Konkurentni i distribuirani sistemi

Stevan Milinković

CET

Računarski fakultet

Konkurentni i distribuirani sistemi

Stevan Milinković

ISBN 978-86-7991-412-5

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

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 razultatima 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 interkonekcionu mrežu, koji je daleko kompleksniji i sa još većim uticajem na performanse nego memorijski podsistem jednoprocesorskih 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 sintaksno 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, zavisiće 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 jednoprocesorskim 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 jednoprocesorskim 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 Pediction*) 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 interkonekcione mreže, 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 interkonekcione mreže

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 interkonekcione mreže.

Multiračunar sa distribuiranom memorijom je računarski sistem koji se sastoji od procesora koji su međusobno povezani preko interkonekcione 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:

- *Eksplicitnom 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. Uzajamno 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 specifikovano 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 interkonekcione mreže. 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* – miltikasta (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 struktuirale 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 (procesi, 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.

Vezivanje 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 prioritetna (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 deklarisati 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 (zavisi 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 deklarisati 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 ciljni 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 notifikacia 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 $9 \cdot 10^9$ 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 proveri 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 ispitaju 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.