სირთულეები მრავალბირთვიან პროგრამირებაში და მათი გადაჭრის გზები

დოქტორანტის სემინარი 1

დოქტორანტი

ლუკა გორგაძე

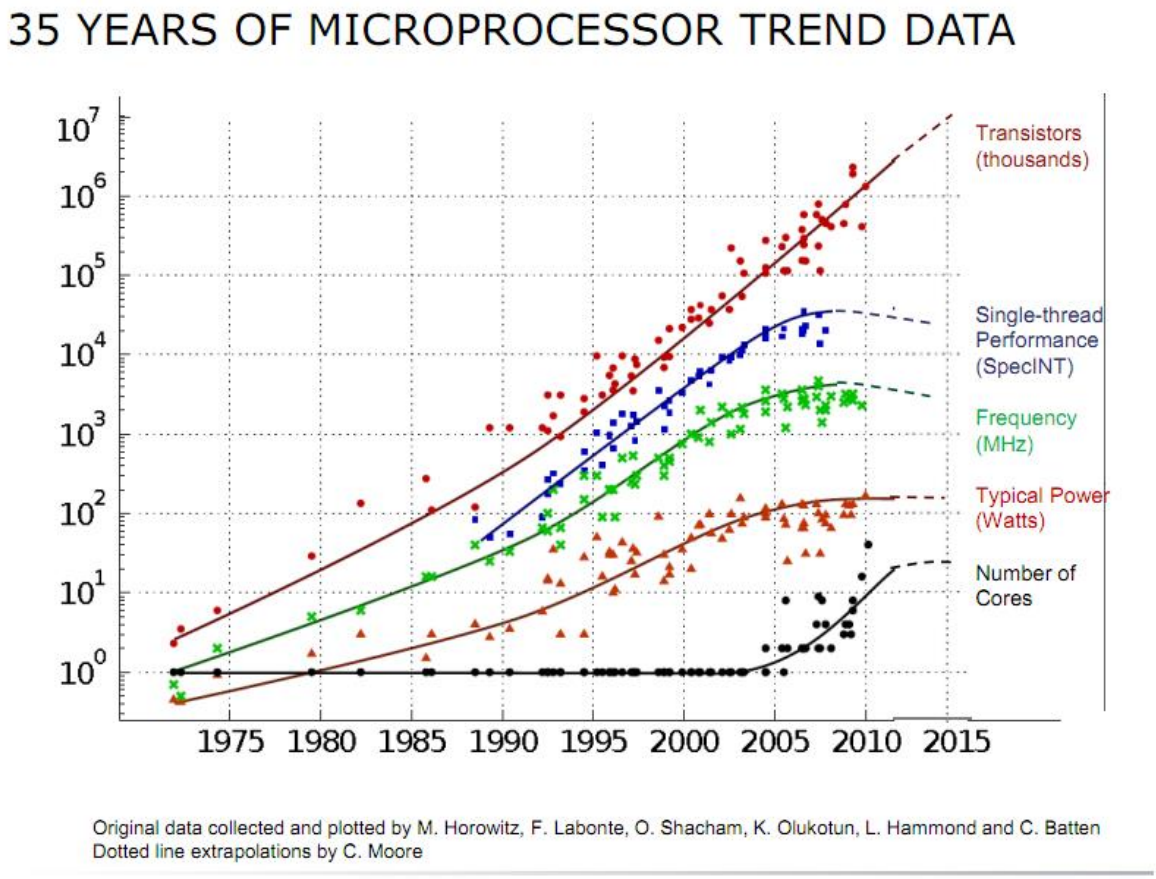
ხელმძღვანელი

სრული პროფესორი

კობა გელაშვილი

1. კონკურენტული პროგრამირების სირთულის მიზეზები

მათთვის, ვისაც შეხება აქვთ კონკურენტულ პროგრამირებასთან, კარგად არის ცნობილი მრავალბირთვიანობის კრიზისის (Multicore Crisis) გამომწვევი მიზეზი (იხ. [2]). კერძოდ ის, რომ მურის (Moore) კანონის თანახმად (იხ. [3]), ყოველ ორ წელიწადში ტრანზისტორების რაოდენობა ჩიპზე აგრძელებს გაორმაგებას, მაშინ როცა უკვე ათეული წელია პროცესორის ბირთვის სისწრაფე ერთ ნიშნულზეა გაჩერებული.



დაახლოებით 2005 წლიდან პროცესორების მწარმოებლებს უწევთ პროცესორების გამოთვლითი სიმძლავრე გაზარდონ მათში ბირთვების რაოდენობის გაზრდით. სწორედ ეს იწვევს მრავალბირთვიანობის კრიზისს, რადგან სტანდარტული მეთოდებით შექმნილი პროგრამები ამ რესურსს ვერ იყენებენ და პროგრამისტებს უწევთ კონკურენტული პროგრამების შექმნა, რაც დიდ სირთულეებთან არის დაკავშირებული.

საინტერესოა ამ სირთულეების გამომწვევი მიზეზები. მათი დიდი ნაწილი მოცემული გრაფიკის მეორე, ხშირად უყურადღებოდ დარჩენილი ნაწილიდან გამომდინარეობს. გრაფიკის ამ ნაწილიდან ჩანს რომ 30 წლის განმავლობაში ტრანზისტორების რაოდენობის ზრდასთან ერთად პროცესორების სიჩქარეც იგივე თანაფარდობით იზრდებოდა, მაგრამ რა კავშირშია ტრანზისტორების რაოდენობა და პროცესორების სისწრაფე? პასუხი არის პარალელიზმი. ტრანზისტორების რაოდენობის ზრდასთან ერთად პროცესორებს უფრო მეტი

და მეტი სამუშაოს პარალელურად შესრულების საშუალება მიეცათ. ეს მოხდა როგორც ბიტურ დონეზე, ასევე ბრძანებების დონეზე. 32 ბიტიანი პროცესორი 8 ბიტიან პროცესორზე სწრაფია რადგან მას შეუძლია, მაგალითად, 32 ბიტიანი რიცხვების შესაკრებად მათი 4 ბიტიან ფრაგმენტებად დაყოფა, შეკრების ოპერაციის პარალელურად განხორციელება თითოეულ ფრაგმენტზე და შედეგად ამ რიცხვების ერთი ოპერაციით შეკრება. 8 ბიტიან პროცესორს კი იგივე სამუშაოსთვის რამდენიმე 8 ბიტიანი ოპერაციის განხორციელება დასჭირდება. ბრძანებების დონეზე თანამედროვე პროცესორები მრავალ სხვადასხვა პარალელურ მეთოდს მიმართავენ ოპერაციების შესრულების სისწრაფის გასაზრდელად, როგორიცაა, მაგალითად, pipelining-ი, არათანმიმდევრული შესრულება და სპეკულაციური შესრულება (იხ [4], [5]). სტანდარტული მეთოდებით პროგრამირებისას ეს პარალელური ოპტიმიზაციები, რის ხარჯზეც პროცესორების სისწრაფე იზრდებოდა წლების განმავლობაში დაფარულია პროგრამისტებისგან, რადგან პროცესორები ოპერაციების თანმიმდევრულად (sequentially) შესრულების ილუზიას ქმნიან. კონკურენტული პროგრამირებისას კი პროცესორებს აღარ შეუძლიათ ამ შიდა პარალელიზმის დამალვა და პროგრამისტებს უწევთ მისი გათვალისწინება. სწორედ ეს არის ერთერთი მნიშვნელოვანი მიზეზი იმისა თუ რატომ არის კონკურენტული პროგრამირება რთული.

პრობლემის თვალსაჩინოდ დასანახად განვიხილოთ მარტივი Java პროგრამა.

```
public class ConcurrencyIsHard {

    public static int answer;
    public static boolean answerIsReady;

    public static Thread t1 = new Thread() {
        public void run() {
            answer = 15;
            answerIsReady = true;
        }
    };

    public static Thread t2 = new Thread() {
        public void run() {
            if(answerIsReady) {
                System.out.println("Answer to the equation is: " + answer);
            } else {
                System.out.println("We are still computing the answer");
            }
        }
    };

    public static void main(String[] args) throws InterruptedException {
        t1.start(); t2.start();
        t1.join(); t2.join();
    }
}
```

რას დაბეჭდავს ეს პროგრამა? იმის მიხედვით თუ რა თანმიმდევრობით შესრულდება ბრძანებები, დაიბეჭდება “Answer to the equation is 15” ან “We are still computing the answer”, მაგრამ კონკურენტული პროგრამირების ერთერთი სირთულე იმაში მდგომარეობს რომ არსებობს მესამე ვარიანტიც - პროგრამამ შეიძლება დაბეჭდოს “Answer to the equation is 0”. ეს იმ შემთხვევაში მოხდება თუ ჯერ answerIsReady-ის მინიჭება მოხდება და მერე answer-ის, ანუ

თუ პროგრამის შესრულებისას ბრძანებების თანმიმდევრობა გადანაცვლდება. ამის მიზეზი შეიძლება იყოს კომპილერი, ვირტუალური მანქანა ან ჰარდვეარი. კომპილერს აქვს უფლება სტატიკური ოპტიმიზაციის განსახორციელებლად გადაალაგოს ბრძანებები, ვირტუალურ მანქანას აქვს უფლება დინამიკური ოპტიმიზაციის განსახორციელებლად გადაალაგოს ბრძანებები, იგივე უფლება აქვს ჰარდვეარსაც.

განვიხილოთ კიდევ ერთი მაგალითი. შევცვალოთ run მეთოდი შემდეგნაირად.

```
public void run() {
    while(!answerIsReady);
    System.out.println("Answer to the equation is: " + answer);
}
```

ამ შემთხვევაში პროგრამამ შეიძლება არასოდეს დაამთავროს მუშაობა, რადგან ერთი დინების (Thread) მიერ მინიჭებული მნიშვნელობა, answerIsReady = true, შეიძლება ვერასოდეს დაინახოს მეორე დინებამ.

როგორც მაგალითებიდან ჩანს, საკმაოდ ძნელია შეცდომებისგან თავისუფალი კონკურენტული პროგრამის შექმნა. ამას ისიც ემატება რომ არ არსებობს ეფექტური ავტომატური ტესტირების მექანიზმი. რადგან კონკურენტული პროგრამა არის არადეტერმინისტული, მან შეიძლება მავალჯერ გაიაროს ტესტირება, მაგრამ მომდევნო ცდაზე, იგივე შემავალი მონაცემებისთვის არასწორად იმუშაოს.

2. სირთულეების დაძლევა ფუნქციონალური პროგრამირების საშუალებით

პრობლემების უდიდესი ნაწილი მრავალ ბირთვიან პროცესორებზე პროგრამირებისას იქმნება მხოლოდ გაზიარებული ცვალებადი მდგომარეობის (shared mutable state) არსებობის შემთხვევაში, ანუ მაშინ როდესაც რამდენიმე დინება ერთი და იგივე ცვლადს იყენებს და ამ ცვლადის შიგთავსი პერიოდულად იცვლება რომელიმე დინების მიერ. თუნდაც წინა პარაგრაფში მოყვანილ მაგალითებში, ერთი დინება მეორე დინების მიერ შეცვლილ ცვლადებზე რომ არ ყოფილიყო დამოკიდებული, მაშინ პროცესორის ოპერაციებში დამალული ოპტიმიზაციებიც არ შეგვიქმნიდა პრობლემებს.

ტრადიციული, პოპულარული პროგრამირების ენები, როგორებიცაა Java და C++-ი, არ არიან ამ იდეოლოგიით შემნილნი, ამიტომაც როგორც წესი ამ ენაზე შექმნილი არატრივიალური აპლიკაციები იყენებენ გაზიარებულ ცვალებად მდგომარეობას. შესაბამისად, პროგრამისტები იღებენ მასთან ასოცირებულ რისკს (accept risk) და სხვადასხვა მეთოდებით და დისციპლინით, როგორებიცაა mutex-ი, სინქრონიზაცია და ჩაკეტვების (lock) თანმიმდევრობის კონტროლი, ცდილობენ ამ რისკების შემცირებას. ეს რათქმაუნდა არ არის პრობლემის იდეალური გადაწყვეტა.

ზემოთ აღნიშნული მიზეზების გამო, ბოლო წლებში პოპულარული გახდნენ ის ენები, რომლებიც ამ პრობლემას უფრო საიმედოდ წყვეტენ, მაგალითად, Erlang-ი, Haskell-ი, Go, Scala და Clojure (იხ. [6]). ეს ენები ორი განსხვავებული ფუნდამენტური მიდგომით უზრუნველყოფენ მეტნაკლებად უსაფრთხო მრავალბირთვიან პროგრამირებას. მიმდინარე პარაგრაფში განვიხილავთ ერთერთს ამ მიდგომებიდან - ფუნქციონალურ პროგრამირებას.

მრავალბირთვიანობის კრიზისის დროს პოპულარული გახდნენ ფუნქციონალური პროგრამირების ენები, რადგან მათთვის საფრთხეს არ წარმოადგენს გაზიარებული ცვალებადი მდგომარეობა (იხ. [7]). ამის მიზეზი მარტივია, ფუნქციონალურ პროგრამებს არ აქვთ ცვალებადი მდგომარეობა (mutable state), შესაბამისად მათ არ აქვთ არც გაზიარებული ცვალებადი მდგომარეობისგან გამოწვეული რისკები. იმპერატიული პროგრამირების ენებისგან განსხვავებით, ფუნქციონალური პროგრამა კომპიუტაციის მოდელირებას ახდენს როგორც გამოსახულებების გამოთვლის (evaluation of expressions) პროცესს. ეს გამოსახულებები შედგებიან სუფთა (pure) მათემატიკური ფუნქციებისგან, მათ არ აქვთ გვერდითი ეფექტები, ანუ ფუნქციის მნიშვნელობის დაბრუნების გარდა არაფერს არ ცვლიან პროგრამის მეხსიერებაში. როდესაც პროგრამისტებს პირველად ესმით ფუნქციონალური პროგრამირების შესახებ, ისინი ხშირად სკეპტიკურად განეწყობიან და ფიქრობენ რომ შეუძლებელია არა ტრივიალური აპლიკაციების შექმნა ცვლადების მოდიფიკაციის გარეშე. თუმცა ეს არ შეესაბამება სიმართლეს, რეკურსია საშუალებას აძლევს ასეთ ენას იყოს ტიურინგ სრული (Turing complete). მეტიც, მთელ რიგ შემთხვევებში აპლიკაციების იმპლემენტაცია უფრო მარტივი და კომპაქტურია ფუნქციონალური ენით ვიდრე იმპერატიულით.

კონტრასტი ფუნქციონალურ და იმპერატიულ ენებს შორის და პირველის უპირატესობა მრავალბირთვიან პროცესორებთან მუშაობისას უფრო ნათელი გახდებოდა კომპლექსური მაგალითის მოყვანით, მაგრამ რადგან მიმდინარე ნაშრომის ფორმატი ამის საშუალებას არ იძლევა განვიხილოთ მარტივი მაგალითი. დავწეროთ მეთოდი რომელიც იპოვის სიაში არსებული ელემენტების ჯამს. იმპერატიულ ენა Java-ში ეს შეიძლება გაკეთდეს შემდეგნაირად:

```
public int sum(int[] numbers) {
    int result = 0;
    for(int i = 0; i < numbers.length; i++) {
        result += numbers[i];
    }
    return result;
}
```

რაც ცხადია არ არის ფუნქციონალური პროგრამა რადგან result და i ცვლადები პროგრამის მუშაობის განმავლობაში იცვლიან მნიშვნელობებს.

ფუნქციონალური პროგრამირების ენა Clojure-ში იგივე მეთოდი რეკურსიის გამოყენებით შემდეგ სახეს მიიღებს:

```
(defn sum [numbers]
    (if (empty? numbers)
        0
        (+ (first numbers) (sum (rest numbers)))))
```

როგორც ვხედავთ result ცვლადის და for ციკლის ფუნქციას, ორივეს, რეკურსია ასრულებს. შესაბამისად პროგრამაში აღარ გვაქვს ცვალებადი მდგომარეობა (mutable state).

აღნიშნული მეთოდი შეგვიძლია გავამარტივოთ ფუნქციონალური პროგრამირებისთვის დამახასიათებელი სამი ფუნდამენტური ფუნქციიდან (filter, map, reduce) ერთერთის, reduce-ს, გამოყენებით. ამ შემთხვევაში ფუნქცია მიიღებს შემდეგ სახეს:

```
(defn sum [numbers]
```

```
    (reduce + 0 numbers))
```

ამჟამად ორივე ენაზე დაწერილი იმპლემენტაცია არის თანმიმდევრული (sequential). შევეცადოთ მათ მოდიფიკაციას რათა მაქსიმალურად გამოიყენონ მრავალბირთვიანი პროცესორის რესურსები.

Clojure-ში ეს ძალიან მარტივია. უბრალოდ შევცვალოთ reduce ფუნქცია მისი პარალელური ეკვივალენტით, fold-ით.

```
(defn sum [numbers]
    (clojure.core.reducers/fold + 0 numbers))
```

ამ შემთხვევაში ბიბლიოთეკა თავად უზრუნველყოფს შესრულებული სამუშაოს სხვადასხვა დინებებისთვის დანაწილებას. მიზეზი იმისა თუ რატომ არსებობს reduce ფუნქციისთვის, ან თუნდაც map და filter ფუნქციებისთვის, ავტომატური, პარალელური ვერსიები, არის თვითონ ამ ფუნქციების ბუნება. ისინი მოქმედებენ სიებზე და ასრულებენ თითოეულ ელემენტზე იმ ოპერაციებს, რომლებიც სრულიად დამოუკიდებლები არიან სხვა ელემენტებისგან ან მათზე ჩატარებული ოპერაციებისგან, შესაბამისად რომელი დინება და როდის ჩაატარებს ამ დამოუკიდებელ ოპერაციას რომელიმე ელემენტზე აღარ არის მნიშვნელოვანი. ამიტომაც თუ გამოვიყენებთ ფუნქციონალურ პროგრამირებას სხვადასხვა ამოცანების გადასაჭრელად და შესაბამისად მათ მოდელირებას გავაკეთებთ map, reduce, filter და სხვა ფუნქციების საშუალებით, მაშინ საშუალება გვექნება ჩვენი პროგრამა ვამუშაოთ მრავალ ბირთვზე ავტომატურად, გაზიარებული ცვალებადი მდგომარეობის და შესაბამისად მასთან დაკავშირებული რისკების გარეშე. ასევე ამ პროცესის ავტომატიზაცია და ბიბლიოთეკის შემქმნელებზე გადაბარება დაგვიცავს ადამიანური შეცდომებისგან და კიდევ უფრო შეამცირებს რისკს.

იგივე პროგრამა Java-ში გამოიყურება შემდეგნაირად:

```
public int sum(int[] numbers) throws ExecutionException, InterruptedException {
    int result = 0;
    int numberOfThreads = Runtime.getRuntime().availableProcessors();
    List<Future<Integer>> results = new ArrayList<>();
    ExecutorService service = null;
    try {
        service = Executors.newFixedThreadPool(numberOfThreads - 1);
        int blockSize = numbers.length / numberOfThreads;
        int startIndex = 0;
        int endIndex = blockSize;
        for(int i = 0; i < numberOfThreads - 1; i++) {
            results.add(service.submit(new ComputeSum(numbers, startIndex,
endIndex)));
            startIndex = endIndex;
            endIndex += blockSize;
        }
        for(int i = startIndex; i < numbers.length; i++) {
            result += numbers[i];
        }
        for(Future<Integer> fractionOfResult : results) {
            result += fractionOfResult.get();
        }
    }
    finally {
        if(service != null) {
            service.shutdown();
```

```
        }
    }
    return result;
}

public static class ComputeSum implements Callable<Integer> {

    public int[] numbers;
    public int startIndex;
    public int endIndex;

    public ComputeSum(int[] numbers, int startIndex, int endIndex) {
        this.numbers = numbers;
        this.startIndex = startIndex;
        this.endIndex = endIndex;
    }

    @Override
    public Integer call() {
        int result = 0;
        for(int i = startIndex; i < endIndex; i++) {
            result += numbers[i];
        }
        return result;
    }

}
```

ეს ბევრად უფრო მოცულობითი და კომპლექსური კოდია. ამასთანავე, Thread-ების გამოყენება პროგრამას არადეტერმინისტულად აქცევს, რაც თავისმხრივ შეცდომების დაშვების რისკთან არის დაკავშირებული.

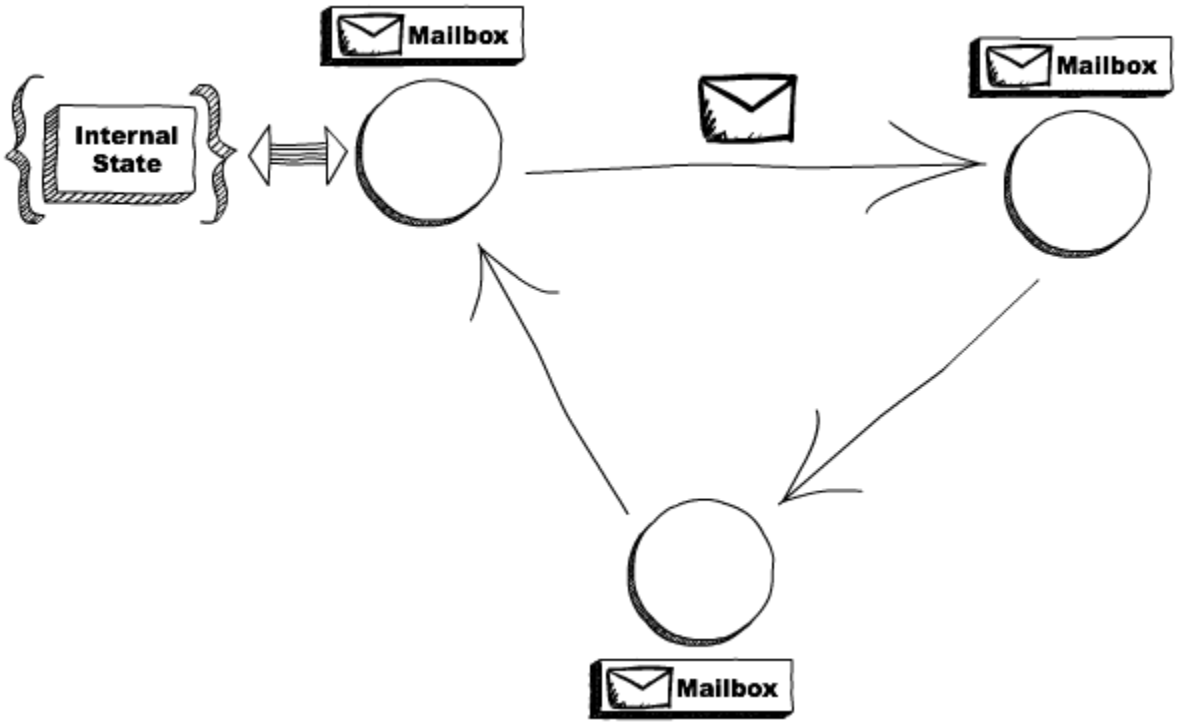
სისრულისთვის უნდა აღინიშნოს, რომ ბოლო წლებში ფუნქციონალური პროგრამირების ეფექტურობის გამო (მრავალბირთვიან პროცესორებთან მუშაობისას) მრავალ იმპერატიულ ენას, მაგალითად, C++-ს, Java-ს, C#-ს, დაემატა ფუნქციონალური პროგრამირების ელემენტები, რომელთა შორის არიან map, filter, reduce და მათი პარალელური ვერსიები. შესაბამისად, დღესდღეისობით Java-შიც შესაძლებელია ზემოთ ხსენებული ამოცანის მოკლედ და უსაფრთხოთ გადაწყვეტა მასში ჩაშენებული ფუნქციონალური ელემენტების გამოყენებით.

3. სირთულეების დაძლევა აქტორ მოდელის (Actor Model) საშუალებით

წინა პარაგრაფში განხილულ იქნა ფუნქციონალური პროგრამირება, რომელიც გაზიარებულ ცვალებად მდგომარეობასთან დაკავშირებულ რისკს, ზოგადად ცვალებადი მდგომარეობის ხმარებიდან ამოღებით გვაცილებდა თავიდან. მაგრამ, ასევე არსებობს მეორე ტიპის მიდგომა, რომელშიც გვაქვს ცვალებადი მდგომარეობა, თუმცა არ შეგვიძლია მისი გაზიარება და შესაბამისად არ გვექმნება ზემოთ აღნიშნული საფრთხე. ამ მიდგომის თვალსაჩინო წარმომადგენელია აქტორ მოდელი (იხ. [8]), რომელზეც დაფუძნებულია დღეს უკვე საკმაოდ პოპულარული ენები Erlang-ი და Elixir-ი (იხ. [9]).

აქტორ მოდელში, როგორც მისი დასახელება მიგვანიშნებს მთავარი მოქმედი პირები არიან აქტორები. აქტორი გავს ობიექტს, ობიექტზე ორიენტირებული ენიდან, იმ გაგებით რომ ის ენკაფსულაციას უკეთებს მდგომარეობას (state) და კომუნიკაციას ამყარებს სხვა აქტორებთან შეტყობინებების საშუალებით. თითოეულ აქტორს აქვს თავისი კოდი, რომელსაც ის

თანმიმდევრულად ასრულებს. ასერომ, აქტორ მოდელის მიხედვით შექმნილი პროგრამა შედგება აქტორებისგან, რომლებიც კონკურენტულად ასრულებენ თავთავიანთ ბრძანებებს, აქვთ დამოუკიდებელი მეხსიერება, რომელზეც სხვა აქტორებს არ აქვთ წვდომა და რომლებიც კომუნიკაციისთვის ერთმანეთის საფოსტო ყუთებზე გზავნიან შეტყობინებებს. თითოეულ აქტორს აქვს თავისი საფოსტო ყუთი, რომელზეც სხვა აქტორებს შეუძლიათ შეტყობინების გაგზავნა ადრესატის საიდენტიფიკაციო ნომრის (id) მითითებით. ვიზუალურად გამოზახული აქტორ მოდელი გამოიყურება შემდეგნაირად.



სწორედ ეს აქტორების დამოუკიდებლობა, ენკაფსულაცია, აქრობს ამ მოდელის მიხედვით შექმნილ პროგრამებში გაზიარებული ცვალებადი მდგომარეობიდან გამოწვეულ რისკებს. ეს კი როგორც ვთქვით, კრიტიკული მნიშვნელობისაა და განაპირობებს ამ მოდელის პოპულარულობას.

მნიშვნელოვანია ასევე რომ საფოსტო ყუთები წარმოადგენენ დინებებისგან დაცულ რიგებს (thread safe queue), რაც აქტორს დროის მიხედვითაც დამოუკიდებლს, განცალკევებულს (decoupled) ხდის. რაც იმას ნიშნავს რომ სხვადასხვა აქტორებს შეუძლიათ შეტყობინებები მაშინ გაუგზავნონ რომელიმე სხვა აქტორს როდესაც ადრესატი დაკავებულია რაიმე სხვა ოპერაციის განხორციელებით. რადგან აღნიშნული შეტყობინებები შეინახება რიგში, ადრესატი, გათავისუფლებისთანავე, თანმიმდევრულად წაიკითხავს მიღებულ შეტყობინებებს და მოახდენს შესაბამის რეაგირებას.

ბოლო მნიშვნელოვანი საკითხი რასაც შევეხებით არის ის რომ, აქტორის შექმნა, დინებისგან განსხვავებით (რომელიც საკმაოდ დიდ რესურსს მოიხმარს და რომლის შექმნაც დიდ დროს მოითხოვს) , არის ბევრად იაფი რესურსების მოხმარების მხრივაც და ჩატვირთვის მხრივაც

(startup cost). შესაბამისად, მაგალითად Elixir-ში შექმნილი პროგრამები ათასობით აქტორებს ქმნიან და არ სჭირდებათ საზიარო დინებების (thread pool) რაიმე ეკვივალენტის გამოყენება. ეს კი პროგრამის შექმნის პროცესს ამარტივებს.

საკმაო თეორიული განხილვის შემდეგ, დროა განვიხილოთ მარტივი მაგალითი. კერძოდ პრობლემური შემთხვევა პირველი პარაგრაფიდან, სადაც გვინდოდა ერთ დინებას გამოეთვალა განტოლების ფესვი და მეორე დინებას კი ეს ინფორმაცია ეკრანზე დაებეჭდა. აღნიშნული პროგრამა Elixir-ში შემდეგნაირად გამოიყურება.

```
defmodule Main do

    def main do
        reporterId = spawn(&Main.reporter/0)
        Process.register(reporterId, :reporter)
        spawn(&Main.solver/0)
    end

    def solver do
        answer = 15
        reporterId = Process.whereis(:reporter)
        send(reporterId, {:answerIsReady, answer})
    end

    def reporter do
        receive do
            {:answerIsReady, answer} -> IO.puts("Answer to the equation is:
#{answer}")
        end
    end

end
```

როგორც ვხედავთ, პროგრამაში გვაქვს ორი აქტორი, ერთი ითვლის განტოლების ფესვს და გამოთვლისთანავე უგზავნის ამ ინფორმაციის შემცველ შეტყობინებას მეორე აქტორს, რომელიც ამავდროულად ელოდება შეტყობინებას და მიღების თანავე ბეჭდავს მას ეკრანზე.

შეჯამების სახით შეგვიძლია ვთქვათ რომ აქტორ მოდელი არის ძალიან მოსახერხებელი, რადგან მასში შემავალი თითოეული აქტორი მოქმედებს სხვებთან ერთად თანადროულად, მაგრამ მათ არ აქვთ ერთმანეთის მიერ გამოყოფილ და მოხმარებულ მეხსიერებასთან წვდომა. თითოეულის კოდი სრულდება თანმიმდევრულად (sequentially) და შესაბამისად პროგრამისტს კონკურენციაზე ფიქრი უწევს მხოლოდ შეტყობინებათა დინების განხილვის დროს. ეს უდიდესი დახმარებაა პროგრამისტისთვის, რადგან აღნიშნული ფაქტი საშუალებას გვაძლევს აქტორები გავტესტოთ იზოლირებულად (შესაძლო შეტყობინებების ხელოვნურად გადაგზავნით), რითიც შეგვიძლია დიდი ალბათობით დავრწმუნდეთ რომ თითოეული გატესტილი აქტორის კოდი შეცდომებს არ შეიცავს. ასეთ შემთხვევაში, პროგრამაში თუ მაინც აღმოჩნდება კონკურენტულობასთან დაკავშირებული შეცდომა, უკვე გვეცოდინება სადაც არის ის დამალული - აქტორებს შორის, შეტყობინებების დინებებში.

სისრულისთვის უნდა აღინიშნოს რომ მართალია აქტორ მოდელით შექმნილი პროგრამა მარტივი გასატესტია ვიდრე დინებებით და საკეტებით (lock) შექმნილი პროგრამა, მაგრამ მასში მაინც შეიძლება მოხდეს ზოგიერთი კონკურენტულობასთან დაკავშირებული

პრობლემა, როგორიცაა მაგალითად deadlock-ი ან აქტორებისთვის უნიკალური - საფოსტო ყუთის გადავსება.

4. დასკვნა

როგორც ნაშრომიდან ჩანს, მრავალბირთვიანი პროგრამირება დიდ სირთულეებთან და რთულად აღმოსაჩენ შეცდომებთან არის დაკავშირებული, რისი მთავარი მიზეზიც გაზიარებული ცვალებადი მდგომარეობაა. პოპულარული, ტრადიციული პროგრამირების მეთოდები, რომლებიც ერთ ბირთვიანი პროცესორების ერაში ჩამოყალიბდნენ, იყენებენ გაზიარებულ ცვალებად მდგომარეობას და შესაბამისად მნიშვნელოვნად ართულებენ შეცდომებისგან თავისუფალი კონკურენტული პროგრამების შექმნას. ამიტომაც, მრავალ ბირთვიანი პროცესორების ფართო გავრცელებასთან ერთად, პოპულარული გახდნენ ის პროგრამირების მეთოდები რომლებიც ამარტივებენ და მეტნაკლებად უსაფრთხოს ხდიან ასეთ პროცესორებზე მომუშავე პროგრამების შექმნას. ნაშრომში განხილულ იქნა ასეთი ორი ძირითადი მიდგომა. ფუნქციონალური პროგრამირება, რომელიც გაზიარებული ცვალებადი მდგომარეობიდან გამომდინარე რისკს ცვალებადი მდგომარეობის ხმარებიდან ამოღებით გვარიდებს და მეორე მიდგომა, რომელიც იგივე რისკს გაზიარებული მდგომარეობის ანულირებით გვაცილებს თავიდან.

ლიტერატურა

1. Paul Butcher. Seven Concurrency Models in Seven Weeks: When Threads Unravel (1st edition). 2014.
2. Saman Amarasinghe. The Looming Software Crisis due to the Multicore Menace. See http://groups.csail.mit.edu/commit/papers/06/MulticoreMenace.pdf
3. Moore's law. See https://en.wikipedia.org/wiki/Moore%27s_law
4. Out-of-order execution. See https://en.wikipedia.org/wiki/Out-of-order_execution
5. Speculative execution. See https://en.wikipedia.org/wiki/Speculative_execution
6. Clojure. See https://clojure.org/
7. Functional programming. See https://en.wikipedia.org/wiki/Functional_programming
8. Carl Hewitt. Actor Model of Computation. 2010.
9. Elixir. See https://elixir-lang.org/