Laboratorium Node.js

Łukasz Wala
AGH, Wydział Informatyki, Elektroniki i Telekomunikacji
Teoria Współbieżności 2022/23

Kraków, 14 grudnia 2022

1 Treść zadań

1. a) Zaimplementuj funkcję loop, wg instrukcji z załączonego pliku.
   b) wykorzystaj funkcję waterfall biblioteki *async*.
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 i asynchronicznym przetwarzaniem plików.

2 Implementacja zadania 1

Do zadania załączony został kod:

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

Polega ono na napisaniu funckji *loop(m)*, która m razy wykona poniższy fragment kodu (nie jest to sprecyzowane w poleceniu, ale zakładam, że nie chodzi o zwykłą pętlę *for* wywołującą fragment w każdej iteracji, ale o synchroniczne wykonanie kolejnych wywołań fragmentu).

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

Poniżej implementacja funkcji *(*loop):

```
function loop(m) {
  if (m === 0) return;
  console.log("RUN", m)

  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');
  }).then(() => loop(m-1));
}
```

Funkcja jest wywoływana rekurencyjnie w ostatnim callbacku *then*, dzięki czemu kolejne wywołania fragmentu są synchroniczne.

Drugim podejściem będzie użycie funkcji *waterfall* z modułu *async*. Funkcja ta przyjmuje m. in. tablicę asynchronicznych funkcji, które wykonuje synchronicznie przekazując wartość zwracaną z funkcji $n-1$ do funkcji n:

```
function loop(m) {
  const tasks = [];
  for (let i=0; i<m; ++i) {
    tasks.push((callback) => {
    console.log("RUN", i)
    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');
    }).then(() => callback(null));;
    })
  }

  async.waterfall(tasks, (err, res) => console.log(err));
}
```

Do funkcji przekazywana tablicę funkcji asynchronicznych (w tym przypadku są takie same). Wykonywane są one synchronicznie.

3 Implementacja zadania 2

Rozwiązanie z asynchronicznym przetwarzaniem zostało zaimplementowane następująco:

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

function lc_async() {
    console.time("walk");

    files = walk.sync("PAM08/");
    callbacks = files.map((filepath) => {
        return new Promise((resolve, reject) => {
            lines = countLines(filepath, (n) => resolve(n));
        })
    });
```

```
    Promise.all(callbacks).then((result) => {
        console.log("Lines total: ", result.reduce((acc, cur) => acc + cur, 0));
        console.timeEnd("walk");
    });
}
```

Przy użyciu funkcji *walk.sync()* wyznaczane są wszystkie potencjalne pliki do przetworzenia. Następnie są one asynchronicznie przetwarzane (zostają opakowane w *Promise*). Funkcja *Promise.all()* pozwala na zebranie wyników od każdej z funkcji przetwarzających i ich wyświetlenie.

Całkowity czas działania dla tego wariantu wyniósł 190.186 ms dla danych testowych.

Poniżej implementacja z przetwarzaniem synchronicznym (z użyciem tej samej funkcji *countLines*):

```
function countLines(files, index, totalCount) {
    if (index >= files.length) {
        console.log("Total count:", totalCount)
        console.timeEnd("walk");
        return;
    };
    let count = 0;
    fs.createReadStream(files[index]).on('data', function(chunk) {
        count += chunk.toString('utf8')
        .split(/\r\n|[\n\r\u0085\u2028\u2029]/g)
        .length-1;
    }).on('end', function() {
        console.log(files[index], count);
        countLines(files, index+1, totalCount + count,);
    }).on('error', function(err) {
        countLines(files, index+1, totalCount);
    });
}

function lc_sync() {
    console.time("walk");

    counter = 0;
    files = walk.sync("PAM08/");

    countLines(files, 0, 0);
}
```

Czas jej wykonania to 444.882 ms.

Jak widać, istnieje różnica pomiędzy czasami wykonania programów, jednak nie taka, jakiej można by się spodziewać po programie działającym prawdziwie

równolegle a kompletnie synchronicznie. Z racji architektury *Node.js*, zadanie nie są wykonywanie równolegle, ale nadal na jednym wątku, jedynie asynchronicznie. Niewielki, uzyskany tutaj narzut może wynikać np. z szybszej implementacji przy użyciu *Promises*, ale nie z prawdziwej wielowątkowości.

4 Wnioski

Node.js udostępnia wiele mechanizmów asynchroniczności, jak np. *Promises*, *async/await* czy moduł *async*. Architektura *Node.js* pozwala na tak wygodną asynchroniczność ze względu na użycie pętli zdarzeń (*Event Loop*), która synchronizuje wykonanie zadań. Taka architektura, pomimo tego, że pętla zdarzeń działa na jedym wątku, pozwala na oszczędzeniu czasu np. na oczekiwanie na wydarzenia wejścia/wyjścia.

5 Bibliografia

1. Dokumentacja języka JavaScript - https://developer.mozilla.org
2. Dokumentacja *Node.js* - https://nodejs.org/api
3. Dokumentacja modułu *async* - https://caolan.github.io/async/v3