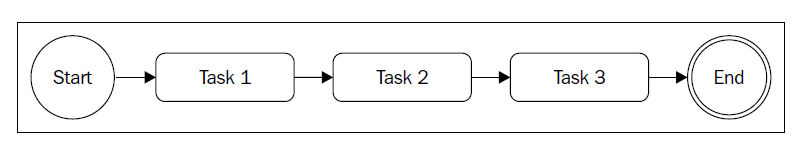
**Sequential execution**

We now begin our exploration of the asynchronous control flow patterns. We will

start by analyzing the **sequential execution** flow.

Executing a set of tasks in sequence means running them one at a time, one after the

other. The order of execution matters and must be preserved, because the result of a

task in the list

function task1(callback) {

asyncOperation(function() {

task2(callback);

});

}

function task2(callback) {

asyncOperation(function(result) {

task3(callback);

});

}

function task3(callback) {

asyncOperation(function() {

callback();

});

}

task1(function() {

//task1, task2, task3 completed

});

**The pattern**

function iterate(index) {

if(index === tasks.length) {

return finish();

}

var task = tasks[index];

task(function() {

iterate(index + 1);

});

}

function finish() {

//iteration completed

}

iterate(0);

It's important to notice that these types of algorithms become really

recursive if task() is a synchronous operation. In such a case, the

stack will not unwind at every cycle and there might be a risk of

hitting the maximum call stack size limit.

**Parallel execution**



Concurrency

**The pattern**

Also, for the parallel execution flow, we can extract our nice little pattern, which we

can adapt and reuse for different situations. We can represent a generic version of

the pattern with the following code:

var tasks = [...];

var completed = 0;

tasks.forEach(function(task) {

task(function() {

if(++completed === tasks.length) {

finish();

}

});

});

function finish() {

//all the tasks completed

}

import fs from 'fs';

import path from 'path';

import superagent from 'superagent';

import mkdirp from 'mkdirp';

import { urlToFilename, getPageLinks } from './utils.js';

Import necessary modules: **fs** for file system operations, **path** for path manipulation, **superagent** for making HTTP requests, **mkdirp** for creating directories, and two functions (**urlToFilename** and **getPageLinks**) from a file named **utils.js** for URL and page-related operations.

function saveFile(filename, contents, cb) {

mkdirp(path.dirname(filename), err => {

if (err) {

return cb(err);

}

fs.writeFile(filename, contents, cb);

});

}

* Define a function named **saveFile** that takes a filename, contents, and a callback (**cb**).
* Use **mkdirp** to create directories as needed based on the dirname of the filename.
* Write the contents to the specified filename using **fs.writeFile**.
* Call the provided callback (**cb**) once the operation is completed.

function download(url, filename, cb) {

console.log(`Downloading ${url}`);

superagent.get(url).end((err, res) => {

if (err) {

return cb(err);

}

saveFile(filename, res.text, err => {

if (err) {

return cb(err);

}

console.log(`Downloaded and saved: ${url}`);

cb(null, res.text);

});

});

}

* Define a function named **download** that takes a URL, filename, and a callback (**cb**).
* Log a message indicating the start of the download.
* Use **superagent** to make an HTTP GET request to the specified URL.
* If there is an error during the request, invoke the callback with the error.
* If successful, save the response text to the specified filename using the **saveFile** function.
* Log a message indicating the completion of the download.
* Call the provided callback (**cb**) once the operation is completed.

function spiderLinks(currentUrl, body, nesting, cb) {

if (nesting === 0) {

return process.nextTick(cb);

}

const links = getPageLinks(currentUrl, body);

if (links.length === 0) {

return process.nextTick(cb);

}

let completed = 0;

let hasErrors = false;

function done(err) {

if (err) {

hasErrors = true;

return cb(err);

}

if (++completed === links.length && !hasErrors) {

return cb();

}

}

links.forEach(link => spider(link, nesting - 1, done));

}

* Define a function named **spiderLinks** that takes a current URL, page body, nesting level, and a callback (**cb**).
* If nesting level is 0, immediately invoke the callback on the next tick of the event loop.
* Get an array of links from the page body using the **getPageLinks** function.
* If there are no links, immediately invoke the callback on the next tick of the event loop.
* Initialize variables to track the number of completed link processes and whether there are errors.
* Define a function named **done** to handle the completion of each link process.
* If there's an error, set **hasErrors** to true and invoke the callback with the error.
* If all links are processed without errors, invoke the callback.

export function spider(url, nesting, cb) {

const filename = urlToFilename(url);

fs.readFile(filename, 'utf8', (err, fileContent) => {

if (err) {

if (err.code !== 'ENOENT') {

return cb(err);

}

return download(url, filename, (err, requestContent) => {

if (err) {

return cb(err);

}

spiderLinks(url, requestContent, nesting, cb);

});

}

spiderLinks(url, fileContent, nesting, cb);

});

}

* Export a function named **spider** that takes a URL, nesting level, and a callback (**cb**).
* Derive the filename from the URL using the **urlToFilename** function.
* Read the content of the file asynchronously using **fs.readFile**.
* If there is an error and it's not due to the file not existing, invoke the callback with the error.
* If the file doesn't exist, trigger a download using the **download** function.
* Once downloaded, call **spiderLinks** with the content of the downloaded file.
* If the file exists, call **spiderLinks** with the content of the file.

This code defines a set of functions for downloading web pages, saving them to files, and recursively crawling through links. It utilizes asynchronous operations for file handling and HTTP requests and modular design for maintainability.

In spider function we got a race condition:

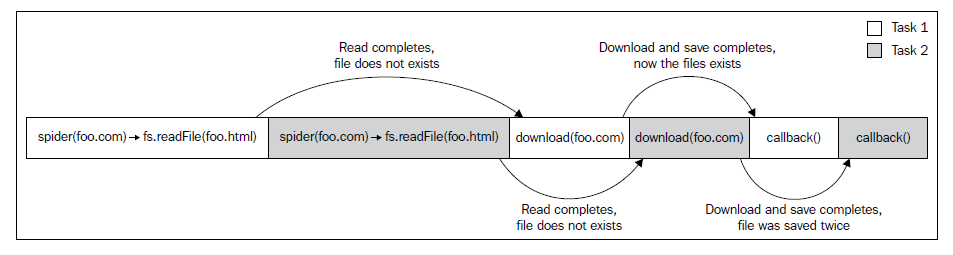
fs.readFile(filename, 'utf8', (err, fileContent) => {

    if (err) {

      if (err.code !== 'ENOENT') { // file or directory does not exist

        return cb(err)

      }



Fixing spider function:

const spidering = {}; // solving the race condition

export function spider(url, nesting, cb) {

if (spidering[url]) {

return process.nextTick(cb);

}

spidering[url] = true;

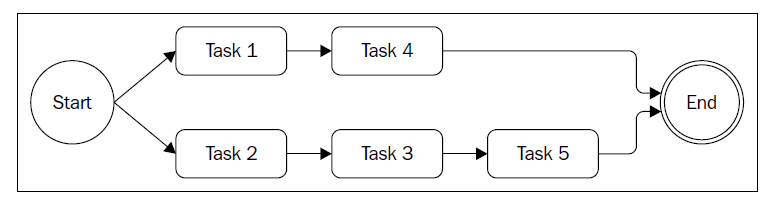
// [...]

}

* **const spidering = {};**: This line declares a constant variable named **spidering** and initializes it as an empty object. This object is going to be used to keep track of URLs that are currently being spidered.
* **export function spider(url, nesting, cb) {**: This line defines a function named **spider**. This function takes three parameters: **url** (the URL to spider), **nesting** (the nesting level), and **cb** (a callback function to be called when the spidering is complete).
* **if (spidering[url]) {**: This condition checks if the **url** is already present in the **spidering** object. If it is, that means the spider is already in the process of spidering this URL.
* **return process.nextTick(cb);**: If the URL is already being spidered, the code immediately schedules the callback function (**cb**) to be called on the next tick of the event loop using **process.nextTick**. This helps avoid the race condition by allowing the current call stack to clear before invoking the callback.
* **spidering[url] = true;**: If the URL is not in the **spidering** object (i.e., it's not currently being spidered), the code sets the value of **spidering[url]** to **true**, indicating that this URL is now being spidered.

The purpose of this mechanism is to prevent the spider from attempting to spider the same URL concurrently, which could lead to race conditions and unpredictable behavior. By using the **spidering** object, it ensures that each URL is processed sequentially, avoiding conflicts in the spidering process.

**Limited parallel execution**



**Limiting the concurrency**

var tasks = [...];

var concurrency = 2, running = 0, completed = 0, index = 0;

function next() { //[1]

while(running < concurrency && index < tasks.length) {

task = tasks[index++];

task(function() { //[2]

if(completed === tasks.length) {

return finish();

}

completed++, running--;

next();

});

running++;

}

}

next();

function finish() {

//all tasks finished

}

**Promises**

**What is a promise?**

In very simple terms, promises are an abstraction that allow an asynchronous

function to return an object called a **promise**, which represents the eventual result

of the operation. In the promises jargon, we say that a promise is **pending** when

the asynchronous operation is not yet complete, it's **fulfilled** when the operation

successfully completes, and **rejected** when the operation terminates with an error.

Once a promise is either fulfilled or rejected, it's considered **settled.**

To receive the fulfillment value or the error (*reason*) associated with the rejection,

we can use the then() method of the promise. The following is its signature:

promise.then([onFulfilled], [onRejected])

Where onFulfilled() is a function that will eventually receive the fulfillment value

of the promise, and onRejected() is another function that will receive the reason of

the rejection (if any). Both functions are optional.

To have an idea of how Promises can transform our code, let's consider the following:

asyncOperation(arg, function(err, result) {

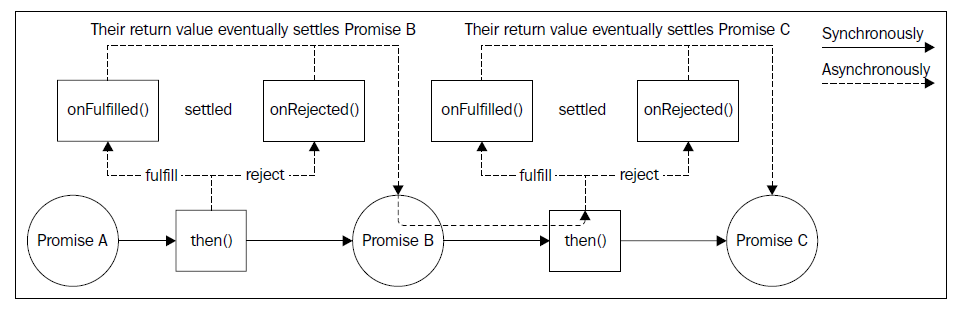
if(err) {

//handle error

}

//do stuff with result

});



**Sequential iteration – the pattern**

To conclude this section on sequential execution, let's extract the pattern to iterate

over a set of promises in sequence:

var tasks = [...]

var promise = Promise.resolve();

tasks.forEach(function(task) {

promise = promise.then(function() {

return task();

});

});

promise.then(function() {

//All tasks completed

});

An alternative to using the forEach() loop is to use reduce(), allowing an even

more compact code:

var tasks = [...]

var promise = tasks.reduce(function(prev, task) {

return prev.then(function() {

return task();

});

}, Promise.resolve());

promise.then(function() {

//All tasks completed

});

Pattern (sequential iteration with promises): dynamically builds a

chain of promises using a loop.

**Parallel execution Promises**

TaskQueue class:

export class TaskQueue {

constructor(concurrency) {

this.concurrency = concurrency;

this.running = 0;

this.queue = [];

}

* **constructor(concurrency) {**: This is the constructor method for the **TaskQueue** class, which gets executed when an instance of the class is created.
* **this.concurrency = concurrency;**: Initializes the **concurrency** property of the instance with the provided value.
* **this.running = 0;**: Initializes the **running** property to 0, representing the number of currently running tasks.
* **this.queue = [];**: Initializes an empty array to store tasks in the queue.

runTask(task) {

return new Promise((resolve, reject) => {

this.queue.push(() => {

return task().then(resolve, reject);

});

process.nextTick(this.next.bind(this));

});

}

* **runTask(task) {**: Declares a method **runTask** that takes a **task** function as an argument.
* **return new Promise((resolve, reject) => {**: Creates and returns a new Promise.
* **this.queue.push(() => {**: Adds a function to the **queue** that, when executed, will execute the provided **task** function and resolve or reject the promise based on its outcome.
* **process.nextTick(this.next.bind(this));**: Schedules the execution of the **next** method on the next tick of the event loop, ensuring the task is picked up asynchronously.

next() {

while (this.running < this.concurrency && this.queue.length) {

const task = this.queue.shift();

task().finally(() => {

this.running--;

this.next();

});

this.running++;

}

}

* **next() {**: Defines the **next** method.
* **while (this.running < this.concurrency && this.queue.length) {**: A loop that continues as long as there are available slots (**this.running < this.concurrency**) and tasks in the queue (**this.queue.length**).
* **const task = this.queue.shift();**: Dequeues the next task from the queue.
* **task().finally(() => { this.running--; this.next(); });**: Executes the task and, after it completes, decreases the running count and calls **next** to process the next task in the queue.
* **this.running++;**: Increments the running count, indicating that a task is now running.

**Generators**

function\* fruitGenerator() {

yield 'apple';

yield 'orange';

return 'watermelon';

}

* **function\* fruitGenerator() {**: This line declares a generator function named **fruitGenerator**.
* **yield 'apple';**: The first **yield** statement produces the value **'apple'** when the generator is iterated.
* **yield 'orange';**: The second **yield** statement produces the value **'orange'** when the generator is iterated.
* **return 'watermelon';**: The **return** statement ends the generator, producing the value **'watermelon'**. Note that this value won't be produced by the regular **next** calls but can be accessed if the generator is exhausted.

**Generators as iterators**

while(!currentItem.done) {

console.log(currentItem.value);

currentItem = iterator.next();

}

* **while(!currentItem.done)**: This starts a **while** loop that continues as long as **currentItem.done** is falsy (undefined in this case).
* **console.log(currentItem.value)**: This logs the value of the current item to the console.
* **currentItem = iterator.next()**: This gets the next item from the iterator.

const asyncFlow = generatorFunction => {

function callback(err) {

if (err) {

return generator.throw(err);

}

const results = [].slice.call(arguments, 1);

generator.next(results.length > 1 ? results : results[0]);

}

const generator = generatorFunction(callback);

generator.next();

}

* **const asyncFlow = generatorFunction => {**: This line declares a function named **asyncFlow** that takes a **generatorFunction** as an argument. This function is designed to work with generator functions.
* **function callback(err) {**: This inner function named **callback** is defined within **asyncFlow**. It will be used as a callback to handle errors and pass results back to the generator.
* **if (err) { return generator.throw(err); }**: If an error (**err**) is passed to the callback, it throws the error into the generator using **generator.throw(err)**.
* **const results = [].slice.call(arguments, 1);**: This line creates an array **results** containing the arguments passed to the **callback** function, excluding the first argument (which is **err**).
* **generator.next(results.length > 1 ? results : results[0]);**: This line calls **generator.next** with the results. If there is more than one result, it passes an array of results; otherwise, it passes a single result.
* **const generator = generatorFunction(callback);**: It invokes **generatorFunction** with the **callback** function, creating a generator.
* **generator.next();**: This starts the execution of the generator by calling **generator.next()**.

In summary, the **asyncFlow** function is designed to simplify the handling of asynchronous flows using generator functions. It provides a callback (**callback**) to the generator, which can be used to handle errors and pass results back into the generator. The generator is then started with **generator.next()**.

asyncFlow(function\* (callback) {

const filename = path.basename(\_\_filename);

const myself = yield fs.readFile(filename, 'utf-8', callback);

yield fs.writeFile('clone\_of\_' + filename, myself, callback);

console.log('Clone created');

})

* **asyncFlow(function\* (callback) {**: This line uses the **asyncFlow** function, passing in a generator function. The generator function takes a **callback** as an argument.
* **const filename = path.basename(\_\_filename);**: This line uses the **path.basename** method to get the base name of the current filename (excluding the directory).
* **const myself = yield fs.readFile(filename, 'utf-8', callback);**: This line yields a promise returned by **fs.readFile**, reading the content of the current file. The result is stored in the **myself** variable.
* **yield fs.writeFile('clone\_of\_' + filename, myself, callback);**: This line yields a promise returned by **fs.writeFile**, writing the content of the file with a new name ('clone\_of\_' + filename).
* **console.log('Clone created');**: After the file is written, this line logs a message to the console.

Here's a quick explanation of how it works:

1. It reads the content of the current file using **fs.readFile**.
2. It writes the content to a new file with a modified name using **fs.writeFile**.
3. It logs a message to the console after the writing is complete.

There are two other variations of this technique, one involving the use of promises

and the other using *thunks*.

A **thunk** used in generator-based control flow is just a function that

partially applies all the arguments of the original function except

its callback. The return value is another function that accepts only

the callback as an argument. For example, the *thunkified* version of

fs.readFile() would be as follows:

function readFileThunk(filename, options) {

return function(callback) {

fs.readFile(filename, options, callback);

}

}

In JavaScript, a "thunk" is a function that wraps an expression to delay its evaluation. Thunks are often used in the context of asynchronous programming or lazy evaluation.

Here are a few key points about thunks:

1. **Delaying Execution:** Thunks are functions that encapsulate a computation or an action, allowing you to defer the execution of that computation until a later time.
2. **Used in Asynchronous Programming:** In the context of asynchronous programming, thunks are often used to represent asynchronous operations. Instead of directly calling a function that returns a result, you call a thunk that will eventually produce the result through a callback or a promise.
3. **Example in Asynchronous Code:**

// Regular function returning a thunk

function asyncOperation(value, callback) {

return function thunk() {

// Some asynchronous operation

setTimeout(function () {

callback(null, value \* 2);

}, 1000);

};

}

// Using the thunk

const myThunk = asyncOperation(21, function (err, result) {

console.log(result); // Will be executed after the asynchronous operation

});

// ES6 version of a thunk for an asynchronous operation

const asyncOperationThunk = (value) => (callback) => {

// Simulating an asynchronous operation

setTimeout(() => {

callback(null, value \* 2);

}, 1000);

};

// Using the thunk

const myThunk = asyncOperationThunk(21);

// Execute the thunk to start the asynchronous operation

myThunk((err, result) => {

if (err) {

console.error(err);

} else {

console.log(result); // Will be executed after the asynchronous operation

}

});

myThunk(); // Execute the thunk to start the asynchronous operation

**Producer-consumer pattern**

export class TaskQueuePC {

constructor(concurrency) {

this.taskQueue = [];

this.consumerQueue = [];

// spawn consumers

for (let i = 0; i < concurrency; i++) {

this.consumer();

}

}

async consumer() {

while (true) {

try {

const task = await this.getNextTask();

} catch (err) {

console.error(err);

}

}

}

getNextTask() {

return new Promise((resolve) => {

if (this.taskQueue.length !== 0) {

return resolve(this.taskQueue.shift());

}

this.consumerQueue.push(resolve);

});

}

runTask(task) {

return new Promise((resolve, reject) => {

const taskWrapper = () => {

const taskPromise = task();

taskPromise.then(resolve, reject);

return taskPromise;

};

if (this.consumerQueue.length !== 0) {

const consumer = this.consumerQueue.shift();

consumer(taskWrapper);

} else {

this.taskQueue.push(taskWrapper);

}

});

}

}

1. **constructor(concurrency) {**: The constructor initializes the **TaskQueuePC** instance with a specified **concurrency** level. It sets up the **taskQueue** to hold tasks and the **consumerQueue** to manage consumers.
2. **for (let i = 0; i < concurrency; i++) { this.consumer(); }**: It spawns a specified number of consumers (equal to **concurrency**) by calling the **consumer** method in a loop during initialization.
3. **async consumer() { while (true) { try { const task = await this.getNextTask(); } catch (err) { console.error(err); } } }**: The **consumer** method runs indefinitely in a loop. It asynchronously waits for the next task using **await this.getNextTask()**. If an error occurs during task retrieval, it is caught and logged.
4. **return new Promise((resolve) => { if (this.taskQueue.length !== 0) { return resolve(this.taskQueue.shift()); } this.consumerQueue.push(resolve); });**: The **getNextTask** method returns a promise that resolves with the next task. If the task queue is not empty, it resolves immediately. Otherwise, it adds the resolver function to the consumer queue.
5. **runTask(task) { return new Promise((resolve, reject) => { const taskWrapper = () => { const taskPromise = task(); taskPromise.then(resolve, reject); return taskPromise; }; if (this.consumerQueue.length !== 0) { const consumer = this.consumerQueue.shift(); consumer(taskWrapper); } else { this.taskQueue.push(taskWrapper); } });**: The **runTask** method runs a specified task asynchronously. It returns a promise that resolves or rejects based on the task execution.
   * **const taskWrapper = () => { ... };**: Wraps the provided task in a function that returns a promise.
   * **if (this.consumerQueue.length !== 0) { ... } else { ... }**: If there is an available consumer, it dequeues one from the consumer queue and assigns the task to it. Otherwise, it adds the task to the task queue.

This class implements a task queue with a producer-consumer pattern, where tasks are processed concurrently by multiple consumers. It ensures that tasks are executed in the order they are received, and consumers are efficiently utilized.