7 reads a small piece of WebAssembly text and explains what it needs as it goes.

What you'll need

A current version of Chrome and Node.js 20 or later. Everything else is in the code download that goes with the book, a single zip file at https://drive.google.com/file/d/1uXT32Mr7WYbPXagdg-EeMaclQJQEp8Me7/view?usp=sharing. Chapter 1 shows how to run it.

How the book is organized

Chapter 1 explains the problem. Chapters 2 to 5 cover the kinds of worker and the canvas. Chapters 6 and 7 cover shared memory and WebAssembly threads. Chapters 8 to 10 cover scheduling, files and event streams. The appendix has a summary of browser support, notes on running the demos, a glossary and a list of further reading.

The early chapters build on each other, but after Chapter 2 most stand alone. If you only need one tool, go straight to its chapter and follow the cross-references back when you need them.

Conventions

Code appears in `this font`, and longer listings are set apart. Every listing in the book comes from a working demo, trimmed for the page. Figures numbered with the chapter, such as Figure 4.2, are either diagrams or screenshots of the demos running.

When we say “in our test,” we mean a measurement from the demos, run in Chrome 155 on one laptop. We give ranges where the numbers vary, and Chapter 1 explains how to read them.

A note on browsers

Some features in this book, such as `scheduler.yield()` and Observables, are new, and support in browsers other than Chrome is changing. Check the compatibility tables on MDN and on caniuse.com before you depend on a feature. The appendix says what we verified.

CHAPTER 1

The Main Thread Problem

You click a button and nothing happens. The spinner stops turning. Text you type shows up a second late, all at once. A minute later the page wakes up.

That's the main thread being busy. This book is about the tools the browser gives you to prevent it: workers, shared memory, a scheduler, and a few storage and event features that go with them. Each chapter takes one tool, explains the idea behind it, shows where real software uses it, and walks through a small demo you can run and break yourself.

This chapter covers the problem first. Once you know why a page freezes, the tools make sense; without that they look like unrelated APIs.

1.1 Technical requirements

You'll need a current version of Chrome, Node.js 20 or later, and a text editor. The code download is a single zip file, `beyond-the-main-thread-code.zip`, at https://drive.google.com/file/d/1uX-T32Mr7WYbPXagdgEeMaclQJQEp8Me7/view?usp=sharing. Every demo has its own folder under `public/`. Unzip it, open a terminal in the unzipped folder, and run:

```
npm install
npm run build:wasm
npm start
```

Then open `http://localhost:3000`. We use Chrome for every screenshot and every measurement in the book, and the appendix lists where other browsers differ.

1.2 One thread, many jobs

A browser tab runs your JavaScript, handles clicks and keystrokes, calculates layout, and paints pixels, and most of that happens on a single thread. We call it the main thread.

One thread means one job at a time. While your code runs, the browser can't respond to a click, and it can't repaint. It can't interrupt your code, so it waits.

For most code this is fine, because most code finishes in a millisecond or two. The trouble starts with longer work: sorting a hundred thousand rows, resizing an image, parsing a large file. Any of those can hold the thread for hundreds of milliseconds, and people notice. Past about fifty milliseconds a task counts as a *long task*, the threshold browsers and performance tools use to flag a sluggish page.

Figure 1.1 shows the problem and the usual fix.

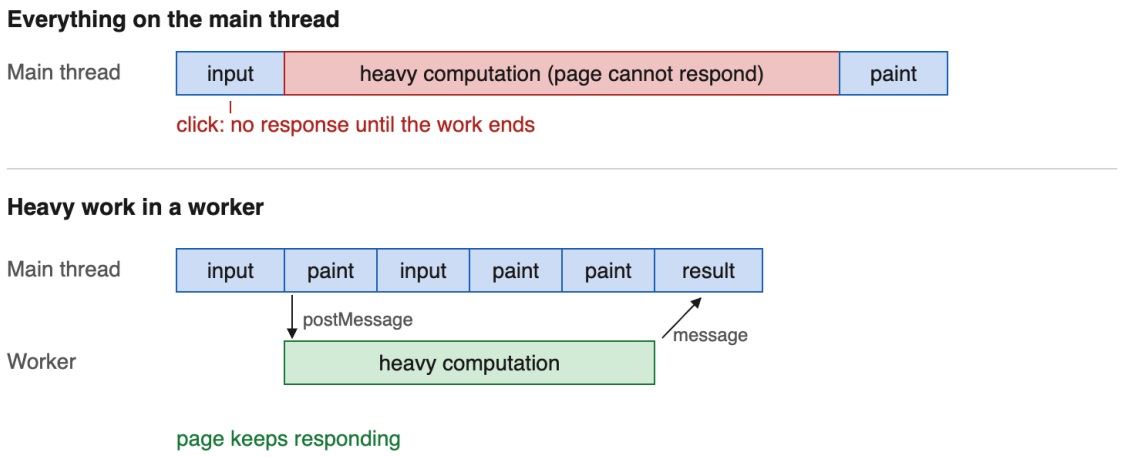


Figure 1.1: Heavy computation on the main thread blocks input and painting. The same work in a worker leaves the main thread free.

In the top lane, the computation blocks everything else: the click arrives, but the browser can't get to it. In the bottom lane, the main thread hands the work off with a message and carries on, and the worker sends a message back when it finishes.

1.2.1 The frame budget

The threshold comes from the screen. A typical display refreshes 60 times a second, which gives the browser about 16.7 milliseconds to produce each frame. In that time it has to handle any input, run whatever JavaScript is due, lay out anything that changed, and paint. Figure 1.2 shows a healthy frame and what happens when your own code takes too long.

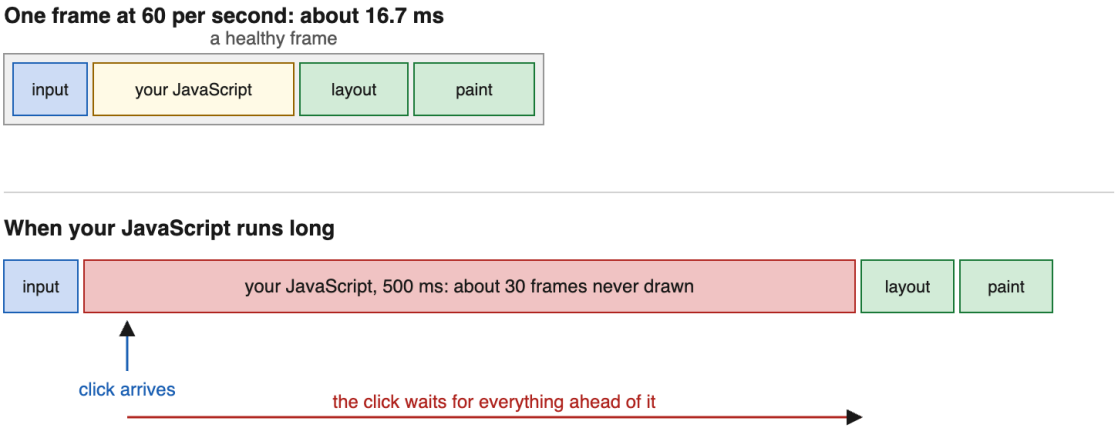


Figure 1.2: A frame has about 16.7 ms to fit input, JavaScript, layout and paint. A 500 ms task takes the place of about 30 frames, and a click that arrives at the start has to wait for all of it.

A task of 16 milliseconds eats a whole frame. A task of 500 milliseconds eats about thirty. The user doesn't see "slow". They see a broken page that has stopped responding, with no way to tell whether it's working or dead.

Google's Core Web Vitals include a measure called *Interaction to Next Paint*, or INP, that measures this. It records how long a page takes to show a response after the user does something. A result of 200 milliseconds or less counts as good, and over 500 as poor, measured across real visits. Long tasks are a common reason pages do badly on it.

1.2.2 Why you can't just be careful

Writing fast code helps, but the main thread is shared with your framework, every library, every third-party script, and the browser's own work, and you control only some of it. "Fast" is also relative to the machine. A loop that takes 20 milliseconds on your development laptop can take several times that on a modest phone, and the phone is where many of your users are.

The goal is to make sure that when code is slow, it doesn't get in the way.

1.3 Three ways out

There are three things you can do about a long task.

Make it shorter. Sometimes the work is wasteful and a better algorithm fixes it. Check this first: it's the cheapest option and needs no new APIs.

Break it up. If the work can pause, split it into small pieces and let the browser handle input and paint between them. The Scheduler API in Chapter 8 is built for this. Nothing runs in parallel, but the page stays responsive.

Move it. Run the work on another thread. This is what workers are for, and it's the subject of most of the book. The work can use another CPU core while the main thread stays free.

Moving work has a price. A worker is a separate world: it has its own memory, it can't touch the page, and talking to it means sending messages. Much of what follows is about making that conversation cheap, and about the cases where two threads need to look at the same bytes.

1.4 A map of the tools

The rest of the book covers tools that attack this problem from different angles. Figure 1.3, at the end of this section, is a guide to choosing between them. We start with the basic worker in Chapter 2 and build outward.

Chapters 2 to 5 cover the workers you'll meet in everyday projects: dedicated workers for computation, shared workers for state that spans tabs, service workers for caching and offline support, and OffscreenCanvas for drawing without the main thread.

Chapters 6 and 7 go deeper. They cover shared memory, the atomic operations that make it safe, and how WebAssembly uses both to run real multithreaded code in the browser.

Chapters 8 to 10 cover the Scheduler API for breaking up work, the Origin Private File System for fast storage inside workers, and Observables for handling streams of events.

Each chapter describes the problem, explains how the feature works, shows where it appears in real software, builds the demo, and ends with common mistakes.

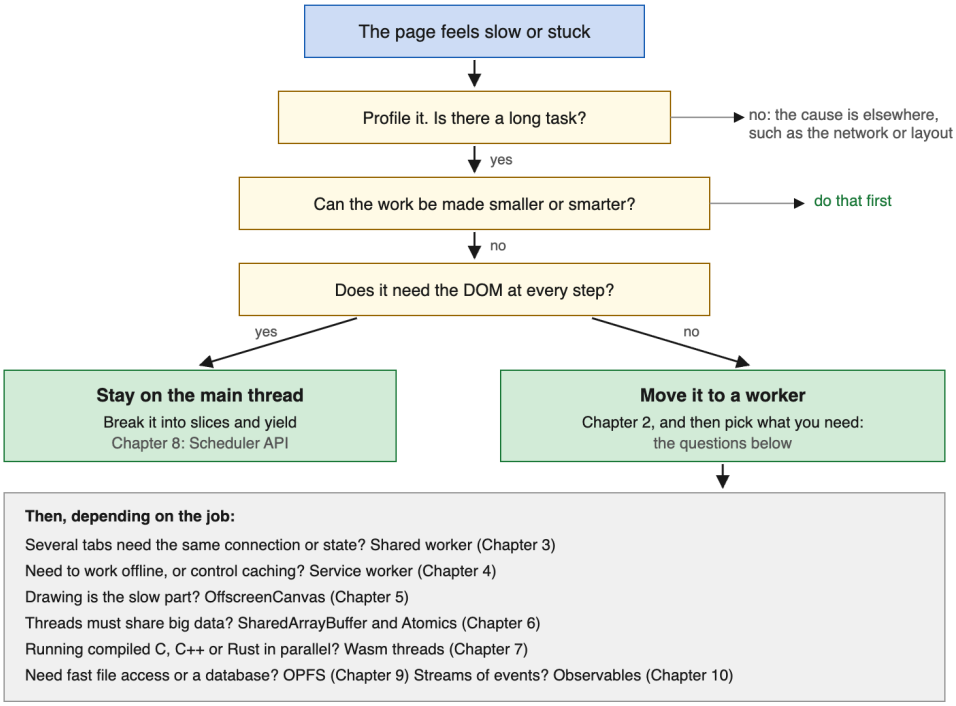


Figure 1.3: A guide to the tools in this book. Start with the profile, make the work smaller if you can, and decide from there whether to slice it up on the main thread or move it to a worker.

1.5 What the numbers will look like

Each demo reports numbers, and each chapter quotes them. The table previews them, with the chapter each comes from.

Chapter	What we measured	What we found
2	A 1.5-second job on the main thread, then in a worker	The same time, but the page stayed alive. Sending 256 MB by copy blocked the sender for 50 to 175 ms; transferring it took about a millisecond
3	A tab crashing while sharing a worker	Noticed within 158 to 214 ms with a lock, and never with a `pagehide` handler
4	Three caching strategies against a slow server	About 810 ms for network-first, a few milliseconds from the cache

Chapter	What we measured	What we found
5	A canvas during a 3-second main-thread freeze	0 frames drawn on the page's canvas, 360 on the worker's
6	Four workers adding to one counter	58 to 69 percent of the updates lost with a plain `counter[0]++`
7	A Mandelbrot render on 1 to 8 threads	Six to seven times faster on eight; a fixed split of the rows was twice as slow as a shared counter
8	A keystroke during a 3-second job	Waited 2,900 ms, or 40 ms with `scheduler.yield()`
9	20,000 small file writes	About 70 ms with a synchronous handle, about 3,500 ms without
10	Eleven keystrokes through a debounce	Two searches

Two results are counterintuitive. In Chapter 7, plain JavaScript beat our hand-written WebAssembly on a single thread. In Chapter 2, the worker wasn't faster at all.

1.5.1 How we measured

All the numbers and screenshots in the book come from Chrome 155 on one Apple-silicon laptop with 11 logical cores. Its screen refreshes at 120 Hz, so a healthy page shows 120 frames a second in our screenshots and in the frame counts in Chapter 5, where many articles assume 60. We ran most tests several times and report ranges, because timings vary from run to run. Some effects, such as the lost updates in Chapter 6, differ every time.

Treat the numbers as examples; your machine will give different ones. What carries over is which approach is faster, by roughly how much, and why. If a rerun looks nothing like ours, investigate.

1.6 A word on the demos

The demos use no framework and no build step beyond the one Wasm file.

A screenshot can't capture a frozen page, since it waits for the page to wake up. Where a demo shows something like that, we describe it and tell you how to trigger it yourself.

To see the problem now, open the Chapter 2 demo and the DevTools Performance panel together, and record a trace while you click **Run on main thread**. The main thread's track shows a single block spanning the whole job, and Chrome marks long tasks like it with a red flag in the corner. The rest of this book is about getting rid of blocks like that.

1.7 Summary

The main thread does many jobs on one thread, so anything slow on it makes the whole page feel stuck. A frame has about 16.7 milliseconds, and a long task takes the place of many of them. You can shorten the work, split it into pieces, or move it to a worker. The rest of the book is about doing the last two well and choosing between the tools that help.

1.8 Check your understanding

1. What is a long task, and why does it matter for how a page feels?
2. At 60 frames a second, how long does the browser have to produce each frame? What happens to a click that arrives during a 500 millisecond task?
3. Name the three ways to deal with slow work on the main thread.
4. Why can’t the browser interrupt a function that’s running on the main thread?

CHAPTER 2

Dedicated Workers

When you move a slow function into a worker, it takes about as long as before. What changes is the page: the spinner keeps turning, buttons respond, and text you type appears as you type it. The work didn't get faster; it stopped getting in the way of everything else.

The dedicated worker is the foundation for almost everything else in this book. This chapter covers what a worker is, how it talks to the page, what that conversation costs, and where it goes wrong. Then we build a demo that measures all of it.

2.1 Technical requirements

You'll need Chrome and the code download from Chapter 1, running with `npm start`. This chapter's demo lives in `public/01-dedicated-worker`, and you can open it at `http://localhost:3000/01-dedicated-worker/`. It has three files that matter: `index.html`, `worker.js`, and `primes.js`.

2.2 The idea

A dedicated worker is a second JavaScript environment that runs alongside your page, on its own thread. You start it from a script, and it's "dedicated" because exactly one page owns it. When that page goes away, so does the worker. (Chapter 3 covers the shared kind, which several pages can connect to.)

A worker isn't a function that happens to run elsewhere; it's a separate *environment*, with its own global object, variables, event loop and memory. Nothing it creates is visible to the page, and nothing the page creates is visible to it. If you declare `let count = 0` in the page, the worker can't see `count`. It can't even tell the page has one.

This is a limitation, but it's also why workers are safe. Two threads reading and writing the same variables is the classic source of hard-to-reproduce bugs, and separate memory avoids it. The only way in or out is a message, and messages are delivered one at a time.

Figure 2.1 shows the two worlds.

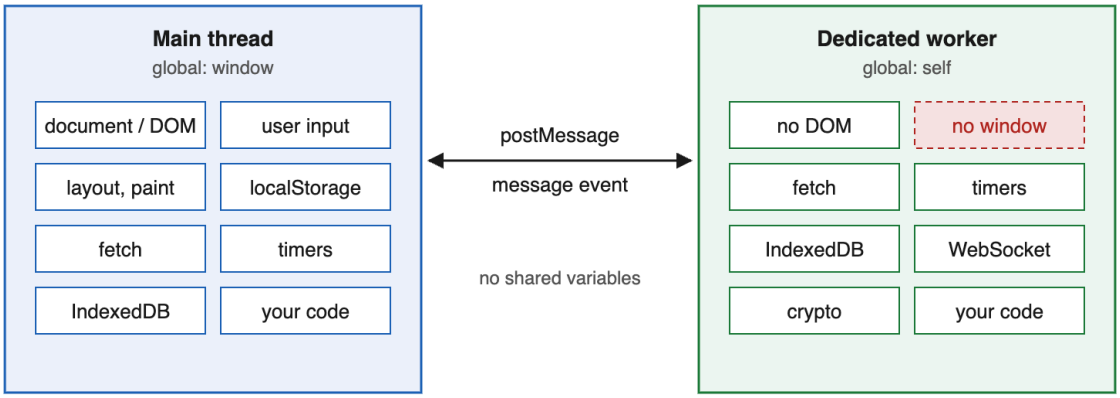


Figure 2.1: The page and a dedicated worker each have their own global object, memory and event loop. They only communicate through messages.

2.2.1 What a worker can and can't do

Inside a worker, the global object is `self`, not `window`. There's no `document`, so no DOM, and no way to touch what's on screen. There's no `localStorage` either.

What you do get is most of the rest of the platform: `fetch`, timers, `WebSocket`, `IndexedDB`, the Web Crypto API, `Blob`, `URL`, `TextEncoder`, and WebAssembly. A worker can start its own network requests, wait on timers, and read and write to databases. It can even create more workers.

A rough rule: if the job is "take some data, do something expensive with it, and give me the result," it belongs in a worker. If it's "change what the user sees," it belongs on the main thread. Most designs end up as a pipeline: the page collects input, the worker computes, the page displays.

2.2.2 What it costs

Workers aren't free. Each one is a separate thread with its own JavaScript heap and runtime. Starting one means the browser fetches your script, parses it, and runs its top level before it can handle the first message. That takes a few milliseconds on a fast machine and a lot more on a slow phone, so measure it in your own app.

Memory is the other cost. Ten workers hold ten copies of everything they load. If your worker script imports a large library, every worker pays for it.

Sending data between threads isn't always cheap either, and sometimes it costs more than the work you were trying to offload.

2.3 How it works

2.3.1 Starting a worker

You create a worker by handing the browser a URL:

```
const worker = new Worker('./worker.js', { type: 'module' });
```

The call returns immediately. The browser starts a thread in the background, fetches `worker.js`, and runs it. Anything you post before the script is ready is queued and delivered once it is.

The script has to come from the same origin as the page. (A `blob:` URL also works, which suits tests and tools that generate worker code on the fly.) A page opened straight off the disk with a `file://` address has no real origin. In our test, Chrome refused to start a worker from one and threw a `SecurityError`. That is one reason the demos run behind a small local server.

The `type: 'module'` option makes the worker a module, so it can use `import` and `export`. Without it you get a classic worker, which loads other scripts with `importScripts()`. Module workers are supported in all current major browsers, and unless you support old browsers they're the better choice, because they let the worker share code with the page, as our demo does with `primes.js`.

If you use a bundler such as Vite or webpack, you'll often see the URL written like this:

```
new Worker(new URL('./worker.js', import.meta.url), { type: 'module' });
```

This does the same thing when the page and the script sit in the same folder, as ours do. The `new URL(...)` form is a pattern the bundler recognizes, so it compiles `worker.js` as a separate file.

2.3.2 A worker's life

Figure 2.2 shows a worker's lifecycle.

After the script runs, the worker settles into its event loop and waits. When a message arrives, your `onmessage` handler runs. While the handler runs, the worker is busy, and any further messages wait in line. When the handler returns, the worker goes back to waiting, or picks up the next message.

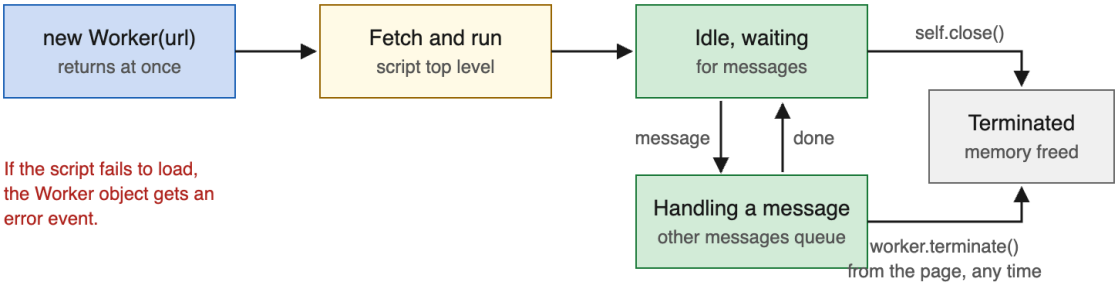


Figure 2.2: A worker's lifecycle. It sits idle between messages, handles one at a time, and stops when either side closes it.

There are two ways to end a worker's life. The page can call `worker.terminate()`, which stops the worker immediately, even in the middle of a calculation. Or the worker can call `self.close()` when it decides it's finished. Either way, its memory goes back to the browser. A worker doesn't shut down just because it's idle, so if you create workers and forget about them, they stay around until the page goes away.

2.3.3 Sending messages

Everything else happens through `postMessage`. On the page side:

```
worker.postMessage({ type: 'primes', limit: 250_000_000 });
worker.onmessage = (event) => console.log(event.data);
```

And on the worker side, the same thing in mirror image, using `self.postMessage` and `self.onmessage`. The data you send arrives as `event.data`.

Nothing here returns a value. `postMessage` is fire-and-forget: a reply may or may not come back later. That's why code that uses workers tends to need some scaffolding, which we build below.

2.3.4 What can go in a message

The browser doesn't send your object across; it makes a copy. The algorithm is called *structured clone*, and it walks your data and rebuilds an equivalent copy on the other side. The copy is independent, so changing it doesn't affect the original.

Structured clone handles most plain data: numbers, strings, booleans, `BigInt`, arrays, plain objects, `Date`, `RegExp`, `Map`, `Set`, typed arrays, `ArrayBuffer`, `Blob`, `File` and `ImageData`. It follows references, so an object that contains itself survives the trip.

It doesn't handle everything. Functions can't be cloned, DOM nodes can't be cloned, and symbols can't be cloned. If you try, `postMessage` throws a `DataCloneError`. Class instances arrive as plain objects, with their fields but not their methods or prototype, so a `Point` class becomes `{ x, y }` and loses `distanceTo()`. The usual fix is to send plain data and rebuild the class on the other side.

For performance, the point is that cloning takes time proportional to the size of the data, and the sender pays it up front. We measure it below.

2.3.5 Transferring instead of copying

For big binary data, the browser offers a way around the copy. You can *transfer* certain objects, which means ownership moves from one thread to the other and no bytes are copied. You list what you want to transfer in the second argument:

```
worker.postMessage({ buffer }, [buffer]);
```

Transferable objects include `ArrayBuffer`, `MessagePort`, `ImageBitmap`, `OffscreenCanvas` and the stream types, among others. Typed arrays aren't transferable themselves, but the `ArrayBuffer` behind them is, so you transfer `array.buffer`.

Once you've transferred something, you no longer have it. The buffer on the sender's side is *detached*: its `byteLength` becomes zero and you can't read or write it. Only one thread owns the memory at a time. Figure 2.3 compares copy and transfer.

A third option, shared memory, lets both threads see the same bytes at once. It's dangerous, and Chapter 6 is devoted to it.

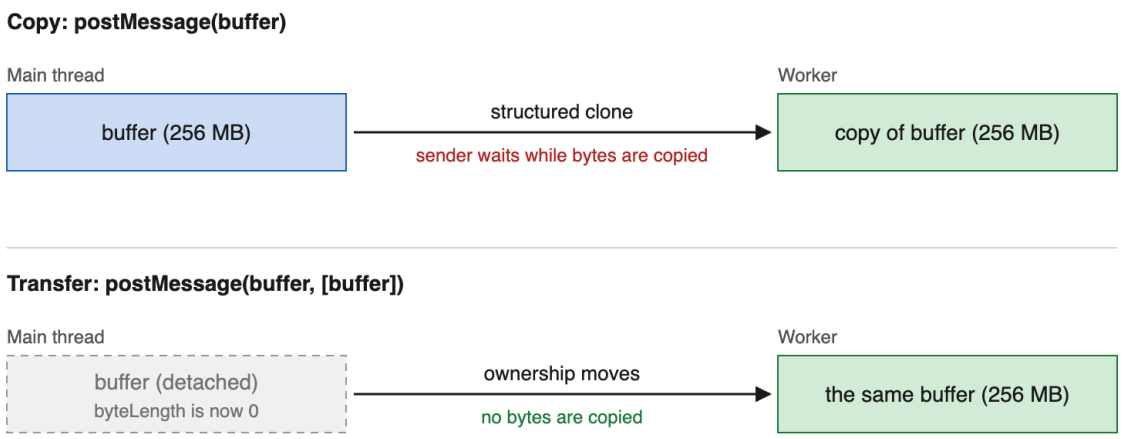


Figure 2.3: Sending a buffer by copy duplicates the bytes and makes the sender wait. Transferring it moves ownership and leaves the sender with an empty, detached buffer.

2.3.6 Making calls feel like calls

Because `postMessage` doesn't return anything, you'll quickly want a way to ask the worker a question and wait for the answer. The standard pattern is to attach an `id` to every request and have the worker echo it back in the reply. The page keeps a promise for each outstanding id and resolves it when the matching reply arrives. Figure 2.4 shows two requests in flight.

The worker in the diagram handles messages in order, one at a time. If request 1 takes two seconds, request 2 waits two seconds, even if it would have taken a millisecond. Keep that in mind when you design what a worker does.

There are libraries that wrap all this up. Comlink, a Google Chrome Labs project, is a popular one: it lets you call a worker's functions as though they were ordinary async functions. To learn the pattern, though, write it once by hand, as we do in the demo.

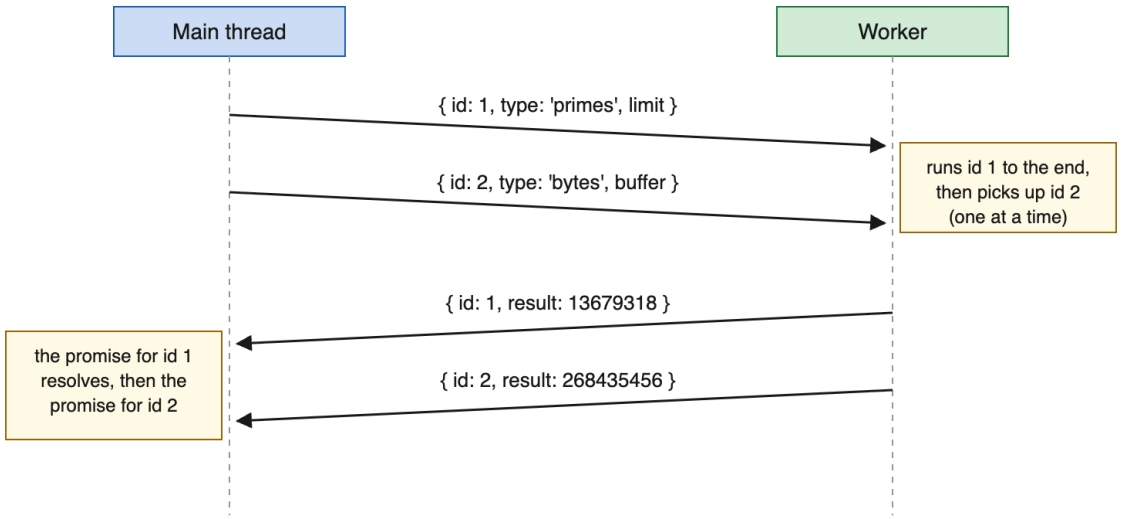


Figure 2.4: Requests and replies carry an id so the page can match each reply to its promise. The worker handles one request at a time.

2.3.7 Errors

If your worker script throws an uncaught exception, the worker object on the page receives an `error` event with the message, file name and line number. If a message can't be deserialized on the receiving end, you get a `messageerror` event instead. And if the worker's script can't be loaded at all, because the path is wrong or the server returns a 404, you get an `error` event with very little detail. That last case is a common source of confusion: when a worker silently does nothing, check the Network tab first.

Inside the worker, rejected promises that nobody handled raise an `unhandledrejection` event on `self`, the same as on the page.

2.4 Where it is used

Workers have been in browsers for well over a decade.

PDF.js, the library behind Firefox's built-in PDF viewer, loads and parses PDF data in a separate worker script. Parsing a large PDF is the sort of heavy, self-contained job workers suit. Monaco, the code editor from VS Code, creates web workers for its language services, as its own FAQ says, "to compute heavy stuff outside of the UI thread." A keystroke shouldn't have to wait on a type checker.

Other jobs that tend to move to a worker:

- **Parsing and transforming large files.** Reading a big CSV, a spreadsheet, a log, or a JSON payload with hundreds of thousands of records. The parser is CPU-bound, and the output is plain data that's easy to send back.
- **Image and audio processing.** Resizing, filtering or compressing before upload. Pixel data is a typed array, so it transfers cleanly. Chapter 5 shows how to draw from a worker as well.
- **Search and indexing.** Building a full-text index or filtering a large in-memory list as the user types, so the input box never stutters.
- **Cryptography and hashing.** Hashing a large file or deriving a key can take long enough to notice.
- **Compression and encoding.** Zipping files in the browser, or encoding data into a binary format.

Don't reach for a worker when the job is small. If the job takes a few milliseconds, the cost of starting a worker and passing messages will exceed what you save. If the job needs the DOM on every step, a worker can't do it. And if the work is slow because it makes a network request, a worker doesn't help, because the main thread was never blocked while waiting for the network. Profile first, and add a worker when you've seen a long task in the performance panel and know what's in it.

2.5 Building the demo

The demo has three parts: a prime-counting job you can run on the main thread or in a worker, a copy-versus-transfer comparison, and a button that sends the worker a message it can't handle.

2.5.1 The job

The job counts the primes up to a limit using the Sieve of Eratosthenes, which fills an array of flags and crosses off multiples. It's pure CPU work with no I/O, and on large inputs it runs long enough to freeze the main thread visibly.

```
export function countPrimes(limit) {
  const sieve = new Uint8Array(limit + 1);
  let count = 0;
  for (let i = 2; i <= limit; i++) {
    if (!sieve[i]) {
      count++;
      for (let j = i * i; j <= limit; j += i) sieve[j] = 1;
    }
```

```
  }
  return count;
}
```

It lives in its own file, `primes.js`, and is exported as a module. The page imports it for the main-thread run and the worker imports it for the worker run, so both runs execute the same code.

The sieve allocates one byte per number, so a limit of 250 million allocates about 250 MB, in whichever thread runs it. That's fine on a laptop, but much higher limits run into the browser's memory limits.

2.5.2 The worker

The worker listens for messages and answers them:

```
import { countPrimes } from './primes.js';

self.onmessage = ({ data }) => {
  const { id, type } = data;
  try {
    if (type === 'primes') {
      self.postMessage({ id, result: countPrimes(data.limit) });
    } else if (type === 'bytes') {
      self.postMessage({ id, result: data.buffer.byteLength });
    } else {
      throw new Error(`unknown message type: ${type}`);
    }
  } catch (err) {
    self.postMessage({ id, error: err.message });
  }
};
```

Every message has an `id` and a `type`. The worker echoes the `id` in whatever it sends back, which is what lets the page match replies to requests. The `try` and `catch` turn any exception into an ordinary message with an `error` field, so the page can reject the right promise instead of seeing a worker-wide failure.

The `bytes` case only reports the length of the buffer it received, so the copy-versus-transfer test times the delivery and not work on the data.

2.5.3 The page

On the page, we create the worker once and write a small `ask` function around `postMessage`:

```
const worker = new Worker('./worker.js', { type: 'module' });
worker.onerror = (e) => log(out, `worker error: ${e.message}`);

let nextId = 0;
function ask(msg, transfer = []) {
  return new Promise((resolve, reject) => {
    const id = ++nextId;
    const onMessage = (e) => {
      if (e.data.id !== id) return;
      worker.removeEventListener('message', onMessage);
      if (e.data.error) reject(new Error(e.data.error));
      else resolve(e.data.result);
    };
    worker.addEventListener('message', onMessage);
    worker.postMessage({ id, ...msg }, transfer);
  });
}
```

This is the pattern from Figure 2.4. Each call picks a fresh `id`, adds a listener that ignores every message except its own reply, and posts the request. When the reply arrives, the listener removes itself and settles the promise.

It works, with one caveat: every outstanding call adds its own listener, and every incoming message wakes all of them up just to check the `id`. For a handful of calls that's nothing. If you were sending thousands of messages a second, you'd keep a single listener and a `Map` from ids to pending promises. The idea is identical and the cost is lower.

With `ask` in place, running the job in the worker reads like normal async code:

```
$('#worker').onclick = async () => {
  log(out, 'worker: start');
  setMetric('where', 'worker', 'good');
  const t = performance.now();
  const n = await ask({ type: 'primes', limit: limit() });
  setMetric('count', n.toLocaleString());
  setMetric('time', Math.round(performance.now() - t) + ' ms');
  log(out, `worker: ${n} primes`);
};
```

The main-thread version calls `countPrimes` directly instead of `ask`. It first waits 50 milliseconds with `setTimeout`, so the browser can paint the "start" message before the freeze begins.

2.5.4 Running it

The spinner at the top of the page is driven by `requestAnimationFrame`, which asks the browser for a callback before each repaint. The frame-counter next to it works out how many callbacks fired per second. When the main thread is healthy, that matches the screen's refresh rate: 60 on many displays, and 120 on the one we used. When it's blocked, the callbacks stop, the spinner stops, and the number drops.

Click **Run on main thread** and the page locks up: the spinner stops and nothing is clickable. After about a second and a half, the page recovers. Figure 2.5 shows it just afterwards.

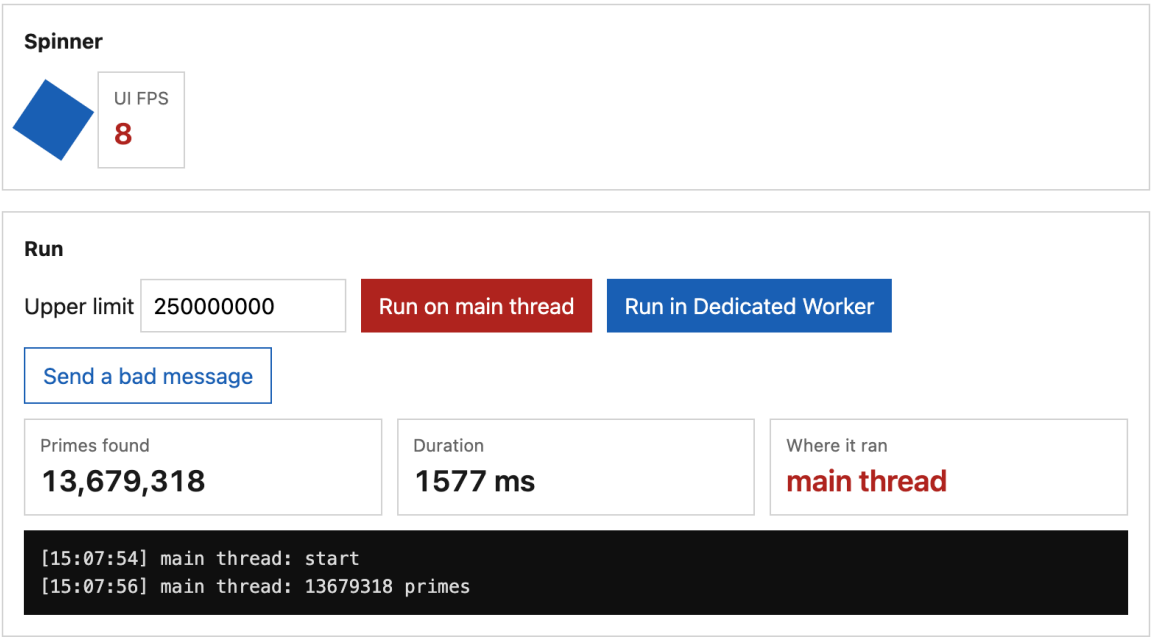


Figure 2.5: Counting primes up to 250 million on the main thread. The job took 1,577 ms, and the frame counter, averaged over the frozen stretch, dropped to 8.

The frame counter reads 8, in red. The rate during the freeze was essentially zero; 8 is the average over a stretch that includes the frozen time and the frames either side of it, since the counter updates twice a second. The 1,577 ms duration is the reliable figure.

Now click **Run in Dedicated Worker**. The same job runs, the spinner keeps turning, and the frame counter holds steady (Figure 2.6).

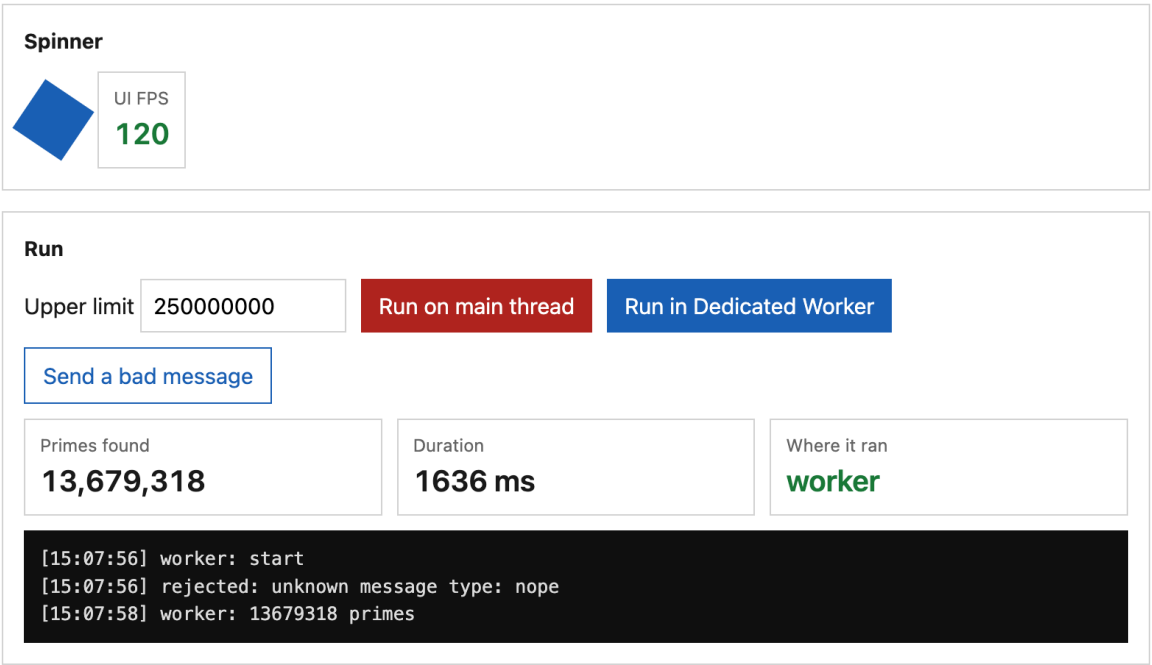


Figure 2.6: The same job in a worker. The duration is about the same, but the page stayed responsive, and the frame counter read 120, the refresh rate of the screen we used. The log also shows the rejected bad message.

On our machine, an Apple-silicon laptop with 11 logical cores, running Chrome 155, the main-thread run took about 1.5 seconds, and the worker runs took between 1.4 and 1.65 seconds. The worker isn't faster, because it's the same code on the same kind of core, and it pays a small messaging overhead. What it bought us is a usable page for those 1.5 seconds.

Moving work to a worker doesn't make it faster. It keeps the main thread free, so the page stays responsive. To make the work itself faster, you have to split it across several workers, and Chapters 6 and 7 show how.

2.5.5 Copy versus transfer

The **Copy vs transfer** card sends a buffer to the worker twice, once by copy and once by transfer, using the `send` function:

```
async function send(transferIt) {
  const buffer = new ArrayBuffer(Number($('#mb').value) * 1048576);
  const t = performance.now();
  const reply = ask({ type: 'bytes', buffer }, transferIt ? [buffer] : []);
  setMetric('posted', (performance.now() - t).toFixed(1) + ' ms');
```

```
  await reply;
  setMetric('round', (performance.now() - t).toFixed(1) + ' ms');
  setMetric('after', buffer.byteLength.toLocaleString() + ' bytes',
            buffer.byteLength ? '' : 'warn');
}
```

The `Promise` constructor runs its function immediately, so by the time `ask` returns, `postMessage` has already finished. The time between `t` and the first `setMetric` is therefore the cost of the call itself, with the sender blocked while the bytes are cloned. Then we wait for the reply and take the round-trip time.

With a 256 MB buffer, the copy is shown in Figure 2.7.

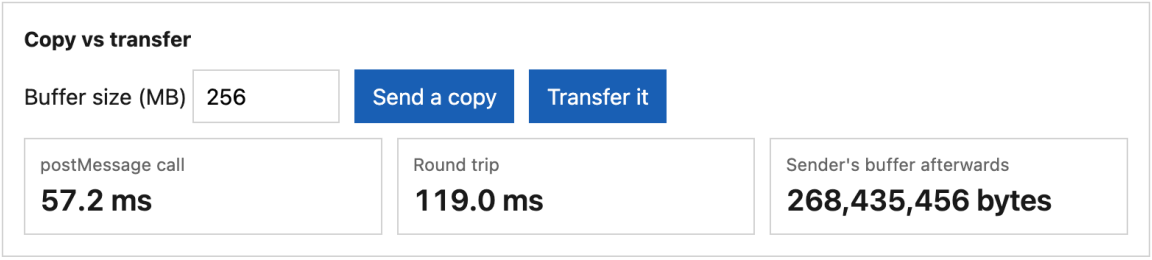


Figure 2.7: Sending 256 MB by copy. The postMessage call blocked the sender for 57 ms, and the sender still has all its bytes afterwards.

and the transfer in Figure 2.8.

Across repeated runs, the copy kept the sender blocked for somewhere between about 50 and 175 milliseconds, with a round trip of roughly 90 to 250. The transfer call took between 0.1 and about a millisecond, and the round trip stayed in the low milliseconds. The last number on the card, the sender's buffer afterwards, shows 268,435,456 bytes after the copy and zero after the transfer.

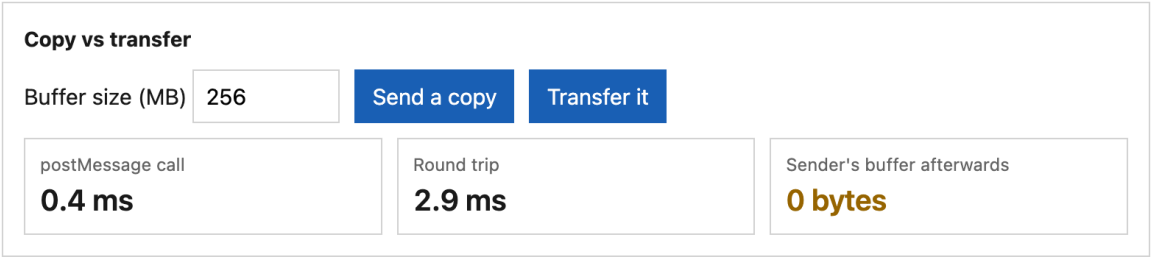


Figure 2.8: Transferring the same 256 MB. The call took 0.4 ms, and the sender's buffer is now empty.

A 100-millisecond block is well over the 50-millisecond mark where a task counts as long. Sending a large buffer by copy, to offload work that was meant to keep the page smooth, can cause the very stall you were trying to avoid. If you can give up the buffer, transfer it.

2.5.6 Sending something the worker can't handle

The third button, **Send a bad message**, posts `{ type: 'nope' }`. The worker throws inside its handler, the `catch` block turns that into an error reply, and `ask` rejects. The log in Figure 2.6 shows the result: `rejected: unknown message type: nope`.

We recommend this error-handling shape. Failures the worker knows about come back as ordinary messages, tied to a specific request. The `onerror` handler on the worker is left for the unexpected: an exception you didn't catch, or a script that failed to load.

2.6 Pitfalls and browser support

A busy worker can't hear you. Because a worker handles one message at a time, a long calculation stops it from seeing *any* other message until it finishes. If you send a "cancel" message while the worker is mid-sieve, the worker won't read it until the sieve is done, which defeats the point. You can call `worker.terminate()`, which kills it on the spot (you'll start a fresh one for the next job), or you can write the long job so it works in small slices, giving the worker's event loop a turn between them so it can read new messages. Chapter 8 shows the slicing idea on the main thread; the same shape works in a worker.

Transferred buffers vanish. If you transfer a buffer and then try to use it, you'll find it empty, and reading it won't throw. Typed arrays on a detached buffer have a length of zero. When code seems to be quietly doing nothing after a `postMessage`, check whether you transferred something you still needed.

Class instances lose their methods. Structured clone sends data, not behavior. If a worker returns an instance of your class, the page gets a plain object. Send plain data and rebuild.

Too many workers. Don't start a worker per task. Each one costs a thread and a heap, and once you have more busy workers than CPU cores, they compete for time. A sensible ceiling is `navigator.hardwareConcurrency`, which reports the number of logical cores (it said 11 on our test machine), and often fewer, because the main thread and the browser need room too. For many small jobs, create a small fixed pool and queue the work, rather than spawning workers on demand.

Forgotten workers. A worker that's no longer needed still holds memory until you call `terminate()`. Anything that creates workers dynamically should clean up after itself.

Debugging. Chrome shows each worker as a separate thread in the Sources panel, and you can set breakpoints in worker code like any other. The Console has a context dropdown that lets you switch into a worker and run code there. If a worker never seems to start, look at the Network tab to confirm its script actually loaded.

Browser support. Dedicated workers work in every browser you're likely to encounter. Module workers, using `type: 'module'`, are supported in all current major browsers, though they arrived later than classic ones, so if you support older versions you'll need to check. The appendix has a table.

2.7 Summary

A dedicated worker is a second thread with its own memory, its own event loop and no access to the DOM. It talks to the page only through messages, which are copied unless you transfer them. Running work in a worker doesn't make it finish sooner; it keeps the page responsive while the work runs. The costs to watch are startup time, memory, and the time it takes to move data across, which can be large for big buffers sent by copy.

The demo covers the whole pattern: a worker that answers requests, an `ask` function that turns messages into promises, and transfer for large buffers. That's enough for most uses of workers and the base for later chapters.

2.8 Check your understanding

1. In our demo, the main-thread run and the worker run took about the same time. So what did the worker actually improve?
2. You send a 200 MB buffer to a worker with a plain `postMessage`, and the page stutters. What's happening, and what's the fix? What do you give up by applying it?
3. Why does the worker in our demo echo an `id` in every reply? What would go wrong without it if two requests were in flight?
4. A long calculation is running in a worker, and the user clicks Cancel. Why might a "cancel" message do nothing, and what are two ways to deal with it?