Asynchrony in Chrome Extensions

Creating a Chrome extension is not trivial, especially with Manifest 3. You've got three components, a content script that runs on the webpage, a popup that provides the UI for your extension, and a service worker that coordinates the extension's activities. Complicating matters is the fact that the Manifest 3 service worker gets shut down after an interval, 5 minutes if it's active, 30 seconds if not. Anything you need the service worker to know when it restarts must be retrieved from storage.

Other complications arise from the different ways asynchrony is handled in async function calls, calls to asynchronous functions that do not return promises, such as the Chrome and file system APIs, and event handlers. You can get surprising results if you're not careful. This discussion includes a proposal for avoiding those surprises.

Async Functions

There are two ways to write async code. The first form looks sequential.

```
let response = await foo();
…
```

The second syntax makes it clear that this is not sequential code.

```
foo().then((response) => {
   …
});
…
```

Any code following the `then` block will execute before the code in the block.

You may need to create a continuation to get things to execute in the desired order with the `then` syntax. For example, if the synchronous version of the code was

```
let result;
if (condition) {
   result = foo();
} else {
   result = “”;
}
…
```

With await, you'd write

```
let result;
if (condition) {
    result = await foo();
} else (
    result = “”;
}
…
```

With `then` syntax you may have to write something like

```
if (condition) {
    foo().then((response) => {
       rest(response);
    });
} else {
   rest(“”);
}
function rest(result) {
    …
}
```

While the await syntax is a lot easier to read and understand, it can trick you into synchronous thinking. The danger comes when returning from the function running this code. It’s easy to forget that you may have to await all calls in the call chain to get things to execute in the order you want.

Chrome API and Friends

The Chrome API is next trickiest because these calls do not return promises.[1] Instead, you provide a callback.

```
chrome.runtime.sendMessage(args, (response) => {
    …
});
…
```

As with async functions, any code following the callback block is executed before the body of that block. You may need to introduce a continuation to force the desired ordering, but here there is no simpler syntax as there is with async functions. It also means you have to be careful because the routine running this code will return before the callback executes, and you don’t have await at your disposal.

---

[1] Strangely, JavaScript allows you to await them, but the await is ignored.

As if that's not enough complexity, dealing with restarts of the service worker creates its own problems. First is the delay; it can take 100 ms or more for the service worker to be ready to process the event that woke it up, a time long enough to produce a noticeable UI lag.

To make matters worse, the message that wakes up the service worker is usually lost.[2] Since you never know when the service worker might be idled, you must be prepared for any service worker event to be lost. One strategy that appears to work reliably is to keep sending wakeup messages until the service worker responds and only then trigger the event you want processed.

```
async function wakeup() {
    await new Promise((resolve) => {
        chrome.runtime.sendMessage("wakeup", async (x)=> {
            if (!x) await wakeup();
            resolve("awake");
        });
    });
}
await wakeup();
chrome.runtime.sendMessage …
```

This example relies on the fact that a lost event produces a runtime error and returns undefined, but you can use a reject block instead.

Experiments show that no more than two wakeup messages are ever needed. The wakeup message response time is about 10 ms or less if the service worker is awake and 100 ms or more if it is idle.

Event Handlers

Event handlers present special problems. They take no parameters and return no values. The only way to provide for inputs and return values is via the DOM or the storage API. It also means there's no way to await them. An example of an event handler that can cause unexpected behavior is

```
element.onpaste = function() {
    setTimeout(() => {
       …
    }, 0);
}
```

---

[2] There is a year's old bug report, but other sources state that this behavior is expected.

You need the set timeout because a paste event fires before the value of the element is updated.[3]

You can get surprising results because the event handler returns before the code inside the setTimeout runs. For example, a change to the contents of the DOM element that happens before the body of the setTimeout runs can be lost. It's also the case that a single action can trigger multiple events. For example, clicking on an input field will trigger a click event and a blur event if another element had the focus.

A Proposal: Wrap in Promises

Dealing with all three forms of asynchrony individually is likely to be a source of bugs. Having a single pattern should help. One approach is to wrap setTimeout and the Chrome API in event handlers in promises.

```
element.onpaste = async function() {
    await new Promise(resolve) => {
        setTimeout(() => {
            …
            resolve();
        }, 0);
    });
});
```

Putting this block in an event handler delays its return until the setTimeout body has completed. Even if a user generates more events before this one finishes, they will be processed to completion in the order they are generated because of the way JavaScript schedules events.

Using this pattern lets you control the order events are processed at the cost of any performance gains you might get from concurrent execution. Experience shows that's not a problem for user interaction. In fact, the only noticeable delay is the time it takes the service worker to wake up, something that is out of your control.

You don't need to wrap all event handlers in promises. Those that don't involve asynchronous operations or only call async functions, *e.g.,*

```
element.onblur = async function {
    await foo();
}
```

will behave as you expect. On the other hand, using `then` syntax

---

[3] The 0 in setTimeout is actually some minimum time, currently 4 ms.

```
element.onblur = function () {
    foo().then(() => {
        …
    });
}
```

might not because the handler will return before the body of the `then` executes.

Testing

A similar trick can be used in UI test code. (There are some wonderful test frameworks, but you might want to roll your own.) In tests you often want to trigger an event and verify the effects are what you expect before triggering the next event.

A straightforward approach is to wrap the event handler body in a promise that the test can await. The problem is that you end up with a lot of test-only code in your event handlers. Another approach is to produce the promise in your test module and add one line in the event handler to resolve it. For example, your test code could be

```
let promise = new Promise((resolve) => {
    resolvers[“blurResolve”] = resolve;
});
element.dispatchEvent(new Event(“blur”);
await promise;
```

Then import `resolvers` and add one line to the event handler.

```
element.onblur = async function() {
    await foo();
    if (resolvers.blurResolve) resolvers.blurResolve();
});
```

Be careful here. Your tests won’t expose bugs in your event handlers if you haven’t constrained their order of execution the way you did in the test. You might think that’s not a problem because people can’t act fast enough to trigger ordering bugs. However, service worker startup or the garbage collector can delay processing enough that ordering bugs get exposed. In one extension, an ordering error would show up roughly once a month in normal use. Tracking down that problem was a lot of fun.