# Remove Job from Promise Resolve Functions

Making Promise Adoption Faster

# How many ticks?

- A logs in tick 0.
- B logs in tick 1.
- C logs in tick 2.
- When is D logged?

```
1  // Pretend this increments every tick.
2  let ticks = 0;
3  new Promise((res) => {
4    console.log('A', ticks);
5    res('A');
6
7  }).then(() => {
8    console.log('B', ticks);
9    return 'B';
10
11 }).then(() => {
12   console.log('C', ticks);
13   return Promise.resolve('C');
14
15 }).then(() => {
16   console.log('D', ticks);
17 });
```

# Context

Promise constructor takes an executor function.

Executor receives `resolve` and `reject` functions. I'll refer to `resolve` as the "executor's resolve".

Executor's resolve is a closure that performs "Promise Resolve Functions" steps.

If `inner` is thenable, we *adopt* its state to settle the outer promise.

```
1  const outer = new Promise(executor);
2  function executor(resolve, reject) {
3    // resolve is PromiseResolveFns
4  }
5
6  function PromiseResolveFns(inner) {
7    if (typeof inner !== 'object')
8      return FulfillPromise(inner);
9
10   const then = inner.then;
11   if (typeof then !== 'function)
12     return FulfillPromise(inner);
13
14   NEXT_TICK(() => {
15     then.call(inner, settleOuter);
16   });
17 }
```

# Promise.p.then

Settles a new promise with the result of the `onFul` callback using executor's resolve.

The `onFul` call ****must**** happen in a new tick, according to Promises A+ spec. This guarantees that the `onFul` callbacks don't release Zalgo.

https://blog.izs.me/2013/08/designing-apis-for-asynchrony/

```
1  // Glossing over pending and rejected
2  // states, but they're not important
3  // for this discussion.
4  Promise.p.then = function(onFul) {
5    const C = this[Symbol.species];
6
7    // Pretend this promise is fulfilled
8    // already with a value.
9    return new C(res => {
10     NEXT_TICK(() => {
11       res(onFul([[PromiseResult]]));
12     });
13   });
14 };
15
16
17
```

# So resolving a promise with a promise?

When that `promise.then(...)` fires its callback:

Returning an inner promise immediately resolves the outer with the result (`inner`). We need to adopt `inner`'s state to settle `outer`.

To do that, we wait 1 tick before calling `inner.then(settleOuter)`.

We wait 1 tick before calling `settleOuter([[Res]])`, which fulfills the outer promise.

Now that outer is settled, we wait 1 tick before calling `log([[Res]])`.

It requires 2 ticks to adopt, 1 tick to fire chained thens.

```
1  const outer = promise.then(() => {
2    const inner = Promise.resolve('A');
3    return inner;
4  });
5  outer.then(log);
6  // promise.then(retInner) becomes:
7  new Promise(
8    resOuter => resOuter(inner));
9  // resOuter(inner) becomes:
10 NEXT_TICK(
11   () => inner.then(settleOuter));
12 // inner.then(settleOuter) becomes:
13 NEXT_TICK(
14   () => settleOuter(inner.[[Res]]));
15 // outer.then(log) becomes:
16 NEXT_TICK(
17   () => log(outer.[[Res]]));
```

# How many ticks!?

- A logs in tick 0.
- B logs in tick 1.
- C logs in tick 2.
- D logs in tick 5.

It requires 2 ticks to adopt, 1 tick to fire chained thens.

```
1  // Pretend this increments every tick.
2  let ticks = 0;
3  new Promise((res) => {
4    console.log('A', ticks);
5    res('A');
6
7  }).then(() => {
8    console.log('B', ticks);
9    return 'B';
10
11 }).then(() => {
12   console.log('C', ticks);
13   return Promise.resolve('C');
14
15 }).then(() => {
16   console.log('D', ticks);
17 });
```

# Why does this matter?

Promise adoption is everywhere. Even async functions.

The function body's Completion value is passed to the executor's resolve.

- A logs in tick 1.
- B logs in tick 2.
- C logs in tick 3.
- D logs in tick 2.

It's faster to await a promise then return its value than it is to return the promise directly.

(We fixed await to fast-path native promises in #1250)

```
1  let ticks = 0;
2
3  // Return Direct Primitive
4  (async () => 1)()
5  .then(() => console.log('A', ticks));
6
7  // Return Awaited Primitive
8  (async () => await 1)()
9  .then(() => console.log('B', ticks));
10
11 // Return Direct Promise
12 (async () => Promise.resolve(1))()
13 .then(() => console.log('C', ticks));
14
15 // Return Awaited Promise
16 (async() => await Promise.resolve(1))()
17 .then(() => console.log('D', ticks));
```

# Proposal

Remove the tick before invoking `then.call(...)`. Thenable adoption will take 1 tick instead of 2.

Remember, `then` can't invoke `onFul` immediately unless we want to release Zalgo.

But there's no need to wait before calling the thenable's `then`. In fact, Promises A+ says you're supposed to call `then` immediately.

```
1  function PromiseResolveFns(inner) {
2    if (typeof inner !== 'object')
3      return FulfillPromise(inner);
4
5    const then = inner.then;
6    if (typeof then !== 'function')
7      return FulfillPromise(inner);
8
9    // No need to wait.
10   then.call(inner, settleOuter)
11 }
12
13
14
15
16
17
```

# What'll change?

D logs in tick 4 instead of 5.

Everything else is the same:

- A logs in tick 0.
- B logs in tick 1.
- C logs in tick 2.

Now it takes 1 tick to adopt, 1 tick to fire chained then.

```
1  // Pretend this increments every tick.
2  let ticks = 0;
3  new Promise((res) => {
4    console.log('A', ticks);
5    res('A');
6
7  }).then(() => {
8    console.log('B', ticks);
9    return 'B';
10
11 }).then(() => {
12   console.log('C', ticks);
13   return Promise.resolve('C');
14
15 }).then(() => {
16   console.log('D', ticks);
17 });
```

# What'll change?

C logs in tick 2 instead of 3.

Everything else is the same:

- A logs in tick 1.
- B logs in tick 2.
- D logs in tick 2.

Now it takes 1 tick to adopt, 1 tick to fire chained then.

```
1  let ticks = 0;
2
3  // Return Direct Primitive
4  (async () => 1)()
5  .then(() => console.log('A', ticks));
6
7  // Return Awaited Primitive
8  (async () => await 1)()
9  .then(() => console.log('B', ticks));
10
11 // Return Direct Promise
12 (async () => Promise.resolve(1))()
13 .then(() => console.log('C', ticks));
14
15 // Return Awaited Promise
16 (async() => await Promise.resolve(1))()
17 .then(() => console.log('D', ticks));
```

# What'll change?

- B logs in tick 0 instead of 1.
- C logs in tick 2 instead of 3.

Everything else is the same:

- A logs in tick 0.

Now then is called immediately, it takes 1 tick to adopt, 1 tick to fire chained then.

```
1  let ticks = 0;
2
3  Promise.resolve({
4    get then() {
5      console.log('A', ticks);
6
7      return (res) => {
8        console.log('B', ticks);
9        NEXT_TICK(() => {
10         res('B');
11       });
12     };
13   },
14 }).then(() => {
15   console.log('C', ticks);
16 });
17
```

# Alternative Proposal

Fast path `%Promise.p.then%`

# Fast Path

If the thenable's `then` is `Promise.p.then`, then just call it.

`Promise.p.then` does sync access `val[Symbol.species]`, but otherwise unobservable. Well, besides the adoption taking 1 less tick.

Non-native promise thenables are likely rare at this point?

```
1  function PromiseResolveFns(val) {
2    if (val === [[Promise]])
3      return RejectPromise(new Error);
4    if (typeof val !== 'object')
5      return FulfillPromise(val);
6    const then = val.then;
7    if (typeof then !== 'function')
8      return FulfillPromise(val);
9
10   if (then === %Promise.p.then%) {
11     then.call(val, PromiseResolveFns);
12     return;
13   }
14   NEXT_TICK(() => {
15     then.call(val, PromiseResolveFns);
16   });
17 }
```

# Why did we wait before?

# Promise.resolve

`Promise.resolve` “casts” a value to a promise.

If `value` is not already a promise (has internal slot), then we run the executor’s resolve to create a new promise.

```
1  Promise.resolve = function(value) {
2    const C = this;
3    if (!isFunction(C)) throw new Error;
4
5    if (value.[[PromiseState]]) {
6      const vC = value.constructor;
7      if (vC === c) return value;
8    }
9
10   return new C(res => {
11     res(value);
12   });
13 };
14
15
16
17
```

# Promise.resolve

Apparent design was to safely cast values to a known good promise without running untrusted code during this tick.

But, `.constructor` is sync accessed. And `.then` is not guaranteed.

```
1  const p = new Promise(r => r(1));
2  Object.defineProperty(
3    p,
4    'constructor',
5    {
6      get() {
7        alert('constructor');
8        return Promise;
9      },
10   },
11 );
12 p.then = () => { alert('gotcha') };
13
14
15 Promise.resolve(p).then(() => {});
16 // constructor
17 // gotcha
```

# Promise.resolve

When the object doesn't have a `[[PromiseState]]`, `.then` is sync accessed (inside Promise Resolve Functions).

```
1  const val = {
2    get then() {
3      alert('then');
4      return (res) => {
5        res(1);
6      };
7    }
8  };
9
10
11
12
13
14
15 Promise.resolve(val).then(() => {});
16 // then
17
```

# Discussion?

# Consensus?

# Zalgo

If a callback can be called sync, then it must always be called sync.

If it can be called async, then it must always be called async.

Anything else releases Zalgo.

https://blog.izs.me/2013/08/designing-apis-for-asynchrony/

```
function zalgoTest(promise) {
  let sync = true;
  promise.then(() => {
    console.assert(sync === false);
  }, () => {
    console.assert(sync === false);
  });
  sync = false;
}

// If then's onFul/onRej params _can_
// be called async, they must always
// be called async.
zalgoTest(new Promise(setTimeout));
zalgoTest(Promise.resolve(1));
zalgoTest(Promise.reject(1));

```