Javascript runtime:

v8: runtime inside chrome

**V8 contains:**

* heap (where memory is allocated && management)
* stack (where stack frames are, a stack of execution contexts)

**In addition to the runtime, we also have:**

* we have web apis, which are extra functionality that the browser provides (DOM, ajax (XMLHttpRequest, setTimeout))
* event loop
* callback queue

**The call stack**

one thread == one call stack == one thing at a time

javascript is a single threaded programming language, a single threaded runtime,

which means it has one call stack, which means it can do execute one piece of code at a time

Example:

Function multiple(a,b) { return a \* b; }

Function square(n) { return multiply(n, n); }

Function printSquare(n) { console.log(square(n); };

printSquare(4);

Call stack: a data structure which essentially records where in the program we are.

1. When we step into a function, we push something onto the top of the stack,
2. When we return from a function, we pop off from the top of the stack

Call Stack v1: main function pushed onto top of stack

|  |
| --- |
| main() |

Call Stack v2: printSquare pushed onto top of stack

|  |
| --- |
| printSquare(4) |
| main() |

Call Stack v3: printSquare calls square, so square is pushed onto top of stack

|  |
| --- |
| square(n) |
| printSquare(4) |
| main() |

Call Stack v4: square calls multiply, so multiply is pushed onto top of stack

|  |
| --- |
| multiply(n, n) |
| square(n) |
| printSquare(4) |
| main() |

Call Stack v5: multiply returns 16, multiply is popped off the top of the stack

|  |
| --- |
| square(n) |
| printSquare(4) |
| main() |

Call Stack v6: square returns 16, square is popped of the top of the stack

|  |
| --- |
| printSquare(4) |
| main() |

Call Stack v7: printSquare calls console.log, console.log is pushed on top of the stack

|  |
| --- |
| console.log(squared) |
| printSquare(4) |
| main() |

Call Stack v8: console.log returns; console.log is popped on the top of the stack

|  |
| --- |
| printSquare(4) |
| main() |

Call Stack v9: printSquare implicitly returns, printSquare is popped of the top of the stack

|  |
| --- |
| main() |

Example 2: Errors in the stack

If you were to run the following code:

Function foo() { throw new Error(‘Oops!’); };

Function bar() { foo(); };

Function baz() { bar(); };

baz();

You would get an uncaught error in the stack, and the stacktrace (state of the stack when the error happended) would be printed to your console with the following message:

“Uncaught Error: Oops!

Foo

Bar

Baz

(anonymous function aka main() in other languages)

Blowing the stack:

Running the following code will cause a stack overflow, in which the maximum call stack size is exceeded.

Function foo() { return foo(); }

foo();

**Blocking:**

What happens when things are slow.

Code that is slow ( large loops, image processing is slow, network requests are slow, counting from 1 to 10 billion is slow, etc)

**Concurrency & The Event Loop**

Javascript the runtime can only do one thing at a time

The reason we can do things concurrently is because the browser is not just the runtime,

We also have web apis (setTimeout, XHR/HTTP, etc), which are effectively threads that we can access through calls to their apis

For node, the same principle holds true, with the web apis switched out for C++ apis, which run threading in C++ land

So the full picture looks like the following:



For given the following code:

console.log(‘hi’);

setTimeout(function cb() {

console.log(‘there’);

}, 0);

console.log(‘JSConfEU’);

The sequence of events is as follows:

1. Console.log(‘hi’) is pushed onto the stack
2. console.log(‘hi’) executes and is popped off the stack
3. setTimeout is pushed on the stack
   1. setTimeout calls the setTimeout webapi
   2. The setTimeout webapi begins counting down
   3. When countdown is complete, setTimeout webapi pushes callback onto the task queue
   4. When the stack is clear, the event loop dequeues the item, and pushes it on the stack.
   5. The callback executes and is popped off the stack.
4. setTimeout is popped off the stack
5. console.log(‘JSConfEU’) is pushed on the stack.
6. console.log(‘JSConfEU’) is popped off the stack.

**Timeouts are not a time to execution, they are a minimum time to execution**

**Callbacks vs Async callbacks:**

Callbacks can be any function another function calls

Or they can be more explicitly async callbacks

Which will be pushed onto the callback queue

// non async callback

[1,2,3].forEach(function(i) {

console.log(i);

});

// async callback

Function asyncForEach(array, cb) {

array.forEach(function() {

setTimeout(cb, 0);

})

};

asyncForEach([1,2,3,4], function(i) {

console.log(i);

});

**Browser repainting**

Given the following code:

[1,2,3].forEach(function(i) {

console.log(‘sync’);

delay();

});

// async callback

Function asyncForEach(array, cb) {

array.forEach(function() {

setTimeout(cb, 0);

})

};

asyncForEach([1,2,3,4], function(i) {

console.log(‘async’);

delay();

});

The blocking callback blocks the browser’s render queue, meaning that you can’t click things, highlight text, etc.

The non blocking callback through being put on the callback queue, gives the render queue the opportunity to render before the event queue pushes code to the stack.



You have roughly 16ms to get synchronous work done within the browser before the user notices that the browser is not repainting.

However, when you consider that the browser actually takes about 8ms to repaint when you modify the DOM, you really only have about 8ms to do your sync work.

Operations that can be resource & time intensive:

* Parsing a large JSON data set
* Image processing
* Sound processing
* Running a prediction algo
* Generating a very large table
* Real time analytics
* Data visualization

Fortunately, today even underpowered devices generally will have multiple cores (multi thread capable)

Challenges with Web Workers:

* Similar to UNIX process model
* No access to dom
* Don’t share memory with main process
* All data is passed through serializable messages, which runs into concurrency issues.

Why not do it on the server?

It costs more to transmit a byte than compute it

You are not charged for client resource usage, wheras you are for server usage.

Angular’s solution is to run everything in web workers

|  |  |
| --- | --- |
| L1 cache reference | 0.5 ns |
| Branch mispredict | 5 ns |
| L2 cache reference | 7 ns |
| Mutex lock/unlock | 25 ns |
| Main memory reference | 100 ns |
| Compress 1k bytes with Zippy | 3,000 ns |
| Send 2k butes over 1 Gbps network | 20,000 ns |
| Read 1 MB sequentially from memory | 250,000 ns |
| Round trip within same datacenter | 500,000 ns |
| Disk seek | 10,000,000 ns |
| Read 1 MB sequentially from disk | 20,000,000 ns |
| Send packet CA -> NYC -> CA | 150,000,000 ns |

* Takes Six order of magnitude longer to send a packet back and forth then reading from main memory
* You have to be sure that running a given computation on the server will be at least that much faster than running on the client.