# Lab 09

## Przetwarzanie asynchroniczne (wstęp do Node.js)

Adrian Madej 27.11.2023

## 1. Treść zadań

Zadanie 1:

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

b) Wykorzystaj funkcję waterfall biblioteki async.

Zadanie 2:

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. Rozwiązywanie zadań

### Zadanie 1a

#### Koncepcja

W celu zapewnienia poprawnej kolejności wykonywania zadań wykorzystamy rekurencję oraz mechanizm Promise. Funkcja task() zwraca obiekty Promise, co

pozwala oddzielić wykonywane po sobie sekwencje, tzn. wiemy kiedy zakończyła się poprzednia i kiedy należy uruchomić nową.

**Implementacja**

```
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);
        });
    });
}

function task_sequence() {
    return 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');
    });
}

function loop(m) {
    if (m === 0) {
        return;
    }

    task_sequence().then(() => {
        console.log("next sequence")
        loop(m - 1)
    })
}

loop(4);
```

**Wyniki**

W wyniku wywołania programu otrzymujemy:

```
      1
 task 1 done
      2
 task 2 done
      3
 task 3 done
    done
next sequence
      1
 task 1 done
      2
 task 2 done
      3
 task 3 done
    done
next sequence
      1
 task 1 done
      2
 task 2 done
      3
 task 3 done
    done
next sequence
      1
 task 1 done
      2
 task 2 done
      3
 task 3 done
    done
next sequence
```

Jak widzimy zadania są wykonywane w poprawnej kolejności, a sam program działa prawidłowo.

## Zadanie 1b

**Koncepcja**

Tym razem do zapewnienia sekwencyjności wykorzystamy funkcję waterfall z biblioteki async. Wykonuje ona funkcje zwrotne (callback functions) jedna po drugiej

w kolejności, w jakiej są dostarczone w tablicy. Wynik takiej funkcji jest przekazywany jako argument do następnej funkcji w kolejności.

**Implementacja**

```
let async = require("async");

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);
        });
    });
}

function task_sequence(cb) {
    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');
        console.log("next sequence")
        cb()
    });
}

function loop(m) {
    let task_list = Array.from({length: m}, () => task_sequence);
    async.waterfall(task_list);
}

loop(4);
```

**Wyniki**

W wyniku wykonania programu otrzymaliśmy następujące wyjście:

```
      1
 task 1 done
      2
 task 2 done
      3
 task 3 done
    done
next sequence
      1
 task 1 done
      2
 task 2 done
      3
 task 3 done
    done
next sequence
      1
 task 1 done
      2
 task 2 done
      3
 task 3 done
    done
next sequence
      1
 task 1 done
      2
 task 2 done
      3
 task 3 done
    done
next sequence
```

Podobnie jak poprzednio udało nam się zapewnić odpowiednią sekwencję wypisywanych poleceń, co wskazuje na poprawność implementacji

**Zadanie 2**

**Koncepcja**

Do rekurencyjnego przechodzenia przez katalogi wykorzystamy moduł walkdir. Następnie zaimplementujemy dwie funkcje zliczające linie w plikach: synchroniczą

`measureSync()` używającą waterfall() oraz asynchroniczną `measureAsync()` zliczają przy użyciu metodyPromise.all().

Aby wypisywać liczbę linii w każdym pliku należy odkomentować linię 15 w pliku zad2.js:

```
//console.log(path, cnt);
```

Została ona zakomentowana dla większej czytelności wyników.

**Implementacja**

```
const walk = require('walkdir');
const fs = require('fs');
const async = require('async');
const performance = require('perf_hooks').performance;


function countLines(path) {
    return new Promise(((resolve, reject) => {
        let cnt = 0;
        fs.createReadStream(path).on('data', function (chunk) {
            cnt += chunk.toString('utf8')
                .split(/\r\n|[\n\r\u0085\u2028\u2029]/g)
                .length - 1;
        }).on('end', function () {
            //console.log(path, cnt);
            resolve(cnt);
        }).on('error', function (err) {
            console.error(err);
            reject(err);
        });
    }));
}


function syncCount(paths) {
    let totalLines = 0;

    const tasks_arr = paths.map((p) => (cb) => {
        countLines(p).then(l => {
            totalLines += l;
            cb();
        });
    });

    return new Promise(((resolve) => {
        async.waterfall(tasks_arr)
            .then(() => {
```

```
                resolve(totalLines)
            })
    }));
}

const PATH = './pam08'
const paths = walk.sync(PATH).filter(p => {
    return fs.lstatSync(p).isFile()
})

function measureSync(){
    const start = performance.now()
    syncCount(paths).then((totalLines) => {
        const timeElapsed = performance.now() - start;
        console.log("Synchronicznie: " + Math.round(timeElapsed) + "ms")
        console.log(totalLines + " linii")

        measureAsync();
    })
}

function measureAsync(){
    const start = performance.now()
    Promise.all(
        paths.map(p => countLines(p))
    ).then((lines) => {
        const totalLines = lines.reduce((acc, val) => acc + val)

        const timeElapsed = performance.now() - start;
        console.log("Asynchronicznie: " + Math.round(timeElapsed) + "ms")
        console.log(totalLines + " linii")
    })
}

measureSync()
```

**Wyniki**

W wyniku wywołania programu dostajemy następujące wyniki:

```
Synchronicznie: 281ms
     61823 linii
Asynchronicznie: 74ms
     61823 linii
```

Jeśli odkomentujemy linię

```
//console.log(path, cnt);
```

to program wypisze nam poszczególne linie uwzględniając ścieżki do plików.

(Kilka wybranych ścieżek do pliku)

```
c:\Users\ltmol\IdeaProjects\tw-
lab9\pam08\PAM08\ArizonaC\ArizonaC_www.msn.com.html 45
c:\Users\ltmol\IdeaProjects\tw-
lab9\pam08\PAM08\ArizonaC\ArizonaC_www.myspace.com.html 45
c:\Users\ltmol\IdeaProjects\tw-
lab9\pam08\PAM08\ArizonaC\ArizonaC_www.orkut.com.html 26
c:\Users\ltmol\IdeaProjects\tw-
lab9\pam08\PAM08\ArizonaC\ArizonaC_www.qq.com.html 45
```

Jak widzimy obie metody obliczyły jednakową liczbę linii co świadczy o poprawności wykonania zadania.

Zliczanie asynchroniczne jest dużo szybsze niż synchroniczne, czego można się spodziewać, gdyż działa ono równolegle.

## 3. Wnioski

- Mechanizm Promise gwarantuje sekwencyjność w kontekście łańcuchów Promise, co oznacza, że kolejne operacje zdefiniowane za pomocą .then() zostaną wykonane po kolei w ustalonej kolejności. To pozwala na jedno po drugim wykonanie operacji asynchronicznych, co jest przydatne w wielu przypadkach.
- Promises pozwalają na łatwe tworzenie równoległych operacji przy użyciu Promise.all() lub innych technik zarządzania wieloma Promise'ami. Promise.all() czeka na zakończenie wszystkich obiektów Promise w danym zestawie i zwraca jedno Promise, które jest rozwiązane, gdy wszystkie obietnice w zestawie zostały rozwiązane.
- Waterfall z biblioteki async dostarcza prosty i czytelny sposób na kontrolowanie przepływu w sekwencji operacji, gdzie wynik jednej operacji jest ważny dla kolejnej. Każda funkcja zwrotna przyjmuje argumenty, które są wynikami poprzedniej operacji. Łańcuch jest przerwany, jeśli jedna z funkcji zwrotnych zwróci błąd. Wówczas funkcja końcowa otrzyma ten błąd jako pierwszy argument.
- Waterfall umożliwia unikanie tzw. "callback hell", czyli głębokich zagnieżdżeń funkcji zwrotnych, poprzez zapewnienie bardziej płaskiej struktury kodu.
- Promise jest wbudowanym mechanizmem języka JavaScript, który oferuje bardziej ogólną obsługę operacji asynchronicznych. Jest bardziej elastyczny i może być używany do zarządzania wieloma operacjami asynchronicznymi jednocześnie, dzięki np. Promise.all.

- Asynchroniczność w języku JavaScript jest bardzo istotnym aspektem i odgrywa kluczową rolę w obszarach takich jak programowanie front-end, operacje sieciowe, obsługa zdarzeń czy też operacje wejścia/wyjścia (I/O).

## 4. Bibliografia

(Każdy podpunkt jest hiperłączem)


- Materiały laboratorium
- Javasript.info
- Synchronously asynchronous
- Dokumentacja Node.js