Laboratorium IX – Teoria współbieżności

Weronika Szybińska, 18.12.2022

Treść zadania:

1. Zadanie 1:
    A. Zaimplementuj funkcję loop, wg instrukcji w pliku z Rozwiązaniem 3.
    B. Wykorzystaj funkcję waterfall biblioteki async.
2. 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:
    A. z synchronicznym (jeden po drugim) przetwarzaniem plików,
    B. z synchronicznym (jeden po drugim) przetwarzaniem plików,

Rozwiązanie:

Zgodnie z dostarczonym do zadania przykładem implementacji obiektu Promise() wykonane zostało zadanie 1a. Funkcja loop() jako argument przyjmuje liczbę wywołań funkcji tasks(). Funkcja ta, za to wykorzystuje asynchroniczne wywołania funkcji task(), która zwraca obiekt klasy Promise() zawierający wskaźnik do funkcji printAsync(). Funkcja printAsync czeka wylosowaną ilość czasu a następnie wypisuje liczbę, którą dostaje jako argument. Poniżej przedstawiona została implementacja wykorzystująca mechanizm async/await:

```
const asyncLib = 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);
      });
    });
  }
```

```
async function tasks() {
  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');
  });
 }

const loop = async m => {
  for (let i = 0; i < m; i++) {
    await tasks();
  }
};
```

W zadaniu 1 b funkcja loop() została zmodyfikowana oraz na początku programu dodana została biblioteka async. W implementacji tej funkcji, użyta została funkcja waterfall dostępna w dodanej bibliotece. Na początku tworzony jest wektor zawierający zadania, które program powinien wykonać. Następnie wektor ten jest przekazany do wywołania funkcji waterfall(), która uruchamia wszystkie znajdujące się w nim zadania w kolejności i kończy program.

```
async function loop(m) {
 const tasksToRun = [];
 for (let i = 0; i < m; i++) {
 tasksToRun.push(async () => await tasks());
 }
 asyncLib.waterfall(tasksToRun, (err, result) => {
 console.log('done');
 });
};
```

Wyniki uzyskane po uruchomieniu kolejno pierwszej wersji kodu i drugiej przedstawione zostały poniżej:

```
task 1 done
1
task 1 done
1
task 1 done
2
task 2 done
1
task 1 done
2
task 2 done
task 3 done
done
3
task 3 done
done
3
task 3 done
done
3
task 3 done
done
```

```
task 1 done
1
task 1 done
1
task 1 done
2
task 2 done
1
task 1 done
2
task 2 done
3
task 3 done
done
2
task 2 done
3
task 3 done
done
2
task 2 done
3
task 3 done
done
3
task 3 done
done
```

Zadanie 2

W zadaniu 2 napisany został program zliczający liczbę linii we wszystkich plikach tekstowych znajdujących się w zadanym folderze. Zadanie zostało wykonane w dwóch wersjach, z asynchronicznym oraz synchronicznym przetwarzaniem plików. Do wykonania wykorzystane zostały funkcje: walkdir – która umożliwia przechodzenie przez kolejne pliki w katalogu , fs – która umożliwia operacje na systemie plików oraz szkielet kodu obliczający liczbę linii w pliku tekstowym. Zaimplementowane zostały dwie funkcję sync oraz async odpowiedzialnie za dwie wersje zadania. Funkcje te są bardzo podobne w implementacji. Na początku tworzona jest ścieżka do pliku oraz tablica zadań. Następnie zapisane zadania są wykonywane w przypadku synchronicznego wykonywania plików przez funkcję waterfall, a w przypadku asynchronicznego funkcję paralel. Do przetestowania programu wykorzystane zostały pliki z przekazanego zbioru danych. Poniżej przedstawiona została implementacja:

```
const walkdir = require("walkdir");
const fs = require("fs");
const asyncLib = require("async");

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

```
const sync = async () => {
  const paths = walkdir.sync("./PAM08");
  const tasks = paths.map(path => cb => countLines(path, cb));

  asyncLib.waterfall(tasks, () => {
    console.log(count);
  });
};

const async = async () => {
  const paths = walkdir.sync("./PAM08");
  const tasks = paths.map(path => cb => countLines(path, cb));

  asyncLib.parallel(tasks, () => {
    console.log(count);
  });
};
```

Następnie zmierzone zostały czasy działania funkcji sync oraz async, na podstawie których obliczone zostały średnie. Wyniki podane są w milisekundach.

	1	2	3	4	5	ŚREDNIA
Async	224	182	201	237	550	278,8
Sync	1029	876	1033	597	414	789,8

WNIOSKI:

Na podstawie pomiarów z zadania 2 można wywnioskować, że asynchroniczne wywołanie funkcji znacząco zmniejsza długość działania programu. JavaScript sama w sobie jest jednowątkowa, lecz dzięki zastosowaniu platformy Node.js jesteśmy w stanie zastosować w programie wielowątkowość. Asynchroniczne wywołanie funkcji powoduje, że operacje wykonują się w tle nie blokując głównego wątku co umożliwia tworzenie responsywnych stron internetowych. Dzięki funkcjom z biblioteki async oraz Promises, udało się nam zastosować ten mechanizm w wykonywanych zadaniach i usprawnić pisane programy.

BIBLIOGRAFIA:

https://home.agh.edu.pl/~funika/tw/lab-js/

https://webroad.pl/javascript/746-synchroniczna-asynchronicznosc

https://tworcastron.pl/blog/jak-dziala-asynchronicznosc-w-javascript/