Laboratorium Nr 9
"Przetwarzanie asynchroniczne (wstęp do Node.js)"
Radosław Kopeć
01.12.2020, 08.12.2020

I. Zadanie 1

1. **Treść zadania**

a. Zaimplementuj funkcję loop, wg instrukcji w pliku z Rozwiązaniem 3.

```
/*
** Zadanie:
** Napisz funkcje loop(m), ktora powoduje wykonanie powyzszej
** sekwencji zadan m razy.
**
*/
```

b. wykorzystaj funkcję waterfall biblioteki async

2. **Koncepcja rozwiązania**

Wykorzystamy podstawowe mechanizmy Node.js.

3. **Implementacja:**

1a)

```
function printAsync(s, cb) {
  var delay = Math.floor((Math.random() * 1000) + 500);
  setTimeout(function () {
    console.log(s);
    if (cb) cb();
  }, delay);
}

function task(n) {
  return new Promise((resolve, reject) => {
    printAsync(n, function () {
      resolve(n);
    });
  });
}
```

```
async function loop(times){

    for(i=0;i<times;i++){
        await task(1).then((n) => {
      console.log('task', n, 'done');
      return task(2);
    }).then((n) => {
      console.log('task', n, 'done');
      return task(3);
    }).then((n) => {
      console.log('task', n, 'done');
      console.log('done');
    });
    }

}

loop(4);

console.log("I am not stopped")
```

1b)

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

function first(callback) {
    callback(null, 1);
}

function last(number, callback){
    console.log("DONE")
    callback(null,1)
  }

function task(number, callback) {
    console.log(number)
    callback(null,number + 1)
  }

async function loop(times){
    execTable = [first]
    for(i=0;i<times;i++){
        for(j=0;j<3;j++){
        execTable.push(task)
        }
        execTable.push(last)
    }

    async.waterfall(execTable, function (err, result) {
    });
}

loop(4)
```

4. Wnioski i wyniki

1a)

```
C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad1>node 1a.js
I am not stopped
1
task 1 done
2
task 2 done
3
task 3 done
done
1
task 1 done
2
task 2 done
3
task 3 done
done
1
task 1 done
2
task 2 done
3
task 3 done
done
1
task 1 done
2
task 2 done
3
task 3 done
done
```

Wykonaliśmy asynchroniczną sekwencję wywołań nieblokująco.

1b)

```
C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad1>node 1b.js
1
2
3
DONE
1
2
3
DONE
1
2
3
DONE
1
2
3
DONE
```

Udało nam się to samo z wykorzystaniem funkcji waterfall z biblioteki async. Pozwala ona przekazać wyjście jednej funkcji na drugą i wykonać całą listę takich funkcji.

Wnioski:

- Nigdy nie powinno się umieszczać żadnych ciężkich zadań w wywołaniach zwrotnych, ponieważ są one wykonywane przez pętlę zdarzeń. W ten blokujemy **wątek pętli zdarzeń**, a inne zadania nie zostaną wykonane, na przykład wywołania API na serwerze, program zawiesi się na dłuższą chwilę.

- Do przetwarzania równoległego powinniśmy wykorzystywać mechanizm **Promises**, słowa kluczowe **async/awit**. Wskazane też jest korzystanie z biblioteki **async**.

II. Zadanie 2

1. Treść zadania

Proszę napisać program obliczający liczbę linii we wszystkich plikach tekstowych z danego drzewa katalogów. Do testów proszę wykorzystać zbiór danych Traceroute Data. Program powinien wypisywać liczbę linii w każdym pliku, a na końcu ich globalną sumę. Proszę zmierzyć czas wykonania dwóch wersji programu:

- z synchronicznym (jeden po drugim) przetwarzaniem plików,
- z asynchronicznym (jednoczesnym) przetwarzaniem plików.

2. Koncepcja rozwiązania

Do synchronicznego wywołania funkcji wykorzystamy metodę waterfall z biblioteki async natomiast do równoległego zgodnie z zaleceniem (stack) funkcję async.parallel.

3. **Implementacja:**

Importujemy potrzebne moduły i dostarczoną funkcję:

```
const walk = require('walkdir');
const fs = require('fs');
const async = require("async")
const {performance} = require('perf_hooks')

let count = 0;
const countLines = (file, cb) => { fs.createReadStream(file).on('data', function(chunk) {
            let lines = chunk.toString('utf8')
            .split(/\r\n|[\n\r\u0085\u2028\u2029]/g)
            .length-1;
            count +=lines
            // console.log(file, lines)
        }).on('end', function() {
            cb()
        }).on('error', function(err) {
            cb()
        });

    }
```

Tworzymy funkcje obliczające linie:

```
const paths = walk.sync('PAM08');

const pathTasks = paths.map(path => (cb) => countLines(path, cb))
```

Tworzymy funkcję obliczającą synchronicznie:

```
const callSync = async () => {
    const t1 = performance.now()
    async.waterfall(pathTasks, () => {
        const t2 = performance.now()
        console.log(count);
        console.log(t2 - t1);
    })
}
```

Tworzymy funkcję obliczającą asynchronicznie:

```
const callAsync = async () => {
    const t1 = performance.now()
    async.parallel(pathTasks, () => {
        const t2 = performance.now()
        console.log(count);
        console.log(t2 - t1);
    })
}
```

Wywołujemy jedną z nich:

```
callSync()
// callAsync()
```

4. Wyniki i wnioski

Dla wywołania synchronicznego:

```
C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
215.8358990000561

C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
214.49360099993646

C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
203.5726990001276

C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
190.11470000073314

C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
206.77980000060052

C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
191.87819899991155

C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
212.56150100007653

C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
194.40190099924803
```

Dla wywołania asynchronicznego:

```
C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
107.12099999934435

C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
106.88149999920279

C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
102.74990100041032

C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
108.52129999920726

C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
108.29689999949187

C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
105.32120100036263

C:\Users\Radek\Desktop\SemestrV\theory-of-concurrent-computing\Lab9\zad2>node 2sync.js
61823
102.11909999977797
```

Przy wypisywaniu ilości linii w każdym z pliku "console.log" strasznie spowalniał wszystko co zamazywało różnice między wyołaniami, dlatego wyniki zdecydowałem się zaprezentować dla pomiaru czasu bez wypisywania ilości linii w każdym z plików:

- Wyniki dla wywołania asynchronicznego są lepsze, całość zajęła mniej czasu.

- Funkcja async.parallel pozwala nam uruchomić kolekcję zadań równolegle bez czekania na to, że poprzednia funkcja się wykona. Gdy wszystkie zadania się zakończą rezultat jest wstawiany jako tablica do funkcji powrotu (**callback**).
- Funkcja async.waterfall pozwala przekazać wyjście jednej funkcji na drugą co wymusza przetwarzanie jedna funkcja po drugiej.

5. Bibliografia

http://home.agh.edu.pl/~funika/tw/lab-js/
https://stackoverflow.com/questions/4631774/coordinating-parallel-execution-in-node-js
https://nodejs.org/api/fs.html

https://www.npmjs.com/package/walkdir
https://caolan.github.io/async/v3/docs.html
https://ichi.pro/pl/post/116517108030208
https://levelup.gitconnected.com/javascript-and-asynchronous-magic-bee537edc2da
https://flaviocopes.com/javascript-event-loop/