
Konkurentni i distribuirani sistemi

Stevan Milinković



Konkurentni i distribuirani sistemi

Stevan Milinković

ISBN 978-86-7991-412-5

Copyright © 2019. CET Computer Equipment and Trade, Beograd i Računarski fakultet, Beograd.

Sva prava zadržana. Nijedan deo ove knjige ne može biti reprodukovan, snimljen, ili emitovan na bilo koji način: elektronski, mehanički, fotokopiranjem, ili drugim vidom, bez pisane dozvole izdavača. Informacije korišćene u ovoj knjizi nisu pod patentnom zaštitom. U pripremi ove knjige učinjeni su svi napori da se ne pojave greške. Izdavač i autori ne preuzimaju bilo kakvu odgovornost za eventualne greške i omaške, kao ni za njihove posledice.

Recenzija	dr Branko Perišić, redovni profesor Fakulteta Tehničkih Nauka Univerziteta u Novom Sadu dr Dragan Urošević, redovni profesor Računarskog fakulteta.
Lektura	Milanka Vorkapić Stojanović
Urednik	Dubravka Dragović Šehović
Tehnički urednik	Vesna Petrinović
Izdavač	CET Computer Equipment and Trade Beograd, Skadarska 45 tel/fax: 011 3243-043, 3235-139, 3237-246 http://www.cet.rs Računarski fakultet Beograd, Knez Mihailova 6/VI tel: 011 2627-613, 2633-321 www.raf.edu.rs
Za izdavača	Dragan Stojanović, direktor
Tiraž	1000
Štampa	„SaTCIP”, Vrnjačka Banja

Nastavno-naučno veće na 131. sednici održanoj 27.12.2018. godine donelo je odluku da knjiga „Konkurentni i distribuirani sistemi” bude štampana kao univerzitetski udžbenik.

Sadržaj

Predgovor	xv
0. Terminologija i klasifikacija	1
0.1 Paradigme konkurentnog programiranja	1
0.2 Terminologija	2
0.3 Klasifikacija konkurentnih i distribuiranih sistema	4
0.4 Klasifikacija paradigmi konkurentnog programiranja	5
0.5 Programiranje pomoću deljenih promenljivih	7
0.6 Distribuirano programiranje	8
0.7 Virtuelni prostori	9
0.8 Programske niti	10
0.9 Java	11
0.9.1 Kreiranje i kontrola rada niti	11
0.9.2 Sinhronizacija	12
1. Uvod	15
1.1 Deljeni objekti i sinhronizacija	15
1.2 Nova priča o Alisi i Bobu	17
1.2.1 Osobine uzajamnog isključivanja	20
1.2.2 Naravoučenje	20
1.3 Proizvođač – potrošač	21
1.4 Čitaoci i pisci	23
1.5 Surova realnost paralelizacije	24
2. Uzajamno isključivanje	27
2.1 Vreme	27
2.2 Kritične sekcije	27
2.3 Rešenje za dve niti	30
2.3.1 Klasa LockOne	30
2.3.2 Klasa LockTwo	31
2.3.3 Petersonov algoritam	32
2.4 Filter algoritam	34
2.5 Da li je ovo fer algoritam?	37
2.6 Lamportov pekarski algoritam	37
2.7 Vremenski žigovi	39
2.8 Donja granica broja memorijskih lokacija	42
3. Konkurentni objekti	47
3.1 Konkurentnost i korektnost	47
3.2 Sekvencijalni objekti	50
3.3 Mirna konzistentnost	51

3.4	Sekvencijalna konzistentnost	53
3.5	Linearizacija	56
	Tačke linearizacije	56
	Napomene	56
3.6	Formalne definicije	57
	3.6.1 Linearizacija	58
	3.6.2 Kompoziciona linearizacija	58
	3.6.3 Neblokirajuće svojstvo	59
3.7	Uslovi za progres	60
	3.7.1 Zavisni uslovi za progres	61
3.8	Java memorijski model	62
	3.8.1 Ključevi i sinhronizacioni blokovi	63
	3.8.2 Volatile	64
	3.8.3 Final	64
3.9	Napomene	65
4.	Spin-Lock i konflikt	66
4.1	Realni svet	66
4.2	Test And Set	69
4.3	Revidirani TAS Spin Lock	71
4.4	Eksponencijalni Backoff	72
4.5	Redovi čekanja za niti	74
	4.5.1 Ključevi zasnovani na nizu	74
	4.5.2 CLH Queue Lock	77
	4.5.3 MCS Queue Lock	79
4.6	Queue Lock sa tajmautom	81
4.7	Kompozitni ključ	84
	4.7.1 Kompozitni ključ sa brzom putanjom	89
4.8	Klasteri i hijerarhijski ključevi	92
	4.8.1 Hijerarhijski Backoff	92
	4.8.2 Hijerarhijski CLH Queue ključ	93
4.9	Jedan ključ za sve probleme	98
4.10	Napomene	98
4.11	Literatura	99
5.	Monitori	101
5.1	Potreba za monitorima	101
5.2	Ključevi i uslovi monitora	101
	5.2.1 Objekat uslova	102
	5.2.2 Problem propuštenog buđenja	105
5.3	Čitaoci i pisci	107
	5.3.1 Jednostavan Readers–Writers ključ	107
	5.3.2 Readers–Writers fer ključ	109

5.4 Reentrant ključ	112
5.5 Semafori	112
6. Barijere	114
6.1 Uvod	114
6.2 Implementacija barijere	115
6.3 Barijera sa promenom parnosti.	116
6.4 Barijera sa kombinacionim stablom	117
6.5 Barijera sa statičkim stablom	120
6.6 Barijere sa detekcijom završetka	121
7. Transakciona memorija	125
7.1 Uvod	125
7.1.1 Šta nije u redu sa zaključavanjem?	125
7.1.2 Šta nije u redu sa compareAndSet()?	125
7.1.3 Šta nije u redu sa kompozicijom?	127
7.1.4 Šta mi možemo da uradimo po tom pitanju?	128
7.2 Transakcije i atomičnost	128
7.3 Softverska transakciona memorija	131
7.3.1 Transakcije i transakcione niti	134
7.3.2 Konkurentnost i zombi transakcije	135
7.3.3 Atomični objekti	136
7.3.4 Zavisni i nezavisni progres?	138
7.3.5 Upravljanje konfliktima.	138
7.3.6 Implementacija atomičnih objekata.	140
7.3.7 Atomični objekti bez opstrukcije.	142
7.3.8 Atomični objekti sa ključevima	145
7.4 Hardverska transakciona memorija	152
7.4.1 Koherentnost keša	152
7.4.2 Transakciona koherentnost keša	153
7.4.3 Poboljšanja	154
8. Multiprocesorsko raspoređivanje	155
8.1 Uvod	155
8.2 Analiza paralelizma	161
8.3 Realistično multiprocesorsko raspoređivanje.	164
8.4 Raspodela opterećenja	167
8.4.1 Krađa poslova	167
8.4.2 Predaja kontrole i multiprogramiranje.	167
8.5 DEQueue sa krađom poslova	168
8.5.1 Ograničeni DEQueue sa krađom poslova	169
8.5.2 Neograničeni DEQueue sa krađom poslova	172
8.5.3 Balansiranje poslova	176
8.6 Raspoređivanje među heterogenim procesorima	178

9. Distribuirano izračunavanje	180
9.1 Distribuirani program	180
9.1.1 Programiranje upotrebom soketa	180
9.1.2 Povezivanje distribuiranih procesa	183
9.1.3 Poziv udaljene metode (RMI)	187
9.2 Model distribuiranog izvršavanja	191
9.2.1 Relacija kauzalnog prethođenja	192
9.2.2 Logička i fizička konkurentnost	193
9.3 Modeli komunikacionih mreža	193
9.4 Globalno stanje distribuiranog sistema	194
9.5 Preseci kod distribuiranog izračunavanja	196
9.6 Konusi prošlosti i budućnosti događaja	197
9.7 Modeli komunikacija među procesima	198
9.7.1 Tranzijentne komunikacione primitive	199
9.7.2 Sinhronizam procesora	203
9.7.3 Biblioteke i standardi	203
9.7.4 Sinhrona i asinhrona izvršavanja	204
9.7.5 Literatura	205
10. Logičko vreme	207
10.1 Sistem logičkih časovnika	208
10.1.1 Definicija	208
10.1.2 Implementacija logičkih časovnika	209
10.2 Skalarno vreme	209
10.2.1 Definicija	209
10.2.2 Osnovna svojstva	210
10.3 Vektorsko vreme	212
10.3.1 Definicija	212
10.3.2 Osnovna svojstva	213
10.3.3 Veličina vektorskih časovnika	214
10.4 Implementacija vektorskih časovnika	217
10.4.1 Diferencijalna tehnika (Singhal–Kshemkalyani)	218
10.4.2 Tehnika direktne zavisnosti (Fowler–Zwaenepoel)	220
10.4.3 Adaptivna tehnika (Jard–Jourdan)	224
10.5 Matrično vreme	227
10.5.1 Definicija	227
10.5.2 Osnovne osobine	229
10.6 Virtuelno vreme	229
10.6.1 Definicija virtuelnog vremena	230
10.6.2 Poređenje sa Lamportovim logičkim časovnicima	231
10.6.3 Mehanizam vremenskog zakrivljenja	232
10.6.4 Lokalni kontrolni mehanizam	233

10.6.5 Globalni kontrolni mehanizam	235
10.7 Sinhronizacija fizičkih časovnika	238
10.7.1 Definicije i terminologija	239
10.7.2 Odstupanja časovnika	240
11. Globalno stanje i algoritmi za njegovo snimanje.	243
11.1 Uvod	243
11.2 Model sistema i definicije.	245
11.2.1 Model sistema	245
11.2.2 Konzistentno globalno stanje.	246
11.2.3 Interpretacija upotrebom preseka	246
11.2.4 Problemi u snimanju globalnog stanja.	247
11.3 Algoritmi za FIFO kanale	248
11.3.1 Chandy–Lampport algoritam.	248
11.3.2 Osobine snimljenog globalnog stanja	250
11.3.3 Implementacija Chandy–Lampport algoritma.	252
11.4 Varijacije Chandy–Lampport algoritma	255
11.4.1 Spezialetti–Kearns algoritam.	255
11.4.2 Venkatesan inkrementalni algoritam	257
11.4.3 Helary talasni sinhronizacioni metod	258
11.5 Algoritmi za ne-FIFO kanale	258
11.5.1 Lai–Yang algoritam	259
11.5.2 Li algoritam	260
11.5.3 Mattern algoritam	261
11.6 Snimanje stanja u sistemima sa kauzalnom isporukom	263
11.6.1 Snimanje stanja procesa.	263
11.6.2 Snimanje stanja kanala kod Acharya–Badrinath algoritma.	263
11.6.3 Snimanje stanja kanala kod Alagar–Venkatesan algoritma	264
11.7 Praćenje globalnog stanja	266
11.8 Potrebni i dovoljni uslovi	266
11.9 Nalaženje konzistentnih globalnih snimaka	270
11.9.1 Pronalaženje konzistentnih globalnih snimaka	271
11.9.2 Manivannan–Netzer–Singhal algoritam	274
11.9.3 Nalaženje Z-putanja.	275
12. Osnovni distribuirani algoritmi	278
12.1 Topološka apstrakcija i prekrivajuće mreže	278
12.2 Klasifikacija i osnovni koncepti	280
12.2.1 Izvršavanje aplikacija i koordinacionih algoritama.	280
12.2.2 Centralizovani i distribuirani algoritmi	280
12.2.3 Simetrični i asimetrični algoritmi	281
12.2.4 Anonimni algoritmi	281
12.2.5 Uniformni algoritmi.	281

12.2.6	Adaptivni algoritmi	282
12.2.7	Determinističko i nedeterminističko izvršavanje	282
12.2.8	Inhibicija izvršavanja.	282
12.2.9	Sinhroni i asinhroni sistemi	284
12.2.10	On-line i offline algoritmi	284
12.2.11	Modeli otkaza	284
12.2.12	Algoritmi bez čekanja	286
12.2.13	Komunikacioni kanali	286
12.3	Metrike i mere kompleksnosti	286
12.4	Programska struktura	288
12.5	Elementarni grafovski algoritmi.	289
12.5.1	Sinhroni algoritam razapinjućeg stabla sa jednim inicijatorom i plavljenjem	289
12.5.2	Asinhroni algoritam razapinjućeg stabla sa jednim inicijatorom i plavljenjem	291
12.5.3	Asinhroni algoritam razapinjućeg stabla sa konkurentnim inicijatorima i plavljenjem	294
12.5.4	Asinhroni algoritam razapinjućeg stabla sa konkurentnim inicijatorima i DFS	297
12.5.5	Brodkast i konvergekast na stablu	299
12.5.6	Najkraći put sa jednim izvorom: sinhroni Bellman–Ford	300
12.5.7	Rutiranje sa vektorom rastojanja	301
12.5.8	Najkraći put sa jednim izvorom: asinhroni Bellman–Ford	302
12.5.9	Najkraći putevi od svih izvora: asinhroni distribuirani Floyd–Warshall algoritam	303
12.5.10	Algoritmi bez razapinjućeg stabla sa ograničenim plavljenjem.	307
12.5.11	Razapinjuće stablo minimalne težine kod sinhronih sistema	308
12.5.12	Algoritam razapinjućeg stabla minimalne težine kod asinhronih sistema	313
12.5.13	Primeri implementacije	314
12.6	Sinhronizatori	317
12.6.1	Opšta zapažanja kod sinhronih i asinhronih algoritama	317
12.6.2	Primeri implementacije	323
12.7	Maksimalni nezavisni skup	328
12.8	Povezani dominantni skup	330
12.9	Kompaktne tabele rutiranja.	331
12.10	Izbor lidera	333
12.11	Problemi u projektovanju grafovskih algoritama	336
12.12	Problemi replikacije objekata	336
12.12.1	Definicija problema	337
12.12.2	Prikaz algoritma.	337
12.12.3	Čitaoci i pisci.	338

12.12.4 Konvergencija ka šemi replikacije	338
12.13 Pomoćni programi	342
12.14 Literatura.	343
13. Redosled poruka i grupna komunikacija	345
13.1 Paradigme redosleda poruka.	345
13.1.1 Asinhrona izvršavanja	346
13.1.2 FIFO izvršavanje	346
13.1.3 Izvršavanje kauzalnim redosledom	347
13.1.4 Sinhrono izvršavanje	349
13.2 Asinhrono izvršavanje sa sinhronom komunikacijom	351
13.2.1 Izvršavanje ostvarivo sa sinhronom komunikacijom	351
13.2.2 Hijerarhija paradigmi uređenja	354
13.2.3 Simulacije	355
13.3 Redosled u sinhronom programu na asinhronom sistemu	356
13.3.1 Randevu.	357
13.3.2 Algoritam za binarni randevu	358
13.4 Grupna komunikacija	362
13.5 Kauzalno uređenje	362
13.5.1 Raynal–Schiper–Toueg algoritam	363
13.5.2 Kshemkalyani–Singhal algoritam	365
13.6 Totalno uređenje	371
13.6.1 Centralizovani algoritam za totalno uređenje	372
13.6.2 Trofazni distribuirani algoritam.	372
13.7 Nomenklatura za multikast.	376
13.8 Propagaciona stabla za multikast	377
13.9 Klasifikacija multikast algoritama na aplikacionom nivou	382
13.10 Semantika grupne komunikacije tolerantne na otkaze	384
13.11 Distribuirani multikast algoritmi na mrežnom sloju.	386
13.11.1 Prosleđivanje reverznom putanjom za ograničeno plavljenje	387
13.11.2 Štajnerova stabla	387
13.11.3 Funkcija cene multikasta	388
13.11.4 Štajnerova stabla sa ograničenim kašnjenjem	389
13.11.5 Stabla na bazi jezgra	392
14. Distribuirano uzajamno isključivanje.	393
14.1 Uvod	393
14.2 Preliminarna razmatranja	394
14.2.1 Model sistema	394
14.2.2 Zahtevi koje mora da ispuni algoritam	394
14.2.3 Metrike performansi	395
14.3 Lamportov algoritam	396
14.4 Ricart–Agrawala algoritam.	400

14.5 Singhal algoritam	403
14.5.1 Opis algoritma	405
14.5.2 Korektnost	407
14.5.3 Analiza performansi	408
14.6 Lodha-Kshemkalyani fer algoritam	409
14.6.1 Model sistema	409
14.6.2 Opis algoritma	409
14.6.3 Kompleksnost poruka	414
14.7 Uzajamno isključivanje na bazi kvoruma	415
14.8 Maekawa algoritam	416
14.8.1 Problem uzajamnog blokiranja	417
14.9 Agarwal–El Abbadi algoritam na bazi kvoruma	418
14.9.1 Konstrukcija kvoruma sa strukturom stabla	419
14.9.2 Analiza algoritama za konstrukciju kvoruma sa strukturom stabla	420
14.9.3 Validacija	420
14.9.4 Primeri kvoruma sa strukturom stabla	421
14.9.5 Algoritam za distribuirano uzajamno isključivanje	422
14.9.6 Dokaz korektnosti	423
14.10 Algoritmi na bazi tokena	423
14.11 Suzuki–Kasami brodkast algoritam	424
14.12 Raymond algoritam na bazi stabla	426
14.12.1 Promenljiva HOLDER	427
14.12.2 Funkcionisanje algoritma	428
14.12.3 Opis algoritma	429
14.12.4 Korektnost	431
14.12.5 Analiza cene i performansi	433
14.12.6 Inicijalizacija algoritma	434
14.12.7 Otkazi i oporavak čvorova	434
14.13 Literatura	435
15. Distribuirana deljena memorija	436
15.1 Apstrakcija i prednosti	436
15.2 Modeli konzistentnosti memorije	439
15.2.1 Striktna konzistentnost/atomična konzistentnost/mogućnost linearizacije	440
15.2.2 Sekvencijalna konzistentnost	444
15.2.3 Kauzalna konzistentnost	447
15.2.4 PRAM (Pipelined RAM) ili konzistentnost procesora	448
15.2.5 Spora memorija	450
15.2.6 Hijerarhija modela konzistentnosti	450
15.2.7 Modeli konzistentnosti kod sinhronizacionih instrukcija	451

15.3	Uzajamno isključivanje u deljenoj memoriji	453
15.3.1	Lamportov pekarski algoritam	454
15.3.2	Lamportov WRWR mehanizam i brzo uzajamno isključivanje	455
15.3.3	Hardverska podrška uzajamnom isključivanju	458
15.4	Algoritmi bez čekanja	460
15.5	Hijerarhija registara i simulacija izvršavanja bez čekanja	461
15.5.1	SRSW bezbedan u MRSW bezbedan	464
15.5.2	SRSW regularni u MRSW regularni	464
15.5.3	Bulov MRSW bezbedni u celobrojni MRSW bezbedni	465
15.5.4	Bulov MRSW bezbedan u Bulov MRSW regularan	466
15.5.5	Bulov MRSW regularni u celobrojni MRSW regularni	467
15.5.6	Bulov MRSW regularni u celobrojni MRSW atomični	468
15.5.7	Celobrojni MRSW atomični u celobrojni MRMW atomični	471
15.5.8	Celobrojni SRSW atomični u celobrojni MRSW atomični	472
15.6	Deljeni objekat - atomični snimak bez čekanja	474
15.7	Literatura	478
16.	Dogovor i konsenzus	480
16.1	Definicija problema	480
16.1.1	Vizantijski dogovor i ostali problemi	482
16.1.2	Ekvivalentnost problema i notacije	483
16.2	Pregled rezultata	483
16.3	Dogovor u sinhronim i asinhronim sistemima bez otkaza	485
16.4	Dogovor u sinhronim sistemima sa otkazima	486
16.4.1	Algoritam konsenzusa kod sinhronih sistema sa otkazima	486
16.4.2	Konsenzus kod sinhronih sistema sa Vizantijskim otkazima	487
16.5	Dogovor u asinhronim sistemima sa prenosom poruka i sa otkazima	499
16.5.1	Rezultat nemogućnosti za problem konsenzusa	499
16.5.2	Terminacioni pouzdani brodkast	501
16.5.3	Predaja kod distribuiranih transakcija	502
16.5.4	k-set konsenzus	502
16.5.5	Približni dogovor	503
16.5.6	Problem preimenovanja	509
16.5.7	Pouzdan brodkast	515
16.6	Konsenzus bez čekanja kod asinhronih sistema sa deljenom memorijom	515
16.6.1	Rezultat nemogućnosti	515
16.6.2	Brojevi konsenzusa i hijerarhija	518
16.6.3	Univerzalnost konsenzus objekata	523
16.6.4	k-set konsenzus u deljenoj memoriji	528
16.6.5	Preimenovanje u deljenoj memoriji	529
16.6.6	Preimenovanje kod deljene memorije upotrebom splitera	531
16.7	Literatura	533

17. Detekcija otkaza535
17.1 Uvod535
17.2 Nepouzdana detektori otkaza535
17.2.1 Model sistema536
17.2.2 Detektori otkaza537
17.2.3 Osobine kompletnosti i tačnosti537
17.2.4 Tipovi detektora otkaza539
17.2.5 Redukcija detektora otkaza540
17.2.6 Redukovanje slabog detektora W na jaki detektor S541
17.2.7 Redukcija slabog detektora sa odlaganjem $\diamond W$ na jaki detektor sa odlaganjem $\diamond S$543
17.3 Problem konsenzusa545
17.3.1 Rešenja problema konsenzusa545
17.3.2 Rešenje upotrebom jakog detektora otkaza S545
17.3.3 Rešenje upotrebom jakog detektora otkaza sa odlaganjem $\diamond S$..	.547
17.4 Atomični brodkast550
17.5 Rešenje za atomični brodkast551
17.6 Najslabiji detektori otkaza koji rešavaju osnovni problem dogovora ..	.552
17.6.1 Realistični detektori otkaza553
17.6.2 Najslabiji detektor otkaza za konsenzus555
17.6.3 Najslabiji detektor otkaza za terminacioni pouzdani brodkast ..	.555
17.7 Implementacija detektora otkaza556
17.8 Protokol za adaptivnu detekciju otkaza558
17.9 Literatura562

Predgovor

Konkurentni i distribuirani sistemi više nisu egzotična oblast koja se povremeno izučava na master ili doktorskim studijama. Današnji programi su inherentno konkurentni i/ili distribuirani, počev od multiprocesorskih sistema, implementacija GUI (sistemi zasnovani na događajima), preko operativnih sistema, sistema u realnom vremenu pa sve do internet aplikacija kao što su IoT, blockchain, P2P, itd. pri čemu tu treba uključiti infrastrukturu i samog interneta (algoritmi i protokoli prenosa i rutiranja informacija).

Ova knjiga je nastala kao rezultat višegodišnjeg iskustva u nastavi na predmetu [8015] *Konkurentni i distribuirani sistemi*, koji se izvodi na studijskim programima osnovnih akademskih studija Računarske nauke i Računarsko inženjerstvo na Računarskom fakultetu Univerziteta Union u Beogradu. Iako je u početku bila namenjena isključivo kao udžbenik za ovaj predmet, ispostavilo se da ona ima i širu primenu. Knjiga može da koristi svakome ko želi da nauči kako konkurentni i distribuirani sistemi funkcionišu i zbog čega nekada, pored svog uloženog truda u njihov razvoj, ne funkcionišu. Potrebno predznanje studenata je na nivou analize sekvencijalnih algoritama. Poželjno je poznavanje funkcionisanja operativnih sistema i programskog jezika Java.

Primeri u knjizi, koji su dati u programskom jeziku Java, namenjeni su isključivo obrazovanju. Namerno su izostavljeni delovi koji proveravaju moguće greške, izuzetke i slično, a pojedini delovi koda i nisu u izvršnom obliku, već predstavljaju neku vrstu pseudo koda.

Knjiga nema nameru da bude sveobuhvatna. Opisani su osnovni pojmovi i principi iz kojih se dalje mogu izvoditi novi rezultati koji su specifični za datu primenu. Ukazano je na važnost generalizacije, apstrakcije i transparentnosti pojedinih rešenja, kao npr. kod transakcione memorije ili distribuirane deljene memorije.

Materijal za knjigu prikupljan je iz različitih izvora, pri čemu je u početku korišćen u okviru *Beleški sa predavanja* koje su bile osnovna literatura za predmet *Konkurentni i distribuirani sistemi*. Sasvim je moguće da se neke rečenice i fraze pojavljuju bez navođenja izvora. Autor je uložio maksimalan napor da naknadno citira upotrebljene resurse, pri čemu se citirana literatura nalazi na kraju pojedinih poglavlja. Svaka sličnost sa literaturom koja nije citirana je nenamerna i autor na te rezultate ne polaže nikakva autorska prava. U pojedinim delovima knjige korišćene su informacije i iz drugih knjiga koje se bave konkurentnim i distribuiranim sistemima, i to:

- Terminologija i klasifikacija preuzeta je iz knjige: Igor Ikodinović, Zoran Jovanović, *Konkurentno programiranje: Teorijske osnove sa zbirkom rešenih zadataka*, Akademska misao, 2004.
- Prvi deo knjige, koji se odnosi na konkurentne sisteme, zasnovan je na rezultatima koji su opisani u knjizi: Maurice Herlihy, Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2012.

- Drugi deo knjige, koji pokriva distribuirane sisteme, zasnovan je rezultatima koji su opisani u knjizi: Ajay D. Kshemkalyani, Mukesh Singhal, *Distributed Computing: Principles, Algorithms, and Systems*, Cambridge University Press, 2008.
- Neki primeri u Javi za distribuirane sisteme preuzeti su iz knjige: Vijay K. Garg, *Concurrent and Distributed Computing in Java*, John Wiley & Sons, 2004.

Bez obzira na uloženi napor, u tekstu sigurno postoje greške. Mole se savesni čitaoci da greške koje pronađu prijave izdavaču, a autor im na tome unapred zahvaljuje.

U Beogradu, januara 2019. godine

Prof. dr Stevan A. Milinković

0. Terminologija i klasifikacija

Napredak u oblasti arhitekture multiprocesorskih računarskih sistema prirodno je pratila i potreba za razvojem formalnih programskih metoda koje bi omogućile njihovo efikasno korišćenje. Od početka šezdesetih godina prošlog veka, kada je počeo razvoj ove oblasti, pa do danas, formulisan je velik broj paradigmi konkurentnog programiranja oličenih u raznim programskim jezicima i bibliotekama. Pri tom se može uočiti jaka veza između načina pisanja programa (programerske paradigme) i tipa izvršne platforme. To je posledica činjenice da programski prevodioci nisu u stanju da generišu efikasan kôd za različite računarske arhitekture na osnovu izvornog koda pisanog upotrebom jedne iste programerske paradigme. Ovo se prvenstveno odnosi na multiprocesorske arhitekture. Kod njih je, pored arhitekture samih procesora, od vitalnog značaja i memorijski podsistem, uključujući i interkonekciju mrežu, koji je daleko kompleksniji i sa još većim uticajem na performanse nego memorijski podsistem jednoprocorskih sistema. Nove arhitekture često traže stvaranje novih, njima prilagođenih formi konkurentnog programiranja.

Odvajanje paradigme programiranja od izvršne platforme ni do danas nije postignuto. Sa jedne strane, postoje mašinski programi koji potpuno zavise od izvršne platforme i imaju fiksiran algoritam rada kojeg programski kôd samo implementira na odgovarajućoj računarskoj arhitekturi. Zadatak asemblera sveden je na puko sintakso prevođenje asemblerskih naredbi u odgovarajuće mašinske instrukcije. S druge strane nalaze se programi koji su potpuno nezavisni od načina implementacije izvršne platforme i ne definišu neki fiksni algoritam rada. Odgovarajući prevodilac bi u tom slučaju imao zadatak da samostalno pronalazi algoritme i konstruiše mašinske programe u skladu sa tipom izvršne platforme, ali i drugim kriterijumima, kao što su vrsta problema, performanse, itd. Savremeni programski prevodioci imaju mogućnosti koje se nalaze negde između ove dve krajnosti. Oni omogućuju određeni nivo nezavisnosti programa od izvršne platforme, ali to i dalje nije na tako visokom nivou da bi prevodilac mogao da generiše efikasan kôd za različite tipove multiprocesorskih arhitektura. Zato je u praksi i dalje prisutan velik broj paradigmi konkurentnog programiranja.

0.1 Paradigme konkurentnog programiranja

Podrška za različite tipove paradigmi programiranja može se realizovati na različitim nivoima:

1. Na nivou arhitekture, tj. naredbi mašinskog jezika. Određene arhitekture podržavaju hardverski implementirane LOCK/UNLOCK ili BARRIER primitive (npr. Intel Itanium).

2. Na nivou operativnog sistema. Skoro svi savremeni operativni sistemi imaju mogućnost raspoređivanja sa istiskivanjem, što dozvoljava efikasno programiranje upotrebom niti.
3. Na nivou programskih prevodilaca za odgovarajuće programske jezike. Npr., u okviru programskog jezika Java implementirana je paradigma monitora.
4. Na nivou programskih biblioteka. Na primer, MPI biblioteka implementira primitive za programiranje putem prosleđivanja poruka.

Zavisno od toga na kom nivou je implementirana podrška za neku paradigmu programiranja, zavisice performanse, prenosivost i lakoća razvoja i održavanja određenog programa. Programi pisani na mašinskom jeziku obično su veoma efikasni, ali očigledno nisu prenosivi i teški su za razvoj i održavanje. Nasuprot tome, ako je podrška paradigme programiranja implementirana na nivou biblioteke za neki programski jezik, efikasnost možda nije maksimalna, ali su programi prenosivi i relativno laki za održavanje, s obzirom na viši nivo apstrakcije korišćenih primitiva. Ove druge dve osobine obično imaju prevagu nad prvom, u slučaju kada razlika u performansama nije velika.

0.2 Terminologija

U literaturi se mogu sresti različiti pojmovi u vezi sa oblašću konkurentnog programiranja koji su često nedovoljno jasno definisani ili se koriste sa različitim značenjima u različitim kontekstima. Zato ćemo ovde definisati neke osnovne pojmove.

- *Paralelno izvršavanje* označava istovremeno izvršavanje više računarskih operacija, sekvenci operacija, programa ili delova jednog programa.
- *Paralelni program* je svaki program koji koristi posebnu sintaksu sa označavanjem delova programskog koda koji se mogu izvršavati paralelno.
- *Paralelni računarski sistem* je svaki računarski sistem koji je u stanju da istovremeno izvršava dva ili više delova jednog ili više programa.
- *Sekvencijalno izvršavanje* je takvo izvršavanje kod kog se sledeća računarska operacija ili programska naredba izvršava tek nakon što je prethodna završena, u skladu sa redosledom koji je zadat programom.
- *Sekvencijalni program* je program kod kog postoji samo jedan sekvencijalni tok izvršavanja u vremenu. To znači, bez obzira na kakvoj računarskoj platformi se program izvršava, da će se, kada su ulazni parametri isti, uvek izvršavati iste naredbe i istim jedinstvenim redosledom koji je zadat izvornim programom.
- *Nit toka kontrole* (engl. *Thread of Control*), *tok izvršavanja programa* (engl. *Execution Flow*), *programski tok* (engl. *Program Flow*) ili *instrukcijski tok* (engl. *Instruction Flow*) je naziv za skup naredbi programa koje se izvršavaju određenim sekvencijalnim redosledom. Program može imati jednu ili više niti. Sekvencijalni program ima samo jedan tok izvršavanja u vremenu (jednu nit),

mada može da ima više statičkih tokova izvršavanja. Na primer, svako uslovno grananje u programu deli odgovarajući statički tok na dva dela, ali će tokom izvršavanja programa samo jedan od njih biti izabran, tj. postojaće samo jedan tok izvršavanja u vremenu. Paralelni program može imati kako više statičkih tokova izvršavanja, tako i više programskih tokova koji se mogu izvršavati istovremeno.

Teorijski gledano, moguće je konstruisati takav programski prevodilac koji bi i sekvencijalne programe prema nekom algoritmu prevodio tako da se izvršavaju paralelno na odgovarajućem računarskom sistemu. I obrnuto, kao što paralelno izvršavanje u krajnjoj liniji ne mora podrazumevati postojanje paralelnih programa, tako ni paralelni programi ne moraju podrazumevati paralelno izvršavanje. Danas postoje prevodioci koji omogućavaju paralelno izvršavanje sekvencijalnih programa, ali samo na nivou paralelnog izvršavanja mašinskih instrukcija (na primer izvršavanje sekvencijalnih programa na superskalarnim procesorima). Drugi slučaj, gde se sekvencijalno izvršavaju paralelni programi, često se sreće u jednoprocorskim sistemima (sa jednim jezgrom) sa operativnim sistemima koji podržavaju *raspoređivanje sa istiskivanjem*.

Konkurentno izvršavanje podrazumeva izvršavanje više programskih tokova jednog programa tako da oni napreduju u vremenu (bar dva od njih), ali se oni ne moraju obavezno izvršavati istovremeno, kao što je to bio slučaj kod paralelnog izvršavanja.

Mada pojmovi konkurentnog i paralelnog izvršavanja zvuče slično, među njima postoji razlika. Paralelno izvršavanje podrazumeva da je izvršna platforma paralelni računarski sistem na kom se zaista mogu istovremeno izvršavati delovi paralelnog programa, dok se pojam konkurentnog izvršavanja može primeniti i na jednoprocorskim sistemima. Termin *konkurentni program*, koji bi označavao program kod koga se dva ili više programskih tokova konkurentno izvršavaju, je sinonim za paralelni program, jer bi razlika bila samo u načinu njihovog izvršavanja i izvršnoj platformi, a ne u sintaksi i semantici samog programa. Kao što smo već ranije rekli, program i izvršna platforma su različite kategorije i spaja ih, odnosno deli, programski prevodilac. Na sličan način, termin *konkurentno programiranje* bio bi sinonim za paralelno programiranje, pod uslovom da on podrazumeva izvršnu platformu. Mi ćemo koristiti termine konkurentno izvršavanje i konkurentno programiranje, jer oni mogu da podrazumevaju bilo kakvu izvršnu platformu. Pojam *konkurentnost* označava potencijal za paralelizam koji postoji u programu, odnosno statički, programom definisani potencijalni paralelizam u izvršavanju.

Zabunu često unose i hardverski termini koji prate novije procesore. Koristeći tzv. *instrukcijski paralelizam* u programu ili tzv. *paralelizam fine granularnosti* (engl. *Fine Grained Parallelism*), procesori sa protočnom arhitekturom, kao i superskalarni procesori, omogućavaju istovremeno izvršavanje više faza različitih mašinskih instrukcija. Međutim, ovde je reč o vrsti paralelnog izvršavanja *sekvencijalnih* programa, tako da procesori ovog tipa nemaju direktan uticaj na paradigme konkurentnog programiranja. Ovakvi procesori spadaju u kategoriju SISD (engl. *Single*

Instruction Single Data) arhitektura, gde svaka mašinska naredba sadrži i podatke (operande) i odgovarajući instrukcijski kôd jedne operacije koju treba izvršiti nad tim podacima.

Danas se često sreću i procesori sa ugrađenom podrškom za tzv. SIMD (engl. *Single Instruction Multiple Data*) obradu, gde jedna ista mašinska instrukcija određuje paralelno izvršavanje iste operacije nad skupom različitih podataka. U ovom slučaju, reč je o hardverskoj podršci paralelnom programiranju, što podrazumeva i postojanje eksplicitnih sintaksnih elemenata u okviru paradigmi koje se koriste za njihovo programiranje. Paradigma programiranja prilagođena ovakvoj arhitekturi naziva se *Data Parallel Programming*.

SMT (engl. *Simultaneous Multi-Threading*) procesori su sposobni za istovremeno izvršavanje više programskih niti, što podrazumeva i korišćenje odgovarajućih paradigmi konkurentnog programiranja. Ovi procesori spadaju u MIMD (engl. *Multiple Instruction Multiple Data*) arhitekture. Njihova prednost je što koriste *paralelizam veće granularnosti* (engl. *Coarse Grained Parallelism*) u odnosu na instrukcijski paralelizam. Instrukcijski nivo paralelizma ograničen je zavisnošću između podataka kao i prosečnom frekvencijom skokova u programu. Prvi problem se ublažava primenom tehnika optimizacije koda (npr. softverske protočnosti – engl. *Software Pipelining*), a drugi tehnikama predviđanja skokova (engl. *Branch Prediction*) i keširanja. Međutim, i ove tehnike imaju svoja ograničenja. Paralelizam veće granularnosti dozvoljava da programer, prilikom pisanja programa, implementacijom odgovarajućih paralelnih algoritama omogući veće iskorišćenje resursa paralelnog sistema, a time i brže izvršavanje programa.

0.3 Klasifikacija konkurentnih i distribuiranih sistema

Za oblast konkurentnog programiranja od značaja su najviše sistemi sa MIMD arhitekturom koji se dele na: *sisteme sa deljenom memorijom* (engl. *Shared-Memory Systems*), *multiračunare sa distribuiranom memorijom* (engl. *Distributed-Memory Multicomputers*) i *mreže računara*.

Sistemi sa deljenom memorijom imaju veći broj potklasa, zavisno od načina međusobnog povezivanja procesora i memorije. Navešćemo samo najvažnije potklase.

- *Sistemi sa uniformnim pristupom memoriji* (engl. *Uniform Memory Access – UMA*), poznati i kao sistemi sa simetričnim pristupom memoriji (engl. *Symmetric MultiProcessor Systems – SMP*), imaju fizički jedinstvenu memoriju kojoj svi procesori pristupaju na isti način preko interkonekcionih mreža, obično jednostavnije topologije tipa magistrale (engl. *bus*) ili mreže (engl. *grid*), i to tako da je vreme pristupa svih procesora memoriji isto (uniformno). Međutim, procesori mogu imati svoje privatne keš memorije.
- *Sistemi sa neuniformnim pristupom memoriji* (engl. *Non-Uniform Memory Access – NUMA*) imaju fizički jedinstvenu, ali hijerarhijski organizovanu memoriju. Procesori su povezani sa memorijom preko interkonekcionih mreža

koja obično ima složeniju topologiju. Vreme pristupa svakog procesora memoriji nije uniformno i zavisi od lokacije podataka. Što se podatak nalazi dublje u memorijskoj hijerarhiji, više vremena je potrebno za pristup do njega.

- *Sistemi sa distribuiranom deljenom memorijom* (engl. *Distributed Shared Memory – DSM*) imaju logički jedinstvenu, ali fizički distribuiranu memoriju. Svaki procesor ima svoju privatnu memoriju koja je istovremeno deo zajedničkog adresnog prostora sa drugim privatnim memorijama u sistemu. Procesori su međusobno povezani preko interkonekcionih mreže.

Multiračunar sa distribuiranom memorijom je računarski sistem koji se sastoji od procesora koji su međusobno povezani preko interkonekcionih mreže, a svaki od njih ima svoju privatnu memoriju. Za razliku od DSM, privatna memorija svakog procesora nije deo globalnog adresnog prostora. Komunikacija između procesora postiže se prosleđivanjem poruka kroz mrežu. Postoje različite topologije interkonekcionih mreža (engl. *switch, grid, mesh, hypercube*, itd.) kao i različiti protokoli za slanje poruka kroz njih.

Mreža računara se, kao što i samo ime kaže, sastoji od više nezavisnih računara koji su fizički povezani preko računarske mreže. Računari komuniciraju prosleđivanjem poruka koristeći za to predviđene protokole.

0.4 Klasifikacija paradigmi konkurentnog programiranja

Različiti modeli programiranja (paradigme) za konkurentno programiranje formirani su u skladu sa različitim računarskim arhitekturama i njihovim tipičnim parametrima, kao što su prosečno vreme kašnjenja prilikom pristupa memoriji, tip arhitekture memorijskog podsistema, broj procesora u sistemu itd., a na višem nivou i prema domenu primene (naučna izračunavanja, inteligentni sistemi, itd.). Njihov zadatak je da programeru omoguće da na efikasan i razumljiv način može da formuliše programe koji će postići željeni cilj.

Postoji više različitih paradigmi konkurentnog programiranja, koje se prema tipu interakcije među procesima mogu podeliti na dve velike grupe:

Programiranje pomoću deljenih promenljivih (engl. *Shared Variable Programming*). Program sadrži aktivne procese i pasivne deljene promenljive. Interakcija među procesima je putem čitanja i pisanja u deljene promenljive.

Distribuirano programiranje (engl. *Distributed Programming*). U programu nema deljenih objekata. Interakcija među procesima je zasnovana na:

- *Eksplisitnom slanju i primanju poruka*. Poruka je neki objekat koji se prosleđuje drugom procesu upotrebom komunikacionih primitiva (npr. *send* i *receive*).
- *Pozivima udaljenih procedura* (engl. *Remote Procedure Call – RPC*, ili *Remote Method Invocation – RMI*), gde pozivajuća i pozvana procedura pripadaju različitim procesima. Prilikom svakog poziva udaljene procedure, pozvanom procesu se šalju odgovarajući parametri, nakon čega on izvršava

pozvanu proceduru i šalje rezultat nazad pozivajućem procesu (pozivajući proces je do tog trenutka blokiran). Prihvatanje poziva od strane pozvanog procesa je implicitno, tj. unutar pozvanog procesa nema eksplicitnih naredbi za prihvatanje poziva, već se one prihvataju čim stignu. Po prihvatanju poziva najčešće se kreira nova programska nit ili proces u okviru kog se izvršava pozvana procedura. Ređe se primenjuje rešenje kod kog postoji samo jedna nit ili proces za izvršavanje pozvane procedure za sve pozivajuće procese. Tada pozivajući procesi čekaju blokirani u redu čekanja za pozivanje udaljene procedure. Interakcija pomoću poziva udaljenih procedura je uvek potpuno sinhrona.

- *Mehanizmu randevua* (engl. *Rendezvous*). Prilikom poziva servisne procedure dva procesa čekaju jedan drugog, pre nego što nastave sa izvršavanjem. Pozivajući proces čeka da pozvani proces dođe do tačke prihvatanja randevua, a pozvani proces čeka da pozivajući proces stigne do tačke poziva (zavisno od toga koji proces stigne prvi do tačke poziva/prihvatanja, desiće se jedan od ova dva scenarija). Razlika između randevua i poziva udaljenih procedura je što u slučaju randevua kod pozvanog procesa postoji eksplicitno definisana tačka prihvatanja poziva na kojoj se on blokira dok ne stigne poziv. Kod poziva udaljenih procedura nema blokiranja pozvanog procesa, već se po dobijanju poziva po pravilu odmah kreira nova nit ili proces od strane pozvanog procesa.

Pored navedenih, postoje i drugi parametri prema kojima se može vršiti klasifikacija paradigmi konkurentnog programiranja. Jedan od najopštijih i najčešće korišćenih je prema modelu koordinacije. Kod ovog tipa klasifikacije polazi se od činjenice da se sve aktivnosti programa mogu podeliti na izračunavanje (engl. *Computation*) i koordinaciju (engl. *Coordination*). Ova dva aspekta su međusobno ortogonalna i mogu se posmatrati nezavisno. Pojam koordinacije, pored načina na koji međusobno interaguju aktivni delovi programa, obuhvata i način na koji su oni organizovani u programsku celinu (kreiranje i uklanjanje aktivnih delova programa, njihovu prostornu distribuciju i sinhronizaciju u vremenu). Svaki koordinacioni model ima tri elementa:

- *Entitete čiji se rad koordinira*. To su aktivni delovi programa koji čine osnovne gradivne komponente sistema. Npr. procesi, niti, agenti, itd.
- *Koordinacioni medijum*. Medijum koji služi za koordinaciju programskih entiteta. Npr. kanali (engl. *Channels*), deljene promenljive (engl. *Shared Variables*), prostori podataka (engl. *Data Spaces*), itd.
- *Pravila koordinacije*. Ova pravila određuju kako entiteti koordiniraju rad korišćenjem koordinacionog medijuma. Npr. skup primitiva za upis i čitanje u prostor podataka, itd.

U praksi se najčešće sreću sledeća dva koordinaciona modela:

- *Model virtuelnog prostora* predstavlja vrstu proširenja modela programiranja pomoću deljene memorije. Virtuelni prostor je koordinacioni medijum koji predstavlja poseban deo programskog prostora kome može da pristupa više aktivnih delova programa. Ukoliko je taj prostor rezervisan samo za podatke, onda se za njega koristi naziv *virtuelna deljena memorija* (engl. *Virtual Shared Memory*). U opštem slučaju, virtuelni prostor osim pasivnih objekata može da sadrži i aktivne delove programa.
- *Programiranje upotrebom programskih niti*. Koordinacioni model se zasniva na nezavisnim programskim tokovima (nitima) koji međusobno mogu da komuniciraju primenom neke od komunikacionih paradigmi. Ovakav način programiranja naziva se *Multithreading*.

0.5 Programiranje pomoću deljenih promenljivih

Programiranje pomoću deljenih promenljivih najčešće se koristi za programiranje na računarskim sistemima sa deljenom memorijom. Ovaj način programiranja podrazumeva da se za komunikaciju između procesa u konkurentnom programu koriste globalne promenljive kojima mogu da pristupe svi procesi bez ograničenja (deljene promenljive). Nasuprot njima, nalaze se privatne (lokalne) promenljive kojima mogu da pristupe samo procesi kojima one pripadaju. Postoji više različitih programskih paradigmi koje se koriste za pisanje programa u kojima se komunikacija i sinhronizacija između procesa vrši pomoću deljenih promenljivih. Među njima su najpoznatiji semafori, uslovni kritični regioni i monitori.

Koncept *semafora*, kao i samo ime, potiče od mehanizma koji se koristi u železničkom saobraćaju za sprečavanje sudara vozova. U programskom smislu, vozovi su *procesi*, železničke šine su *kritične sekcije*, a semafori imaju ulogu da skreću i zaustavljaju procese da ne bi došlo do „sudara“, tj. da više vozova ne bi ušlo u kritičnu sekciju. Pomoću njih se jednostavno realizuju međusobno isključivanje procesa i uslovna sinhronizacija procesa. Iz tog razloga prirodna je i njihova upotreba za realizaciju konkurentnih programa u vidu posebne paradigme programiranja. Zbog jednostavnosti korišćenja, semafori su uključeni u gotovo sve postojeće biblioteke za konkurentno programiranje. Semafori mogu da budu implementirani na različitim nivoima, a najčešće se realizuju na nivou operativnog sistema ili na nivou programskih biblioteka.

Kod paradigme *kritičnih regiona*, glavni akcenat se stavlja na kritičnu sekciju kao deo programa u kome se pristupa deljenim promenljivama. Zbog toga je realizovana apstrakcija ovakvog pristupa uvođenjem posebno označenih sintaksnih celina – kritičnih regiona. Mehanizam uzajamnog isključivanja realizovan je transparentno za programera, čime je njegov posao značajno olakšan, a čitljivost i razumljivost programa povećana.

Kod paradigme *uslovnih kritičnih regiona*, izvršeno je dodatno sintaksno proširivanje u odnosu na obične kritične regione, kako bi se omogućila uslovna sinhronizacija procesa. Uvedena je naredba `await` koja dozvoljava da se sinhronizacioni uslov postavi unutar kritičnih regiona.

Monitori predstavljaju još viši nivo apstrakcije sinhronizacionog mehanizma u odnosu na uslovne kritične regione. Monitor enkapsulira skup deljenih promenljivih u objekat nad kojim su definisane moguće operacije u vidu skupa procedura. Uza- jamno isključivanje procesa prilikom pristupa monitoru za korisnika je obezbeđeno potpuno transparentno. To se postiže zabranom konkurentnog izvršavanja procedura monitora. Uslovna sinhronizacija obavlja se eksplicitno, preko uslovnih promenljivih. Uslovne promenljive su donekle slične semaforima, pri čemu je red čekanja na uslovnoj promenljivoj po definiciji tipa FIFO. Kod semafora nije specificovano kog tipa mora da bude red čekanja, pa postoje *jaki semafori* (red čekanja je FIFO) i *slabi semafori* (nije definisan red čekanja).

0.6 Distribuirano programiranje

Distribuirano programiranje se najčešće koristi za programiranje na računarskim sistemima sa distribuiranom memorijom. Kod njih ne postoji zajednička memorija koju dele svi procesori, već svaki procesor ima svoju privatnu memoriju, pa međusobno mogu da komuniciraju samo razmenom poruka preko interkonekcionu mrežu. Da bi razmena poruka bila omogućena u programu, mora se definisati odgovarajući programski interfejs preko koga će se implementirati primitive za slanje i prijem poruka. On može da bude uprošćen, kakav bi npr. bio interfejs koji definiše samo operacije za čitanje i upis podataka preko mreže (za pristup podacima koji pripadaju procesima koji se izvršavaju na drugim procesorima i ne nalaze se u lokalnoj memoriji) – analogno operacijama čitanja i pisanja nad deljenim promenljivama. Međutim, u tom slučaju bi sinhronizacija između procesa pošiljaoca i procesa primaoca morala da se radi pomoću uposlenog čekanja, što je nepraktično. Zato se u okviru komunikacionog interfejsa obično definišu nešto složenije primitive za slanje i prijem poruka, koje u sebi uključuju i sinhronizaciju. Takve primitive za slanje i prijem poruka mogu se smatrati nekom vrstom semafora koji, osim što vrše sinhronizaciju, ujedno prenose i podatke. Komunikacija, tj. prosleđivanje poruka između procesa, vrši se preko *komunikacionih kanala*. Kanal apstrahuje interkonekcionu mrežu i fizičku vezu koja postoji između procesora, tako da program ne mora da zna ništa o hardverskoj arhitekturi sistema.

Konkurentni programi koji koriste prosleđivanje poruka nazivaju se *distribuirani programi*, jer procesi mogu biti raspoređeni i izvršavati se na više procesora u okviru računarskog sistema sa distribuiranom memorijom. Međutim, distribuirani program se može isto tako izvršavati i na sistemu sa zajedničkom memorijom, realizovanjem komunikacionih kanala pomoću deljenih promenljivih (moguće je i obrnuto, da se deljene promenljive implementiraju na distribuiranoj platformi). Ovo je još jedan primer koji pokazuje da su paradigma programiranja i izvršna platforma u

suštini nezavisni. Naravno, popularnost primene neke paradigme uvek je uslovljena efikasnošću izvršavanja na datoj platformi.

Zavisno od toga kakvi su komunikacioni kanali (jednosmerni ili dvosmerni), kako se imenuju (direktno ili indirektno) i kako se vrši sinhronizacija prilikom slanja i prijema poruka (sinhrono ili asinhrono), mogu se kreirati različiti programski interfejsi i odgovarajuće primitive za komunikaciju putem prosleđivanja poruka. Neka paradigma može da sadrži i primitive za realizaciju slanja i prijema poruka preko više različitih tipova kanala.

Paradigma programiranja putem *javnog emitovanja* ili brodkasta (engl. *Broadcast*) podrazumeva postojanje komunikacionog kanala koji istovremeno povezuje sve procese. Komunikacija se odvija tako što proces pošiljalac istovremeno šalje poruku svim procesima, a svaki proces primalac odlučuje za sebe da li će je prihvatiti ili ne. Ova paradigma predstavljena je pomoću programskog modela *Broadcasting Sequential Processes* (BSP). Pored brodkasta, BSP model nudi i mogućnost *usmerenog emitovanja* – multikasta (engl. *Multicast*) i direktne – P2P (engl. *Point-to-Point*) komunikacije između dva procesa. Komunikacioni kanal za slanje i prijem poruka uvek je jednosmeran, a koristi se direktno imenovanje procesa. Brodkast uvek podrazumeva asinhrono slanje, jer je nerealno očekivati da pošiljalac čeka da svi procesi budu spremni da prime poruku. Takođe, prenos može da bude nebaferovan ili baferovan. Kod nebaferovanog prenosa, poruka koju proces primalac propusti da primi zato što nije bio spreman, biva zauvek izgubljena. Kod baferovanog prenosa, poruke koje stižu ostaju u baferu procesa sve dok on ne bude spreman da ih primi.

Communicating Sequential Processes (CSP) je programski model kojim je predstavljena paradigma programiranja sa sinhronim slanjem i sinhronim (blokirajućim) prijemom poruka, uz korišćenje direktnog imenovanja i dvosmernih kanala. Kod slanja i prijema poruka koristi se jedna vrsta jednostavnog prepoznavanja šablona (engl. *Pattern Matching*) kako bi se lakše strukturale poruke i kako bi se jednostavnije pristupalo njihovim komponentama kod prijema, obezbeđujući ujedno da proces primalac ne prima poruke koje ne odgovaraju zadatom šablonu.

Kod programske paradigme sa *indirektnim imenovanjem procesa*, ovo imenovanje se realizuje uvođenjem komunikacionih portova, pri čemu se tip kanala (jednosmeran ili dvosmeran) definiše prilikom deklaracije porta. Vremenski ograničeno (koristi se i termin uslovljeno) asinhrono slanje poruka vrši se pomoću opcione `wait` primitive unutar komande `send`, koja omogućava vremenski ograničeno blokiranje procesa pošiljaoca prilikom čekanja na odgovor.

0.7 Virtuelni prostori

Programski prostor se može definisati kao prostor koji vidi programer kada piše program. U njemu se nalaze kako aktivni objekti (proces, niti, agenti), tako i pasivni podaci. Taj prostor ne mora da bude jednodimenzionalan, kao kod klasičnih paradigmi konkurentnog programiranja koje koriste programski prostor i kao prostor za

koordinaciju i kao prostor za izračunavanja, već se može konceptualno organizovati na različite načine. Način organizacije programskog prostora često je uslovljen primenom u nekoj oblasti.

Virtuelni programski prostor je izdvojeni deo programskog prostora koji služi za koordinaciju aktivnih delova programa. To je njihov koordinacioni medijum. U opštem slučaju, virtuelni prostor može obuhvatati i podatke i aktivne programske entitete. Ukoliko je virtuelni prostor rezervisan samo za podatke, onda se za njega koristi naziv *virtuelna deljena memorija* (engl. *Virtual Shared Memory*) ili *prostor podataka* (engl. *Data Space*). Sam naziv „virtuelni“ potiče od činjenice da izdvojeni prostor postoji samo u kontekstu koordinacionog modela, dok je fizički on realizovan u globalnom programskom prostoru.

0.8 Programske niti

Aktivnost procesa karakteriše redosled u kome se izvršavaju naredbe programa. Ovaj redosled se naziva *trag* (engl. *trace*) procesa. Trag procesa može da se prikaže kao *programska nit* (engl. *thread*) koja povezuje izvršene naredbe u redosledu njihovog izvršavanja. Ako se za proces veže jedna nit, tada ona ima atribute procesa. U toku aktivnosti niti izvršavaju se naredbe koje ona povezuje.

Veživanje samo jedne niti za proces je posledica pretpostavke da proces odgovara izvršavanju sekvencijalnog programa. Za sekvencijalni program se podrazumeva da se naredbe izvršavaju jedna za drugom po redosledu koji zavisi od obrađivanih podataka. Ovakve naredbe obrazuju sekvencu naredbi.

Praksa pokazuje da je nekada korisno za procese vezati više od jedne niti. U tom slučaju stek, prioritet i stanje se ne vezuju za proces, nego za njegove niti. Znači, svaka od niti procesa ima svoj stek, svoj prioritet i svoje stanje, kao i svoj deskriptor (ID). Na primer, za editovanje teksta su važne dve aktivnosti. Prva je interakcija sa korisnikom, a druga je zadužena za periodično smeštanje unesenog teksta na disk, radi sprečavanja gubljenja teksta. Ako bi za editorski proces bila vezana jedna nit, tada bi se pomenute aktivnosti odvijale jedna za drugom (sekvencijalno). Znači, za vreme trajanja druge aktivnosti nije moguća interakcija sa korisnikom, što je ozbiljan nedostatak. Taj nedostatak se može otkloniti, ako se za editorski proces vežu dve niti raznih prioriteta. Nit višeg prioriteta se pridružuje prvoj aktivnosti, a manje prioriteta (pozadinska) nit se pridružuje drugoj aktivnosti. Ako nit višeg prioriteta čeka, pozadinska nit može da bude aktivna sve dok, na primer, pritisak tastera na tastaturi ne najavi početak interakcije sa korisnikom. Tada obrada odgovarajućeg prekida prevodi nit višeg prioriteta u spremno stanje, pa se procesor prebacuje sa prekinute pozadinske niti na nit višeg prioriteta. U toku svoje aktivnosti, kada nit višeg prioriteta obavi interakciju sa korisnikom, ona se vraća u stanje čekanja, što dovodi do vraćanja procesora na prekinutu pozadinsku nit.

Kod paradigme programiranja pomoću programskih niti (engl. *multithreading*), koordinacioni prostor i računski prostor nisu razdvojeni. Glavni gradivni elementi programa kod ovog koordinacionog modela jesu programske niti. Programske niti

su nezavisni tokovi koji se konkurentno izvršavaju na datom računarskom sistemu. Sinhronizacija i koordinacija rada niti vrši se pomoću nekih od sinhronizacionih i komunikacionih paradigmi (semafori, monitori, prosleđivanje poruka, itd.).

0.9 Java

Java podržava rad sa više paradigmi konkurentnog programiranja. Ona obuhvata paradigme za programiranje pomoću deljenih promenljivih (npr. monitori su sastavni deo jezika, postoje biblioteke za rad sa semaforima, itd.) i distribuirano programiranje (biblioteke za rad sa pozivima udaljenih procedura, prosleđivanjem poruka, itd.). Posebnu pažnju zaslužuje model programiranja pomoću programskih niti. Rad sa nitima u jeziku Java implementiran je na nivou korisničkih niti.

0.9.1 Kreiranje i kontrola rada niti

Svaka klasa se može deklarirati kao nit na dva načina. Prvi način je prostim nasleđivanjem klase *Thread* koja je ugrađena u jezik i redefinisanjem njene metode *run*:

```
import java.lang.*;
public class MyThread extends Thread
{
    public void run()
    {
        . . .
    }
}
```

Drugi način za kreiranje niti je implementiranjem interfejsa *Runnable* koji je ugrađen u jezik i redefinisanjem njene metode *run*:

```
import java.lang.*;
public class MyThread implements Runnable
{
    Thread T;
    public void run()
    {
        . . .
    }
}
```

Nit se instancira sa:

```
MyThread = new MyThread(...);
```

a pokreće sa:

```
MyThread.start();
```

Nit se zaustavlja sama kada dođe do kraja metode `run()`.

Nakon pokretanja niti, nit koja ju je pokrenula uporedo nastavlja sa radom. Za čekanje da se nit završi koristi se:

```
MyThread.join();
```

Naredba

```
MyThread.yield();
```

predstavlja eksplicitni zahtev predaje kontrole raspoređivaču, koji može, ali ne mora (zavisí od implementacije), da nastavi izvršenje neke druge niti.

Pored standardnog načina blokiranja, nit može da se blokira i tokom nekog vremenskog perioda `t` sa:

```
MyThread.sleep(t);
```

0.9.2 Sinhronizacija

Deo koda, objekat ili metoda mogu se deklarirati kao kritična sekcija upotrebom ključne reči *synchronized*. Na primer:

```
synchronized (this) { . . . }
```

znači da je označeni deo koda kritična sekcija,

```
synchronized (referenca_na_objekat)
```

znači da se objektu na koji ukazuje referenca pristupa upotrebom uzajamnog isključivanja, a

```
public class nekaKlasa
{
    public synchronized void fun(int vrednost) {
        . . .
    }
}
```

znači da metoda `fun` ne može da se izvrši istovremeno ako je dve niti pozovu preko istog objekta, ali može ako se poziva preko dva različita objekta iste klase. Samo jedna nit dobija pravo pristupa monitoru kome pripada *synchronized* metoda, dok ostale niti moraju da čekaju da prethodna nit izađe. Monitori su u jeziku Java vezani za objekte, a ne za klase.

Operacija `wait` prekida izvršavanje niti i stavlja je u blokirano stanje (stanje čekanja), kada se privremeno skida zabrana drugim nitima da pristupaju monitoru. Nit će se odblokirati tek kada neka druga nit izvrši `notify()` ili `notifyAll()` naredbu za ciljani objekat. Postoje tri oblika operacije `wait`:

```
wait();  
wait(t_ms);  
wait(t_ms, t_ns);
```

`wait()` čeka neograničeno, a u druga dva slučaja se čeka određeno vreme da stigne notifikacija da može da se pristupi ciljnom objektu. Operacija:

```
notifyAll();
```

šalje notifikaciju svim nitima koje pomoću operacije `wait` čekaju da pristupe ciljnom objektu, da to mogu da učine. Ako ima više takvih niti, bira se jedna, a ostale i dalje čekaju. `notify()` radi isto što i `notifyAll()`, ali notifikuje samo jednu nit.

1. Uvod

1.1 Deljeni objekti i sinhronizacija

Prvog dana vašeg novog programerskog posla, vaš šef je zatražio da nađete sve proste brojeve između 1 i 10^{10} (nije važno zašto to od vas traži), upotrebom paralelnog računara koji podržava izvršavanje deset konkurentnih niti. Ovaj računar se iznajmljuje i plaća na minut, pa ako vaš program traje dugo, njegovo izvršavanje će više koštati vašu kompaniju. Vi, naravno, želite da ostavite dobar utisak na svog šefa. Šta ćete uraditi?

```
1 void primePrint {
2     int i = ThreadID.get(); // ID niti {0..9}
3     int block = power(10,9);
4     for (int j = (i*block)+1; j <= (i+1)*block; j++) {
5         if (isPrime(j))
6             print(j);
7     }
8 }
```

Slika 1.1. Balansiranje opterećenja deljenjem ulaznog opsega. Svaka nit {0..9} dobija jednaki podopseg.

U prvom pokušaju, mogli biste svakoj niti da dodelite isti deo ukupnog ulaznog prostora. Npr., svaka nit može da ispituje 10^9 brojeva, kao što je prikazano na slici 1.1 (testiranje da li je neki konkretan broj prost nije predmet našeg razmatranja). Predloženi pristup nije dobar zbog elementarnog, ali važnog razloga. Naime, jednaki ulazni opsezi brojeva ne znače automatski i jednaku dužinu izračunavanja. Prosti brojevi ne pojavljuju se uniformno. U opsegu između 1 i 10^9 ima dosta prostih brojeva, ali ih je teško naći u opsegu između $9 \cdot 10^8$ i 10^{10} . Da bi stvar bila još gora, vreme ispitivanja da li je neki broj prost nije isto u svim opsezima. Obično je potrebno više vremena da se proverí da li je neki veliki broj prost nego što je to slučaj sa malim brojem.

```
1 Counter counter = new Counter(1); // brojac kojeg dele sve niti
2 void primePrint {
3     long i = 0;
4     long limit = power(10,10);
5     while (i < limit) { // traje dok se ne ispituju svi brojevi
6         i = counter.getAndIncrement(); // sledeci netestirani broj
7         if (isPrime(i))
8             print(i);
9     }
10 }
```

Slika 1.2. Balansiranje opterećenja upotrebom zajedničkog deljenog brojača. Svaka nit dinamički dobija podopseg brojeva za testiranje.