Asynchronous programming & mutlithreading in Node.js

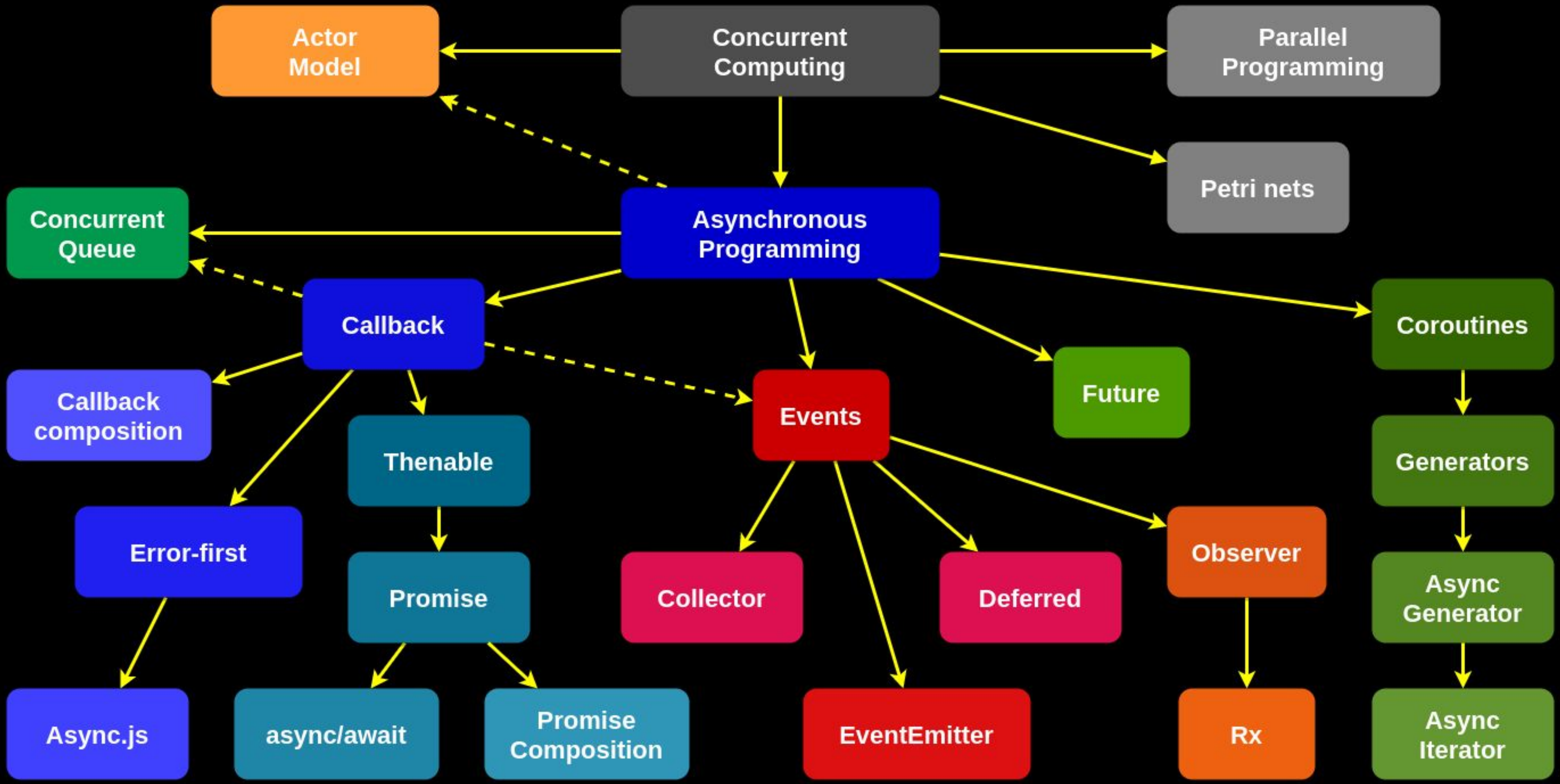
github.com/HowProgrammingWorks

Timur Shemsedinov

Chief Technology Architect at Metarhia
Lecturer at Kiev Polytechnic Institute

github.com/tshemsedinov

Concurrent Computing



Async. prog. in JavaScript as of today

- callbacks: callback-last / error-first contract
- async.js (and other async.js-like utilities)
- promises and async/await
- generators/yield (including async generators)
- observable: event streams like Rx.js
- asynchronous composition
- for await and Symbol.asyncIterator
- multiple different abstractions and primitives

Callbacks

```
// An idea
(callback) => callback(data)

// But we have conventions or contracts
(...args, callback) => callback(err, data)

// Hints:
//   * Use contracts: callback-last / error-first
//   * You can implement callback-hell easely
```

Callback-hell recipe

```
// HOWTO: implement callback-hell
readConfig('myConfig', (err, data) => {
  query('select * from cities', (err, data) => {
    httpGet('http://kpi.ua', (err, data) => {
      readFile('README.md', (err, data) => {
      });
    });
  });
});
// but this is not a problem at the moment...
```

Callbacks: separate named functions

```
const handleQueryResylt = (err, record) => {
  // do something
  httpGet('http://kpi.ua', handleHttpResult);
};

const handleConfig = (err, config) => {
  // do something
  query('select * from cities', handleQueryResylt);
};

// and so on...

readConfig(handleConfig);
```

Callbacks: Error ignoring

```
fn1(arg1, arg2, (err, res1) => {
  fn2(res1, arg3, (err, res2) => {
    fn3(res2, arg4, arg5, (err, res3) => {
      doSomething(arg5, res3);
    });
  });
});
```

Callbacks: Error ignoring

```
const cb3 = (err, res3) => {
  doSomething(arg6, res3);
};

const cb2 = (err, res2) => {
  fn3(res2, arg4, arg5, cb3);
};

const cb1 = (err, res1) => {
  fn2(res1, arg3, cb2);
};

fn1(arg1, arg2, cb1);
```

Library async.js or analogs

```
async.<methodName>(
  [
    // collection of functions
    (data, cb) => cb(err, result)
  ],
  (err, result) => {} // finally handler
);

// Use callback-last, error-first
// Define functions separately, descriptive names
// We need nested calls and hell remains
```

Asynchronous programming with EE

```
const ee = new EventEmitter();
const f1 = () => ee.emit('step2');
const f2 = () => ee.emit('step3');
const f3 = () => ee.emit('done');
ee.on('step1', f1.bind(null, par));
ee.on('step2', f2.bind(null, par));
ee.on('step3', f3.bind(null, par));
ee.on('done', () => console.log('done'));
ee.emit('step1');

// looks terrible :)
```

Thenable & Promises

```
// Contract: Thenable

const thenable = {
  then(onFulfilled[, onRejected]) {}
};


// Contract: Promise

const promise = new Promise()
  .then(onFulfilled[, onRejected])
  .catch(onRejected)
  .finally(onFinally);
```

Promises

```
// Contract

new Promise((resolve, reject) => {
  resolve(data);
  reject(new Error(...));
})
  .then(result => {}, reason => {})
  .catch(err => {});

// Separated control flow for success and fail
// Hell remains for complex parallel/sequential code
```

Promise sequential execution

```
Promise.resolve()
  .then(readConfig.bind(null, 'myConfig'))
  .then(query.bind(null, 'select * from cities'))
  .then(httpGet.bind(null, 'http://kpi.ua'))
  .catch(err => console.log(err.message))
  .then(readFile.bind(null, 'README.md'))
  .catch(err => console.log(err.message))
  .then(data => {
    console.dir({ data });
  });
```

Promise parallel exacution

```
Promise.all([
  readConfig('myConfig'),
  doQuery('select * from cities'),
  httpGet('http://kpi.ua'),
  readFile('README.md')
]).then(data => {
  console.log('Done');
  console.dir({ data });
});
```

Promise mixed: parallel / sequential

```
Promise.resolve()
  .then(readConfig.bind(null, 'myConfig'))
  .then(() => Promise.all([
    query('select * from cities'),
    gttpGet('http://kpi.ua')
  ]))
  .then(readFile.bind(null, 'README.md'))
  .then(data => {
    console.log('Done');
    console.dir({ data });
  });
```

Why do we need Promise.allSettled ?

```
const p1 = Promise.resolve('p1');
const p2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, 'p2');
});
const p3 = new Promise((resolve, reject) => {
  setTimeout(reject, 100, 'p3');
});

Promise.all([p1, p2, p3]).then(values => {
  console.log(values);
});
```

Why do we need Promise.allSettled ?

```
Promise.all([p1, p2, p3]).then(values => {
  console.log(values);
});
```

```
(node:26549) UnhandledPromiseRejectionWarning: p3
(node:26549) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error
originated either by throwing inside of an async function without a catch block, or by
rejecting a promise which was not handled with .catch(). (rejection id: 1)
(node:26549) [DEP0018] DeprecationWarning: Unhandled promise rejections are
deprecated. In the future, promise rejections that are not handled will terminate the
Node.js process with
a non-zero exit code.
```

Why do we need Promise.allSettled ?

```
Promise.all([p1, p2, p3]).then(values => {
  console.log({ values });
}).catch(err => {
  console.log({ err });
});

// Console output:
{ err: 'p3' }
```

Promise.allSettled

```
Promise.allSettled([p1, p2, p3]).then(values => {
  console.log(values);
});

// Console output:
[
  { status: 'fulfilled', value: 'p1' },
  { status: 'fulfilled', value: 'p2' },
  { status: 'rejected', reason: 'p3' }
]
```

async/await

```
// Async function definition:
async function f() {
  return await new Promise(...);
}

// Usage:
f().then(console.log).catch(console.error);

// Promises under the hood, Control-flow separated
// Hell can be implemented, Performance reduced
```

Error ignoring in async/await

```
(async () => {
  const config = await readConfig('myConfig');
  const res = await doQuery('select * from cities');
  const json = await httpGet('http://kpi.ua');
  const file = await readFile('README.md');
  console.dir({ config, res, json, file });
})();
```

Error handling in async/await

```
(async () => {
  let config, res, json, file;
  try {
    config = await readConfig('myConfig');
  } catch (err) {
    // handle err
  }
  try {
    res = await doQuery('select * from cities');
  } catch (err) {
    // handle err and so on...
```

Error handling in async/await

```
try {
  const config = await readConfig('myConfig');
  const res = await doQuery('select * from cities');
  const json = await httpGet('http://kpi.ua');
  const file = await readFile('README.md');
  console.dir({ config, res, json, file });
} catch (err) {
  // handle all err
  // if... if... if...
}
```

Functional object + chaining + composition

```
// npm i do

const c1 = chain()
  .do(readConfig, 'myConfig')
  .do(doQuery, 'select * from cities')
  .do(httpGet, 'http://kpi.ua')
  .do(readFile, 'README.md');

c1();

// We may compose c1 again
```

Functional object + chaining + composition

```javascript
function chain(prev = null) {
  const cur = () => {
    if (cur.prev) {
      cur.prev.next = cur;
      cur.prev();
    } else {
      cur.forward();
    }
  };
  cur.prev = prev;
  cur.fn = null;
  cur.args = null;
  cur.do = (fn, ...args) => {
    cur.fn = fn;
    cur.args = args;
    return chain(cur);
  };
  cur.forward = () => {
    if (!cur.fn) return;
    cur.fn(cur.args, () => {
      if (cur.next) cur.next.forward();
    });
  }
  return cur;
}
```

Catch unhandled errors

```
process.on('uncaughtException', err => {
  console.log({ uncaughtException: err });
  process.exit(1);
});

process.on('multipleResolves', (type, p, reason) => {
  console.log({ type, promise: p, reason });
});
```

Catch unhandled errors

```
process.on('unhandledRejection', (err, promise) => {
  console.log({ err, promise });
});

process.on('rejectionHandled', promise => {
  console.log({ promise });
});
```

Problems

All primitives and syntaxes are not universal (callbacks, async.js, Promise, async/await, do, ...)

- Nesting and syntax
- Different contracts
- Not cancellable, no timeouts
- Complexity and Performance
- Ignoring errors and complex error handling

Asynchronous Tricks & adapters

Callback with timeout

```
// Callback function
const callback = (err, data) => {
  console.log({ err, data });
};

// Wrap to 1s timeout
const callback1s = timeout(1000, callback);

// Pass as callback
asyncFunctionWithCallback(...args, callback1s);
```

Callback with timeout

```
const timeout = (msec, f) => {
  let timer = setTimeout(() => {
    if (timer) console.log('Function timed out');
    timer = null;
  }, msec);
  return (...args) => {
    if (!timer) return;
    clearTimeout(timer);
    timer = null;
    return f(...args);
  };
};
```

Make function cancelable

```
const callback = (err, data) => {
  console.log({ err, data });
};

const cc = cancelable(callback);

doSomethisn(...args, () => {
  cc.cancel(); // Cancel from different place
});

asyncFunctionWithCallback(...args, cc);
```

Make function cancelable

```
const cancelable = fn => {
  const wrapper = (...args) => {
    if (fn) return fn(...args);
  };
  wrapper.cancel = () => {
    fn = null;
  };
  return wrapper;
};
```

Cancelable Promise

```
const cancelable = promise => {
  let cancelled = false;
  return {
    promise: promise.then(val => {
      if (!cancelled) return val;
      return Promise.reject(new Error('Canceled'));
    }),
    cancel: () => {
      cancelled = true;
    }
  };
};
```

Cancelable Promise

```
// Usage

const { cancel, promise } = cancelable(
  new Promise(resolve => {
    setTimeout(() => { resolve('first'); }, 10);
  })
);

// You can call cancel() from different place...

promise.then(console.log).catch(console.log);
```

More wrappers

```
const f1 = timeout(1000, fn);
const f2 = cancelable(fn);
const f3 = once(fn);
const f4 = limit(10, fn);
const f5 = throttle(10, 1000, fn);
const f6 = debounce(1000, fn);
const f7 = utils(fn)
  .limit(10)
  .throttle(10, 100)
  .timeout(1000);
```

Promisify and Callbackify

```
// callback-last to Promise-returning
const promiseReturning = promisify(callbackLast);
promiseReturning(...args).then(...).catch(...);

// Promise-returning to callback-last
const callbackLast = callbackify(promiseReturning);
callbackLast(...args, (err, value) => {});

// Supported in Node.js
const { promisify, callbackify } = require('util');
```

Promisify

```
const promisify = fn => (...args) => (
  new Promise((resolve, reject) => (
    fn(...args, (err, data) => (
      err ? reject(err) : resolve(data)
    ))
  ))
);
```

Callbackify

```
const callbackify = fn => (...args) => {
  const callback = args.pop();
  fn(...args)
    .then(value => {
      callback(null, value);
    })
    .catch(reason => {
      callback(reason);
    });
};
```

Function composition

```
inc = a => ++a;
square = a => a * a;
lg = x => log(10, x);

f = compose(inc, square, lg);
```

...but it’s synchronous

Function composition

Function composition is a great idea for asynchronous I/O but there are questions:

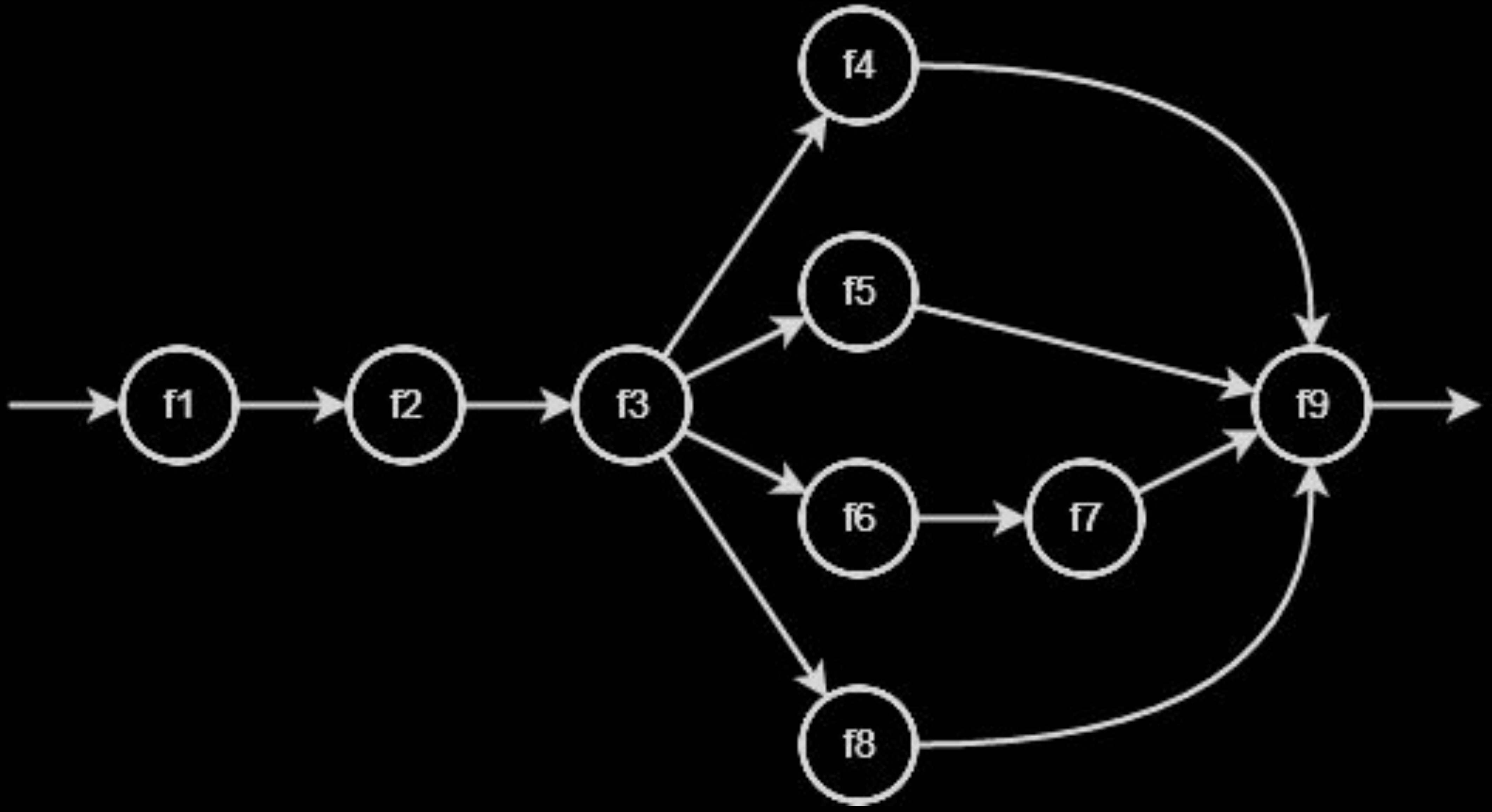
- What about contracts?
  - for calls and callbacks, arguments and errors
  - timeouts, queueing, throttling
- How to add asynchronicity?
  - parallel and sequential

Asynchronous function composition

```
// Just to show contracts
const readCfg = (name, cb) => fs.readFile(name, cb);
const netReq = (data, cb) => http.get(data.url, cb);
const dbReq = (query, cb) => db.select(query, cb);

// We need two types of composition
const f1 = sequential(readCfg, netReq, dbReq);
const f2 = parallel(dbReq1, dbReq2, dbReq3);
// f1 & f2 contracts (...args, cb) => cb(err, data)
```

Asynchronous function composition

```
// npm i metasync
```



```
const fx = metasync(
  [f1, f2, f3, [[f4, f5, [f6, f7], f8]], f9]
);
```

Data collector

```
// https://github.com/metarhia/metasync
// npm i metasync

const dc1 = metasync
  .collect(3)
  .timeout(5000)
  .done((err, data) => {});

dc1(item);
```

Key collector

```
// https://github.com/metarhia/metasync
// npm i metasync

const dc2 = metasync
  .collect(['key1', 'key2', 'key3'])
  .timeout(5000)
  .done((err, data) => {});

dc2(key, value);
```

Universal collector from do library

```
// npm i do

const collect = require('do');
const fs = require('fs');
const dc = collect.do(6);

dc('user', null, { name: 'Marcus Aurelius' });

fs.readFile(
  'HISTORY.md',
  (err, data) => dc.collect('history', err, data)
);
```

Universal collector from do library

```
fs.readFile('README.md', dc.callback('readme'));

fs.readFile('README.md', dc('readme'));

dc.take('readme', fs.readFile, 'README.md');

setTimeout(
  () => dc.pick('timer', { date: new Date() }),
  1000
);

// https://github.com/metarhia/do
```

Concurrent Queue

```
const queue = metasync.queue(3)
   .wait(2000)
   .timeout(5000)
   .throttle(100, 1000)
   .process((item, cb) => cb(err, result))
   .success(item => {})
   .failure(item => {})
   .done(() => {})
   .drain(() => {});
```

Loop: for await of is blocking

```
(async () => {
  let ticks = 0;
  const timer = setInterval(() => ticks++, 10);
  const numbers = new Array(1000000).fill(1);
  let i = 0;
  for await (const number of numbers) i++;
  clearInterval(timer);
  console.dir({ i, ticks });
})();

// { i: 1000000, ticks: 0 }
```

AsyncArray (short version)

```
class AsyncArray extends Array {
  [Symbol.asyncIterator]() {
    let i = 0;
    return {
      next: () => new Promise(resolve => {
        setTimeout(() => resolve({
          value: this[i], done: i++ === this.length
        }), 0);
      })
    };
  }
}  // github.com/HowProgrammingWorks/NonBlocking
```

Loop: for await of + AsyncArray

```
(async () => {
  let ticks = 0;
  const timer = setInterval(() => ticks++, 10);
  const numbers = new AsyncArray(1000000).fill(1);
  let i = 0;
  for await (const number of numbers) i++;
  clearInterval(timer);
  console.dir({ i, ticks });
})();

// { i: 1000000, ticks: 1163 }
// https://github.com/HowProgrammingWorks/NonBlocking
```

Multi-core support

child_process and worker_threads

How to use workers_threads

Node.js: The Road to Workers
Anna Henningsen
https://youtu.be/pO5a10YPQG4

A Crash Course on Worker Threads
Rich Trott
https://youtu.be/GRb-XQ5JRA8

We are ready for Parallel programming

- Stable worker_threads and messaging API
  https://nodejs.org/api/worker_threads.html
- Atomics for Compare-and-Swap operations
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Atomics
- SharedArrayBuffer to share memory
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer

Concurrency Problems

- Race conditions
- Deadlock
- Livelock
- Resource starvation
- Resource leaks

and other interesting thing from parallel world...

Synchronization Primitives

Semaphore

Binary semaphore

Counting semaphore

Condition variable

Spinlock

Mutex

Timed mutex

Shared mutex

Recursive mutex

Monitor

Barrier

Why do we need Web Locks API ?

- Do you know what is
  mutex, locks, critical section, race condition,
  parallel programming at all?

- Congrats!
  It’s is very likely that
  all your JavaScript code broken)))

How race conditions in single threaded JavaScript are possible?

Everybody knows...

javascruptissinglethreaded

Everybody knows...

nodejsissinglethreaded

Everybody knows...

Promises
async/await

Race Condition

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  async move(dx, dy) {
    this.x = await add(this.x, dx);
    this.y = await add(this.y, dy);
  }
}
```

Race Condition

```
const random = (min, max) => Math
  .floor(Math.random() * (max - min + 1)) + min;

const add = (x, dx) => new Promise(resolve => {
  setTimeout(() => {
    resolve(x + dx);
  }, random(20, 100));
});
```

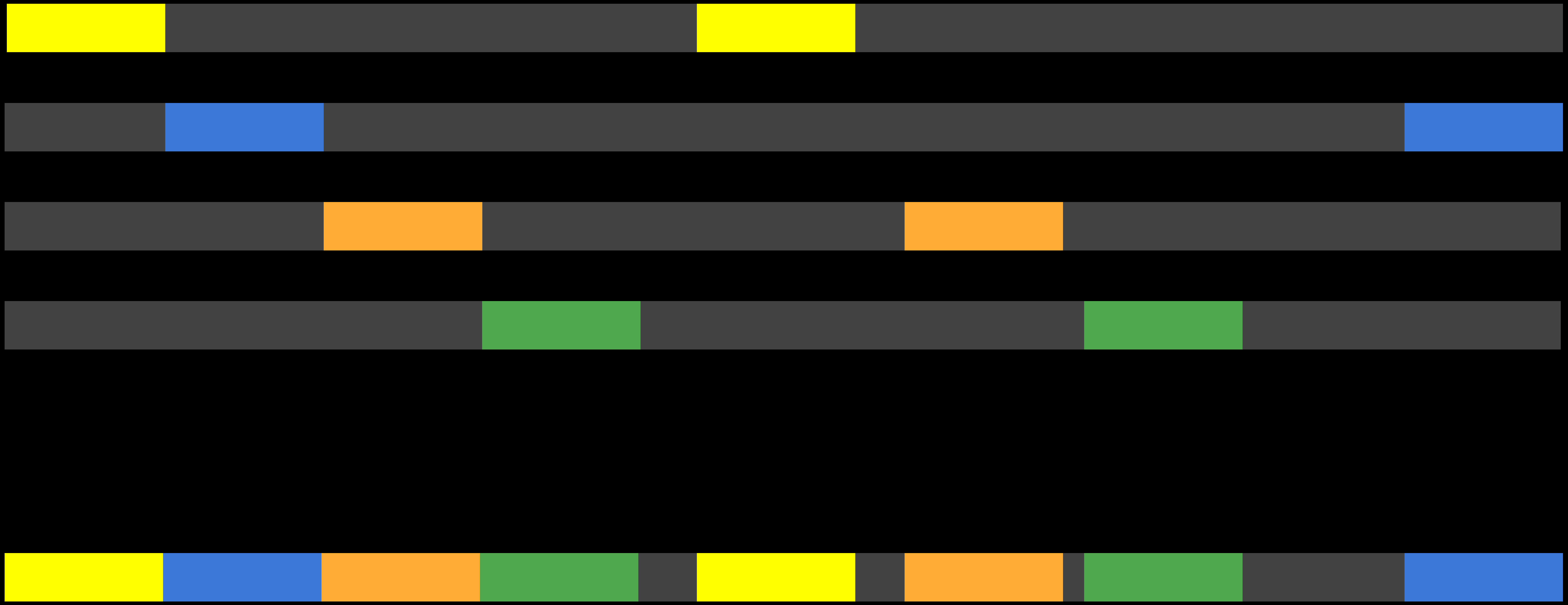
Race Condition

```
const p1 = new Point(10, 10);
console.log(p1);

p1.move(5, 5);
p1.move(6, 6);
p1.move(7, 7);
p1.move(8, 8);

setTimeout(() => {
  console.log(p1);
}, 1000);
```

Race Condition

```
Initial
Point { x: 10, y: 10 }

Expected
Point { x: 36, y: 36 }

Actual
Point { x: 18, y: 25 }
```

Race Condition



Possible Solutions

- Synchronization
- Resource locking
- Special control flow organization
- Queuing theory
- Actor model
- Use DBMS transactions

Semaphore

```
class Semaphore {
  constructor()
  enter(callback)
  leave()
}
semaphore.enter(() => {
  // do something
  semaphore.leave();
});
```

```
class Semaphore {
  constructor()
  async enter()
  leave()
}
await semaphore.enter();
// do something
semaphore.leave();
```

github.com/HowProgrammingWorks/Semaphore

Mutex

```
class Mutex {
  constructor()
  async enter()
  leave()
}

await mutex.enter();
// do something with shared resources
mutex.leave();
```

https://github.com/HowProgrammingWorks/Mutex

Resource Locking

```
class Lock {
  constructor() {
    this.active = false;
    this.queue = [];
  }

  leave() {
    if (!this.active) return;
    this.active = false;
    const next = this.queue.pop();
    if (next) next();
  }
}
```

```
enter() {
  return new Promise(resolve => {
    const start = () => {
      this.active = true;
      resolve();
    };
    if (!this.active) {
      start();
      return;
    }
    this.queue.push(start);
  });
}
```

Resource Locking

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
    this.lock = new Lock();
  }

  async move(dx, dy) {
    await this.lock.enter();
    this.x = await add(this.x, dx);
    this.y = await add(this.y, dy);
    this.lock.leave();
  }
}
```

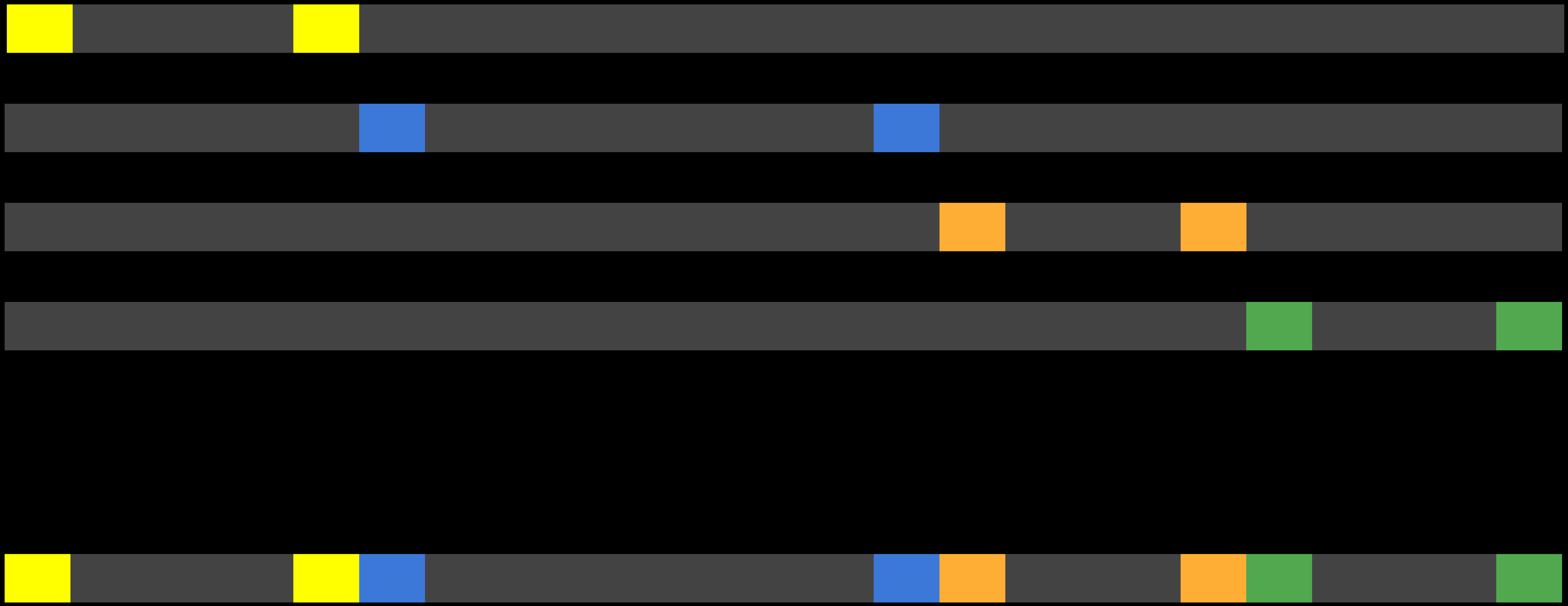
Resource Locking

```
const p1 = new Point(10, 10);
console.log(p1);

p1.move(5, 5);
p1.move(6, 6);
p1.move(7, 7);
p1.move(8, 8);

setTimeout(() => {
  console.log(p1);
}, 1000);
```

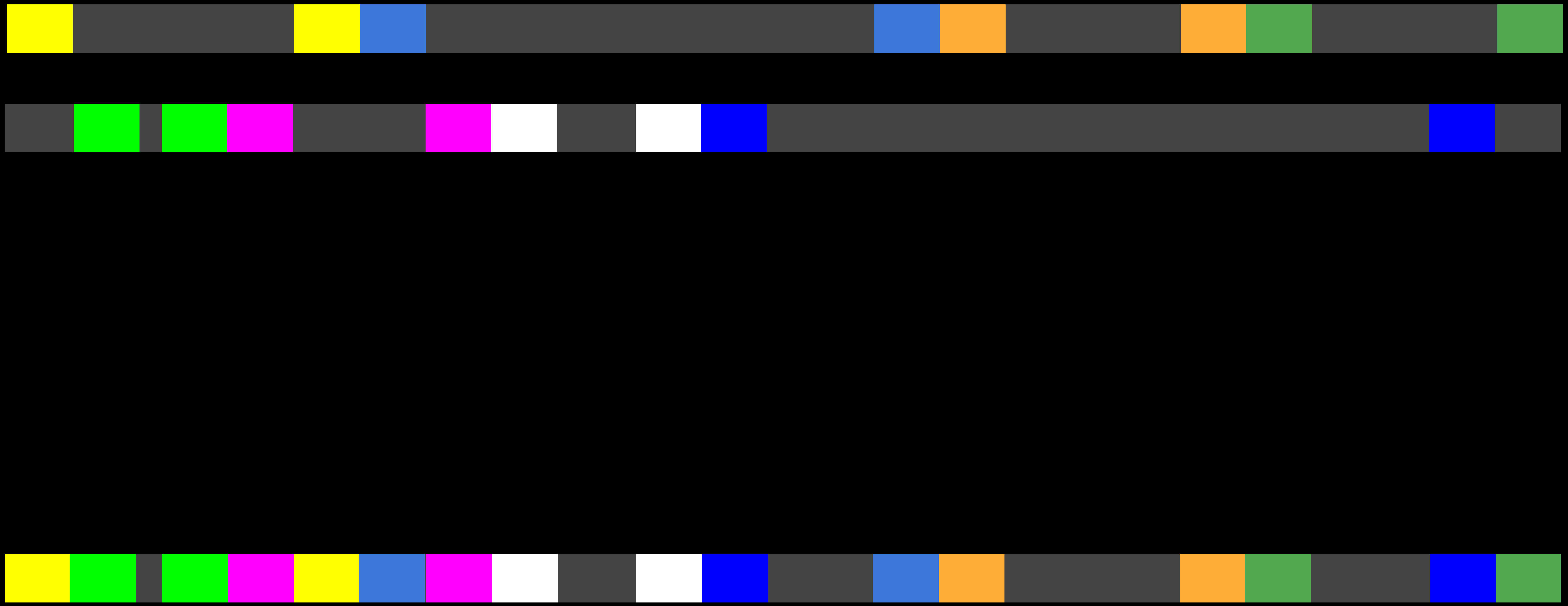
Resource Locking

```
Initial
Point { x: 10, y: 10 }

Expected
Point { x: 36, y: 36 }

Actual
Point { x: 36, y: 36 }
```

Resource Locking



Asynchronous Res Locks



Real-life Example

Warehouse API

- Check balances
- Ship goods
- Lock balances

github.com/HowProgrammingWorks/RaceCondition

Web Locks API

```
locks.request('resource', opt, async lock => {
  if (lock) {
    // critical section for `resource`
    // will be released after return
  }
});
```

https://wicg.github.io/web-locks/

Web Locks: await

```
(async () => {
  await something();
  await locks.request('resource', async lock => {
    // critical section for `resource`
  });
  await somethingElse();
})();
```

Web Locks: Promise

```
locks.request('resource', lock => new Promise(
  (resolve, reject) => {
    // you can store or pass
    // resolve and reject here
  }
));
```

Web Locks: Thenable

```
locks.request('resource', lock => ({
  then((resolve, reject) => {
    // critical section for `resource`
    // you can call resolve and reject here
  })
}));
```

Web Locks: Abort

```
const controller = new AbortController();
setTimeout(() => controller.abort(), 200);

const { signal } = controller;

locks.request('resource', { signal }, async lock => {
  // lock is held
}).catch(err => {
  // err is AbortError
});
```

Web Locks for Node.js

github.com/nodejs/node/issues/22702

Open

github.com/nodejs/node/pull/22719

Closed

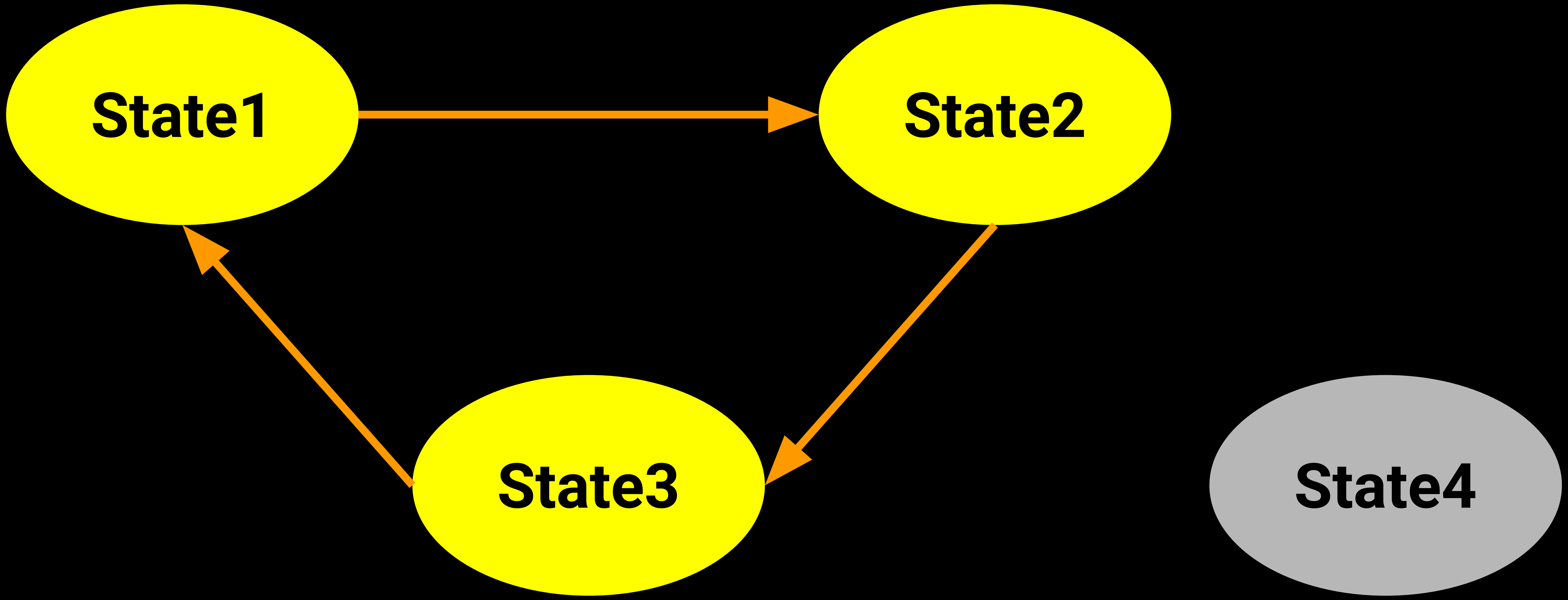
Safe data structures

- Low-level structures
  e.g. Register, Counter, Buffer, Array, Lists, etc.
- Abstract structures
  e.g. Queue, Graph, Polyline, etc.
- Subject-domain classes
  e.g. Sensors, Payment, Biometric data, etc.
- Resources and handles
  e.g. Sockets, Connections, Streams, etc.

Deadlock

```
(async () => {
  await locks.request('A', async lock => {
    await locks.request('B', async lock => {
    });
  });
})(); (async () => {
  await locks.request('B', async lock => {
    await locks.request('A', async lock => {
    });
  });
})();
```

Livelock



Alternative Solutions

- Thread safe data structures
- Lock-free data structures
- Wait-free algorithms
- Conflict-free data structures

Links

github.com/HowProgrammingWorks/RaceCondition
github.com/HowProgrammingWorks/Semaphore
github.com/HowProgrammingWorks/Mutex

github.com/metarhia/web-locks
wicg.github.io/web-locks

Node.js Starter Kit

25 kb core,
minimum dependencies:
pg (1.2 mb) and ws (0.24 mb)

Starter Kit Purpose

- Demonstrate modern node.js features (v14.x)
- Optimize for readability and understanding
- Minimum code size and dependencies
- Give structure and architecture examples
- Show patterns and code cohesion
- Not for production use

Starter Kit Feature List

- Serve API with auto-routing, HTTP(S), WS(S)
- Server code live reload with file system watch
- Graceful shutdown and application reload
- Code isolation, sandboxing and security
- Implemented dependency injection
- Layered architecture: core, domain, API, client

Starter Kit Feature List

- Multi-threading for CPU utilization
- Serve multiple ports in threads
- Serve static files with memory cache
- Request queue with timeout and size
- Execution timeout and error handling

Starter Kit Feature List

- Application configuration
- Simple logger and to terminal and file
- Database access layer (Postgresql)
- Persistent sessions (stored in DB)
- Unit-tests and API tests example

github.com
/HowProgrammingWorks
/NodejsStarterKit

Contacts

github.com/tshemsedinov
youtube.com/TimurShemsedinov
github.com/HowProgrammingWorks
patreon.com/tshemsedinov
t.me/HowProgrammingWorks
t.me/NodeUA

timur.shemsedinov@gmail.com