

# Event Loop & EventEmitter

simen

[simenkid@gmail.com](mailto:simenkid@gmail.com)

[github.com/simenkid](https://github.com/simenkid)

# Materials

- Demo Code

- <https://github.com/simenkid/talks>

## Materials

- [20161025\\_event\\_loop.zip](https://github.com/simenkid/talks/blob/master/20161025_event_loop.zip) Node.js 線上讀書會分享 Event Loop 與 EventEmitter, 內含 demo examples

- Read More...

- <https://simeneer.blogspot.tw/2016/09/nodejs-eventemitter.html>

## 非同步程式碼之霧：Node.js 的事件迴圈與 EventEmitter

By Simen | 下午9:12 | 軟體, Node | [Edit](#)

身為一個 Node.js 工程師，怎麼可以不夠了解「非同步程式碼」的行為？我希望能綜合自己的一點心得與經驗，寫一篇探討 Node.js Event Loop 與 Event Pattern 的文章，而且還不能只是泛泛之談，必須稍微有點深度，然後還期待大家能夠很容易地讀懂。

這篇文章是我為這個想法所作的努力，它花了我好幾個晚上，寫了將近 20 個小時左右（天吶～～）。雖然極力想要用更短的篇幅把一切說明清楚，卻發現這實在沒辦法用短短的幾句話就講完。然而，即便寫得夠多了，但難免還是有疏漏之處，也要請大家有發現錯誤之處，踴躍提出糾正！讓這篇文章能夠呈現最正確的內容！

# Is Node.js Single Threaded?

“... Actually **only your ‘userland’ code runs in one thread.**  
... Node.js in fact spins up a number of threads.”

- D. Khan, “How to track down CPU issues in Node.js”

- Let's run a simple http server

```
var http = require('http');
```

01\_server.js

```
var server = http.createServer(function (req, res) {  
  // Request Handler  
});
```

```
server.listen(3000);
```

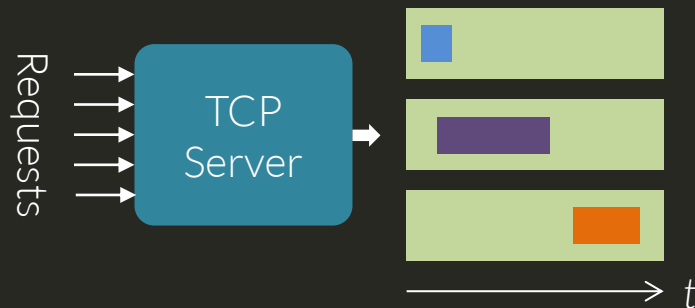
```
simen@ubuntu:~$ pstree -ap | grep node  
| | | | | `--node,5653 test.js  
| | | | |   |--{node},5654  
| | | | |   |--{node},5655  
| | | | |   |--{node},5656  
| | | | |   |--{node},5657  
| | | | |   `--{node},5658
```

# Concurrency

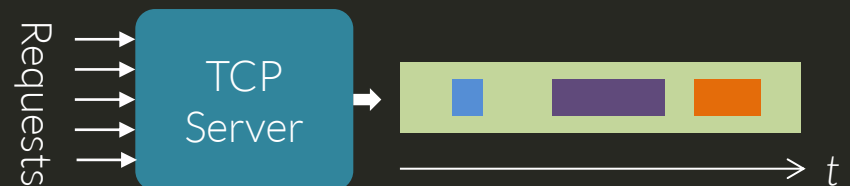
*“Concurrency is a way to structure a thing so that you can, maybe, use parallelism to do a better job. But parallelism is not the goal of concurrency; **concurrency's goal is a good structure.**”*

- R. Pike, Golang co-inventor

Tasks spread over *threads*



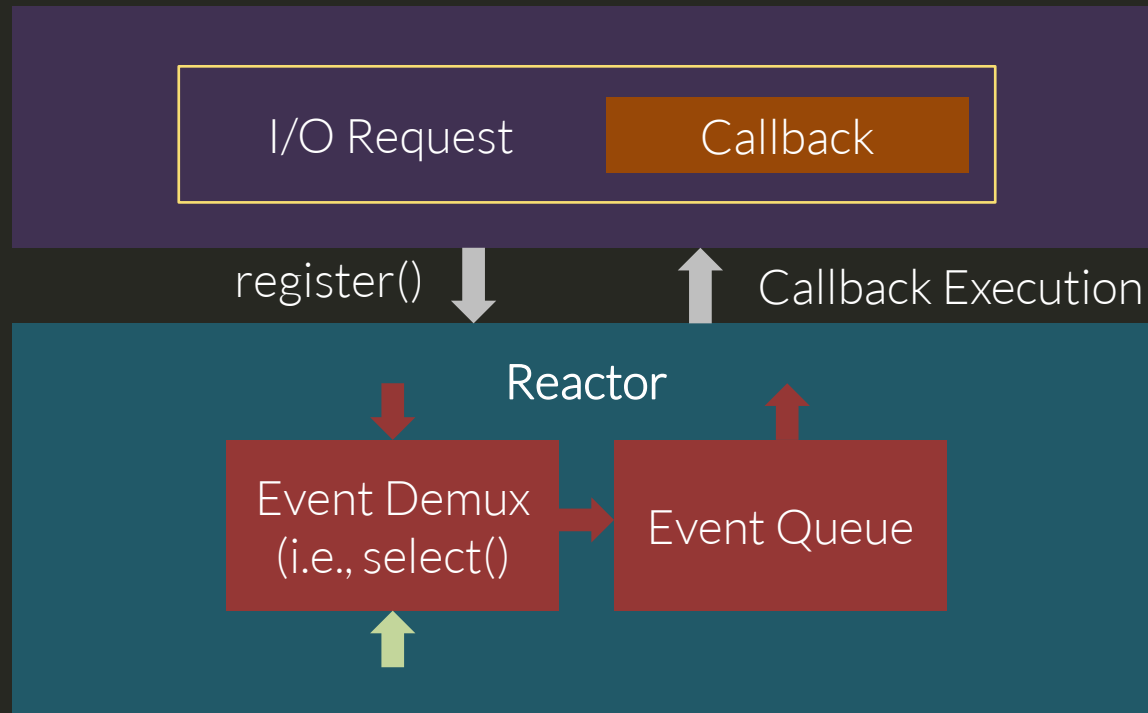
Tasks spread over *time*



# Blocking and Non-Blocking I/O

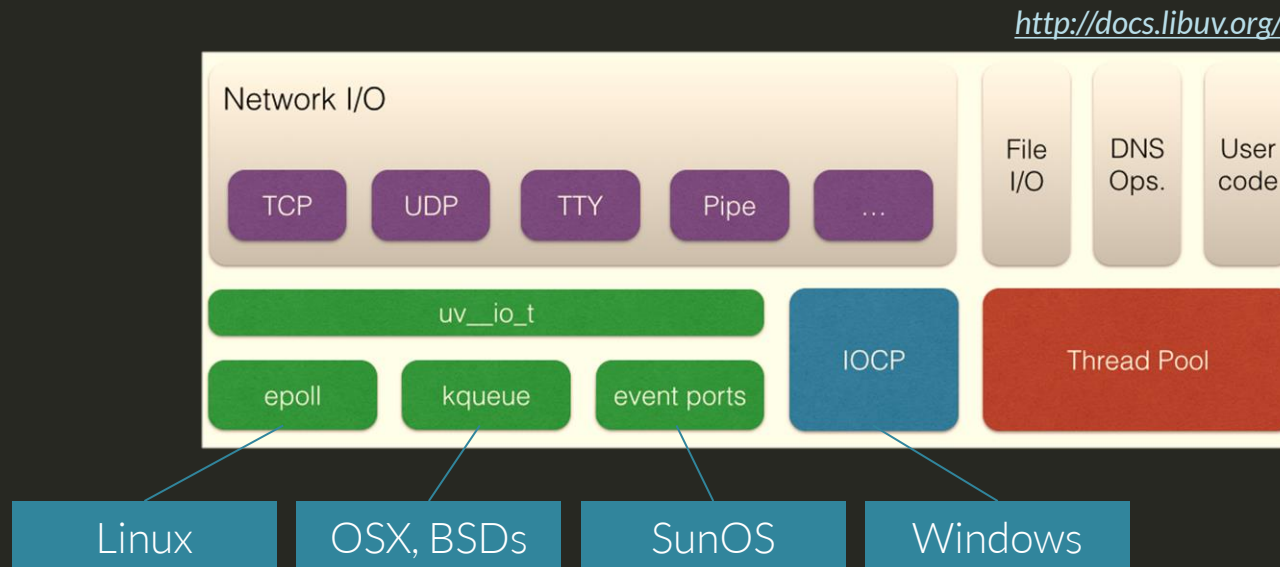
- Simplified concepts of
  - Blocking I/O
  - Non-blocking I/O
- I/O multiplexing
  - Reactor
  - `select()` or `poll()`
  - Event notification, synchronous event demux

# Reactor Pattern

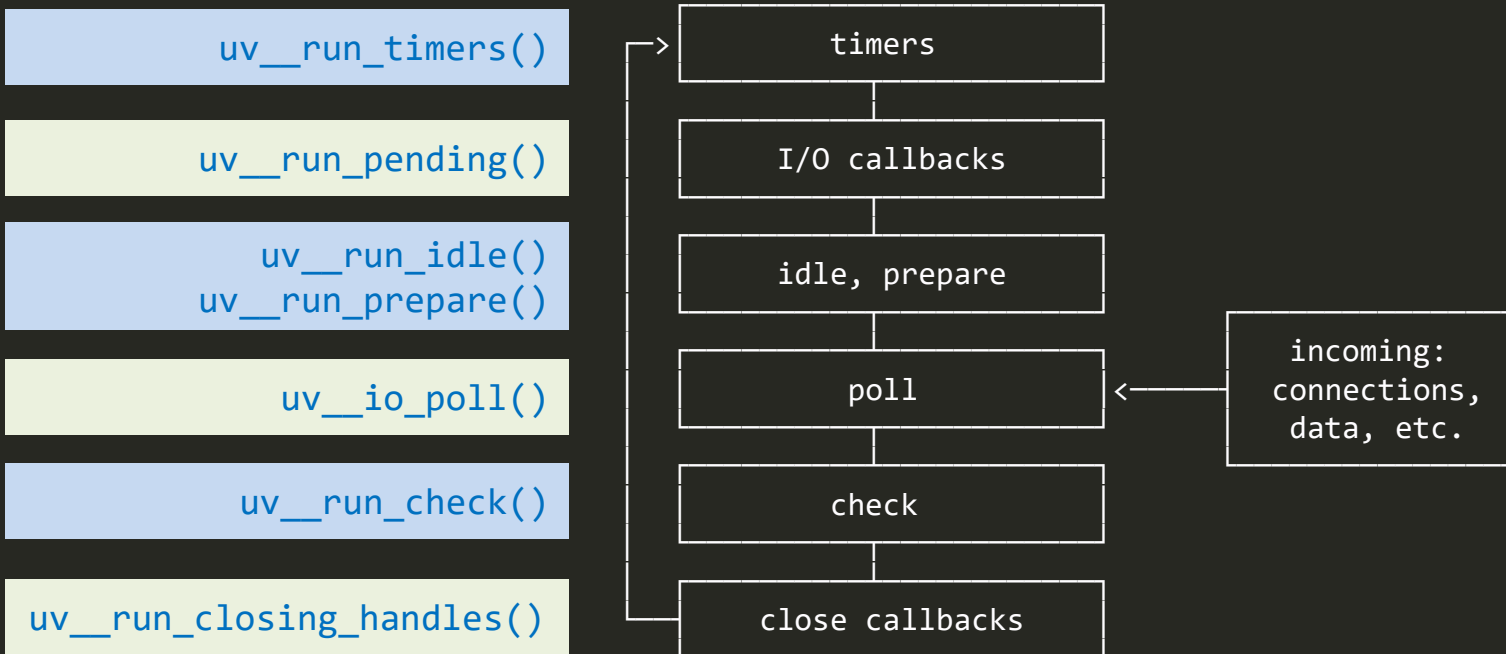


# Non-Blocking I/O Engine

- libuv
  - Node.js, Luvit, ...
  - multi-platform support library with a focus on asynchronous I/O



# Phases In the Loop





# libuv core.c

```
int uv_run(uv_loop_t* loop, uv_run_mode mode) { core.c
    // ...
    r = uv__loop_alive(loop);
    // ...

    while (r != 0 && loop->stop_flag == 0) {
        uv__update_time(loop);
        uv__run_timers(loop);
        ran_pending = uv__run_pending(loop);
        uv__run_idle(loop);
        uv__run_prepare(loop);

        timeout = 0;
        if ((mode == UV_RUN_ONCE && !ran_pending) || mode == UV_RUN_DEFAULT)
            timeout = uv_backend_timeout(loop);

        uv__io_poll(loop, timeout);
        uv__run_check(loop);
        uv__run_closing_handles(loop);

        if (mode == UV_RUN_ONCE) {
            uv__update_time(loop);
            uv__run_timers(loop);
        }

        r = uv__loop_alive(loop);
        // ...
    }
}
```

<https://github.com/libuv/libuv/blob/v1.x/src/unix/core.c#L332-L372>


# Event Loop in Node.js (I)

node.cc

```
// Entry point for new node instances, ...
static void StartNodeInstance(void* arg) {
    // ...
    {
        SealHandleScope seal(isolate);
        bool more;
        do {
            v8::platform::PumpMessageLoop(default_platform, isolate);
            more = uv_run(env->event_loop(), UV_RUN_ONCE);

            if (more == false) {
                v8::platform::PumpMessageLoop(default_platform, isolate);
                EmitBeforeExit(env);

                // Emit `beforeExit` if the loop became alive either after emitting
                // event, or after running some callbacks.
                more = uv_loop_alive(env->event_loop());
                if (uv_run(env->event_loop(), UV_RUN_NOWAIT) != 0)
                    more = true;
            }
        } while (more == true);
    }
    // ...
}
```



# Event Loop in Node.js (II)

node.cc

```
Environment* CreateEnvironment(
    Isolate* isolate,
    uv_loop_t* loop,      Each node instance has its own event loop.
    // ...
    const char* const* exec_argv)
{
    // ...

    uv_check_init(env->event_loop(), env->immediate_check_handle());
    uv_unref(
        reinterpret_cast<uv_handle_t*>(env->immediate_check_handle()));

    uv_idle_init(env->event_loop(), env->immediate_idle_handle());
    uv_prepare_init(env->event_loop(), env->idle_prepare_handle());
    uv_check_init(env->event_loop(), env->idle_check_handle());
    uv_unref(reinterpret_cast<uv_handle_t*>(env->idle_prepare_handle()));
    uv_unref(reinterpret_cast<uv_handle_t*>(env->idle_check_handle()));

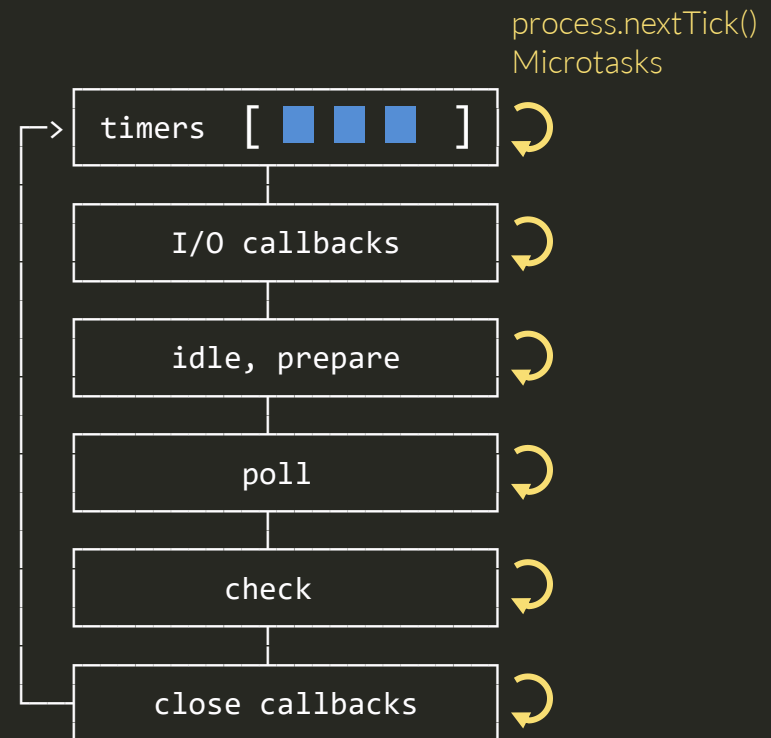
    // Register handle cleanups
    env->RegisterHandleCleanup(
        reinterpret_cast<uv_handle_t*>(env->immediate_check_handle()),
        HandleCleanup,
        nullptr);
    // ...

    return env;
}
```

# Put Tasks/Callbacks To Event Loop

- Non-blocking I/O APIs
  - `fs.readFile(path, cb)`
- Timer Phase
  - `setTimeout(cb, time)`
  - `setInterval(cb, time)`
- Check Phase
  - `setImmediate(cb)`
- At Each Phase End
  - `process.nextTick(cb)`
  - Microtasks (Promise)

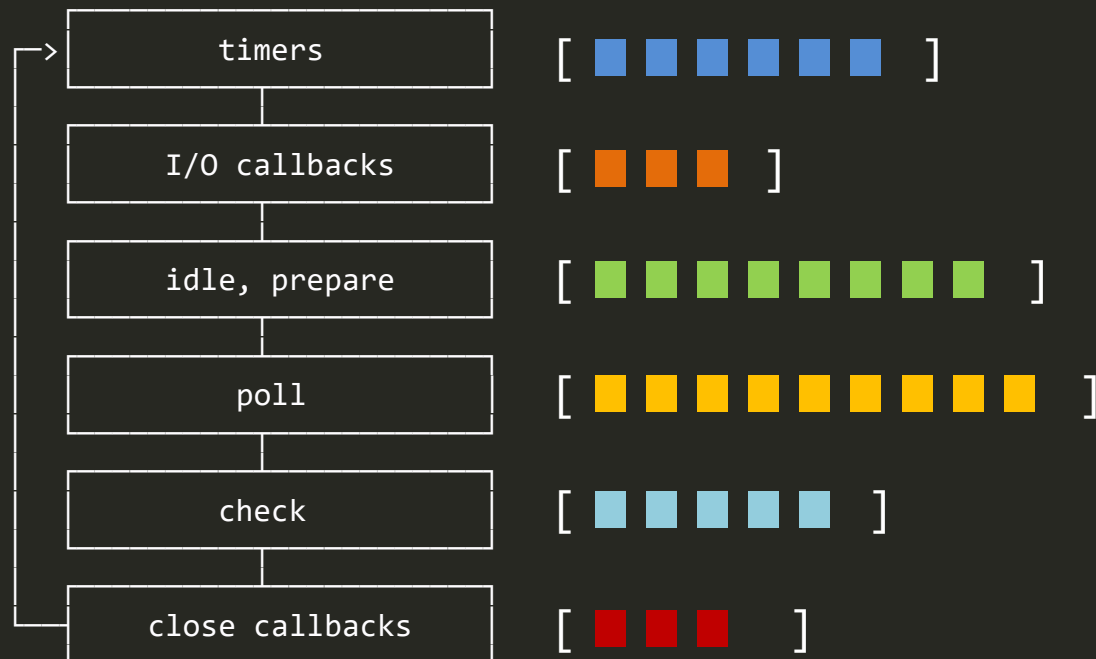
*"Each phase has a FIFO queue of callbacks to execute."*



# How Long a Tick Is?

*"This is a tick: the synchronous invocation of zero or more callback functions associated with any external events. **Once the queue is emptied out and the last function returns, the tick is over.**"*

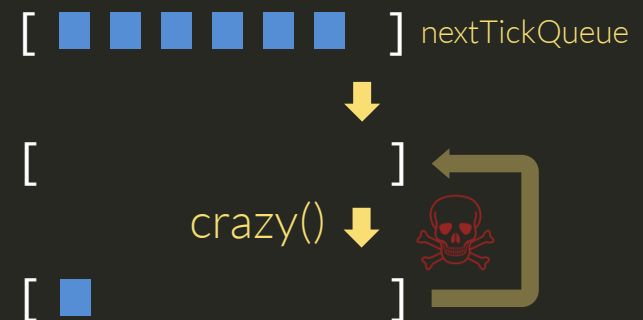
- josh3736, "What exactly is a Node.js event loop tick?" Answered@stackoverflow



# I/O Starvation

“ ... any time you call `process.nextTick()` in a given phase, all callbacks passed to `process.nextTick()` will be resolved before the event loop continues. ... *"starve" your I/O* by making recursive `process.nextTick()` calls, ... ”

```
function crazy() {  
  console.log('Are you crazy?');  
  
  process.nextTick(function () {  
    crazy();  
  });  
}  
  
crazy();
```



# EventEmitter

# EventEmitter

*The base class accommodates **Observer Pattern** in Node.js  
(Publish/Subscribe, Mediator Patterns)*

```
function EventEmitter() { events.js
  EventEmitter.init.call(this);
}
module.exports = EventEmitter;

// Backwards-compat with node 0.10.x
EventEmitter.EventEmitter = EventEmitter;

// ...
EventEmitter.init = function() {
  // ...
  if (!this._events || this._events === Object.getPrototypeOf(this)._events) {
    this._events = {};
    this._eventsCount = 0;
  }

  this._maxListeners = this._maxListeners || undefined;
};
```



# APIs

```
EventEmitter.prototype.setMaxListeners events.js  
EventEmitter.prototype.getMaxListeners  
EventEmitter.prototype.emit (publish)  
EventEmitter.prototype.addListener (subscribe)  
EventEmitter.prototype.on = EventEmitter.prototype.addListener  
EventEmitter.prototype.once  
EventEmitter.prototype.removeListener  
EventEmitter.prototype.removeAllListeners  
EventEmitter.prototype.listeners  
EventEmitter.prototype.listenerCount
```

# .on()

```
EventEmitter.prototype.addListener = function addListener(type, listener) { events.js
  var m;
  var events;
  var existing;

  if (typeof listener !== 'function')
    throw new TypeError('listener must be a function');

  events = this._events;
  // ...
  if (!existing) {
    // Optimize the case of one listener.
    // Don't need the extra array object.
    existing = events[type] = listener;
    ++this._eventsCount;
  } else {
    if (typeof existing === 'function') {
      // Adding the second element, need to change to array.
      existing = events[type] = [existing, listener];
    } else {
      // If we've already got an array, just append.
      existing.push(listener);
    }

    // ...
  }

  return this;
};
```

event\_name1: [       ]

event\_name2: 

event\_name3: [    ]

# .emit()

```
EventEmitter.prototype.emit = function emit(type) {  
  // ...  
  events = this._events;  
  // ...  
  handler = events[type];  
  if (!handler)  
    return false;  
  
  // ...  
  switch (len) {  
    // fast cases  
    case 1:  
      emitNone(handler, isFn, this);  
      break;  
    case 2:  
      emitOne(handler, isFn, this, arguments[1]);  
      break;  
    // ...  
    // slower  
    default:  
      args = new Array(len - 1);  
      for (i = 1; i < len; i++)  
        args[i - 1] = arguments[i];  
      emitMany(handler, isFn, this, args);  
    }  
  // ...  
  return true;  
};
```

events.js

The diagram shows three event listener arrays: `event_name1` with 6 grey squares, `event_name2` with 1 grey square, and `event_name3` with 3 purple squares. A dashed line connects the `handler = events[type];` line in the code to the `event_name3` array. Inside the `event_name3` array, two yellow curved arrows point from the first and second purple squares to the third purple square, indicating a loop or iteration.

event\_name1: [ ■ ■ ■ ■ ■ ■ ]  
event\_name2: [ ■ ]  
event\_name3: [ ■ ■ ■ ]

```
function emitMany(handler, isFn, self, args) {  
  if (isFn)  
    handler.apply(self, args);  
  else {  
    var len = handler.length;  
    var listeners = arrayClone(handler, len);  
    for (var i = 0; i < len; ++i)  
      listeners[i].apply(self, args);  
  }  
}
```

# Quick Demo

# Who Prints First?

```
console.log('<0> schedule with setTimeout in 1-sec');                                02_prints.js
    setTimeout(function () { console.log('[0] setTimeout in 1-sec boom!'); }, 1000);
console.log('<1> schedule with setTimeout in 0-sec');
    setTimeout(function () { console.log('[1] setTimeout in 0-sec boom!'); }, 0);
console.log('<2> schedule with setImmediate');
    setImmediate(function () { console.log('[2] setImmediate boom!'); });
console.log('<3> A immediately resolved promise');
    aPromiseCall().then(function () { console.log('[3] promise resolve boom!'); });
console.log('<4> schedule with process.nextTick');
    process.nextTick(function () { console.log('[4] process.nextTick boom!'); });

function aPromiseCall () {
    return new Promise(function(resolve, reject) { return resolve(); });
}
```

What if these things are arranged in an I/O callback?

# EventEmitter - Dead Loop

```
var EventEmitter = require('events');

var crazy = new EventEmitter();

crazy.on('event1', function () {
  console.log('event1 fired!');
  crazy.emit('event2');
});

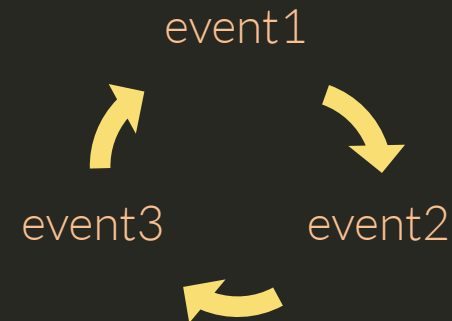
crazy.on('event2', function () {
  console.log('event2 fired!');
  crazy.emit('event3');
});

crazy.on('event3', function () {
  console.log('event3 fired!');
  crazy.emit('event1');
});

crazy.emit('event1');
```

03\_deadloop.js

Synchronous Execution Loop



What if scheduled with...

- process.nextTick()
- setImmediate()

# Long-Running Tasks?

- How to deal with my long-running tasks?
  - Cut it down or use a worker
- A ridiculous heavy task

```
function doHeavy () {  
  // Counts how many 1s occurred  
  var count = 0;  
  
  for (var i = 0; i < 1e8; i++) {  
    if (Math.round(Math.log(  
      Math.sqrt(Math.abs(Math.round(Math.random() * 1000)))  
    )) === 1)  
      count++;  
  }  
  return count;  
}
```

04\_heavy.js

```
setInterval(function () { console.log('I am not blocked'); }, 1000);  
console.log(doHeavy()); // Takes around 10 seconds on my machine
```

# Cut It Down (I)

```
function doNotSoHeavy (times) {
  var count = 0;

  for (var i = 0; i < times; i++) {
    if (Math.round(Math.log(
      Math.sqrt(Math.abs(Math.round(Math.random() * 1000)))
    )) === 1)
      count++;
  }
  return count;
}

function doHeavy() {
  var total = 1e8,
      cuts = 100,
      counts = 0;

  for (var i = 0; i < cuts; i++) {
    counts = counts + doNotSoHeavy(total/cuts);
  }
  return counts;
}

setInterval(function () { console.log('I am not blocked'); }, 1000);
console.log(doHeavy()); // Takes around 10 seconds on my machine
```

04\_heavy\_cut\_sync.js

← Synchronous. Blocks?



# Cut It Down (II)

```
function doHeavy(callback) { 04_heavy_cut_async1.js
  var total = 1e8,
      cuts = 100,
      counts = 0,
      remains = cuts;

  for (var i = 0; i < cuts; i++) {
    setImmediate(function () {
      counts = counts + doNotSoHeavy(total/cuts);
      remains--;

      if (!remains) {
        process.nextTick(function () {
          callback(counts);
        });
      }
    });
  }
}

doHeavy(function (counts) {
  console.log(counts);
});
```

Need a callback to get the asynchronous result

Asynchronous. Blocks?

# Cut It Down (III)

```
function doHeavy(callback) {
    var total = 1e8,
        cuts = 100,
        counts = 0,
        remains = cuts;

    function doPerLoopIter() {
        setImmediate(function () {
            counts = counts + doNotSoHeavy(total/cuts);
            remains--;
            if (!remains) {
                process.nextTick(function () {
                    callback(counts);
                });
            } else {
                doPerLoopIter();
            }
        });
    }
    doPerLoopIter();
}

doHeavy(function (counts) {
    console.log(counts);
});
```

04\_heavy\_cut\_async2.js

← Asynchronous. Blocks?

# Cut It Down (IV)

```
var heavyJobs = {
  counts: 0,
  queue: [],
  _callback: null,
  add: function (task) {
    this.queue.push(task);
  },
  next: function (callback) {
    var self = this,
        task = this.queue.shift();

    if (!task) return;

    setImmediate(function () {
      self.counts = self.counts + task();
      if (self.queue.length === 0)
        self._callback(self.counts);
      else
        self.next();
    });
  },
  do: function (callback) {
    this._callback = callback;
    this.next();
  }
};
```

```
var total = 1e8,                                04_heavy_queue.js
    cuts = 100;

for (var i = 0; i < cuts; i++) {
  heavyJobs.add(function () {
    return doNotSoHeavy(total/cuts);
  });
}

setInterval(function () {
  console.log('I am not blocked');
}, 1000);

heavyJobs.do(function (counts) {
  console.log(counts);
});
```

There are many ways to make your heavy jobs happy...

This example is for demonstrating the idea. Not very thoughtful.

# Run With Another Process

## 05\_heavy\_fork.js

```
var fork = require('child_process').fork;

function doHeavyWithWorker(callback) {
  var worker = fork('./heavy_jobs.js');

  worker.once('message', function (counts) {
    callback(counts);
  });
}

setInterval(function () {
  console.log('I am not blocked');
}, 1000);

doHeavyWithWorker(function (result) {
  console.log(result.counts);
});
```

## heavy\_jobs.js

```
// heavy_jobs.js
function doHeavy () {
  // Counts how many 1s occurred
  var count = 0;

  for (var i = 0; i < 1e8; i++) {
    if (Math.round(Math.log(
      Math.sqrt(Math.abs(
        Math.round(Math.random() * 1000))
      )) === 1)
      count++;
  }
  return count;
}

var counts = doHeavy();
process.send({
  counts: counts
});
```

fork() establishes the IPC channel between parent and child for your convenience to run your node.js code. There are others ways to do such a job with `child_process`.

# That's It!

```
simen@ubuntu:~$ pstree -ap | grep node
`-node,7130 test.js
    -node,7136 ./heavy_jobs.js
        |-{node},7137
        |-{node},7138
        |-{node},7139
        |-{node},7140
        `-{node},7141
    |-{node},7131
    |-{node},7132
    |-{node},7133
    |-{node},7134
    `-{node},7135
|-grep,7144 --color=auto node
```

```
simen@ubuntu:~$ pstree -ap | grep node
|-node,7130 test.js
|   |-{node},7131
|   |-{node},7132
|   |-{node},7133
|   |-{node},7134
|   `--{node},7135
|-grep,7146 --color=auto node
```

Thank You!