Web workers

Lecture 8

Problem: slow scripts

- If you've spent enough time with JavaScript or browsing the Web you've probably seen the "slow script" message.
- And, with all those multicore processors sitting in your new machine how could a script be running *too slow*?
- It's because JavaScript can only do one thing at a time.

JavaScript is single-threaded

- Great thing: JavaScript is “single-threaded.”
- Why’s that great? Because it makes programming straightforward.

- The downside:
  - Cannot process large amount of data
  - When doing heavy computation: UI becomes unresponsive.

Most of Web apps work well

- JavaScript steps through everything it has to do, one after the other
- There's no parallel execution

- ❑ Running an init function
- ❑ Handling a user click
- ❑ A timer just went off
- ❑ Handling a submit
- ❑ Process an array of data
- ❑ Handling another user click
- ❑ Updating the DOM
- ❑ Fetching form data
- ❑ Validating user input

Main thread queque

When single-threaded goes bad

- Everything works great until a bit of JavaScript code starts requiring a lot of processing time

- ❑ Running an init function
- ❑ Handling a user click
- ❑ A timer just went off
- ❑ Handling a submit
- ❑ **Process an array of data**
- ❑ Handling another user click
- ❑ Updating the DOM
- ❑ Fetching form data
- ❑ Validating user input

Adding a helper thread: Web workers

- Before HTML5, we were stuck with one thread of control in our pages and apps
- With Web Workers we've now got a way to create another thread of control to help out.
- If you've got code that takes a long time to compute, you can create a Web Worker that will handle that task while the main JavaScript thread of control is making sure everything is good with the browser and the user.

How Web workers work

- The browser creates one or more workers:

  Each worker is defined with its own JavaScript file that contains all the code (or references to code) it needs to do its job.

  Workers live in a very restricted world; they don't have access to many of the runtime objects, like the DOM or any of the variables or functions in your main thread.
- To get a worker to start working, the browser typically sends it a message.

- The worker code receives the message, takes a look at it to see if there are any special instructions, and starts working.

- When the worker completes its work, it then sends a message back, with the final results of what it's been working on.
- The main browser code then takes these results and incorporates them into the page in some way.

Why workers live in a restricted world

- Why not allow workers to access the DOM?
- The reason the DOM and JavaScript have been so successful is that DOM operations are highly optimized, and only one thread can access to the DOM.
- If we let multiple threads concurrently change the DOM, then we'll seriously impact its performance
- Allowing multiple changes to the DOM at the same time can lead to situations where the DOM is in an inconsistent state

Main use of web workers

- Processing large amounts of data in arrays or large JSON responses from web services.
- Analyzing video.
- Processing complex image data for canvas.

Browser support

- Almost all modern browsers support Web Workers, with one exception: Internet Explorer 9. For IE10 and later, you can count on Web Workers

Checking for browser support:

```
if (! window["Worker"]) {
        var status = document.getElementById("status");
        status.innerHTML = "Bummer, no Web Workers";
}
```

Starting: markup

```
<!doctype html>
        <html>
        <head>
                <title>Ping Pong</title>
                <meta charset="utf-8">
                <script src="manager.js"></script>
        </head>
        <body>
                <p id="output"></p>
        </body>
</html>
```

How to create new workers

```
var worker = new Worker("worker.js");
var worker1 = new Worker("worker.js");
var another_worker = new Worker("another_worker.js");
```

Manager.js: sending message to workers

```
window.onload = function() {
        var worker = new Worker("worker.js");
}
worker.postMessage("ping");
```

```
worker.postMessage([1, 2, 3, 5, 11]);
worker.postMessage({"message": "ping", "count": 5});
worker.postMessage(updateTheDOM);
```

Manager waits for a completed job from a worker

```
worker.onmessage = function (event) {
        var message = "Worker says " + event.data;
        document.getElementById("output").innerHTML
                                                = message;
};
```

Worker is waiting for messages from a manager

- To get started on the worker, the first thing we need to do is to make sure the worker can receive messages that are sent from manager.js—that's how the worker gets its work orders.
- Every worker is ready to receive messages, you just need to give the worker a handler to process them.

```
onmessage = pingPong;

function pingPong(event) {
        if (event.data == "ping") {
                postMessage("pong");
        }
}
```

Explaining the code

- First, manager.js creates a new worker, assigns a message handler to it, and then sends the worker a “ping” message.
- The worker, in turn, makes sure pingPong is set up as its message handler, and then it waits.
- At some point, the worker receives a message from the manager, and when it does it checks to see that it contains “ping”, which it does, and then the worker does ~~a lot of~~very little hard work and sends a “pong” message back.
- At this point the main browser code receives a message from the worker, which it hands to the message handler.
- The handler then simply prepends “Worker says ” to the front of the message, and displays it.

link

manager.js

worker.js

Quiz: what will happen 1

- **manager.js**

```
window.onload = function() {
        var worker = new Worker("worker.js");
        worker.onmessage = function(event) {
                alert("Worker says " + event.data);
        }
        for (var i = 0; i < 5; i++) {
                worker.postMessage("ping");
        }
}
```

Quiz: what will happen 2

- **manager.js**

```
window.onload = function() {
        var worker = new Worker("worker.js");
        worker.ommessage = function(event) {
        alert("Worker says " + event.data);
        }
        for(var i = 5; i > 0; i--) {
                worker.postMessage("pong");
        }
}
```

Quiz: what will happen 3

- **manager.js**

```
window.onload = function() {
        var worker = new Worker("worker.js");
        worker.onmessage = function(event) {
                alert("Worker says " + event.data);
                worker.postMessage("ping");
        }

        worker.postMessage("ping");
}
```

Quiz: what will happen 4

- **manager.js**

```
window.onload = function() {
        var worker = new Worker("worker.js");
        worker.onmessage = function(event) {
                alert("Worker says " + event.data);
        }
        setInterval(pinger, 1000);
        function pinger() {
                worker.postMessage("ping");
        }
```

Invoking worker without a message

```
<script>
var worker =
        new Worker("quote.js");
worker.onmessage =
        function(event) {

document.getElementById
    ("quote").innerHTML  =
      event.data;
};
</script>
```

- quote.js

```
var quotes = ["I hope …",
        "There is a light…",
        "Do you believe …"];

var index = Math.floor(
        Math.random() *
        quotes.length);

postMessage(quotes[index]);
```

Simultaneous exhibition: ping-pong

```
window.onload = function() {
        var numWorkers = 3;
        var workers = [];
        for (var i = 0; i < XXX ; i++) {
                var worker = new ("worker.js");
                worker. XXX = function(event) {
                        alert(event.target + " says " + event.XXX );
                };
                workers.push(worker);
        }

        for (var i = 0; i < XXX; i++) {
                workers[i]. ("ping");
        }
}
```

Worker takes initiative

```
var quotes = ["I hope life …", "There is a light at the end…",
"Do you believe …"];

function postAQuote() {
        var index = Math.floor(
                Math.random() * quotes.length);
        postMessage(quotes[index]);}
postAQuote();
setInterval(postAQuote, 3000);
```

Demo: link

Importing computational scripts into Web workers

- With ***importScripts*** you can import one or more JavaScript files into your worker:

```
importScripts("http://bigscience.org/nuclear.js",
        "http://nasa.gov/rocket.js",
        "mylibs/atomsmasher.js");


if (taskType == "songdetection") {
        importScripts("audio.js");
}
```

Using *import scripts* to make JSONP requests from web workers?

Fractals

- The term *fractal* was coined by Benoit Mandelbrot in 1975 in his book **Fractals: Form, Chance, and Dimension**.
- In 1979, while studying the Julia set, Mandelbrot discovered what is now called the Mandelbrot set and inspired a generation of mathematicians and computer programmers in the study of fractals and fractal geometry.
- Like other mathematical ideas, fractals involve numbers and equations.
- Fractals can be used to generate complex images: Swirling spirals, endless self-similar repetitions receding into the distance, geometric objects arranged in infinitely complex patterns, plant-like creations, geologic designs, clouds, and more.
- These wondrous patterns defy logic yet owe their very existence to mathematics and computers

Understanding fractals

Mandelbrot set

- The Mandelbrot Set is one of the most well known fractals
- It is produced by the formula

$$z_{n+1} = z_n^2 + c$$

where z and c are complex numbers, $z_0 = 0$, and c is a point on the plain.

- The formula is iterated until $|z_n|$ (the magnitude of z) is greater than or equal to the bailout value 2.
- Then the pixel that c corresponds to is colored according to the number of iterations that occurred before the process bailed out.
- The uninteresting black area of the image is the actual Mandelbrot Set. It consists of all the values for c where $|z_n|$ never got larger than 2. Of course this area is impossible to compute accurately, so this program decides to colour black all pixels for which $|z_n|$ never gets larger than 2 for a given number of iterations (256).

Single-threaded version: link

Non-mathematical approach

- If you're not a mathematician, the best way to think about the Mandelbrot Set is as an infinitely complex fractal image—meaning an image that you can zoom into, to any level of magnification, because it can be re-calculated based on the initial value

1. It's generated by a very simple equation (the one above) that can be expressed in just a few lines of code
2. Generating the Mandelbrot Set takes a fair number of computing cycles

Using web workers for procedural image generation

- We can do it if we can break up the job into small tasks that each worker can work on independently.
- The browser first creates a bunch of workers to help (but not too many—workers can be expensive).
- Next, the browser code slices out a different part of the image for each worker to compute
- As pre-computed pieces of the image come back from the workers they are aggregated into the image in the browser, and if there are more pieces to compute, new tasks are handed out to the workers that are idle.
- With the last piece of the image computed, the image is complete and the workers sit idle, until the user clicks to zoom in, and then it all starts again...

Pseudocode

```
for (i = 0; i < numberOfRows; i++) {
        var taskForRow = createTaskForRow(i);
        var row = computeRow(taskForRow);
        drawRow(row);
}
```

Is it really faster?

1. Consider an application that has a lot of “computing” going on in the background that also has to be responsive to the user.

   By adding workers to such an app you can immediately improve the feel of the app for your users, because JavaScript has a chance to respond to user interaction in between getting results from the workers, something it doesn’t have a chance to do if everything’s being computed on the main thread. Your app’s just going to *feel* faster (even if it isn’t running any faster under the hood).

2. Almost all modern desktops and devices today are shipping with multicore processors (and even multiple processors).

   With just a single thread of control, JavaScript in the browser doesn’t make use of your extra cores or your extra processors, they’re just wasted. If you use Web Workers, the workers can take advantage of running on your different cores and you’ll see a real speedup in your app

How many workers

- Web Workers aren't intended to be used in large numbers—while creating a worker looks simple in code, it requires extra memory and an operating system thread. So, in general you'll want to create a **limited number** of workers that you reuse over time.

- In theory you could assign a worker to compute every single pixel, which would probably be much simpler from a code design perspective, but given that workers are heavy-weight resources, we'll use 8 workers and structure our computation to take advantage of them.

Preparing a task

- This function packages up all the data needed for the worker to compute a row of pixels, into an object.

```
function createTask(row) {
        var task = {
                row: row,
                width: rowData.width,
                generation: generation,
                r_min: r_min,
                r_max: r_max,
                i: i_max + (i_min - i_max) * row / canvas.height,
                max_iter: max_iter,
                escape: escape
        };
        return task;
}
```

Worker's job – compute row

```
function computeRow(task) {
        var iter = 0;
        var c_i = task.i;
        var max_iter = task.max_iter;
        var escape = task.escape * task.escape;
        task.values = [];
        for (var i = 0; i < task.width; i++) {
                var c_r = task.r_min + (task.r_max - task.r_min) * i / task.width;
                var z_r = 0, z_i = 0;
                for (iter = 0; z_r*z_r + z_i*z_i < escape && iter < max_iter; iter++) {
                // z -> z^2 + c
                ...
                }
                ...
                task.values.push(iter);
        }
        return task;
}
```

Creating an array of workers and giving them an initial task

```
var workers = [];
window.onload = init;
function init() {
        setupGraphics();
        for (var i = 0; i < numberOfWorkers; i++) {
                var worker = new Worker("worker.js");
                worker.onmessage = function(event) {
                processWork(event.target, event.data);
        }
        worker.idle = true;
        workers.push(worker);
    }
    startWorkers();
}
```

StartWorkers

```
var nextRow = 0;
var generation = 0;
function startWorkers() {
        generation++;
        nextRow = 0;
        for (var i = 0; i < workers.length; i++) {
                var worker = workers[i];
                if (worker.idle) {
                        var task = createTask(nextRow);
                        worker.idle = false;
                        worker.postMessage(task);
                        nextRow++;
                }
        }
}
```

Implementing the worker

```
importScripts("workerlib.js");
onmessage = function (task) {
        var workerResult = computeRow(task.data);
        postMessage(workerResult);
}
```

Returning results in the same object

```
task = {
        row: 1,
        width: 1024,
        generation: 1,
        r_min: 2.074,
        r_max: -3.074,
        i: -0.252336,
        max_iter: 1024,
        escape: 1025
};
```

```
workerResult = {
        row: 1,
        width: 1024,
        generation: 1,
        r_min: 2.074,
        r_max: -3.074,
        i: -0.252336,
        max_iter: 1024,
        escape: 1025,
        values: [3, 9, 56, ... -1, 22]
};
```

Reassigning a task to a worker

```
var worker = new Worker("worker.js");
worker.onmessage = function(event) {
        processWork(event.target, event.data);
}

function processWork(worker, workerResults) {
        drawRow(workerResults);
        reassignWorker(worker);
}

function reassignWorker(worker) {
        var row = nextRow++;
        if (row >= canvas.height) {
                worker.idle = true;
        } else {
                var task = createTask(row);
                worker.idle = false;
                worker.postMessage(task);
```

Result

link

mandel.js

Handling a click event

```
canvas.onclick = function(event) {
        handleClick(event.clientX, event.clientY);
};

function handleClick(x, y) {
        var width = r_max - r_min;
        var height = i_min - i_max;
        var click_r = r_min + width * x / canvas.width;
        var click_i = i_max + height * y / canvas.height;
        var zoom = 8;
        r_min = click_r - width/zoom;
        r_max = click_r + width/zoom;
        i_max = click_i - height/zoom;
        i_min = click_i + height/zoom;
        startWorkers();
}
```

Handling onresize event

```
window.onresize = function() {
        resizeToWindow();
};

function resizeToWindow() {
        canvas.width = window.innerWidth;
        canvas.height = window.innerHeight;
        var width = ((i_max - i_min) * canvas.width / canvas.height);
        var r_mid = (r_max + r_min) / 2;
        r_min = r_mid - width/2;
        r_max = r_mid + width/2;
        rowData = ctx.createImageData(canvas.width, 1);
        startWorkers();
}
```

The final test

- Performance
- Single thread: link
- With web workers: link

Terminating a worker

- You've created workers to do a task, the task is done, and you want to get rid of all the workers (they do take up valuable memory in the browser).
- You can terminate a worker from the code in your main page like this:

worker.terminate();

Error handling

- What happens if something goes terribly wrong in a worker? How can you debug it?
- Use the onerror handler to catch any errors and also get debugging information, like this:

```
worker.onerror = function(error) {
        document.getElementById("output").innerHTML =
        "There was an error in " + error.filename +
        " at line number " + error.lineno +
        ": " + error.message;
}
```

Using JSONP to make a server request

```
function makeServerRequest() {
        importScripts("http://SomeServer.com?callback=
                handleRequest");
}

function handleRequest(response) {
        postMessage(response);
}

makeServerRequest();
```

Subworkers

- If your worker needs help with its task, it can create its own workers. Say you're giving your worker regions of an image to work on, the worker could decide that if a region is bigger than some size, it will split it up among its own subworkers:

```
var worker = new Worker("subworker.js");
```

Summary

- Without Web Workers, JavaScript is single-threaded, meaning it can do only one thing at a time.
- If you give a JavaScript program too much to do, you might get the slow script dialog.
- Web Workers handle tasks on a separate thread so your main JavaScript code can continue to run and your UI remains responsive.

More HTML5 and JavaScript: JQuery

```
window.onload = function() {
  alert("the page is loaded!");
}

$(document).ready(function() {
  alert("the page is loaded!");
});

$(function() {
  alert("the page is loaded!");
});
```

```
$(function() {
  $("#buynow").click(function()
  {
    alert("I want to buy now!");
  });
});

$(function() {
  $("a").click(function() {
    alert("I want to buy now!");
  });
});
```

More HTML5: Scalable Vector Graphics SVG

- Including native graphics in your web pages.
- Unlike canvas, SVG graphics are specified with XML
- You create elements that represent graphics, and then you combine those elements together in complex ways to make graphic scenes.
- SVG defines a variety of basic shapes, you can also specify paths
- There are graphical editors that will let you draw a scene and export it as SVG

```
<div id="svg">
 <svg   xmlns
   ="http://www.w3.org/2000/svg">
  <circle id="circle“ cx="50" cy="50"
   r="20"  stroke="#373737"
   stroke-width="2"   fill="#7d7d7d"
  />
 </svg>
</div>
```

Great things about SVG:
You can scale your graphics as big or small as you want and they don’t pixelate
Because SVG is specified with text, SVG files can be searched, indexed, scripted and compressed.

More HTML5

- Built-in native video and audio
- Offline web apps
- Cross-document messaging API
- …