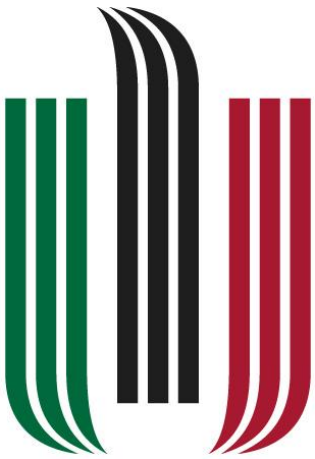
Laboratorium IX

Przetwarzanie asynchroniczne (wstęp do Node.js)

Dominik Marek

2 grudnia 2024



AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Przetwarzanie asynchroniczne

Zadanie 1

1. **Zadanie 1a:** Zaimplementuj funkcję loop, wg instrukcji w pliku z Rozwiązaniem 3.
2. **Zadanie 1b:** 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.

Przydatne moduły:

- walkdir -- trawersacja drzewa katalogów
- fs -- operacje na systemie plików (moduł wbudowany)

Do obliczania liczby linii w pliku tekstowym proszę wykorzystać następujący fragment kodu:

```
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);
      }).on('error', function(err) {
        console.error(err);
      });
```

Fragment ten tworzy strumień i rejestruje trzy funkcje obsługi zdarzeń (wczytanie fragmentu danych, koniec strumienia i wystąpienie błędu). (Zobacz obsługa zdarzeń w Node.js).

W implementacji proszę wykorzystać wzorzec asynchronicznego przetwarzania równoległego opisany tutaj.

Rozwiązania.

Przed rozpoczęciem rozwiązywania zadań z tego laboratorium zapoznałem się ze składnią oraz sposobami zarządzania asynchronicznym wykonywaniem zadań w języku JavaScript. Następnie po pobraniu node.js oraz skonfigurowania środowiska do pracy z językiem przystąpiłem do rozwiązywania poniższych zadań.

1.

a)

Rozwiązanie zadania polega na wykorzystaniu łańcucha obietnic (Promises) w celu sekwencyjnego wykonywania asynchronicznych zadań. Funkcja *loop(m)* tworzy początkowy obiekt Promise za pomocą *Promise.resolve()*, który reprezentuje już zakończoną operację. Następnie, w pętli od 1 do m, do tej obietnicy jest "doklejane" kolejne wywołanie funkcji *task(i)* za pomocą metody *.then().* Dzięki temu każde zadanie jest wykonywane dopiero po zakończeniu poprzedniego.
Funkcja *task(n)* opakowuje asynchroniczną funkcję *printAsync* w obietnicę, co pozwala używać jej w złożonym łańcuchu. Kiedy *task(n)* zostaje wykonane, zwraca obiekt Promise, który zostaje rozwiązany po zakończeniu działania *printAsync.* Po rozwiązaniu obietnicy, w funkcji zwrotnej *.then()* wypisywana jest informacja o zakończeniu zadania.
Cały proces działa sekwencyjnie, ponieważ każda obietnica w łańcuchu czeka na rozwiązanie poprzedniej. Na końcu łańcucha, po zakończeniu ostatniego zadania, wywoływane jest *.then()*, które wypisuje komunikat *done*. To podejście wykorzystuje mocny aspekt *Promises*, czyli kontrolę przepływu asynchronicznego kodu

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

// 'then' returns a new Promise, therefore we can chain another 'then'.
// In this case 'task(x)' directly returns a Promise object, however
// 'then' could also return a value in which case it would be wrapped
// in a Promise that would be automatically resolved with that value.
// 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');
// });

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

function loop(m) {
    let promise = Promise.resolve();

    for (let i = 1; i <= m; i++) {
        promise = promise.then(() => task(i).then((n) => {
            console.log('task', n, 'done');
        }));
    }

    promise.then(() => {
        console.log('done');
    });
}



loop(4);
```

b)

Poniższy kod wykorzystuje bibliotekę *async* i jej funkcję *waterfall*, aby realizować sekwencyjne wykonywanie asynchronicznych zadań. Każde zadanie (*task(n)*) jest opakowane w funkcję, która wywołuje *printAsync* z numerem zadania n. Funkcja *printAsync* symuluje pracę asynchroniczną z opóźnieniem losowym (500-1500 ms) i wypisuje numer zadania, a następnie wywołuje przekazany *callback*, sygnalizując zakończenie.
W funkcji *loop(m)* tworzona jest tablica *tasks*, która zawiera funkcje *task(n)* dla liczb od 1 do m. Te funkcje są przekazywane do *waterfall*, co powoduje ich wywoływanie w kolejności. Po zakończeniu wszystkich zadań, *waterfall* wykonuje końcowy *callback*, wypisując "done". W wersji z *waterfall* uzyskujemy podobny efekt, co w rozwiązaniu z *Promise.then*, gdzie każde zadanie zwraca obiekt Promise, umożliwiający wykonanie kolejnego kroku po zakończeniu bieżącego. W obydwu przypadkach, kod realizuje sekwencję asynchroniczną.

```
const waterfall = require('async/waterfall');

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

function task(n) {
    return function (cb) {

        printAsync(n, function () {
            console.log('task', n, 'done');
            cb();
        });
    };
}

function loop(m) {
    const tasks = [];

        for (let i = 1; i <= m; i++) {
        tasks.push(task(i));
    }


    waterfall(tasks, () => {
        console.log('done');
    });
}

loop(4);
```

Obserwacje:

1. Pierwsze rozwiązanie (z wykorzystaniem Promise i then):
   - *Podejście*:
     Tutaj wykorzystano Promise i łańcuchowanie .then(), aby sekwencyjnie wykonać zadania. Dla każdego zadania tworzona jest obietnica (promise), która rozwiązuje się po zakończeniu zadania asynchronicznego. Każde kolejne zadanie jest dodawane do łańcucha then().
   - *Zalety:*
     - Łatwiejsza do zrozumienia logika, ponieważ wszystkie zadania są wykonywane sekwencyjnie.
     - Lepsza kontrola nad przepływem zadań, ponieważ każde zadanie czeka na zakończenie poprzedniego.
   - *Wady:*
     - W przypadku większej liczby zadań, łańcuch then() może stać się nieczytelny i trudny do utrzymania.
     - Wydajność może nie być optymalna, ponieważ każde zadanie czeka na zakończenie poprzedniego, mimo że operacje są asynchroniczne.

2. Drugie rozwiązanie (z wykorzystaniem waterfall z async):
   - *Podejście*:
     Użycie modułu async/waterfall pozwala na zorganizowanie zadań w łańcuch, gdzie każde kolejne zadanie jest uruchamiane tylko po zakończeniu poprzedniego. Funkcja waterfall przyjmuje tablicę funkcji, które są wywoływane jeden po drugim.
   - *Zalety:*
     - Bardziej strukturalne podejście, które pozwala na łatwe zarządzanie dużą liczbą asynchronicznych zadań w jednym miejscu.
     - Moduł async zapewnia dodatkowe funkcje i wygodę zarządzania asynchronicznością w większych projektach.
   - *Wady:*
     - Użycie zewnętrznej biblioteki (async) może zwiększyć zależności w projekcie.
     - Wymaga większej znajomości dodatkowych narzędzi (biblioteka async), co może zwiększyć złożoność dla osób, które nie są zaznajomione z tym rozwiązaniem.

Podsumowanie:
Obie wersje rozwiązania działają poprawnie, ale różnią się w podejściu do zarządzania asynchronicznością. Pierwsza wersja jest bardziej "czysta" w przypadku prostych przypadków, ale z biegiem czasu może stać się mniej przejrzysta, gdy liczba zadań wzrasta. Druga wersja wykorzystuje async/waterfall, co może być bardziej eleganckim rozwiązaniem w przypadku większych projektów, ale wymaga dodatkowej zależności.

2.

1. Funkcja countLines

Odpowiada za liczenie linii w pojedynczym pliku. Używa fs.*createReadStream* do odczytu pliku w sposób strumieniowy, co pozwala na efektywne przetwarzanie dużych plików bez ładowania ich w całości do pamięci. W trakcie odczytu danych, za pomocą wyrażenia regularnego, linie są dzielone, a ich liczba jest sumowana. Po zakończeniu odczytu (wydarzenie end), wynik jest wyświetlany, a liczba linii jest dodawana do globalnej zmiennej *globalCount*. W przypadku błędu odczytu pliku, funkcja wyświetla komunikat o błędzie.

2. Funkcja processFilesSync

Funkcja processFilesSync realizuje synchroniczne przetwarzanie plików. Korzysta z *asyncLib.waterfall*, co pozwala na wykonanie zadań (liczenia linii w plikach) jeden po drugim. Zadania są przekazywane do *waterfall* jako tablica funkcji, z których każda wywołuje funkcję countLines dla jednego pliku. Funkcja ta mierzy czas wykonania operacji (rozpoczynając go przed przetwarzaniem plików, a kończąc po zakończeniu) i wyświetla całkowitą liczbę linii oraz czas wykonania programu. Po zakończeniu przetwarzania wyświetlane są wyniki.

3. Funkcja processFileAsync

Realizuje asynchroniczne przetwarzanie plików. Zamiast wykonywać zadania jedno po drugim, jak w przypadku wersji synchronicznej, używa *asyncLib.parallel*, co pozwala na jednoczesne przetwarzanie wielu plików. Dzięki temu program działa szybciej, ponieważ operacje na plikach są wykonywane równolegle. Czas wykonania jest również mierzony, a po zakończeniu wszystkich operacji wyświetlana jest całkowita liczba linii oraz czas wykonania.

4. Funkcja main

Funkcja main jest główną funkcją programu. Używa *walkdir*.sync do rekursywnego przeszukiwania katalogu (tutaj ./PAM08), zbierając ścieżki do wszystkich plików w katalogach. Dla każdego pliku tworzy funkcję liczenia linii, która jest przekazywana do tablicy *tasks*. Następnie, wywołuje funkcje *processFilesSync* i *processFileAsync*, aby przetworzyć pliki synchronicznie i asynchronicznie, mierząc czas wykonania obu podejść. Na końcu program wypisuje całkowitą liczbę linii w plikach i czas wykonania obu metod.

Rodzaj przetwarzania	*Liczba zliczonych linii*	*Czas wykonywania programu [ms]*
Synchroniczne	61823	471
Asynchroniczne	61823	254

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

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

let globalCount = 0;


function 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;
            fileCount += add;
        })
        .on("end", function () {
            globalCount += fileCount;
            console.log(`${file}: ${fileCount} lines`);
            cb();
        })
        .on("error", function (err) {
            console.error(`Error reading file ${file}:`, err);
            cb();
        });
};


async function processFilesSync(tasks) {
    let start = new Date().getTime();

    asyncLib.waterfall(tasks, () => {
        console.log(`Total lines: ${globalCount}`);
        var end = new Date().getTime();
        console.log(`Sync execution time: ${end - start} milliseconds`);
    });
}


async function processFileAsync(tasks){
    let start = new Date().getTime();
    await asyncLib.parallel(tasks, () => {
        console.log(`Total lines: ${globalCount}`);
        let end = new Date().getTime();
        console.log(`Async execution time: ${end - start} milliseconds`);
    });
}

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

    console.log('Synchronous processing:')
    await processFilesSync(tasks)

    console.log('Asynchronous processing:')
    await processFileAsync(tasks)

}

main()
```

Na podstawie wyników, które przedstawiają liczbę zliczonych linii i czas wykonania programu dla obu metod (synchronicznej i asynchronicznej), możemy wyciągnąć kilka wniosków i obserwacji.

- **Czas wykonania w zależności od podejścia**:
  - Czas wykonania programu dla podejścia **asynchronicznego** wynosi 254 ms, co jest zdecydowanie szybsze niż 471 ms w przypadku przetwarzania **synchronicznego**.
  - Asynchroniczne podejście wykorzystuje równoczesne przetwarzanie plików, co umożliwia większą wydajność, szczególnie w przypadku dużej liczby plików, gdyż operacje na plikach nie czekają na zakończenie poprzednich. W rezultacie program może jednocześnie przetwarzać wiele plików, co przyspiesza cały proces.
- **Korzyści z równoczesnego przetwarzania**:
  - Asynchroniczne podejście daje znaczną poprawę wydajności w porównaniu do synchronicznego przetwarzania, co wskazuje na dużą przewagę w przypadku większej liczby plików lub dużych plików. Dzięki równoczesnemu odczytowi wielu plików, system nie czeka na zakończenie każdej operacji odczytu, co skraca czas całkowity.
- **Przeciążenie w podejściu synchronicznym**:
  - Podejście synchroniczne przetwarza pliki po kolei, co może prowadzić do długiego czasu oczekiwania, zwłaszcza gdy pliki są duże lub gdy jest ich dużo. Każde przetwarzanie pliku blokuje proces na czas jego odczytu i liczenia linii, co sprawia, że czas wykonania rośnie.
- **Skalowalność asynchroniczna**:
  - W przypadku asynchronicznego przetwarzania, czas wykonania nie rośnie proporcjonalnie do liczby plików w takim stopniu jak w przypadku metody synchronicznej. Zatem w miarę dodawania większej liczby plików, asynchroniczne podejście może wykazywać jeszcze większą przewagę.

Wnioski:

Dla dużych zbiorów danych (jak np. katalogi z wieloma plikami) podejście asynchroniczne jest wyraźnie bardziej efektywne, oferując krótszy czas wykonania programu. W sytuacjach, gdzie czas wykonania jest kluczowy, asynchroniczne przetwarzanie równoległe jest bardziej optymalne.

3.Bibliografia

- https://www.dz5.pl/ti/java/java_skladnia.pdf ~Jacek Rumiński, Język Java – rozdział o wątkach
- http://brinch-hansen.net/papers/1999b.pdf ~ Per Brinch Hansen
- https://www.artima.com/insidejvm/ed2/threadsynch.html ~Bill Venners, Inside the Java Virtual Machine - Charter 20 Thread Synchronization
- https://github.com/creationix/howtonode.org/tree/master/articles
- https://javascript.info/async-await
- https://stackoverflow.com/questions/4631774/coordinating-parallel-execution-in-node-js
- https://www.geeksforgeeks.org/what-is-the-difference-between-async-waterfall-and-async-series