webworker-threads public

Lightweight Web Worker API implementation with native threads

This is based on @xk (jorgechamorro)'s **Threads A GoGo for Node.js**, but with an API conforming to the **Web Worker standard**.

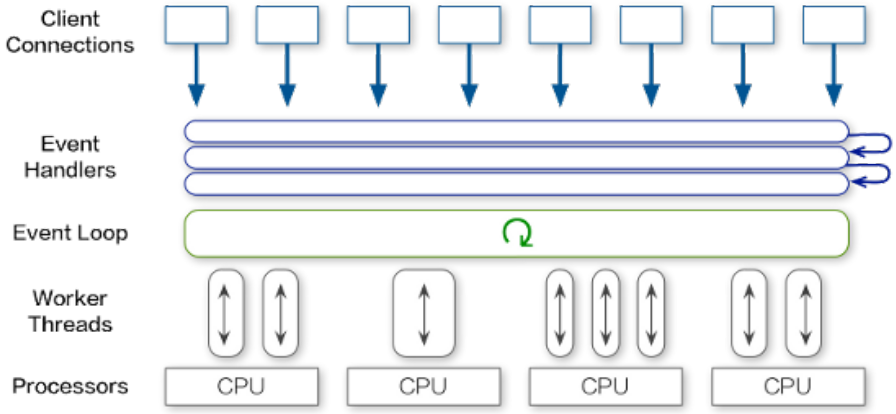
This module provides an asynchronous, evented and/or continuation passing style API for moving blocking/longish CPU-bound tasks out of Node's event loop to JavaScript threads that run in parallel in the background and that use all the available CPU cores automatically; all from within a single Node process.

On Unix (including Linux and OS X), this module requires Node.js 0.8.0+ and a working node-gyp toolchain, which in turn requires make and C/C++. For example, on OS X, you could install XCode from Apple, and then use it to install the command line tools (under Preferences -> Downloads).

On Windows, this module requires Node.js 0.9.3+ and a working **node-gyp toolchain**.

Illustrated Writeup

There is an **illustrated writeup** for the original use case of this module:



Installing the module

With **npm**:

```
npm install webworker-threads
```

Sample usage (adapted from **MDN**):

```
var Worker = require('webworker-threads').Work
// var w = new Worker('worker.js'); // Standar

// You may also pass in a function:
var worker = new Worker(function(){
  postMessage("I'm working before postMessage(
  this.onmessage = function(event) {
    postMessage('Hi ' + event.data);
    self.close();
  };
});
worker.onmessage = function(event) {
  console.log("Worker said : " + event.data);
};
worker.postMessage('ali');
```

A more involved example in **LiveScript** syntax, with five threads:

23 open issues on GitHub

One open pull request on G…

Try it out

T Test webworker-threa…

Keywords

a gogo, web worker, threads

Dependencies (2)

nan, bindings

Dependents

@tradle/otr, otr, synograph, ethercalc, ljve.io, paykoun, labor, minion-job, rabix-cliche, zenjs

```
{ Worker } = require \webworker-threads

for til 5 => (new Worker ->
  fibo = (n) -> if n > 1 then fibo(n - 1) + fi
  @onmessage = ({ data }) -> postMessage fibo
)
  ..onmessage = ({ data }) ->
    console.log "[#{ @thread.id }] #data"
    @postMessage Math.ceil Math.random! * 30
  ..postMessage Math.ceil Math.random! * 30

do spin = -> process.nextTick spin
```

Introduction

After the initialization phase of a Node program, whose purpose is to setup listeners and callbacks to be executed in response to events, the next phase, the proper execution of the program, is orchestrated by the event loop whose duty is to **juggle events, listeners and callbacks quickly and without any hiccups nor interruptions that would ruin its performance**.

Both the event loop and said listeners and callbacks run sequentially in a single thread of execution, Node's main thread. If any of them ever blocks, nothing else will happen for the duration of the block: no more events will be handled, no more callbacks nor listeners nor timeouts nor nextTick()ed functions will have the chance to run and do their job, because they won't be called by the blocked event loop, and the program will turn sluggish at best, or appear to be frozen and dead at worst.

What is WebWorker-Threads

`webworker-threads` provides an asynchronous API for CPU-bound tasks that's missing in Node.js:

```
var Worker = require('webworker-threads').Work
require('http').createServer(function (req,res
  var fibo = new Worker(function() {
    function fibo (n) {
      return n > 1 ? fibo(n - 1) + fibo(n - 2)
    }
    this.onmessage = function (event) {
      postMessage(fibo(event.data));
    }
  });
  fibo.onmessage = function (event) {
    res.end('fib(40) = ' + event.data);
  };
  fibo.postMessage(40);
}).listen(port);
```

And it won't block the event loop because for each request, the `fibo` worker will run in parallel in a separate background thread.

API

Module API

```
var Threads= require('webworker-threads');
```

.Worker

`new Threads.Worker( [ file | function ] )` returns a Worker object.

.create()

`Threads.create( /* no arguments */ )` returns a thread object.

.createPool(numThreads)

`Threads.createPool( numberOfThreads )` returns a threadPool object.

Web Worker API

```
var worker= new Threads.Worker('worker.js');
var worker= new Threads.Worker(function(){ ...
var worker= new Threads.Worker();
```

.postMessage(data)

`worker.postMessage({ x: 1, y: 2 })` sends a data structure into the worker. The worker can receive it using the `onmessage` handler.

.onmessage

`worker.onmessage = function (event) { console.log(event.data) };` receives data from the worker's `postMessage` calls.

.terminate()

`worker.terminate()` terminates the worker thread.

.addEventListener(type, cb)

`worker.addEventListener('message', callback)` is equivalent to setting `worker.onmesssage = callback`.

.dispatchEvent(event)

Currently unimplemented.

.removeEventListener(type)

Currently unimplemented.

.thread

Returns the underlying `thread` object; see the next section for details. Note that this attribute is implementation-specific, and not part of W3C Web Worker API.

Thread API

```
var thread= Threads.create();
```

.id

`thread.id` is a sequential thread serial number.

.load(absolutePath [, cb])

`thread.load( absolutePath [, cb] )` reads the file at `absolutePath` and `thread.eval(fileContents, cb)`.

.eval(program [, cb])

`thread.eval( program [, cb])` converts `program.toString()` and eval()s it in the thread's global context, and (if provided) returns the completion value to `cb(err, completionValue)`.

.on(eventType, listener)

`thread.on( eventType, listener )` registers the listener `listener(data)` for any events of `eventType` that the thread `thread` may emit.

.once(eventType, listener)

`thread.once( eventType, listener )` is like `thread.on()`, but the listener will only be called once.

.removeAllListeners([eventType])

`thread.removeAllListeners( [eventType] )` deletes all listeners for all eventTypes. If `eventType` is provided, deletes all listeners only for the event type `eventType`.

.emit(eventType, eventData [, eventData ...])

`thread.emit( eventType, eventData [, eventData ... ] )` emits an event of `eventType` with `eventData` inside the thread `thread`. All its arguments are .toString()ed.

.destroy(/* no arguments */)

`thread.destroy( /* no arguments */ )` destroys the thread.

Thread pool API

```
threadPool= Threads.createPool( numberOfThread
```

.load(absolutePath [, cb])

`threadPool.load( absolutePath [, cb] )` runs `thread.load( absolutePath [, cb] )` in all the pool's threads.

.any.eval(program, cb)

`threadPool.any.eval( program, cb )` is like `thread.eval()`, but in any of the pool's threads.

.any.emit(eventType, eventData [, eventData ...])

`threadPool.any.emit( eventType, eventData [, eventData ... ] )` is like `thread.emit()`, but in any of the pool's threads.

.all.eval(program, cb)

`threadPool.all.eval( program, cb )` is like `thread.eval()`, but in all the pool's threads.

.all.emit(eventType, eventData [, eventData ...])

`threadPool.all.emit( eventType, eventData [, eventData ... ] )` is like `thread.emit()`, but in all the pool's threads.

.on(eventType, listener)

`threadPool.on( eventType, listener )` is like `thread.on()`, registers listeners for events from any of the threads in the pool.

.totalThreads()

`threadPool.totalThreads()` returns the number of threads in this pool: as supplied in `.createPool( number )`

.idleThreads()

`threadPool.idleThreads()` returns the number of threads in

this pool that are currently idle (sleeping)

.pendingJobs()

`threadPool.pendingJobs()` returns the number of jobs pending.

.destroy([rudely])

`threadPool.destroy( [ rudely ] )` waits until `pendingJobs()` is zero and then destroys the pool. If `rudely` is truthy, then it doesn't wait for `pendingJobs === 0`.

Global Web Worker API

Inside every Worker instance from webworker-threads, there's a global `self` object with these properties:

.postMessage(data)

`postMessage({ x: 1, y: 2 })` sends a data structure back to the main thread.

.onmessage

`onmessage = function (event) { ... }` receives data from the main thread's `.postMessage` calls.

.close()

`close()` stops the current thread.

.addEventListener(type, cb)

`addEventListener('message', callback)` is equivalent to setting `self.onmesssage = callback`.

.dispatchEvent(event)

`dispatchEvent({ type: 'message', data: data })` is the same as `self.postMessage(data)`.

.removeEventListener(type)

Currently unimplemented.

.importScripts(file [, file...])

`importScripts('a.js', 'b.js')` loads one or more files from the disk and `eval()` them in the worker's instance scope.

.thread

The underlying `thread` object; see the next section for details. Note that this attribute is implementation-specific, and not part of W3C Web Worker API.

Global Thread API

Inside every thread .create()d by webworker-threads, there's a global `thread` object with these properties:

.id

`thread.id` is the serial number of this thread

.on(eventType, listener)

`thread.on( eventType, listener )` is just like `thread.on()` above.

.once(eventType, listener)

`thread.once( eventType, listener )` is just like `thread.once()` above.

.emit(eventType, eventData [, eventData ...])

`thread.emit( eventType, eventData [, eventData ... ] )` is just like `thread.emit()` above.

.removeAllListeners([eventType])

`thread.removeAllListeners( [eventType] )` is just like `thread.removeAllListeners()` above.

.nextTick(function)

`thread.nextTick( function )` is like `process.nextTick()`, but much faster.

Global Helper API

Inside every thread .create()d by webworker-threads, there are some helpers:

console.log(arg1 [, arg2 ...])
Same as `console.log` on the main process.

console.error(arg1 [, arg2 ...])
Same as `console.log`, except it prints to stderr.

puts(arg1 [, arg2 ...])
`puts(arg1 [, arg2 ...])` converts .toString()s and prints its arguments to stdout.

WIP WIP WIP

Note that everything below this line is under construction and subject to change.

Examples

A.- Here's a program that makes Node's event loop spin freely and as fast as possible: it simply prints a dot to the console in each turn:

```
cat examples/quickIntro_loop.js
```

```
(function spinForever () {
  process.nextTick(spinForever);
})();
```

B.- Here's another program that adds to the one above a fibonacci(35) call in each turn, a CPU-bound task that takes quite a while to complete and that blocks the event loop making it spin slowly and clumsily. The point is simply to show that you can't put a job like that in the event loop because Node will stop performing properly when its event loop can't spin fast and freely due to a callback/listener/nextTick()ed function that's blocking.

```
cat examples/quickIntro_blocking.js
```

```
function fibo (n) {
  return n > 1 ? fibo(n - 1) + fibo(n - 2) : 1
}

(function fiboLoop () {
  process.stdout.write(fibo(35).toString());
  process.nextTick(fiboLoop);
})();

(function spinForever () {
  process.nextTick(spinForever);
})();
```

C.- The program below uses `webworker-threads` to run the fibonacci(35) calls in a background thread, so Node's event loop isn't blocked at all and can spin freely again at full speed:

```
cat examples/quickIntro_oneThread.js
```

```
function fibo (n) {
  return n > 1 ? fibo(n - 1) + fibo(n - 2) : 1
}

function cb (err, data) {
  process.stdout.write(data);
  this.eval('fibo(35)', cb);
}

var thread= require('webworker-threads').creat

thread.eval(fibo).eval('fibo(35)', cb);

(function spinForever () {
  process.nextTick(spinForever);
})();
```

D.- This example is almost identical to the one above, only that it creates 5 threads instead of one, each running a fibonacci(35) in parallel and in parallel too with Node's event loop that keeps spinning happily at full speed in its own thread:

```
cat examples/quickIntro_fiveThreads.js
```

```
function fibo (n) {
  return n > 1 ? fibo(n - 1) + fibo(n - 2) : 1
}

function cb (err, data) {
  process.stdout.write(" ["+ this.id+ "]"+ dat
  this.eval('fibo(35)', cb);
}

var Threads= require('webworker-threads');

Threads.create().eval(fibo).eval('fibo(35)', c
Threads.create().eval(fibo).eval('fibo(35)', c
Threads.create().eval(fibo).eval('fibo(35)', c
Threads.create().eval(fibo).eval('fibo(35)', c
Threads.create().eval(fibo).eval('fibo(35)', c

(function spinForever () {
  process.nextTick(spinForever);
})();
```

E.- The next one asks `webworker-threads` to create a pool of 10 background threads, instead of creating them manually one by one:

```
cat examples/multiThread.js
```

```
function fibo (n) {
  return n > 1 ? fibo(n - 1) + fibo(n - 2) : 1
}

var numThreads= 10;
var threadPool= require('webworker-threads').c

threadPool.all.eval('fibo(35)', function cb (e
  process.stdout.write(" ["+ this.id+ "]"+ dat
  this.eval('fibo(35)', cb);
});

(function spinForever () {
  process.nextTick(spinForever);
})();
```

F.- This is a demo of the `webworker-threads` eventEmitter API, using one thread:

```
cat examples/quickIntro_oneThreadEvented.js
```

```
var thread= require('webworker-threads').creat
thread.load(__dirname + '/quickIntro_evented_c

/*
  This is the code that's .load()ed into the c

  function fibo (n) {
    return n > 1 ? fibo(n - 1) + fibo(n - 2) :
  }

  thread.on('giveMeTheFibo', function onGiveMe
    this.emit('theFiboIs', fibo(+data)); //Emi
  });

*/

//Emit 'giveMeTheFibo' in the child/background
thread.emit('giveMeTheFibo', 35);

//Listener for the 'theFiboIs' events emitted
thread.on('theFiboIs', function cb (data) {
  process.stdout.write(data);
  this.emit('giveMeTheFibo', 35);
});

(function spinForever () {
  process.nextTick(spinForever);
})();
```

G.- This is a demo of the `webworker-threads` eventEmitter API, using a pool of threads:

```
cat examples/quickIntro_multiThreadEvented.js
```

```
var numThreads= 10;
var threadPool= require('webworker-threads').c
threadPool.load(__dirname + '/quickIntro_event

/*
  This is the code that's .load()ed into the c

  function fibo (n) {
    return n > 1 ? fibo(n - 1) + fibo(n - 2) :
  }

  thread.on('giveMeTheFibo', function onGiveMe
    this.emit('theFiboIs', fibo(+data)); //Emi
  });

*/

//Emit 'giveMeTheFibo' in all the child/backgr
threadPool.all.emit('giveMeTheFibo', 35);

//Listener for the 'theFiboIs' events emitted
threadPool.on('theFiboIs', function cb (data)
  process.stdout.write(" ["+ this.id+ "]"+ dat
  this.emit('giveMeTheFibo', 35);
});

(function spinForever () {
  process.nextTick(spinForever);
})();
```

More examples

The `examples` directory contains a few more examples:

- **ex01_basic**: Running a simple function in a thread.
- **ex02_events**: Sending events from a worker thread.

- ex03_ping_pong: Sending events both ways between the main thread and a worker thread.
- ex04_main: Loading the worker code from a file.
- ex05_pool: Using the thread pool.
- ex06_jason: Passing complex objects to threads.

Rationale

Node.js is the most awesome, cute and super-sexy piece of free, open source software.

Its event loop can spin as fast and smooth as a turbo, and roughly speaking, **the faster it spins, the more power it delivers**. That's why @ryah took great care to ensure that no -possibly slow- I/O operations could ever block it: a pool of background threads (thanks to Marc Lehmann's libeio library) handle any blocking I/O calls in the background, in parallel.

In Node it's verboten to write a server like this:

```
http.createServer(function (req,res) {
  res.end( fs.readFileSync(path) );
}).listen(port);
```

Because synchronous I/O calls **block the turbo**, and without proper boost, Node.js begins to stutter and behaves clumsily. To avoid it there's the asynchronous version of `.readFile()`, in continuation passing style, that takes a callback:

```
fs.readfile(path, function cb (err, data) { /*
```

It's cool, we love it (*), and there's hundreds of ad hoc built-in functions like this in Node to help us deal with almost any variety of possibly slow, blocking I/O.

But what's with longish, CPU-bound tasks?

How do you avoid blocking the event loop, when the task at hand

isn't I/O bound, and lasts more than a few fractions of a millisecond?

```
http.createServer(function cb (req,res) {
  res.end( fibonacci(40) );
}).listen(port);
```

You simply can't, because there's no way... well, there wasn't before `webworker-threads`.

Why Threads

Threads (kernel threads) are very interesting creatures. They provide:

1.- Parallelism: All the threads run in parallel. On a single core processor, the CPU is switched rapidly back and forth among the threads providing the illusion that the threads are running in parallel, albeit on a slower CPU than the real one. With 10 compute-bound threads in a process, the threads would appear to be running in parallel, each one on a CPU with 1/10th the speed of the real CPU. On a multi-core processor, threads are truly running in parallel, and get time-sliced when the number of threads exceed the number of cores. So with 12 compute bound threads on a quad-core processor each thread will appear to run at 1/3rd of the nominal core speed.

2.- Fairness: No thread is more important than another, cores and CPU slices are fairly distributed among threads by the OS scheduler.

3.- Threads fully exploit all the available CPU resources in your system. On a loaded system running many tasks in many threads, the more cores there are, the faster the threads will complete. Automatically.

4.- The threads of a process share exactly the same address space, that of the process they belong to. Every thread can access every memory address within the process' address space. This is a very appropriate setup when the threads are actually part of the same job and are actively and closely cooperating with each other. Passing a reference to a chunk of data via a pointer is many orders of

magnitude faster than transferring a copy of the data via IPC.

Why not multiple processes.

The "can't block the event loop" problem is inherent to Node's evented model. No matter how many Node processes you have running as a **Node-cluster**, it won't solve its issues with CPU-bound tasks.

Launch a cluster of N Nodes running the example B (`quickIntro_blocking.js`) above, and all you'll get is N -instead of one- Nodes with their event loops blocked and showing a sluggish performance.

About npm

About npm, Inc

Jobs

npm Weekly

Blog

Twitter

GitHub

Legal Stuff

Terms of Use

Code of Conduct

Package Name Disputes

Privacy Policy

Reporting Abuse

Other policies

npm loves you