POGLAVLJE 20

Asinhronizam: pojmovi sada i kasnije

Jedan od najvažnijih, ali uprkos tome često slabo shvaćenih delova programskog jezika kao što je JavaScript jeste kako izraziti ponašanje programa koje je promenljivo tokom vremena.

To se ne svodi samo na pitanje šta se događa od početka petlje `for` do kraja petlje `for`, kojoj je, razume se, potrebno izvesno vreme (više mikrosekundi ili milisekundi) da bi se izvršila. To je pitanje šta se događa kada se deo programa izvršava *sada*, a neki njegov drugi deo se izvršava *kasnije* – tokom perioda između *sada* i *kasnije*, kada program zapravo nije aktivan.

Praktično svaki netrivijalan program koji je dosad napisan (naročito na JS-u) morao je da na jedan ili drugi način upravlja tim intervalom, bez obzira na to da li čeka dok korisnik unosi ulazne podatke, ili čeka podatke iz baze podataka ili sistema datoteka, šalje podatke kroz mrežu i čeka odziv iz nje, ili obavlja posao koji se ponavlja u jednakim vremenskim intervalima (kao što je animacija). U svim tim različitim slučajevima, program mora da upravlja svojim stanjem kroz periode neaktivnosti. Kao što kaže poznata preporuka iz londonskog metroa (u vezi s rastojanjem između vrata vagona i ivice perona): „vodite računa o razmaku“.

U stvari, taj odnos između *sada* i *kasnije* delova programa čini srž asinhronog programiranja.

Nesporno je da mogućnost asinhronog programiranja postoji još od samih početaka JavaScripta. Ipak, većina JS programera nije nikad zaista pažljivo razmatrala kako i gde se ono pojavljuje u njihovim programima, niti istraživala razne druge načine da ga primeni. *Dovoljno dobro* rešenje uvek je bila skromna povratna funkcija. Mnogi i danas tvrde da su povratne funkcije više nego dovoljne.

Ali kako JS nastavlja da raste i širi se i po obimu i po složenosti kako bi zadovoljio neprekidno rastuće zahteve za prvoklasnim jezikom koji radi i u čitačima veba, i na serverima i na svakom zamislivom uređaju između njih, napori koje ulažemo da bismo upravljali asinhronizmom softvera postaju sve teži i zahtevaju nalaženje rešenja koja su istovremeno i razumnija i pružaju više mogućnosti.

Mada vam sve ovo možda zasad izgleda prilično apstraktno, znajte da ćemo se time baviti detaljnije i konkretnije u celom ovom delu knjige. U narednih nekoliko poglavlja proučićemo više novih tehnika koje tek počinju da se primenjuju za asinhrono JavaScript programiranje.

Pre nego što pređemo na to, moramo znatno detaljnije razmotriti šta je asinhroni rad i kako se on postiže u JavaScriptu.

Program u delovima

Ceo svoj JS program možete napisati u jednoj *.js* datoteci, ali je on gotovo uvek sastavljen od više delova, od kojih će se samo jedan izvršiti *sada*, dok će se preostali delovi izvršiti *kasnije*. Najuobičajenija jedinica za svaki *delić* programa jeste *funkcija* (`function`).

Problem koji većina JS programera naizgled ima jeste to što se *kasnije* ne događa neposredno i odmah nakon *sada*. Drugim rečima, poslovi koji se ne mogu završiti *sada* završiće se – po definiciji – asinhrono, pa zato neće imati blokirajuće ponašanje kao što biste intuitivno očekivali ili želeli.

Razmotrite sledeće:

```
// ajax(..) je neka funkcija iz biblioteke
var data = ajax( "http://some.url.1" );

console.log( data );
// Uh! `data` najčešće neće sadržati Ajax rezultate
```

Verovatno znate da se standardni Ajax zahtevi ne izvršavaju sinhrono, što znači da funkcija `ajax(..)` još nema nikakvu vrednost da vrati kako bismo je dodelili promenljivoj `data`. Kada bi funkcija `ajax(..)` mogla da blokira izvršavanje programa dok ne stigne odziv, onda bi se operacija dodeljivanja `data = ..` izvršila kako treba.

Međutim, Ajax ne radi tako. Asinhroni Ajax zahtev šaljemo *sada*, ali ćemo njegove rezultate dobiti *kasnije*.

Najjednostavniji (ali svakako ne i jedini, niti obavezno najbolji!) način da „sačekamo“ od *sada* do *kasnije* jeste da upotrebimo funkciju, poznatu pod nazivom *povratna funkcija* (engl. *callback function*):

```
// ajax(..) je neka funkcija iz biblioteke
ajax( "http://some.url.1", function myCallbackFunction(data){

    console.log( data ); // Ura, sad imamo nešto u `data`!

} );
```

Možda ste čuli da je moguće slati sinhrone Ajax zahteve. Mada je to tehnički izvodljivo, nemojte to nikad i ni u kom slučaju raditi, zato što tako blokirate korisnički interfejs čitača veba (dugmad, menije, pomeranje sadržaja prozora itd.) i sprečavate svaku interakciju s korisnikom. To je užasno loša ideja i trebalo bi da je uvek izbegavate.

Pre nego što počnete da negodujete jer se ne slažete s time, reći ćemo da vaša želja da izbegnete zbrku s povratnim funkcijama ne opravdava upotrebu blokirajućeg, sinhronog Ajaxa.

Na primer, razmotrite sledeći kôd:

```
function now() {
    return 21;
}

function later() {
    answer = answer * 2;
    console.log( "Smisao života je:", answer );
}

var answer = now();

setTimeout( later, 1000 ); // Smisao života je: 42
```

Ovaj program se sastoji od dva dela: dela koji se izvršava *sada* i dela koji će se izvršiti *kasnije*. Trebalo bi da bude prilično jasno koji su to delovi, ali bićemo sasvim eksplicitni:

Sada:

```
function now() {
    return 21;
}

function later() { .. }

var answer = now();

setTimeout( later, 1000 );
```

Kasnije:

```
answer = answer * 2;
console.log( "Smisao života je:", answer );
```

Deo programa *sada* izvršava se odmah, čim pokrenete izvršavanje programa. Ali pošto funkcija `setTimeout(..)` definiše događaj (isticanje vremenskog intervala) koji će se odigrati *kasnije*, sadržaj funkcije `later()` izvršiće se u nekom budućem trenutku (1000 milisekundi od sada).

Kad god deo koda upakujete u funkciju i zadate da se ona izvrši kao odziv na određeni događaj (isticanje vremenskog intervala, pritiskanje tastera miša, prijem Ajax odziva itd.), tako formirate element koji se izvršava *kasnije* i time u program uvodite asinhronizam.

Asinhronizam konzole

Ne postoji specifikacija niti skup zahteva koji bi propisivali kako treba da rade metode `console.*` – zato što one zvanično nisu sastavni deo JavaScripta, nego ih u JS dodaje *radno okruženje* (videti četvrti deo ove knjige – *Tipovi i gramatika*).

Iz tog razloga, različiti čitači veba i JS okruženja rade kako njima odgovara, što ponekad dovodi do zbunjujućeg ponašanja.

Konkretno, postoje čitači veba u kojima – pod određenim uslovima – funkcija `console.log(..)` ne prikazuje odmah ono što joj prosledite. Glavni razlog zbog kojeg se to može dogoditi jeste to što su ulazno/izlazne operacije veoma spore i blokiraju delove mnogih programa (ne samo sâm JS). Zato čitač veba može pružiti bolje performanse (ulazno/izlaznih operacija na veb stranici) ako funkcija `console` U/I zahteve obrađuje asinhrono u pozadini, a da vi možda niste ni svesni da se to događa.

Ne mnogo uobičajen, ali moguć slučaj gde bi to moglo da bude *vidljivo* (ne iz samog koda, nego od spolja):

```
var a = {
    index: 1
};

// kasnije
console.log( a ); // ??

// još kasnije
a.index++;
```

Standardno bismo očekivali da vidimo kako se stanje objekta `a` prikazuje onakvo kakvo je bilo tačno u trenutku izvršavanja iskaza `console.log(..)`, tj. da se ispiše nešto poput `{ index: 1 }`, tako da – kad se izvrši sledeći iskaz `a.index++` – on menja stanje kakvo je bilo neposredno nakon prikazivanja stanja `a`.

Prethodni primer koda predstavljaće objekat na vašoj konzoli najčešće baš onako kako biste očekivali. Ali, može se dogoditi da isti kôd radi u situaciji gde čitač veba smatra da ulazno/izlazne operacije na konzoli treba da izvršava u pozadini, kada se može dogoditi da je – u trenutku kad se objekat prikazuje na konzoli čitača veba – iskaz `a.index++` već bio izvršen, pa zato konzola prikazuje `{ index: 2 }`.

Teško je tačno odrediti pod kojim će uslovima U/I operacije na konzoli biti odložene, pa čak i to da li će to biti uočljivo. Imajte samo u vidu taj mogući U/I asinhronizam u slučaju da pri otkrivanju grešaka imate problema s objektima koji se u programu menjaju *nakon* izvršavanja iskaza `console.log(..)`, ali se uprkos tome na konzoli vide neočekivane naknadne izmene.

Ako vam se dogodi takav redak slučaj, najbolje rešenje je da u svom JS dibageru zadate prekidne tačke umesto da podatke prikazujete pomoću iskaza `console`. Drugo najbolje rešenje je da napravite „snimak“ datog objekta tako što ga serijalizujete u promenljivu tipa `string` – na primer, pomoću metode `JSON.stringify(..)`.

Petlja za obradu događaja

Jedna (možda šokantna) tvrdnja: uprkos tome što očigledno omogućava pisanje asinhronog JS koda (kao što je primer obrade događaja isticanja vremenskog intervala koji smo upravo opisali), sve donedavno (ES6), u sam JavaScript nije bilo ugrađeno zapravo ništa što bi direktno omogućavalo asinhronizam koda.

Molim!? To izgleda kao besmislena tvrdnja, zar ne? U stvari, prilično je tačna. Sama mašina jezika JS nije nikad radila ništa drugo osim što je u svakom datom trenutku izvršavala po jedan delić vašeg programa, kad god se to od nje zahtevalo.

„Od nje zahtevalo". Ko je zahtevao? To je onaj najvažniji deo!

JS mašina ne radi u izolaciji, nego unutar svog *radnog okruženja*, koje je za većinu programera tipičan čitač veba. Pomoću softvera kao što je Node.js, poslednjih nekoliko godina (ali ni u kom slučaju samo tokom tog vremena), upotreba JavaScripta proširila se i izvan čitača veba, na druga okruženja – kao što su serveri. U stvari, danas se JavaScript ugrađuje u sve moguće vrste uređaja – od robota do sijalica.

Zajednička „nit" (ovo je ne baš maštovita asinhrona igra reči) svih tih okruženja jeste to da u sebi imaju mehanizam koji upravlja izvršavanjem programa *tokom vremena* i koji u svakom trenutku poziva JS mašinu. Taj mehanizam se zove *petlja za obradu događaja* (engl. *event loop*).

Drugim rečima, JS mašina nema „urođen" osećaj za vreme, već služi kao okruženje za izvršavanje – na (spoljni) zahtev – proizvoljno zadatog dela JS koda. Spoljno okruženje JS mašine oduvek je određivalo *vremenski raspored* „događaja" (izvršavanja delova JS koda).

Tako, na primer, kada vaš JS program pošalje Ajax zahtev za učitavanje određenih podataka sa servera, kôd koji tumači odgovor servera zadajte unutar zasebne funkcije (čije je uobičajeno ime *povratna funkcija* /engl. *callback*/), a JS mašina kaže radnom okruženju: „Sada ću privremeno prekinuti rad, ali čim završiš obradu ovog zahteva koji sam poslala u mrežu i primiš neke podatke, molim te pozovi ovu povratnu funkciju".

Čitač veba onda osluškuje i čeka odziv iz mreže, a kada dobije nešto za vas, stavlja na raspored za izvršavanje povratnu funkciju tako što je umeće u petlju za obradu događaja.

Dakle, šta je petlja za obradu događaja?

Prvo ćemo je konceptualizovati pomoću malo pseudokoda:

```
// `eventLoop` je niz koji igra ulogu reda čekanja
// (obrada po modelu `prvi došao, prvi uslužen`, FIFO)
var eventLoop = [ ];
var event;

// petlja se izvršava "večno"
while (true) {
    // obrada jednog "podeljka"
    if (eventLoop.length > 0) {
        // učitati sledeći događaj iz reda čekanja
        event = eventLoop.shift();

        // izvršiti sledeći događaj
        try {
            event();
        }
        catch (err) {
            reportError(err);
        }
    }
}
```

Ovo je, razume se, veoma pojednostavljen pseudokod, koji ilustruje koncepte. Ali trebalo bi da bude dovoljan za bolje razumevanje suštine.

Kao što vidite, imamo petlju koja se neprekidno izvršava i koju smo predstavili u obliku petlje `while`, a svaka iteracija te petlje zove se *podeljak* (engl. *tick*). U svakom podeljku, ako postoji događaj koji čeka u redu, on se preuzima odatle i izvršava. Ti događaji su povratne funkcije.

Važno je imati na umu da `setTimeout(..)` ne postavlja vašu povratnu funkciju u red čekanja petlje za obradu događaja. Metoda `setTimeout(..)` pokreće odbrojavanje vremenskog intervala; kada taj interval istekne, radno okruženje postavlja vašu povratnu funkciju u petlju za obradu događaja, da bi je neki budući podeljak pokupio i izvršio.

Šta ako u tom trenutku u petlji događaja već ima 20 stavki na čekanju? Onda vaša povratna funkcija čeka. Ona staje u red i čeka iza svih ostalih – standardno ne postoji način za zaobilaženje i preskakanje mesta u redu prema napred. To objašnjava zašto intervali čekanja koje definišete pomoću metode `setTimeout(..)` nisu uvek tačne dužine. Garantovano je (grubo rečeno) da se vaša povratna funkcija neće pokrenuti *pre* kraja intervala čekanja koji zadate, ali se to može dogoditi tačno u tom trenutku ili posle njega, zavisno od stanja petlje za obradu događaja.

Drugim rečima, vaš program je najčešće izdeljen na više malih delova, koji se izvršavaju jedan za drugim u petlji za obradu događaja. Tehnički govoreći, u istom redu čekanja za izvršavanje mogu biti isprepleteni i događaji koji nisu u direktnoj vezi s vašim programom.

Uz tvrdnju da ES6 menja prirodu mesta na kome se upravlja redom čekanja u petlji za obradu događaja, upotrebili smo izraz „donedavno“. To je najvećim delom formalna tehnička napomena, ali pošto ES6 sada tačno propisuje kako radi petlja za obradu događaja, to znači da je, tehnički gledano, to u „nadležnosti“ JS mašine, a ne radnog okruženja JavaScripta. Jedan od najvažnijih razloga za tu promenu bilo je uvođenje ES6 obećanja, koja ćemo razmotriti u poglavlju 22, zato što ona zahtevaju direktnu i preciznu kontrolu nad operacijama raspoređivanja u redu čekanja petlje događaja (videti opis metode `setTimeout(..0)` u odeljku „Saradnja“ na strani 423).

Paralelni višenitni rad

Vrlo je uobičajeno da ljudi brkaju značenje reči „asinhrono“ i „paralelno“ ali one znače zapravo sasvim različite stvari. Ne zaboravite, „asinhrono“ se odnosi na interval između *sada* i *kasnije*, a „paralelno“ opisuje stvari koje se mogu odvijati istovremeno.

Najuobičajenije alatke za paralelan rad koda jesu *procesi* i *niti izvršavanja* (engl. *threads*). Procesi i niti izvršavaju se nezavisno jedni od drugih i mogu se izvršavati istovremeno: na zasebnim procesorima, pa čak i na zasebnim računarima, ali više niti može deliti memoriju istog procesa.

Nasuprot tome, petlja za obradu događaja deli svoje zadatke na poslove koje izvršava serijski (jedan za drugim), što sprečava upotrebu paralelnih procesa i menjanje sadržaja deljene memorije. Paralelizam i „serijalizam“ mogu postojati istovremeno u obliku petlji za obradu događaja koje međusobno sarađuju ali rade u zasebnim nitima.

Do preplitanja paralelnih niti izvršavanja i preplitanja asinhronih događaja može doći na vrlo različitim nivoima granularnosti.

Na primer:

```
function later() {
    answer = answer * 2;
    console.log( "Smisao života je:", answer );
}
```

Ceo sadržaj funkcije `later()` bio bi jedna stavka u redu čekanja petlje za obradu događaja, ali ako razmatramo nit u kojoj bi se ovaj kôd izvršavao, imali bismo zapravo desetak operacija niskog nivoa. Na primer, `answer = answer * 2` zahteva da se prvo učita tekuća vrednost promenjive `answer`, zatim da se negde smesti vrednost `2`, čemu sledi operacija množenja, a onda se rezultat te operacije smešta u `answer`.

U jednonitnom okruženju, nije baš bitno da li su stavke u redu čekanja niti operacije niskog nivoa, zato što ništa ne može da prekine izvršavanje niti. Ali ako imate paralelan sistem, gde dve različite niti rade za isti program, vrlo je moguće nepredvidljivo ponašanje.

Razmotrite sledeće:

```
var a = 20;

function foo() {
    a = a + 1;
}

function bar() {
    a = a * 2;
}

// ajax(..) je neka funkcija iz biblioteke
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

U jednonitnom ponašanju JavaScripta, ako se funkcija `foo()` izvrši pre funkcije `bar()`, rezultat je to da `a` ima vrednost `42`, ali ukoliko se funkcija `bar()` izvrši pre funkcije `foo()`, rezultat u `a` biće `41`.

Međutim, ako se JS događaji koji dele iste podatke izvršavaju paralelno, problemi bi bili znatno suptilniji. Razmotrite sledeće dve liste poslova (izražene pomoću pseudokoda) kao niti koje izvršavaju kôd funkcije `foo()`, odnosno `bar()` i pogledajte šta se događa kada se obe niti izvršavaju tačno u isto vreme:

Nit 1 (`X` i `Y` su privremene lokacije u memoriji):

```
foo():
  a. učitati vrednost `a` u `X`
  b. upisati `1` u `Y`
  c. sabrati `X` i `Y`, upisati rezultat u `X`
  d. vrednost `X` upisati u `a`
```

Nit 2 (`X` i `Y` su privremene lokacije u memoriji):

```
bar():
  a. učitati vrednost `a` u `X`
  b. upisati `2` u `Y`
  c. pomnožiti `X` sa `Y`, upisati rezultat u `X`
  d. vrednost `X` upisati u `a`
```

A sada, recimo da se obe niti izvršavaju potpuno paralelno. Verovatno već možete da uočite problem, zar ne? Obe niti koriste za svoje privremene korake iste deljene memorijske lokacije `X` i `Y`.

Koji je konačan rezultat u `a` ako se koraci izvrše sledećim redosledom?

```
1a  (učitati vrednost `a` u `X` ==> `20`)
2a  (učitati vrednost `a` u `X` ==> `20`)
1b  (upisati `1` u `Y` ==> `1`)
2b  (upisati `2` u `Y` ==> `2`)
1c  (sabrati `X` i `Y`, upisati rezultat u `X` ==> `22`)
1d  (vrednost `X` upisati u `a` ==> `22`)
2c  (pomnožiti `X` sa `Y`, upisati rezultat u `X` ==> `44`)
2d  (vrednost `X` upisati u `a` ==> `44`)
```

Rezultat u `a` biće `44`. Ali šta ako je redosled sledeći?

```
1a  (učitati vrednost `a` u `X` ==> `20`)
2a  (učitati vrednost `a` u `X` ==> `20`)
2b  (upisati `2` u `Y` ==> `2`)
1b  (upisati `1` u `Y` ==> `1`)
2c  (pomnožiti `X` sa `Y`, upisati rezultat u `X` ==> `20`)
1c  (sabrati `X` i `Y`, upisati rezultat u `X` ==> `21`)
1d  (vrednost `X` upisati u `a` ==> `21`)
2d  (vrednost `X` upisati u `a` ==> `21`)
```

Rezultat u `a` biće `21`.

Dakle, programiranje niti može biti vrlo zapetljano, zato što ako ne preduzmete posebne mere da sprečite događanje te vrste prekidanja/preplitanja, možete dobiti veoma iznenađujuće i nedeterminističko ponašanje koje često dovodi do glavobolja.

JavaScript nikada ne deli podatke između niti, što znači da taj nivo nedeterminizma ne pravi problem. Ali to ne znači da je JS uvek deterministički. Već smo videli da različit redosled izvršavanja funkcija `foo()` i `bar()` proizvodi dva različita rezultata (`41` ili `42`).

Možda još nije očigledno, ali nije svaki nedeterminizam loš. Nedeterminizam je ponekad nebitan, a ponekad je nameran. Više primera toga videćemo u ovom i u sledećih nekoliko poglavlja.

Potpuno izvršavanje

Zbog jednonitne prirode JavaScripta, kôd unutar funkcije `foo()` (i `bar()`) čini jednu jedinicu, što znači sledeće: čim funkcija `foo()` počne da se izvršava, ceo njen kôd će se izvršiti pre nego što počne izvršavanje koda funkcije `bar()`, i obrnuto. To ponašanje se zove *potpuno izvršavanje* (engl. *run-to-completion*).

U stvari, semantika potpunog izvršavanja postaje jasnija kada funkcije `foo()` i `bar()` sadrže više koda, na primer:

```
var a = 1;
var b = 2;

function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}

// ajax(..) je neka Ajax funkcija iz biblioteke
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

Pošto funkcija `bar()` ne može prekinuti izvršavanje funkcije `foo()`, niti obrnuto, ovaj program može dati samo dva moguća rezultata, u zavisnosti od toga koja funkcija prva počne da se izvršava – kada bi bio moguć višenitni rad i kad bi bilo dozvoljeno da se prepliće izvršavanje pojedinačnih iskaza funkcija `foo()` i `bar()`, broj mogućih rezultata programa značajno bi se povećao!

Deo 1 programa je sinhron (događa se *sada*), ali su delovi 2 i 3 asinhroni (događaju se *kasnije*), što znači da će između njihovog izvršavanja proteći određeno vreme.

Deo 1:

```
var a = 1;
var b = 2;
```

Deo 2 (`foo()`):

```
a++;
b = b * a;
a = b + 3;
```

Deo 3 (`bar()`):

```
b--;
a = 8 + b;
b = a * 2;
```

Pošto se od delova 2 i 3 svaki može izvršiti pre onog drugog, ovaj program može imati dva moguća rezultata, kao što je ovde ilustrovano:

Rezultat 1:

```
var a = 1;
var b = 2;

// foo()
a++;
b = b * a;
a = b + 3;

// bar()
b--;
a = 8 + b;
b = a * 2;

a; // 11
b; // 22
```

Rezultat 2:

```
var a = 1;
var b = 2;

// bar()
b--;
a = 8 + b;
b = a * 2;

// foo()
a++;
b = b * a;
a = b + 3;

a; // 183
b; // 180
```

Dva moguća rezultata istog koda znače da i dalje imamo nedeterminizam! Ali to je na nivou redosleda izvršavanja funkcija (događaja), a ne na nivou redosleda izvršavanja iskaza programa (ili, u stvari, na nivou redosleda operacija nad izrazima) kao u slučaju niti. Drugim rečima, determinizam programa je bolji nego u slučaju višenitnog izvršavanja.

Primenjeno na ponašanje JavaScripta, za taj nedeterminizam usled redosleda izvršavanja funkcija postoji uobičajen izraz *stanje utrkivanja* (engl. *race condition*) jer se funkcije `foo()` i `bar()` međusobno trkaju koja će se prva izvršiti. Konkretno, imamo trku zato što nema pouzdanog načina da predvidimo koju će vrednost dobiti `a` i `b`.

Kada bi u JavaScriptu postojala funkcija čije ponašanje nije potpuno izvršavanje, mogli bismo imati mnogo više mogućih rezultata, zar ne? Ispostavlja se da ES6 uvodi jednu baš takvu stvar (videti poglavlje 23), ali zasad ne brinite – i to ćemo razmotriti!

Istovremenost

Zamislite veb lokaciju koja prikazuje listu ažuriranih statusa (kao što je lista vesti na društvenim mrežama) koja se puni dok korisnik pomera sadržaj prozora. Da bi to radilo kako treba, potrebno je da se (barem) dva zasebna „procesa“ izvršavaju *istovremeno* (na primer, tokom istog vremenskog intervala, ali ne obavezno i u istom trenutku).

Reč „proces“ ovde pišemo između navodnika zato što to nisu pravi procesi operativnog sistema kako ih računarska nauka definiše. To su virtuelni procesi, ili poslovi, koje čine logički povezani sekvencijalni nizovi operacija. Reč „proces“ koristimo umesto „posao“ zato što ona terminološki odgovara definiciji koncepata koje ovde razmatramo.

Prvi „proces“ će se odazivati na događaje `onscroll` (tako što će slati Ajax zahteve za nov sadržaj) kako oni nastaju dok korisnik pomera sadržaj veb stranice nadole. Drugi „proces“ će primati Ajax odgovore (radi prikazivanja novog sadržaja na stranici).

Očigledno, ako korisnik pomera sadržaj stranice dovoljno brzo, možda ćete videti dva ili više događaja `onscroll` unutar jednog vremenskog intervala koji je potreban da stigne prvi odgovor i da se on obradi, pa ćete zato imati brzo odigravanje događaja `onscroll` i događaja Ajax, isprepletene jedne s drugim.

Istovremenost (engl. *concurrency*) imamo kada se dva ili više „procesa“ izvršavaju istovremeno tokom istog perioda, bez obzira na to da li se operacije koje ih čine odvijaju *paralelno* (u istom trenutku na zasebnim procesorma ili jezgrima). Istovremenost možete zamisliti kao paralelizam na nivou „procesa“ (ili poslova), nasuprot paralelizmu na nivou operacija (niti koje rade na zasebnim procesorima).

Istovremenost uvodi i opcioni pojam međusobne interakcije tih „procesa“, na koji ćemo se vratiti u nastavku teksta.

Tokom datog vremenskog intervala (nekoliko sekundi tokom kojih korisnik pomera sadržaj prozora), pokušaćemo da vizuelizujemo svaki nezavisan „proces“ kao niz događaja/operacija:

„Proces“ 1 (događaji `onscroll`):

```
onscroll, request 1
onscroll, request 2
onscroll, request 3
onscroll, request 4
```