Concurrent Async Iterators

update/discussion

github.com/tc39/proposal-async-iterator-helpers

Some good discussion in the issues!

Recap: concurrency in async iterator helpers

```
x = asyncIteratorOfUrls
  .map(u => fetch(u))

await Promise.all([
  x.next(),
  x.next(),
])
```

Things this gives you:

- Concurrency within a mapper.
- Concurrency between mapper and underlying producer.
- Concurrency is driven by the consumer.

Includes extra method to eagerly buffer

```
asyncIteratorOfUrls
  .map(u => fetch(u))
  .buffered(2)
```

Rust has a similar helper

```rust
async fn collect_resources_n_pages_buffered(n: usize, buf_factor: usize) -> Vec<Resource> {
    stream::iter(0..)
        .map(|i| get_page(i))
        .take(n)
        .buffered(buf_factor)
        .flat_map(|page| stream::iter(page))
        .map(|id| fetch_resource(id))
        .buffered(buf_factor)
        .collect()
        .await
}
```

What's not in the MVP?

Unordered helpers

Forcing these to be order-preserving gives up a fair bit of possible concurrency.

A lot of times the user doesn't need order.

But the space of designs for non-order-preserving transforms is *vast*. So leave it for a followup.

Concurrency for consuming helpers

`.forEach`, `.find`, etc will continue to be sequential.

Concurrent versions of these would be nice but there is no obvious best way to do it.

We *might* be able to add a second "concurrency" parameter later, but that may not be web-compatible.

Racing promises

`.toAsync()` turns a (potentially infinite) iterator of Promises into an async iterator, preserving order.

Another possible transform turns a (finite) iterator of Promises into an async iterator by yielding results in the order in which they settle.

Useful, but not really an async iterator helper, and potentially redundant with unordered helpers.

Merging/racing iterators

`AsyncIterator.race([iter1, iter2])` to merge multiple async iterators by pulling from all of them and yielding results as they come in would be nice, someday.

Splitting iterators

`[a, b] = asyncIter.divide(2)` giving you 2 iterators which pull from the same underlying source lets you do concurrency in an entirely different way, especially when combined with `.race()` from previous slide.

It is one possible solution for the "concurrency when we don't care about order" problem.

Splitting iterators

```
AsyncIterator.merge(
  tasks.split(2).map(worker =>
    worker.map(work).filter(predicate)
  )
).toArray();
```

Splitting iterators

Again, not in the MVP. May or may not make sense to include in a followup.

Limiting concurrency

If your underlying iterator does not support concurrent calls to `.next()`, but you want to do concurrent `.map()` over it, you want to limit concurrency of the underlying thing. This doesn't include anything for that. Notably, `buffered(N)` allows pulling more than `N` times.

Nor does it include a way to limit concurrency of your callbacks, though we should do that (seperately).

Basically: the absolute minimum possible set of things.

What *is* included?

`.map`, `.filter`, `.flatMap`, `.toAsync` supporting concurrent calls to `.next`.

`.buffered(N)` for doing such calls.

All the other helpers from sync `Iterator`, with no* additional affordances for concurrency.

Current work / considerations

Original consistency property was too strong

Originally: "you get the same results in the same order as if you had made the calls sequentially".

Now "... as long as there are no errors".

This allows results of `.map()` to settle earlier.

Original consistency property was too strong

```
let iter = naturals.map(async x => {
  if (x === 0) {
    await sleep(5);
    throw new Error; }
  return x;
});

iter.next(); iter.next();
// 2nd promise shouldn't have to wait for 1st
```

`.filter()` still has to settle in order

```
let slowThenFast =
 [sleep(5).then(x => 1), 2]
   .values().toAsync();

let filtered = slowThenFast.filter(x => x > 0);

filtered.next();
filtered.next(); // cannot resolve before 5s
```

Closing iterators

Calling `helper.return()` does two things:

1. marks the iterator as closed
2. calls `.return()` on the underlying iterator(s).

Being closed means future calls to `.next()` settle immediately with `{done: true}`, matching sync helpers.

But earlier promises are not immediately resolved.

```
let iter = slow.map(fn);

let x = iter.next(); // note lack of awaits

iter.return();
// marks `iter` as closed
// calls slow.return()
// x might still end up w/ { done: false }

let y = iter.next();
// resolves immediately w/o calling slow.next()
```

Closing iterators

The callback throwing is treated the same as `.return()` being called: marks this iterator as closed, and calls `.return()` on the underlying iterator(s).

Closing iterators

Errors from the underlying thing also mark this iterator as closed.

Should `.drop()` drop concurrently?

That is, should it `await` each dropped promise between each of its calls to `underlying.next()`?

My inclination: yes, since that's the most predictable thing and avoids walking past errors. Maybe a boolean parameter to opt in to not `await`ing?

`.buffered` is not eager

Calling `.buffered(N)` doesn't start doing work until first pulled from, but thereafter keeps its buffer full.

We could have an opt-in option to start work as soon as the iterator is constructed; it's hard to do otherwise.

Do vended promises count towards the buffer?

`.buffered(5)` creates an iterator with an internal buffer of 5 promises, which it starts filling up when you first pull from it. Does the one you've pulled count towards that 5 while it's still pending?

```
for await (let x of it.map(fn).buffered(5));
```

Does that call `fn` 5 times concurrently, or 6?

Discussion