

Sveučilište u Zagrebu
Prirodoslovno-matematički fakultet
Matematički odsjek

Nikola Kašnar i Magdalena Potočnjak

Filozofi koji ručaju

Projekt iz Distribuiranih procesa

Profesor: Robert Manger

Zagreb, lipanj 2024.

Sadržaj

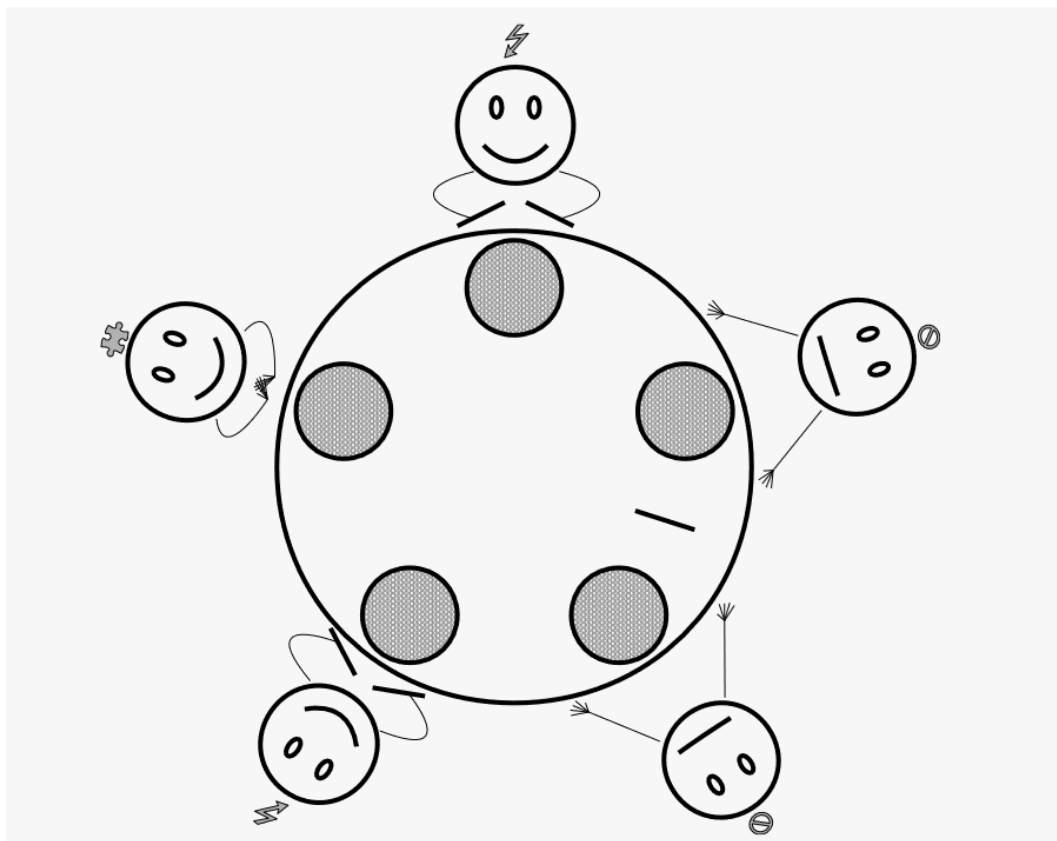
1	Uvod	2
1.1	Detalji problema	2
2	Paralelni algoritam	4
3	Distribuirani algoritam	8
4	Literatura	12

1 Uvod

Problem filozofa koji ručaju predstavlja poopćenje problema međusobnog isključivanja. Osim što imamo više procesa odnosno dretvi, također imamo i više dijeljenih resursa. Jedan proces (dretva) u svojoj kritičnoj sekciji koristi samo neke od resursa. Jedan resurs potreban je samo nekim procesima (dretvama). Više procesa (dretvi) smije istovremeno izvoditi svoje kritične sekcije ako nisu u konfliktu u pogledu korištenja resursa. U projektu ćemo obraditi dva algoritma za rješavanje problema filozofa koji ručaju: jedan za paralelni sustav, drugi za distribuirani sustav. Oba algoritma ćemo implementirati i testirati na računalu.

1.1 Detalji problema

Filozofi koji ručaju je dosta popularan problem te se većinom koristi kao najosnovniji problem prikaza međusobnog isključivanja pri korištenju dijeljenog resursa, tj. ulaska u kritičnu sekciju. Osnovna ideja je da imamo N filozofa od kojih svaki posjeduje jedan štapić. Ono što filozof može raditi je razmišljati, biti gladan i jesti. Jedan filozof može neovisno o drugima razmišljati i biti gladan, ali može jesti samo ako ima dva štapića, što znači da bi mogao jesti, mora posuditi štapić od nekog drugog filozofa. Situaciju možemo zamisliti ovako:



Slika 1: Filozofi koji ručaju

Na slici možemo vidjeti 5 filozofa od kojih 2 jedu(gore i dolje lijevo), jedan razmišlja(lijevo u sredini), te su preostala dva gladna i čekaju da se oslobode neki štapići.

Slijed događaja je takav da na početku svi filozofi razmišljaju, nakon nekog vremena ogladne te čim uđu u tu "fazu" traže vilice kako bi mogli jesti. Nakon što se najedu, ostave štapiće(naravno prije toga ih operu :)), te opet krenu sa razmišljanjem.

Jedno očito rješenje je da napravimo petlju gdje svi filozofi, pri čemu je jedan filozof ustvari jedan proces, čekaju sve dok se ne oslobode dva štapića kako bi mogli krenuti jesti. Nažalost to neće funkcionirati, jer se može dogoditi da svi filozofi u isto vrijeme krenu sa čekanjem, ali nikad neće dočekati drugi štapić, tj. dogodio se deadlock. Razlog je taj što svaki filozof ima točno jedan štapić koju "čuva" te zbog toga nijedan drugi filozof neće imati drugi štapić za uzeti.

U sljedeća dva poglavlja ćemo pokazati kako riješiti ovaj problem sa dva različita algoritma pri čemu je jedan paralelan, a drugi distribuirani.

2 Paralelni algoritam

Za paralelnu implementaciju smo koristili binarne semafore pa ćemo prvo objasniti kako oni funkcioniraju. Ovo je klasa semafora:

```
1 public class BinarySemaphore {
2     boolean value;
3     BinarySemaphore(boolean initialValue) {
4         value = initialValue;
5     }
6     public synchronized void P() {
7         while (value == false)
8             Util.myWait(this); // Čeka u redu
9         value = false;
10    }
11    public synchronized void V() {
12        value = true;
13        notify();
14    }
15 }
16
```

Slika 2: Klasa BinarySemaphore

Binarni semafori mogu poprimiti jednu od dvije vrijednosti: 0(false) ili 1(true). U slučaju da je vrijednost jednaka 0, semafor je neprolazan, a u slučaju 1, prolazan. Tu vrijednost spremamo u varijabli `value`.

Konstruktor `BinarySemaphore()` nam služi za inicijalizaciju vrijednosti `value` na jednu od te dvije vrijednosti.

Metoda `P()` nam služi za postavljanje trenutne dretve u red čekanja. To je napravljeno u `while` petlji tako da ta dretva čeka u redu sve dok semafor ne postane prolazan. `synchronized` u nazivu nam osigurava da samo jedan dretva u tom trenutku može izvršavati ovu metodu na danom semaforu. Kada semafor postane prolazan, izađe iz petlje te postavi `value` na `false`, čime označava ostalim dretvama da je on ušao u kritičnu sekciju. Funkciju `myWait` i klasu `Utils` ću objasniti malo kasnije.

Metoda `V()` se koristi za oslobađanje semafora kako bi ga druge dretve mogle koristiti. Ona ustvari postavlja semafor na `true`, te obavijesti ostale dretve sa `notify()`. Ta funkcija probudi jednu od dretva koje čekaju na semafor.

Sljedeće imamo klasu `Util`:

```
1 public class Util {
2     public static void myWait(Object obj) {
3         try {
4             obj.wait();
5         } catch (InterruptedException e) {
6             e.printStackTrace();
7         }
8     }
9 }
10
```

Slika 3: Klasa Util

Tu nam se nalazi funkcija `myWait()` koju smo koristili u prošlom dijelu. Njena jedina funkcionalnost je da dobivenu dretvu stavi u stanje čekanja. Ona će se probuditi nakon što dobije obavijest funkcijom `notify()`.

Sada prelazimo na glavni dio koda, a to su sami filozofi. Sljedeće tri klase ću odmah sve zapisati jer su dosta međusobno povezane, a zatim slijedi objašnjenje.

```
1 interface Resource {
2     public void acquire(int i);
3     public void release(int i);
4 }
5
```

Slika 4: Klasa Resource

```
1 class DiningPhilosopher implements Resource {
2     int n = 0;
3     BinarySemaphore[] stick = null;
4     public DiningPhilosopher(int initN) {
5         n = initN;
6         stick = new BinarySemaphore[n];
7         for (int i = 0; i < n; i++) {
8             stick[i] = new BinarySemaphore(true);
9         }
10    }
11    public void acquire(int i) {
12        stick[i].P();
13        stick[(i + 1) % n].P();
14    }
15    public void release(int i) {
16        stick[i].V();
17        stick[(i + 1) % n].V();
18    }
19    public static void main(String[] args) {
20        DiningPhilosopher dp = new DiningPhilosopher(5);
21        for (int i = 0; i < 5; i++)
22            new Philosopher(i, dp);
23    }
24 }
25
```

Slika 5: Klasa DiningPhilosopher

```

1      class Philosopher implements Runnable {
2          int id = 0;
3          Resource r = null;
4          public Philosopher(int initId, Resource initr) {
5              id = initId;
6              r = initr;
7              new Thread(this).start();
8          }
9          public void run() {
10             while (true) {
11                 try {
12                     System.out.println("Filozof " + id + " razmislja");
13                     Thread.sleep(3000);
14                     System.out.println("Filozof " + id + " je gladan");
15                     r.acquire(id);
16                     System.out.println("Filozof " + id + " jede");
17                     Thread.sleep(4000);
18                     r.release(id);
19                 } catch (InterruptedException e) {
20                     return;
21                 }
22             }
23         }
24     }
25

```

Slika 6: Klasa Philosopher

Prvo imamo jednostavno sučelje `Resource`. To je naš dijeljeni resurs. Ona sadrži dvije funkcije koje klasa koja nasljeđuje klasu `Resource`, mora implementirati. U našem slučaju to će bit klasa `DiningPhilosopher`. Funkcija `acquire()` će nam služiti za sakupljanje štapića, a funkcija `release()` za ispuštanje.

Naša glavna klasa je klasa `DiningPhilosopher`. Ona sadrži dvije vrijednosti, a to su broj filozofa, a i štapića, `n` (čija je početna vrijednost jednaka 0), te polje binarnih semafora nazvano `fork` čiju smo implementaciju naveli ranije (ono je na početku prazno). U ovom slučaju svaki binarni semafor nam predstavlja jedan štapić.

Konstruktor `Philosopher(int initId, Resource initr)` za ulaznu vrijednost prima broj filozofa u problemu te spremi tu vrijednost u `n`. Zatim za svakog filozofa napravimo i štapić, tj. u polje `forks` spremimo po jedan binarni semafor za svaki štapić te ga postavimo na `true` što označava da je taj štapić slobodan (drugim riječima, svi semafori su prolazni).

Implementacija funkcije `acquire(int i)` prima id filozofa za kojeg dohvaća štapiće. Ta funkcija tada zauzme `i`-ti i $((i+1)\%n)$ -ti štapić koristeći funkciju `P()` iz klase `BinarySemaphore`, tj. štapić sa desne i lijeve strane filozofa. Dijelimo modulo `n` u slučaju da filozof mora uzeti prvi i zadnji štapić u polju.

Implementacija funkcije `release(int i)` je veoma slična prethodnoj, samo što ovdje za dani id oslobađamo `i`-ti i $((i+1)\%n)$ -ti štapić koristeći funkciju `V()` iz klase `BinarySemaphore`.

Za kraj klase imam glavnu funkciju koja pokreće problem filozofa. Za odeđeni `n` (na slici sa klasom `n=5`) stvorimo `n` filozofa pomoću klase `Philosopher` pri čemu svakom filozofu šaljemo njegov id i referencu na dijeljeni resurs.

Na kraju još prolazimo kroz klasu `Philosopher()` koju koristimo za svakog od `n` filozofa. Na

početku klase inicijaliziramo `id` svakog filozofa te referencu na objekt `Resource` koji je postavljen na `null` (u našem slučaju resurs je `DiningPhilosopher`).

Konstruktor `Philosopher(int initId, Resource initr)` sprema `initId` kao `id` filozofa i u `r` referencu na dijeljeni resurs. Također stvorimo i novu dretvu koja pokreće `run` metodu. Time smo ustvari definirali svakog filozofa kao jednu dretvu.

Metoda `run` u sebi ima `while` petlju u kojoj prolazimo kroz cikluse razmišljanja, gladi i jedenja. `Thread.sleep(3000)` simulira razmišljanje filozofa, tu možemo namjestiti dulji ili kraći period ako želimo. Kada filozof predstane s razmišljanjem, pokuša dohvatiti štapiće sa `r.acquire(id)`. Tek nakon što ih je dohvatio kreće za jedenjem. Sa `Thread.sleep(4000)` simuliramo vrijeme jedenja, te kada prođe to vrijeme oslobodimo štapiće sa `r.release(id)`.

Glavna ideja implementacije je da svi filozofi koriste klasu `DiningPhilosophers` te njene metode kako bi izbjegli istovremeno uzimanje jedne vilice.

Sami kod sa malo više komentara možete vidjeti na našem *GitHub* repozitoriju.

Cijeli algoritam se pokreće tako da se prvo sve kompajlira (npr. sa naredbom `"javac *"` u direktoriju), a zatim pokrene klasa `DiningPhilosopher` sa naredbom `"java DiningPhilosopher"`.

Također prilažemo i primjer ispisa programa sa 5 filozofa:

```
$ java DiningPhilosopher
Filozof 3 razmislja
Filozof 4 razmislja
Filozof 1 razmislja
Filozof 2 razmislja
Filozof 0 razmislja
Filozof 0 je gladan
Filozof 2 je gladan
Filozof 1 je gladan
Filozof 2 jede
Filozof 4 je gladan
Filozof 3 je gladan
Filozof 0 jede
Filozof 0 razmislja
Filozof 4 jede
Filozof 2 razmislja
Filozof 1 jede
Filozof 0 je gladan
Filozof 2 je gladan
Filozof 1 razmislja
Filozof 4 razmislja
Filozof 3 jede
Filozof 0 jede
```

Slika 7: Ispis programa

3 Distribuirani algoritam

Temelj na kojem smo gradili algoritam je izbor vođe u prstenu. Algoritam Changa i Robertsa je najpoznatiji za taj problem. Prethodno je već opisana problematika koju rješavamo, a sada dodatno pretpostavimo da svaki od petoro filozofa imaju vrijeme koje im je potrebno da pojedu jelo. Na temelju toga odabiremo vođu tako da prednost ima onaj kojem je potrebno najmanje vremena za pojesti svoje jelo.

Ukoliko je filozof (nakon što je razmišljao i rekao da je gladan) izabran za vođu, uzima oba štapića te kreće jesti. Nakon što pojede, kako bismo izbjegli njegovo prioritiziranje, dodajemo mu maksimalno vrijeme filozofa koji još nije jeo.

Za početak uvodimo sučelje *Election* koje predviđa implementiranje 2 metode:

- *startElection()* - može ju pokrenuti bilo koji filozof čime informira ostale da je gladan
- *getLeader()* - preko nje saznaje koji filozof je trenutno vođa

```
1 public interface Election extends MsgHandler {  
2     void startElection();  
3     int getLeader(); //blocks till the leader is known  
4 }  
5
```

Slika 8: Klasa Election

Zatim nam je potrebna klasa koja će implementirati to sučelje:

```

1      public class RingLeader extends Process implements Election {
2          int number;
3          int leaderId = -1;
4          int next, prev, max;
5          boolean awake = false;
6          boolean blocked = false;
7
8          public RingLeader(Linker initComm, int number) {
9              super(initComm);
10             this.number = number;
11             next = (myId + 1) % N;
12             prev = myId > 0 ? (myId - 1) : N-1;
13             max = -1;
14         }
15
16         public synchronized int getLeader(){
17             while (leaderId == -1) myWait();
18             return leaderId;
19         }
20
21         public synchronized void handleMsg(Msg m, int src, String tag) {
22             int j = m.getMessageInt(); // get the number
23             if (tag.equals("election")) {
24                 if (j < number || blocked)
25                     sendMsg(next, "election", j); // forward the message
26                 else if (j == number) // I won!
27                     sendMsg(next, "leader", myId);
28                 //else if ((j > number) && !awake) startElection();
29             } else if (tag.equals("leader")) {
30                 leaderId = j;
31                 notify();
32                 if (j != myId) {
33                     sendMsg(leaderId, "time", number);
34                     sendMsg(next, "leader", j);
35                     if(j != next && j != prev) {
36                         blocked = false;
37                         startElection();
38                     }
39                     else blocked = true;
40                 } else {
41                     number += max;
42                     awake = false;
43                     Util.println("number = " + number);
44                 }
45             } else {
46                 if ( j > max ) max = j;
47             }
48         }
49
50         public synchronized void startElection() {
51             awake = true;
52             sendMsg(next, "election", number);
53         }
54     }
55
56     public synchronized void startElection() {
57         awake = true;
58         sendMsg(next, "election", number);
59     }
60

```

Slika 9: Klasa RingLeader
9

Klasa *RingLeader* proširuje klasu *Process* pa mora uz prethodno navedene implementirati i metodu *handleMsg()*.

Varijablu *number* koristimo za identifikator pri izboru vođe, *leaderId* nam govori koji je trenutno proces vođa, a ostale varijable su pomoćne za funkciju *handleMsg()*.

Funkcija *getLeader* će biti blokirana sve do trenutka kada je vođa odabran. Funkcija *startElection* brine se za pokretanje ponovnog izbora vođe.

U *handleMsg()* najprije spremamo numerički sadržaj poruke u varijablu *j*. Ovisno o vrijednosti parametra *tag* funkcija radi sljedeće:

- "election" - prosljeđuje poruku za izbor vođe ukoliko ima prednost nad trenutnim procesom ili ako je trenutni proces blokirana odnosno ne može jesti jer je barem jedan štapić zauzet
- "leader" - postavlja varijablu *leaderId* te o tome obavještava ostale filozofe sve dok ne dođe do onog koji je postao vođom, uz poruke o tome tko je vođa, šalje novoproglašeno vođi poruku o svom vremenu potrebnom za jelo temeljem čega će vođa izračunati svoje novo vrijeme, također, kreiraju se paralelni procesi za odabir novog vođe kako bi se maksimizirala učinkovita upotreba resursa što uzrokuje izvođenje u beskonačnost
- "time" - računa maksimum dobivenih vremena koji će pribrojiti varijabli *number*

```
1 public class Philosopher {
2     public static void main(String[] args) throws Exception {
3         int myId = Integer.parseInt(args[1]);
4         int numProc = Integer.parseInt(args[2]);
5         Linker comm = new Linker(args[0], myId, numProc);
6         Election g = new RingLeader(comm, Integer.parseInt(args[3]));
7         for (int i = 0; i < numProc; i++)
8             if (i != myId){
9                 new ListenerThread(i, g).start(); Thread.sleep(3000);
10                //thinking g.startElection();int leader = g.getLeader();
11                System.out.println("The leader is " + leader);
12            }
13     }
14 }
```

Slika 10: Klasa *Philosopher*

Na kraju imamo klasu *Philosopher* u kojoj se nalazi *main* funkcija. Program u komandnoj liniji pozivamo, nakon što smo sve potrebne klase kompilirali i pokrenuli *NameServer*, na sljedeći način:

```
1 $ java Philosopher <bazno-ime> <i> <N> <vrijeme jedenja>$
2
```

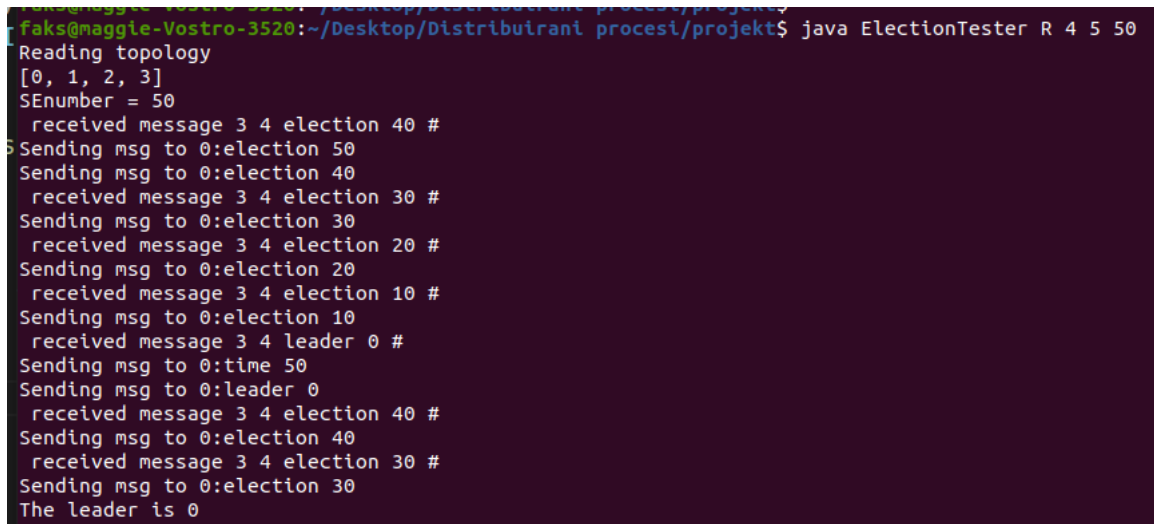
Slika 11: Poziv programa

Dakle, prvi argument nam inicijalizira zajedničko ime za klasu *Linker*, drugi argument nam govori koji redni broj je proces kojeg upravo pokrećemo, treći argument je ukupan broj procesa koje želimo imati, a zadnji argument je vremenska dimenzija pojedinog procesa.

Početno, pokrećemo dretve za slušanje te nakon što su sve uspješno pokrenute, započinje algoritam u kojem najprije svaki filozof razmišlja, zatim ogladni pa poziva funkciju *startElection* i nakon toga ispisuje trenutnog vođu.

Svaki od procesa ima drugačiji ispis pa ovdje stavljamo samo jedan od njih za primjer s 5 filozofa koji redom imaju vremena 10, 20, 30, 40 i 50.

Print screen:



```
faks@maggie-Vostro-3520:~/Desktop/Distribuirani procesi/projekt$ java ElectionTester R 4 5 50
Reading topology
[0, 1, 2, 3]
SNumber = 50
received message 3 4 election 40 #
Sending msg to 0:election 50
Sending msg to 0:election 40
received message 3 4 election 30 #
Sending msg to 0:election 30
received message 3 4 election 20 #
Sending msg to 0:election 20
received message 3 4 election 10 #
Sending msg to 0:election 10
received message 3 4 leader 0 #
Sending msg to 0:time 50
Sending msg to 0:leader 0
received message 3 4 election 40 #
Sending msg to 0:election 40
received message 3 4 election 30 #
Sending msg to 0:election 30
The leader is 0
```

Slika 12: Primjer ispisa za distribuirani program

4 Literatura

- [1] Garg V.K. Concurrent and Distributed Computing in Java . Wiley – IEEE Press, New Jersey, 2004. ISBN-13: 978-0471432302
- [2] Skripta iz distribuiranih procesa (<http://web.studenti.math.pmf.unizg.hr/~manger/dp/>)
- [3] Skripta iz operacijskih sustava (<https://www.zemris.fer.hr/~leonardo/os/math/>)
- [4] Naš Github repozitorij (<https://github.com/NikolaKasnar/Filozofi-koji-rucaju>)