Sveučilište u Zagrebu

Prirodoslovno-matematički fakultet

Tina Marić, Gregor B. Banušić i Mia Filić

Problem međusobnog isključivanja

Zagreb, 2017

Mentor: Robert Manger

Sadržaj

1 Uvod

Na predavanjima, u sklopu cjeline 4, objašnjen je pojam međusobnog isključivanja [3]. Obrađena su 4 algoritma za međusobno isključivanje i navedena njihova svojstva. U sklopu ovoga projekta opisujemo, implementiramo i testiramo još tri dodatna algoritma koji rješavaju isti problem. To su:

1. Singhal-ov algoritam zasnovan na dinamičkoj strukturi podataka,
2. algoritam Lodha & Kshemkalyany koji predstavlja optimizaciju algoritma R & A s predavanja,
3. algoritam Maekawa zasnovan na kvorumu.

2 Algoritmi međusobnog isključivanja zasnovani na dinamičkoj strukturi podataka

Većina algoritama za međusobno isključivanje rade jednako bez obzira na trenutačno stanje sustava u kojemu su pokrenuti. Time su onemogućeni iskoristiti sva svojstva sustava pojedinog trenutka. Postoji mogućnost smanjene efikasnosti.

Ukoliko u sustavu postoje 2 grupe procesa, oni koji često (grupa 1) i oni koji rijetko (grupa 2) ulaze u kritičnu sekciju (CS), povećanje efikasnosti algoritma u odnosu na sustav može se dobiti sljedećim uvjetom:
Procesi koji često ulaze u CS, odobrenje za ulazak u CS traže samo od procesa iste grupe. Ostali procesi traže odobrenje za ulazak u CS od svih procesa.

Jedan algoritam takav algoritam je Singhal-ov algoritam.

2.1 Singhalov algoritam

Singhalov algoritam, nakon pokretanja, uči o stanju sustava. Izvor znanja predstavljaju poruke koje se izmjenjuju unutar sustava. Temeljem znanja o

trenutnom stanju, gradi se dinamička struktura podataka.

Izgradnja algoritma zasnovanog na dinamičkoj strukturi podataka koja prati trenutno stanje sustava ima svoje izazove:

1. Kako efikasno prepoznati koji procesi trenutno zahtjevaju ulaz u CS?
2. Kako projektirati postavljenje zahtjeva za CS za procese iz grupe 2 (manje aktivne procese)?
3. Kako omogućiti prijelaz iz grupe 2 u grupu 1 i obratno?
4. Ukoliko proces ne zahtjeva odobrenje za ulazak u CS od svih procesa, kako osigurati svojstvo *sigurnosti*?
5. Kako osigurati da prikupljanje informacija o trenutnom stanju sustava ne utječe na efikasnost poništavajući dobiveno poboljšanje?

U danjem razmatranju pretpostavljamo kako se distribuirani sustav sastoji od N procesa $P_1, P_2, \ldots, P_N$. Ne postoji gornja ograda na slanje i primanje poruka, ali je ona konačna za svaku poruku. Pretpostavljamo FIFO uređaj na komunikacijske kanale i smatrako kako se poruke ne mogu izgubiti. Također, pretpostavljamo kako procesi rade bez grešaka.

Kako bi se omogućilo korištenje Singhalovog algoritma za međusobno isključivanje, svaki proces P_i čuva treba čuvati dva skupa znanja, R_i i I_i. R_i je skup svih procesa od kojih P_i treba tražiti dopuštenje za ulazak u CS. Skup I_i sadrži sve procese koji traže dopuštenje za ulazak u CS od procesa P_i i imaju zahtjev za CS manjeg prioriteta nego P_i. Zato proces P_i šalje dopuštenje za ulazak u CS tek nakon izlazka iz CS.

Prioritet zahtjeva za kritičnom sekcijom, ostvaruje se pridjeljivanjem vremenskog žiga. Dakle, uz skupove R_i i I_i, zahtjeva se da proces P_i održava logički sat C_i. Koristi se jednostavan Lamportov sat proširen do totalnog uređaja na zahtjeve pomoću vrijednosti identifikatora procesa (kao na predavanjima).

Trenutno stanje, u onosu na zahtjevanje CS-a, proces čuva u 3 lokalne varijable:

1. $zahtjev \in True, False$
2. $u\_CS \in True, False$
3. $mojPrioritet \in True, False$

Varijabla *zahtjev* ima vrijednost $True$ ako i samo ako proces ima postavljen zahtjev za CS. Varijabla u_CS ima vrijednost $True$ ako i samo ako se proces inalazi u CS. Varijabla $mojPrioritet$ ima vrijednost $True$ ako i samo ako se proces ima postavljen zahtjev za CS i on je većeg prioriteta od dolazećeg zahtjeva za CS.

Na početku, skup R_i sadrži procese $P_1, P_2, \ldots, P_{i-1}$, $I_i = P_i$, $C_i = 0$ i $zahtjev = u\_CS = mojPrioritet = False$

Dakle, za niz procesa $P_1, P_2 \ldots, P_N$, interno, proces P_i dijeli niz svih procesaa na 2 podniza: (1) niz procesa od kojih traži dopuštenje, njemu slijeva, i (2) niz procesa koji njega traže dopuštenje za ulazak u CS, njemu zdasna.

$$P_1, P_2, \ldots, P_{i-1} || P_i || P_{i+1}, \ldots, P_N$$

Na ovaj način imamo sljedeće svojstvo: (T) veličina skupa R_i se povećava povećanjem indeksa procesa i.
Ova konfiguracija sustava zato se često naziva *staircase pattern*. Tijekom rada algoritma, održava se *staircase pattern*, ali se redosljed procesa dinamički mijenja. Algoritam teži prema tome da procesi koji češće zahtjevaju CS, imaju manji kardinalitet skupa R, tj. budu što ljevije u nizu.

2.1.1 Opis algoritma

Poruke kojima se traži odobrenje ulaska u CS su tipa *request*. Poruke kojima proces daje odobrenje ulaska u CS su tipa *okay*. Oba tipa poruka sadržavaju

vrijednost logičkog sata pošiljatelja u trenutku slanja poruke.

Slijedi opis rada algoritma upotpunjen pseudokodom na kraju podpoglavlja.
Kada proces P_i želi ući u CS, radi sljedeće:

1. postavlja varijablu $zahtjev$ na $True$,
2. povećava vrijednost logičkog sata za 1,
3. šalje poruke tipa *request* svim procesima iz skupa R_i,
4. čeka da dobije odgovore na postavljeni zahtjev, poruke tipa *okay*, od svih procesa iz R_i,
5. nakon što proces dobije poruku tipa *okay* od svih procesa iz R_i, ulazi u CS.

Slijedi opis akcija procesa P_i, u odnosu na trenutno stanje procesa, u slučaju primitka poruke tipa *request*:

- Kada proces P_i ima $zahtjev = True$ i dobije poruku tipa *request* većeg prioriteta od P_j, radi sljedeće:

  1. poveća vrijednost logičkog sata,
  2. šalje poruku tipa *okay*,
  3. ukoliko P_j nije u R_i, dodaje ga u R_i i šalje poruku tipa *request*.

  Primjetimo kako iz $P_j \notin R_i$ implicira da P_i nije poslao zahtjev za CS procesu P_j koji mu ovim putem šalje kako bi nakon završetka CS procesa P_j, P_j dojavio da je izašao iz CS, tj. da postoji mogućnost da i on uđe u CS.

- Kada proces P_i ima $zahtjev = True$ i dobije poruku tipa *request* nižeg prioriteta od P_j ili kada proces P_i ima $u\_CS = True$ i dobije poruku tipa *request* od P_j, radi sljedeće:

1. poveća vrijednost logičkog sata,
2. doda P_j u R_i.

- Kada proces P_i u stanju $u\_CS = False$ i *zahtjev* $= False$ dobije poruku tipa *request* od P_j, radi sljedeće:

  1. povećaj vrijednost logičkog sata,
  2. doda P_j u I_i,
  3. šalje poruku tipa *okay* procesu P_j.

Kada pak proces P_i primi poruku tipa *okay* od procesa P_j radi:

1. povećava vrijednost logičkog sata po pravilima protjecanja vremena u Lamportovom satu,
2. izbacuje P_j iz R_i,
3. ukoliko je R_i prazan, dopušta ulazak u kritičnu sekciju:
   - postavlja *zahtjev* na $False$,
   - postavlja u_CS na $True$,
   - ulazi u kritičnu sekciju.

Nakon izlaska iz CS, proces P_i radi sljedeće:

1. postavlja varijablu u_CS na $False$,
2. šalje poruke tipa *okay* svim procesima iz $I_i \setminus \{P_i\}$,
3. osvježava vrijednosti skupova R_i i I_i: $R_i = I_i \setminus \{P_i\}$, $I_i = \emptyset$.

Pseudokod algoritma je sljedeći:

Algorithm 1: Singhalov algoritam međusobnog isključivanja

```
1  Korak 1: Zahtjev za CS;
2  zahtjev = True;
3  C_i = C_i + 1;
4  foreach P_j ∈ R_i do
5      pošalji poruku tipa request P_j-u ;
6  while True do
7      if R_i == ∅ then
8          break;
9  zahtjev = False;
10 Korak 2: CS;
11 u_CS = True;
12 kritična sekcija;
13 u_CS = False;
14 Korak 3: Izlazak iz CS;
15 foreach P_j ∈ I_i \ {P_i} do
16     I_i = I_i \ P_j;
17     pošalji poruku tipa okay P_j-u ;
18     R_i = R_i ∪ P_j;
```

Izvedba *Shingalovog algoritma* je napravljena u programskom jeziku *JAVA* klasom *SinghalMutex*. Potrebno je pripaziti na konzistentnost varijabli R_i, I_i, C_i. Funkcije koje ih mijenjaju označavamo sa *syncronized*.

Prevođenje: **javac** SinghalMutex.java
Pokretanje:

1. U terminalu pokrećemo program *NameServer* bez argumenata:
   java NameServer

2. Odlučujemo se za broj procesa distribuiranog sustava u kojem ćemo pokrenuti algoritam, u oznaci N ,

3. Odlučujemo se za *baznoIme* procesa distribuiranog sustava,

4. Otvaramo N terminala, u svakome pokrećemo jedan proces pokretanjem programa *LockTester* s odgovarajućim argumentima:
   java LockTester $< baznoIme > < i > < N > < Singhal >$

   gdje je i identifikator procesa, $i \in \{0, 1, 2, \ldots, N-1\}$. Svakom procesu se pridružuje jedinstveni identifikator.

LockTester.java predstavlja testni program algoritma koji se poslan zajedno s dokumentacijom. Naravno, prije pokretanja samog procesa (programa), potrebno ga je prevesti s **javac** ime_preograma.java.

Ukoliko sve procese pokrenemo pokrećući isti tesni program, svi procesi jednako često zahtjevaju kritičnu sekciju. Program *LockTester.java* je izveden tako da svi procesi često zatražuju CS.
Kako bi pokazali po čemu je objašnjeni algoritam poseban (procesi koji često zahtjevaju CS traže dopuštenje samo od preocesa koji često zahtjevaju CS), napravljen je još jedan testni program koji je izveden tako da su njegovi zahtjevi za CS otprilike 10 puta rjeđi. Ukoliko pokrenemo nekoliko procesa s novim testnim programom (*rijetki* proces), a ostale s *LockTester.java*, uočavamo kako se skup R pojedinog procesa dinamički mijenja. U trenutcima kada *rijetki* proces zatražuje CS, on postaje dio skupa R nekih drugih procesa. Dok ne zatražuje CS, nije element niti jednog skupa R.

2.1.2 Ispravnost algoritma

Najprije primjetimo kako mehanizam rada algoritma održava svojstvo:
U svakom trenutku algoritma za dva različita procesa istog sustava koji istovremeno zatražuju CS, P_i i P_j, vrijedi: $P_i \in R_j$ ili $P_j \in R_i$.

Dokazujemo jače svojstvo.

Na početku postavljanja zahtjeva za CS procesa $P_i \in \{P_0, P_1, \ldots, P_N\}$, za sve druge procese sustava, P_j, vrijedi da $P_i \in R_j$ ili $P_j \in R_i$.

Dokaz. Na početku rada algoritma, svojstvo je trivijalno zadovoljeno. Naime, proces s većim identifikatorom sadrži proces s manjim identifikatorom u svojemu skupu R.
Pretpostvimo sada kako u nekom trenutku rada algoritma P_i želi ući u CS. Neka je P_j neki drugi proces i neka vrijedi $P_j \in R_i$.
Dokazujemo kako je gornje svojstvo zadovoljeno i prilikom sljedećeg postavljanja zahtjeva procesa P_i za CS.

Tokom postavljanja zahtjeva za CS, P_i uklanja P_j iz R_i nakon slanja poruke r_1 tipa *request* procesu P_j, po primitku poruke *okay* procesa P_j.
P_j, po primitku poruke r_1, ili stvalja P_i u R_j ili stavlja P_i u I_j. Ukoliko P_j stvalja P_i u R_j prilikom sljedećeg zahtjevanja za CS od P_i imamo dva moguća scenarija:

- P_j je u međuvremenu (prije P_i) zahtjevao CS:
  Tada je P_j tražio dopuštenje za ulazak u CS i od P_i pa se P_j nalazi u R_i.

- P_j u međuvremenu nije zahtjevao CS:
  Dakle, $P_j \notin R_i$, ali $P_i \in R_j$.

Ukoliko P_j stvalja P_i u I_j, on ne šalje *okay* na postavljeni zahtjev sve do trenutka kada će prebaciti P_i u R_j. Dakle, prilikom sljedećeg postavljanja zahtjeva za CS od P_i, P_i se sigurno nalazio u R_j što dovosi do spomenutih scenarija. Scenarij ovisi o događajima koji su nastupili nakon trenutka prebacivanja P_i u R_j.

Dakle, iskazano svojstvo vrijedi na početku svakoga zahtjeva za CS od P_i. Budući da je P_i proizvan, tvrdnja vrijedi $\forall P_i \in \{P_0, P_1, \ldots, P_{N-1}\}$. □

Pokažimo sada kako algoritam zadovoljava svojstvo ***sigurnosti***.
Potrebno je pokazati kako se dva različita procesa sustava istovremeno ne mogu nalaziti u svojim CS.

Dokaz. Pretpostavimo suprotno. Neka se dogodilo kako se procesi P_i i P_j nalaze istovremeno u svojim CS.
Prema gornjem svojstvu, u trenutku postavljanja zahtjeva $P_i \in R_j$ ili $P_j \in R_i$.
BSOMP $P_i \in R_j$.
Tada P_i traži dopuštenje od P_j za ulazak u CS. Kako vrijedi $i \neq j$, imamo dva slučaja.
Neka je prvo $i < j$. Po primitku zahtjeva za CS od P_i, P_j smatra da P_i-tov zahtjev ima veći prioritet i šalje P_i zahtjev za ulazak u CS. Očito taj zahtjev je ima veći vremenski žig od P_i-tovog. Dakle, tek nakon izlaska iz CS, P_i šalje odobrenje za ulazak u CS procesu P_j pa se oni ne nalaze istovremeno u CS.
Ukoliko $j < i$, tada, P_i-tov zahtjev nema prednost nad P_j-ovim i P_j mu šalje odobrenje za ulazak u CS tek nakon što on sam iz nje izađe. Dakle, opet P_i i P_j nisu istovremeno u CS. □

Odsustvo gladovanja je ispunjeno zato što zahtjev za CS najvišeg prioriteta ne može biti zaustavljen.

Pravednost je posljedica činjenice da procesi ulaze u kritični odsječak u redoslijedu svojih vremenskih žigova.

Primjer:
Neka je $N = 3$. Promatrani distribuirani sustav se sastoji od 3 procesa $P_1, P_2 i P_3$. Neka P_2 i P_3 zahtjevaju ulaz u CS. Smatramo da su to prvi zahtjevi na CS. P_3 i P_2 šalju poruke tipa *request* procesima iz skupa $R_3 = \{P_1, P_2\}$, odnosno $R_2 = \{P_1\}$. Događa se jedna od sljedećih tri situacija:

1. Vremenski žig pridružen zahtjevu od R_3 je manji od vremenskog žiga zahtjeva procesa P_2. Tada, P_2, po primitku *request* poruke od P_3, šalje poruku tipa *okay*, stavlja P_3 u R_2 i šalje poruku tipa *request* procesu P_3. Dakle, ne ulazi u CS prije nego pto dobije *okay* poruku od P_3 što dobiva tek nakon što P_3 izlazi iz CS. P_3, po primitku poruke tipa *okay* procesa P_2, miče P_2 iz R_3.
   P_1, budući da ne zatražuje CS, po primitku poruka *request* od P_2 i P_3 šalje poruke tipa *okay* i stavlja P_2 i P_3 u svoj skup R_1. Naime, postoji mogućnost da se P_2 ili P_3 nalaze u CS u nekom budućem trenutku kada će P_1 tražiti pristup CS.
   P_3, dobivši *okay* i od P_1, miče P_1 iz R_3. Sada je R_3 prazan i P_3 ulazi u CS. Nakon izlaska iz CS, šalje P_2 *okay* i P_2 ulazi u CS.

2. Vremenski žig zahtjeva procesa P_3 je veći od vremenskog žiga zahtjeva P_2. Tada P_2, nakon primitka *request* poruke od P_3, stavlja P_3 u I_2 (engl. Inform set). P_2 obavještava P_3 porukom *okay* o izlasku iz CS. Nakon toga P_2 stavlja P_3 u R_2. Naime, moguće je da se P_3 nalazi u CS sljedeći put kada P_2 zatraži ulazak u CS.

3. P_1 je primio *request* od P_2 i poslao *okay* prije nego što je poruka *request* od P_3 došla do P_2. Dakle, P_2 ulazi prvi u CS i dodaje P_3 u I_2. P_2 šalje *okay* procesu P_3 nakon izlaska iz CS.

2.1.3 Prosječna složenost algoritma u sustavima s rijetkim i čestim zahtjevima na CS

U nastavku analiziramo složenost algoritma za međusobno isključivanje. Raspravljamo o dodatnoj cijeni koju plaćamo kada u mreži rješavamo problem međudobnog isključivanja upotrebom *Singhalovog* algoritma.
Razmatramo dvije vrste sustava:

- sustav u kojemu se zahtjevi za CS događaju rijetko,

- sustav u kojemu se zahtjevi za CS događaju često.

Promotrimo prvo sustav u kojemu se zahtjevi za CS događaju rijetko. U takvom sustavu, broj prisutnih zahtjeva u mreži je mali. Najčešće se radi o jednom ili nijednom zahtjevu. Gotovo nikada 2 ili više procesa istovremeno ne zahtjevaju ulazak u CS. Kako je broj zahtjeva koje proces P_i treba poslati iz skupa $\{0, 1, 2, \ldots, N-1\}$, za svaki proces jedinstven, prosječni broj poruka tipa *request* po jednom korištenju CS iznosi

$$\frac{0+1+2+\ldots+N-1}{2} = \frac{N(N-1)}{2N} = \frac{N-1}{2}.$$

Nadalje, prosječni broj poruka *okay* po jednom korištenju CS-a je $\frac{N-1}{2}$. Naime, za svaku poruku tipa *request*, šalje se po jedna poruka tipa *okay*.
Sveukupan broj dodatnih poruka po jednom korištenju CS-a iznosi

$$2\frac{N-1}{2} = N-1$$

Promotrimo sada sustav u kojemu se zahtjevi za CS događaju često.
Tada u gotovo svakom trenutku postoji po jedan zahtjev za CS na čekanju od svakog procesa. Dok proces čeka odgovore na poslane poruke tipa *request*, prima u prosjeku $\frac{N-1}{2}$ poruka tipa *request*. Na dobivene poruke tipa *request* većeg prioriteta, uz poruku *okay*, šalje i dodatnu poruku tipa *request*. Za vrijeme čekanja na odobronje ulaska u CS-a, proces šalje u prosjeku $\frac{N-1}{4}$ *request* poruka.
Dakle, prosječni broj dodatnih poruka po jednom korištenju CS-a iznosi

$$2\frac{N-1}{2} + 2\frac{N-1}{4} = 3\frac{N-1}{2}.$$

3 Algoritam Lodha & Kshemkalyany

Algoritam Lodha i Kshemkalyani je optimizirana verzija algoritma Ricarta i Agrawale za međusobno isključivanje. Algoritam umanjuje složenost smanjivanjem broja poruka koje procesi izmjenjuju kako bi ušli u kritični odsječak.

Prema algoritmu Ricarta i Agrawale, proces mora primiti poruku tipa *okay* od svih ostalih procea prije ulaska u CS. Algoritam Lodha i Kshemkalyania koristi činjenicu da poruku *okay* treba poslati samo proces koji ima prioritet neposredno veći od procesa koji traži ulazak u kritični odsječak.

Također, algoritam zadovoljava uvijete sigurnosti, pravednosti i odsustvo izgladnjivanja. Dokaza navedenih svojstva su složeni pa ih izostavljamo u ovome radu.

3.1 Ideja algoritma

Uzmimo za primjer da procesi $P_{i_1}, P_{i_2}, \ldots, P_{i_N}$ zahtijevaju ulazak u kritični odsječak, pri čemu P_{i_1} ima zahtjev najvećeg prioriteta te se prioriteti smanjuju redom do P_{i_N} koji ima najniži prioritet. Da bi proces P_{i_j} ušao u kritični odsječak, treba primiti *okay* poruku samo od procesa $P_{i_{j-1}}, 1 < j \leq n$

Svakom procesu pridružen je prioritet te se zahtjevi za kritičnom odsječku odobravaju ovisno o prioritetima. Pretpostavljamo da komunikacijski kanali rade uvijek ispravno.

3.2 Rad algoritma

Za lakše objašnjavanje algoritma, potrebne su sljedeće definicije:

Definicija 3.1. *Zahtjevi R_i i R_j su konkurentni ako i samo ako vrijedi sljedeće:*

- *proces P_i je primio poruku request od P_j nakon što je poslao request*
- *proces P_j je primio poruku request od P_i nakon što je i sam poslao request*

Definicija 3.2. *Definiramo skup:*

$$CSet_i = \{R_j | R_i jekonkurentans R_j\} \cup \{R_i\}$$

Rad algoritma zasniva se na komunikacijskim porukama koje dajemo u nastavku.

3.3 Komunikacija algoritma

Algoritam koristi tri vrste poruke:

- *request*
- *okay*
- *flush*

Svaka od tih poruka nosi sa sobom specifičan vremenski žig. Poruka *request* nosi vremenski žig zahtjeva, dok poruke *flush* i *okay* sadrže vremenski žig zadnjeg ispunjenog zahtjeva pošiljatelja. Porukom *flush* se postiže ušteda na ukupan broj poslanih poruka.

Svaki proces P_i ima svoj lokalni red LQR_i koji sadrži sve konkurentne zahtjeve poredane silazno prema prioritetima. Prioritet zahtjeva određen je njegovim vremenskim žigom. Kad proces primi poruku *request* od nekog drugog procesa, on određuje kada će ga pustiti u kritični odsječak.

3.3.1 Poruka OKAY

Nakon primitka poruke *request*, proces odgovara tom drugom procesu žigosanom porukom *okay* kada on ne želi ući u kritični odsječak.

Poruka *okay* može služiti kao kolektivni *okay* od procesa koji su imali viši prioritet. Kada proces P_i dobije *okay* od P_j koji je završio sa kritičnim odsječkom, slijedi da su svi zahtjevi s prioritetom većim ili jednakim od R_j

gotovi. Proces P_i može maknuti iz svog lokalnog reda LRQ_i sve zahtjeve čiji je prioritet veći ili jednak prioritetu R_j. Poruka *okay* se ponaša kao kolektivni odgovor od svih procesa koji su imali zahtjeve višeg prioriteta.

3.3.2 Poruka FLUSH

Slično kao sa porukom *okay*, poruka flush označava kolektivni odgovor od svih procesa koji su imali zahtjeve višeg prioriteta. Nakon izlaska iz kritičnog odsječka, proces pošalje žigosanu poruku *flush* konkurentnom procesu sa sljedećim najvišim prioritetom (ako postoji). Taj proces nalazi se u njegovom lokalnom redu LRQ u kojem čuva sve konkurentne zahtjeve.

Kada proces P_i izađe iz kritičnog odsječka, proces P_j se može nalaziti u jednom od sljedećih stanja:

- R_j je u lokalnom redu procesa P_i i nalazi se nakon zahtjeva R_i što implicira da su R_j i R_i konkurentni zahtjevi.
- P_j je odgovorio procesu P_i porukom *okay* te je dao zahtjev nižeg prioriteta. R_j i R_i nisu konkurentni zahtjevi.
- R_j je zahtjev višeg prioriteta od R_i, iz čega slijedi da je proces P_j izašao iz kritičnog odsječka te sada ima zahtjev nižeg prioriteta. R_j i R_i nisu konkurentni.

Nakon izlaska iz kritične odsječka, proces P_i pošalje poruku flush procesu s najvećim prioritetom iz prve točke. A procesima iz druge i treće točke šalje poruke *okay* jer njihovi zahtjevi nisu konkurentni sa R_i. Sada proces koji je primio flush i procesi koji su primili *okay* odlučuju koji će proces sljedeći ući u kritični odsječak.

Primjer: Imamo pet procesa $P_0, \ldots, P_4$ s pripadnim zahtjevima $R_0, \ldots, R_4$. Zamislimo situaciju u kojoj imamo red zahtjeva R_3, R_0, R_2, R_4, R_1, koji su poredani silazno po prioritetima te su zahtjevi R_0, R_2, R_4 konkurentni. Dakle,

lokalni red procesa P_0 sadrži zahtjeve R_0, R_2, R_4 te kada izađe iz kritične sekcije, P_0 šalje poruku *flush* samo procesu P_2.

3.3.3 Poruka REQUEST

Ako P_i i P_j nemaju konkurentne zahtjeve, tada proces koji je prvi dao zahtjev za kritičnu odsječak dobiva poruku *okay*.

Ako P_i i P_j imaju konkurentne zahtjeve, tada:

- R_i je višeg prioriteta od R_j. Tada poruka *request* od P_j služi kao implicitni odgovor *okay* na zahtjev procesa P_i. Pritom P_j mora čekati poruku *flush* ili *okay* kako bi ušao u kritični odsječak.
- R_i je nižeg prioriteta od R_j. Tada poruka request od P_i služi kao implicitni odgovor okay na zahtjev procesa P_j. Pritom, P_i mora čekati na poruku *flush* ili *okay* od nekog procesa kako bi ušao u kritični odsječak.

3.4 Broj poruka

Da bi ušao u kritični odsječak, proces P_i šalje $(N-1)$ poruku *request*, a primi $(N - |CSet_i|)$ odgovora u obliku poruka *okay* i *flush*. Pogledajmo sljedeća dva slučaja:

- $|CSet_i| \geq 2$.
  - Postoji barem jedan zahtjev za kritičnim odsječkom čiji je prioritet manji od prioriteta zahtjeva R_i . Stoga će proces P_i poslati jednu *flush* poruku. U ovom slučaju ukupan broj poruka za ulazak u kritični odsječak je $(N - |CSet_i|)$. Kada su svi zahtjevi konkurentni, broj poruka se smanjuje na N.
  - Nema zahtjeva čiji je prioritet manji od prioriteta zahtjeva R_i. P_i neće slati flush poruku.Broj poruka za ulazak u kritični odsječak

je $2N-1-|CSet_i|$. Kada su svi zahtjevi konkurentni, broj poruka se smanjuje na $N-1$

- $|CSet_i| = 1$ je nagori slučaj te ukupan broj poruka iznosi $2(N-1)$.

Algoritam je implementiran pomoću klase *LKMutex*.
Prevođenje: **javac** LKMutex.java
Pokretanje:

1. U terminalu pokrećemo program *NameServer* bez argumenata:
   java NameServer

2. Odlučujemo se za broj procesa distribuiranog sustava u kojem ćemo pokrenuti algoritam, u oznaci N ,

3. Odlučujemo se za *baznoIme* procesa distribuiranog sustava,

4. Otvaramo N terminala, u svakome pokrećemo jedan proces pokretanjem programa *LockTester* s odgovarajućim argumentima:
   java LockTester $< baznoIme > < i > < N > < LKMutex >$

   gdje je i identifikator procesa, $i \in \{0, 1, 2, \ldots, N-1\}$. Svakom procesu se pridružuje jedinstveni identifikator.

4 Algoritmi zasnovani na kvorumu

Algoritmi za isključivanje zasnovani na kvorumu imaju sljedeće dvije karakteristike:

1. Proces ne zahtijeva dopuštenje od svih ostalih procesa nego samo od nekog poskupa procesa. Ovo je znatno drugačiji pristup nego kod Lamport i Ricart–Agrawala algoritma, gdje svi procesi sudjeluju u razrješavanju zahtijeva za kritični odsječak. Podskupovi procesa zadovoljavaju sljedeće pravilo $\forall$ i, j : $1 \leq$ i, j $\leq$ N $\Rightarrow$ Ri $\cap$ Rj $\neq \emptyset$ i svaki skup

naziva se *kvorum*. Posljedica ovog zahtjeva je da svaki par procesa ima proces koji može razriješiti konflikt među njima.

2. Proces može poslati samo jednu REPLY poruku u bilo kojem trenutku. Proces može poslati REPLY poruku samo nakon što je primilo RELEASE poruku za prethodno poslanu REPLY poruku.

Ovakav pristup značajno smanjuje broj poslanih poruka budući da proces traži dopuštenje samo od podskupa skupa svih procesa, a ne od cijelog skupa procesa.

Označimo sa C skup kvoruma. Tada skup C zadovoljava sljedeća svojstva:

- **Svojstvo presjeka** $\forall$ g,h$\in$C, g $\cap$ h $\neq \emptyset$. Na primjer, skupovi {1,2,3}, {2,5,7} i {5,7,9} ne mogu biti kvorumi u skupu C jer prvi i treći skup nemaju zajednički element.

- **Svojstvo minimalnosti** Ne postoje g,h$\in$C takvi da g $\supseteq$ h. Na primjer, skupovi {1,2,3} i {1,3} ne mogu biti kvorumi u skupu C zato što je prvi skup nadskup drugog skupa. Ovo svojstvo osigurava efikasnost, a ne korektnost.

4.1 Algoritam Maekawa

Algortiam Maekawa je prvi algoritam za međusobno isključivanje zasnovan na kvorumu. Skup kvoruma u algoritumu Maekawa konstruiran je tako da zadovoljava sljedeće uvjete:

M1 $(\forall$ i,j : i $\neq$ j $\Rightarrow$ Ri $\cap$ Rj $\neq \emptyset)$

M2 $(\forall$ i : 1$\leq$i$\leq$N $\Rightarrow$ Si $\in$ Ri)

M3 $(\forall$ i : 1$\leq$i$\leq$N $\Rightarrow$ $\mid$ Rt $\mid$ = K)

M4 Svaki proces nalazi se u K kvoruma

Tablica 1: Primjer skupa kvoruma koji zadovoljavaju uvjete algoritma Maekawia gdje je K=2

K=2, N=3	$R_1 = \{1, 2\}$ $R_2 = \{2, 3\}$ $R_3 = \{1, 3\}$

Tablica 2: Primjer skupa kvoruma koji zadovoljavaju uvjete algoritma Maekawia gdje je K=3

K=3, N=7	$R_1 = \{1,2,3\}$ $R_2 = \{2,4,6\}$ $R_3 = \{3,5,6\}$ $R_4 = \{1,4,5\}$ $R_5 = \{2.5.7\}$ $R_6 = \{1,6,7\}$ $R_7 = \{3,4,7\}$

Maekawa je koristio teoriju projektivnih ravnina za konstrukciju kvoruma. Pokazao je da vrijedi N = K(K - 1) + 1. Iz te relacije slijedi | Ri | = $\sqrt{\text{N}}$.

Budući da svaka dva kvoruma sadrže barem jedan zajednički proces (uvjet M1), svaki par procesa ima jedan zajednički proces koji posreduje u rješavanju mogućih konflikata. Proces može imati poslati samo jednu poruku REPLY, to znači da daje dopuštenje procesu samo ukoliko to dopuštenje nije već dano nekom drugom procesu. To daje garanciju za zadovoljavanje svojstva sigurnosti (samo jedan proces može dobiti kritični odsječak). Algoritam također zahtijeva primanje poruka u redoslijedu kojim su poslane. Uvjeti M1 i M2 su nužni za korektnost algoritma, dok uvjeti M3 i M4 pružaju druga poželjna svojstva. Uvjet M3 zahtijeva da svi kvorumi imaju jednak broj procesa što bi trebalo značiti da svi procesi obavljaju jednaku količinu posla u međusobnom isključivanju. Zbog uvjeta M4 svi procesi imaju "jednaku odgovornost" u dodjeljivanju dopuštenja drugim procesima.

4.1.1 Pseudokod

1. **Zahtijevanje kritičnog odsječka**

   (a) Proces Si zatraži kritični odsječak slanjem poruke REQUEST(i) svim procesima koji se nalaze u istom kvorumu Ri

   (b) Kada proces Sj primi poruku REQUEST(i), on šalje poruku REPLY(j) procesu Si, samo ukoliko nakon zadnjeg primanja poruke RELEASE, poruka REPLY nije već poslana nekom procesu. Ukoliko je poruka REPLY već poslana proces Sj stavlja u red čekanja REQUEST(i) za kasniju obradu.

2. **Izvršavanje kritičnog odsječka**

   (a) Proces Si ulazi u kritični odsječak onda kada primi poruku REPLY od svakog procesa u kvorumu Ri.

3. **Izlazak iz kritičnog odsječka**

   (a) Za izlazak iz kritičnog odsječka, proces Si šalje poruku RELEASE(i) svakom procesu u kvorumu Ri.

   (b) Kada proces Sj primi poruku RELEASE(i) od procesa Si, on šalje REPLY poruku sljedećem procesu koji je u njegovom redu čekanja te ga miče iz reda. Ukoliko je red prazan proces ažurira svoje stanje tako da tako da može poslati poruku REPLY kao odgovor na REQUEST poruku

Budući da je veličina svakog kvoruma $\sqrt{N}$, za dobivanje kritičnog odsječka pošalje se $\sqrt{N}$ REQUEST poruka, $\sqrt{N}$ REPLY poruka i $\sqrt{N}$ RELEASE poruka.

Teorem 4.1. *Algoritmom Maekawa postiže se međusobno isključivanje.*

Dokaz. Pretpostavimo da se dva procesa Si i Sj istovremeno nalaze u kritičnom odsječku. To znači da je proces Si primio REPLY poruku od svih procesa u kvorumu Ri i da je istovremeno proces Sj bio u mogućnosti istovremeno primiti REPLY poruku od svih procesa u kvorumu Ri. Budući da vrijedi $Ri \cap Rj \neq \emptyset$ (uvjet M1) postoji proces $Sk \in Ri \cap Rj$. Proces Sk je morao istovremeno poslati REPLY poruku i procesu Si i procesu Sj što je kontradikicija. □

4.1.2 Problem zastoja

U algoritmu Maekawa može doći do zastoja (nije zadovoljeno svojstvo odsustva izgladnjivanja) zato što proces može biti blokiran od strane drugih procesa i poruke REQUEST nisu prioritizirane s obzirom na vrijeme slanja. Dakle, proces može poslati zahtjev procesu i kasnije prisiliti zahtjev većeg prioriteta da čeka (nije zadovoljeno svojstvo pravednosti). Bez smanjenja općenitosti možemo pretpostaviti da procesi Si, Sj i Sk istovremeno zatraže kritični odsječak. Pretpostavimo $Sij \in Ri \cap Rj$, $Sjk \in Rj \cap Rk$ i $Ski \in Rk \cap Ri$. Budući da proces REQUEST poruku ne šalje nekim unaprijed određenim redom i kašnjenja poruka su proizvoljna, moguć je sljedeći redoslijed događaja: proces Si zaustavlja proces Sij (Sj mora čekati Sij), Proces Sj zaustavlja Sjk (Sk mora čekati Sjk) i Sk zaustavlja Ski (Sj mora čekati Ski). Ovakvo stanje prikazuje stanje zastoja u koji su uključeni Si, Sj i Sk.
Da bi se riješio problem zastoja potrebne su dodatne poruke:

- **FAILED** proces Si šalje poruku FAILED procesu Sj ukoliko mu ne može odobriti kritični odsječak jer ga je već odobrio nekom drugom procesu sa većim prioritetom.
- **INQUIRE** proces Si slanjem INQUIRE poruke procesu Sj želi saznati je li proces Sj uspio zaustaviti sve procese koji se nalaze u njegovom kvorumu.

- **YIELD** proces Si slanjem YIELD poruke procesu Sj vraća primljeno dopuštenje za ulazak u kritični odsječak (da proces Sj može dati dopuštenje procesu sa većim prioritetom).

Nadopuna algoritma:

1. Kada proces Sj blokira zahtjev REQUEST(ts, i) [1] procesa Si zato što je već dao dopuštenje procesu Sk, proces Sj šalje procesu Si poruku FAILED(j) ukoliko Si ima manji prioritet od procesa Sk. Inače proces Sj šalje procesu Sk poruku INQUIRE(j).

2. Kao odgovor na primljenu poruku INQUIRE(j) od procesa Sj, proces Sk šalje poruku YIELD(k) procesu Sj nakon što je primio poruku FAILD od nekog procesa u svom kvorumu i ukoliko je poslao YIELD poruke, a nije primio REPLY kao odgovor.

3. Kada proces Sj primi YIELD(k) poruku od procesa Sk, proces Sj sprema zahtjev porocesa Sk na pravo mjesto u svom redu procesa koji čekaju i šalje REPLY(j) poruku procesu koji je prvi u redu.

Sa ovom nadopunom algoritma za jedno izvršavanje kritične sekcije u najgorem slučaju potrebno je $5\sqrt{\text{N}}$ poruka.

Izvedba *Algoritma Maekawa* implemntirana je klasom *MaekawaMutex*. Potrebno je pripaziti na postojanje ispravne topologije. Pokretanje:

1. U terminalu pokrećemo program *NameServer* bez argumenata:
   java NameServer

2. Odlučujemo se za broj procesa distribuiranog sustava u kojem ćemo pokrenuti algoritam, u oznaci N ,

3. Odlučujemo se za *baznoIme* procesa distribuiranog sustava,

[1] `ts je vremenski žig (timestamp)`

4. Otvaramo N terminala, u svakome pokrećemo jedan proces pokretanjem programa *LockTester* s odgovarajućim argumentima:
   java LockTester $< baznoIme > < i > < N > < Maekawi >$

   gdje je i identifikator procesa, $i \in \{0, 1, 2, \ldots, N-1\}$. Svakom procesu se pridružuje jedinstveni identifikator.

[1] V. Garg. *Concurrent and Distributed Computing in Java*. Wiley – IEEE Press, Hoboken NY, 2004.

[2] A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. 2008.

[3] R. Manger. *Distribuirani procesi*. 2016. interna skripta.