Testiranje konkurentskih transakcija u Oracle bazi podataka pomoću Java paralelnog programiranja

Sirotić, Zlatko

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

ZLATKO SIROTIĆ

TESTIRANJE KONKURENTNIH TRANSAKCIJA U ORACLE BAZI PODATAKA POMOĆU JAVA PARALELNOG PROGRAMIRANJA

Diplomski rad

Pula, 2019.

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

ZLATKO SIROTIĆ

TESTIRANJE KONKURENTNIH TRANSAKCIJA U ORACLE BAZI PODATAKA POMOĆU JAVA PARALELNOG PROGRAMIRANJA

Diplomski rad

JMBAG: 0016102643, izvanredni student
Studijski smjer: Diplomski sveučilišni studij Informatika

Predmet: Suvremene tehnike programiranja
Znanstveno područje: Društvene znanosti
Znanstveno polje: Informacijske i komunikacijske znanosti
Znanstvena grana: Informacijski sustavi i informatologija

Mentor: doc. dr. sc. Siniša Sovilj

Pula, studeni 2019.

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Zlatko Sirotić, kandidat za magistra informatike, ovime izjavljujem da je ovaj Diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Diplomskog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, 15. studeni 2019.

IZJAVA
o korištenju autorskog djela

Ja, Zlatko Sirotić, dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom Testiranje konkurentnih transakcija u Oracle bazi podataka pomoću Java paralelnog programiranja koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.
Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, 15. studeni 2019.

Potpis

_____

Sadržaj

1. Uvod

Nije potrebno posebno naglašavati da su sustavi za upravljanje bazama podataka (SUBP) i dalje važni za spremanje i dohvat podataka. Ispravno rukovanje transakcijama u bazama podataka vrlo je bitno za integritet podataka. Nažalost, greške u rukovanju transakcijama obično se kod testiranja teže uoče, jer često ovise o spletu događaja, broju korisnika koji istovremeno rade i sl. Najčešće se puno lakše uoče npr. greške u korisničkom sučelju (što ne znači da su te greške nevažne, ili da ih je uvijek lako ispraviti). Zbog toga je vrlo korisno kada se testiranje može na neki način automatizirati, potpuno ili djelomično.

Kada se nad SUBP sustavima radi s Java programskim jezikom (i pripadajućim okruženjem), koristi se Java Database Connectivity (JDBC). JDBC je aplikacijsko programsko sučelje (API) za programski jezik Java, koje definira kako klijent može pristupiti i koristiti bazu podataka.

Već više od 15 godina brzina rada mikroprocesora vrlo se sporo povećava, iako se broj tranzistora na čipu povećava i dalje po Mooreovom zakonu. Današnji mikroprocesori imaju velike cache memorije (na tri ili više razina) i veći broj jezgri (koje su, zapravo, procesori). Zbog toga je vrlo važno iskoristiti te jezgre, koristeći konkurentne ili paralelne programe. Nažalost, konkurentne i paralelne programe nije lako pisati (barem ne imperativne, za razliku od deklarativnih). To vrijedi i za Java okolinu, koja je od početka imala podršku za konkurentno i paralelno programiranje, a od Jave 5 dalje dobila je i brojna poboljšanja.

U radu će se prikazati testiranje (složenog) poslovnog pravila u Oracle bazi podataka (pisanog pomoću Oracle PL/SQL pohranjenih procedura) pomoću Java paralelnog programa.

Nakon uvoda, u 2. poglavlju prikazuju se osnove rada s transakcijama (uključujući zaključavanje podataka) u Oracle bazi.

U 3. poglavlju prikazuju se primjeri rada sa Oracle bazom iz Java okoline, pomoću JDBC sučelja.

U 4. poglavlju prikazuju se detalji o utjecaju razvoja mikroprocesora na programiranje, kao i različiti sinkronizacijski mehanizmi. Posebno se prikazuju osnove hardverske transakcijske memorije, čime se želi pokazati kako se rješenja u mikro-hardverskom svijetu i makro-softverskom svijetu (npr. baze podataka) danas duboko prožimaju.

U 5. poglavlju prikazuju se tri različita načina rješavanja istog paralelnog programskog zadatka u Java okolini (koja su moguća od Jave 5, 7, i 8): rješenje pomoću Java executora, Java Fork Join frameworka i Java (paralell) Streamsa.

U 6. poglavlju primjenjujemo jednu od metoda paralelnog programiranja iz prethodnog poglavlja (Java executore) za testiranje programskog rješenja (pisanog pomoću Oracle PL/SQL pohranjenih procedura) koje sprečava pojavu petlje (u hijerarhijskoj strukturi podataka) u višekorisničkom radu.

To rješenje je značajno kompleksnije od rješenja koje bi se koristilo u jednokorisničkom radu. Zbog toga ga treba jako dobro testirati. Nažalost, teško je kvalitetno testirati bez velikog broja testera (osoba). Zbog toga koristimo Java paralelni program, koji zamjenjuje taj veliki broj testera. Lako je pokrenuti npr. stotinjak (ili više) paralelnih programa, koji zamjenjuju rad stotinjak testera.

2. Transakcije u Oracle bazi podataka

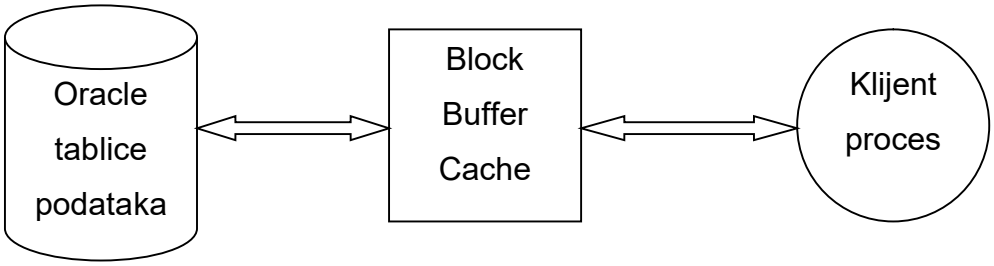
2.1. Osnove arhitekture Oracle DBMS-a (vezano za transakcije)

Oracle sustav čine dva glavna dijela:

- baza podataka (BP), koju čine različite vrste datoteka; najvažnije su datoteke podataka, no postoji i desetak drugih vrsta datoteka, od kojih su posebno važne Redo Log datoteke, koje se dijele na Online i Archive; Online Redo Log datoteke služe (između ostalog) za oporavak baze u slučaju pada sustava (npr. programske greške ili nestanaka struje), dok Archive Redo Log datoteke služe za uspostavu prijašnjeg stanja, zajedno sa backup datotekama (na trakama ili diskovima) u slučaju kvara medija (u pravilu – diska);

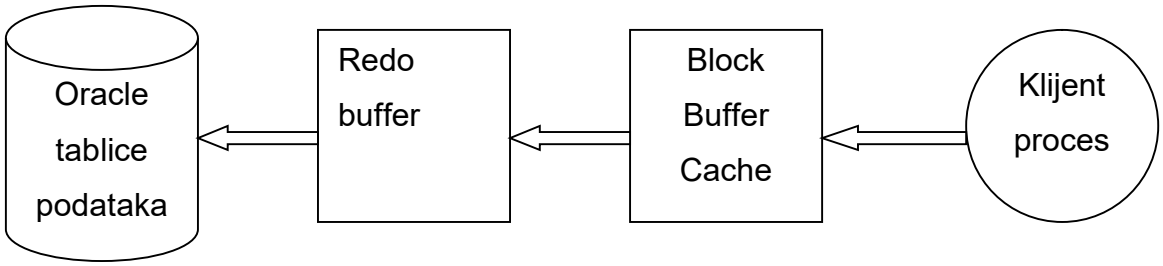
- instanca (jedna ili više) baze podataka, koju čine memorijske strukture i procesi u memoriji; od memorijskih struktura, naročito su zanimljivi Block Buffer Cache i Redo Bufer.

Na koji način radi čitanje (SELECT) i izmjena podataka (INSERT / UPDATE / DELETE) u Oracle sustavu? Klijentski procesi (klijentski proces može biti proces na drugom računalu, proces na istom računalu na kojem se nalazi i Oracle DBMS, ali može biti i neki proces unutar Oracle DBMS-a) nikad ne dobivaju podatke direktno sa diska. Svi podaci koji se čitaju sa diska smještaju se u Block Buffer Cache. Također, kada klijentski proces mijenja podatke, ne mijenja ih direktno na disku, već se promjene prvo spremaju u Block Buffer Cache, kako to prikazuje slika 1.



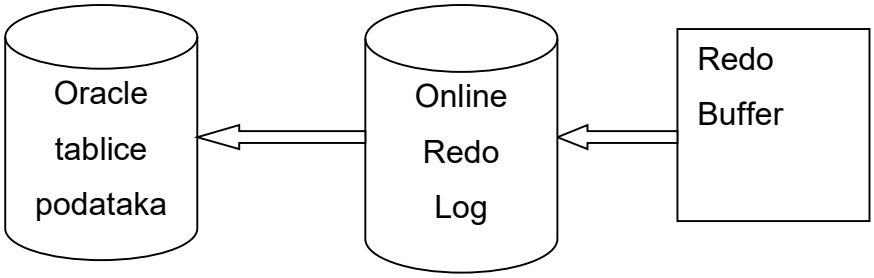
Slika 1. Klijentski procesi uvijek čitaju / mijenjaju podatke preko Block Buffer Cache-a

Zapravo, mijenjanje podataka ne ide tako da se podaci direktno iz Block Buffer Cache-a upisuju na disk, već se upis radi preko Redo Buffer-a, kako prikazuje slika 2.



Slika 2. Izmjena podataka radi se kroz Redo Buffer

Osim toga, mijenjanje podataka ne ide tako da se podaci odmah upisuju u tablice podataka, već se iz Redo Buffer-a prvo upisuju u Online Redo Log datoteku, kako prikazuje slika 3.

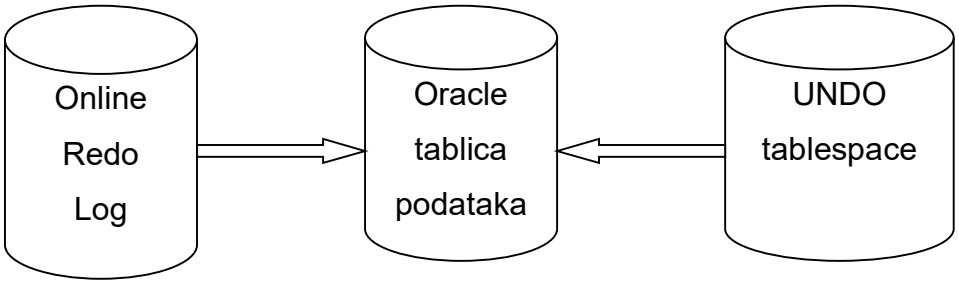


Slika 3. Podaci se na disk prvo zapisuju u Online Redo Log

Online Redo Log čine minimalno dvije datoteke (a preporučljivo je da postoje barem tri) koje se koriste u krug – kada se prva napuni, Oracle počinje pisati u drugu, a kada se druga napuni Oracle se vraća na prvu. Kada su podaci zapisani u Online Redo Log može doći do pada sustava (to nije kvar medija) prije nego Oracle te podatke upiše u prave tablice podataka. No nakon što se Oracle sustav ponovno podigne, pročitat će Online Redo Log i upisati podatke u prave tablice. Naravno, pitanje je da li ti podaci zaista trebaju biti trajno zapisani u tablicu podataka – to ovisi o tome da li su oni COMMIT-irani.

Moglo bi se pomisliti da Oracle sve podatke koji nisu COMMIT-irani drži u memoriji, u Block Buffer Cache-u, a da se tek kod COMMIT-a svi podaci prepisuju na disk. No to je nemoguće, jer Block Buffer Cache, koliko god bio velik, ne može uvijek biti dovoljno velik da svi mijenjani podaci stanu u njega. Stoga se podaci iz Block Buffer Cache-a često upisuju u tablice podataka (posredstvom Redo Buffer-a i Online Redo Log-a) prije nego je transakcija COMMIT-irana. Stoga se nakon pada sustava, i nakon što Oracle

pročita Online Redo Log i upiše podatke (koji još nisu upisani) u prave tablice, tj. nakon faze koju bismo mogli nazvati REDO fazom, zbiva UNDO faza, u kojoj se oni podaci koji nisu COMMIT-irani brišu iz Oracle tablica podataka (slika 4.). Takav postupak, da se prvo radi REDO, a onda UNDO faza, naziva se ARIES (Algorithms for Recovery and Isolation Exploiting Semantics) [6].



Slika 4. Nakon pada sustava, u Oracle tablice podataka prvo se dodaju podaci iz Online Redo Loga, a onda poništavaju necommitirani podaci iz Undo tablespacea

Može se postaviti pitanje – otkuda Oracle uzima stare podatke, koji su mu potrebni da napravi UNDO fazu? Moglo bi se pomisliti da se ti podaci nalaze u Online Redo Log datoteci, tj. da ona čuva sliku redaka kakvi su bili prije promjene (pa bi se ta slika koristila za UNDO) i sliku redaka nakon promjene (pa bi se ta slika koristila za REDO). U stvarnosti Online Redo Log, kako mu i samo ime kaže, sadrži samo sliku za REDO. Podaci potrebni za UNDO nalaze se u jednom posebnom prostoru na disku, koji se naziva UNDO tablespace (naravno, postoje i tablespaceovi za standardne tablice podataka). Oracle zapisuje staro stanje redaka u UNDO tablespace uvijek kada radi izmjenu podataka (INSERT / UPDATE / DELETE).

UNDO tablespace, osim što služi za eliminiranje (iz tablica podataka) promjena koje nisu COMMIT-irane (kod oporavka sustava nakon pada i nakon ROLLBACK-a), služi i za omogućavanje višekorisničkog rada. Naime, jedan od važnijih zahtjeva na RDBMS sustav je da jedna sesija (baze podataka) ne vidi promjene koje je napravila druga sesija, sve dok ta druga sesija ne napravi COMMIT. No budući da se često ne-COMMIT-irani podaci moraju spremati na disk, pitanje je otkuda prva sesija može čitati stare podatke (prije promjene). Oni nisu u tablici podataka (tamo su promijenjeni podaci), nisu niti u Online Redo Log datoteci – oni se nalaze u UNDO tablespaceu. Oracle ih iz UNDO tablespacea čita u Block Buffer Cache, jer klijentski proces sve podatke (pa i stare) čita iz Block Buffer Cache-a, a ne direktno sa diska.

Postojanje UNDO tablespacea (ili neke slične strukture u nekom drugom RDBMS sustavu) omogućava da mijenjanje podataka ne utječe na čitanje podataka, tj. mijenjanje podataka ne sprečava da se ti podaci istovremeno i čitaju (zapravo, čitaju se stara stanja tih podataka). U RDBMS sustavima koji nemaju nešto slično kao što je UNDO tablespace, mijenjanje podataka utječe na čitanje, tj. sesija koja mijenja podatke postavlja lokot nad tim mijenjanim redovima, a taj lokot sprečava druge sesije čak i da čitaju podatke, čime dolazi do znatnog usporavanja korisničkog rada.

Osim UNDO tablespacea i Online Redo Log datoteka, postoje i Archive Redo Log datoteke. One, za razliku od UNDO tablespacea i Online Redo Log datoteka nisu obavezne, tj. Oracle sustav može raditi i bez njih. No Archive Redo Log datoteke su, uz backup datoteke, praktični nužne za oporavak u slučaju kvara medija (diska). Archive Redo Log datoteke dobivaju se kopiranjem napunjenih Online Redo Log datoteka. Naravno, Archive Redo Log datoteke se ne koriste u krug, tako da za njihovo spremanje treba rezervirati dovoljno mjesta na nekom mediju. To može biti traka, ali danas je najpogodnije (zbog relativno niskih cijena diskova) da to budu diskovi - naravno, ne oni isti na koje se spremaju tablice podataka (jer ako se pokvare ti diskovi, izgubili smo oboje).

Napominjemo da je ovaj prikaz arhitekture baze podataka vrlo općenit, bez puno detalja. Preporučamo čitanje izvrsne knjige Toma Kytea [12].

2.2. Transakcije općenito

Kod baza podataka, važni su pojmovi konekcija, sesija i transakcija baze podataka. Moramo napomenuti da se ti pojmovi ponekad brkaju, vjerojatno zbog toga što se na aplikacijskoj strani (npr. kod aplikacijskih servera) često koriste isti pojmovi, ali ne znače uvijek isto što i kod DBMS-a. Npr. u web programiranju, korisnička (user) sesija (neki kažu i aplikacijska sesija) nije isto što i sesija baze. Na bazi, jednu sesiju čini jedna ili više slijednih transakcija, a transakcija uvijek pripada jednoj sesiji (izuzetak su distribuirane transakcije, gdje jednu transakciju realizira više sesije, koje pripadaju različitim bazama podataka). Konekcija na bazu je (pojednostavljeno rečeno) veza između klijentskog programa (to može biti i program na aplikacijskom serveru) i baze, a kroz jednu konekciju može u određenom trenutku ići nula, jedna ili više sesija (baze). Najčešće kroz jednu konekciju ide jedna sesija (baze), pa se često pojmovi konekcija

(na bazu) i sesija (baze) poistovjećuju. No npr. Oracle Forms koristi mogućnost da više sesija (baze) ide kroz samo jednu konekciju.

Možemo reći da su ovo neke najvažnije osobine Oracle transakcije (u ne-distribuiranom slučaju):

1. Sesija (baze) ne vidi promjene koje je napravila druga sesija, dok druga sesija ne napravi COMMIT (ili ROLLBACK). Međutim, sesije nisu nezavisne, jer zaključavanje redaka u jednoj sesiji utječe na drugu sesiju koja pokušava ažurirati redak koji je zaključala prva sesija.

2. Kada sesija izvršava DML naredbu (INSERT / UPDATE / DELETE), automatski se zaključa redak. Redak se može otključati tek na kraju transakcije (COMMIT ili ROLLBACK), ili pomoću ROLLBACK TO SAVEPOINT. Zaključavanje će biti detaljnije prikazano u sljedećem potpoglavlju.

3. Transakcija može ili u cijelosti uspjeti (COMMIT), ili se u cijelosti poništiti (ROLLBACK). Naravno, ne znači da sve DML radnje unutar transakcije moraju uspjeti, jer one DML radnje koje su uzrokovale grešku, ali je greška obrađena, neće uspjeti.

4. DML naredba može ili u cijelosti uspjeti ili se njen efekt u cijelosti poništava. Pritom se mogu desiti tri tipa grešaka:

- narušeno ograničenje na tip stupca, npr. ako u NUMBER (2) pokušamo upisati broj 1000;
- narušeno deklarativno integritetno ograničenje (PK, UK, FK, CK, NOT NULL); zapravo provjera deklarativnih ograničenja može se i odgoditi (najkasnije do COMMIT) - tada naredba može uspjeti i ako deklarativna ograničenja nisu zadovoljena;
- narušena proceduralna ograničenja (okidači baze); to je najkompliciraniji slučaj, jer npr. jedna UPDATE naredba može okinuti različite UPDATE okidače, koji mogu pozivati procedure koje dalje rade DML, čime se okidaju drugi okidači.

5. Ako se desila greška na bazi koja nije obrađena, a početak je DML naredba, svi efekti se poništavaju (prethodna točka). Ako se desila greška na bazi koja nije obrađena, a početak je poziv procedure (ili funkcije) sa strane klijenta (npr. Forms ili Java), ili poziv sa strane druge baze (udaljena procedura), svi efekti procedure se poništavaju.

Često se kaže (npr. [1]) da transakcija treba zadovoljavati ACID svojstva (ACID property):

- A označava atomarnost (Atomicity);
- C označava konzistentnost (Consistency);
- I označava izoliranost (Isolation);
- D označava trajnost (Durability).

Napomenimo da je konzistentnost transakcije usko povezana sa atomarnošću, ali je za konzistentnost odgovoran programer, a ne baza. Za transakciju se često kaže da je ona jedinica integriteta (a unit of integrity). Zanimljivo je naglasiti da Date (npr. u [6]) tvrdi kako bi zapravo i svaka naredba (statement) trebala biti jedinica integriteta (što današnji DBMS-i ne omogućavaju).

Izoliranost se često zove i serijabilnost (serializability). Misli se na to da bi transakcije trebale uvijek ostaviti efekt kao da se izvršavaju serijski (jedna za drugom), iako se izvršavaju konkurentno (paralelno ili kvazi-paralelno). Da bi se postigla serijabilnost, DBMS sustavi primjenjuju dvofazno zaključavanje (two-phase locking), koje ne treba miješati sa dvofaznim commit protokolom (koji se koristi kod commit-iranja distribuirane transakcije). U fazi širenja (growing phase), transakcija zaključava retke, a poslije ih samo otključava, u fazi stezanja (shrinking phase).

Skoro svi DBMS sustavi koriste specifičnu varijantu dvofaznog zaključavanja, striktno dvofazno zaključavanje (strict two-phase locking), kod kojeg se faza stezanja radi neposredno prije kraja transakcije, prije commit-iranja (ili rollback-iranja). Ta varijanta eliminira problem kaskadnog abortiranja (cascading abort).

2.3. Transakcije i zaključavanje

Kako je rečeno u prethodnom potpoglavlju, kada sesija izvršava DML naredbu (INSERT / UPDATE / DELETE), automatski se zaključa redak. Redak se može otključati tek na kraju transakcije (COMMIT ili ROLLBACK), ili pomoću ROLLBACK TO SAVEPOINT. No treba naglasiti da sesiji koja pokuša pristupiti zaključanim redovima prije nego druga sesija napravi ROLLBACK TO SAVEPOINT, ti redovi i daje ostaju zaključani (do COMMIT ili ROLLBACK).

Iako se zaključavanje često radi implicitno, pomoću DML naredbi (INSERT, UPDATE, DELETE), ponekad je potrebno koristiti eksplicitno zaključavanje pomoću

SELECT ... FOR UPDATE (rijetko se koristi LOCK TABLE) kako bi se sačuvao integritet podataka.

Sesija koja čeka na zaključane retke u pravilu čeka neograničeno, tj. dok joj druga sesija ne otključa retke. Postoje dva izuzetka:

1. Sesija može pokušati zaključati retke sa
SELECT ... FOR UPDATE;

i navesti da ne želi čekati
SELECT ... FOR UPDATE NOWAIT;

ili želi čekati određeni broj sekundi
SELECT ... FOR UPDATE WAIT timeout;

Ako je druga sesija zaključala redak, onda se prvoj sesiji javlja greška
ORA-00054: resource busy and acquire with NOWAIT specified;

2. Ako sesija čeka na otključavanje redaka sa udaljene baze, onda je čekanje definirano parametrom DISTRIBUTED_LOCK_TIMEOUT, koji ima standardnu vrijednost 60 sekundi. Ako druga sesija drži zaključan redak, nakon isteka tog vremena prvoj sesiji javit će se greška:
ORA-02049: timeout: distributed transaction waiting for lock;
Ovu smo činjenicu koristili za trik (koji smo prikazali na HrOUG 2003.) – kako izbjeći neograničeno čekanje otključavanja retka kod INSERT naredbe, gdje prethodno korištenje naredbe SELECT .. FOR UPDATE NOWAIT nema efekta (jer uneseni redak za druge sesije nije još vidljiv).

Osim zaključavanja redaka (jednog ili više), ponekad želimo zaključati cijelu tablicu. Zanimljivo je da zaključavanje tablice možemo izvesti u djeljivom ili ekskluzivnom načinu (modu).

Više sesija može zaključati tablicu u djeljivom načinu:
LOCK TABLE tablica IN SHARE MODE NOWAIT;

Samo jedna sesija može zaključati tablicu u ekskluzivnom načinu (ako ju neka druga sesija nije već zaključala u djeljivom ili ekskluzivnom načinu):

LOCK TABLE tablica IN EXCLUSIVE MODE NOWAIT;

Zaključavanje tablice u djeljivom i ekskluzivnom načinu može se iskoristiti npr. za omogućavanje da više sesija mijenja neku tablicu dokumenata (npr. tablicu narudžbi), a da samo jedna sesija može raditi obradu narudžbi. Pritom se može zaključavati izvorna tablica dokumenata (npr. narudžbi), ili neka pomoćna tablica, koja može imati malo redaka (može i jedan, pa i nijedan).

Kod zaključavanja (implicitnog ili eksplicitnog) može se desiti deadlock - ako dvije transakcije pokušaju zaključati dva retka, ali u suprotnom redoslijedu. Npr. u sljedećem primjeru transakcija T1 uspješno zaključa račun broj 1, transakcija T2 uspješno zaključa račun 2, a obje žele zaključati i suprotni račun, pa će se desiti deadlock:

T1: SELECT iznos ... FROM racuni WHERE broj = 1 FOR UPDATE; -- uspješno
T2: SELECT iznos ... FROM racuni WHERE broj = 2 FOR UPDATE; -- uspješno
T1: SELECT iznos ... FROM racuni WHERE broj = 2 FOR UPDATE; -- čeka
T2: SELECT iznos ... FROM racuni WHERE broj = 1 FOR UPDATE; -- čeka

Srećom, Oracle baza otkrit će da je došlo do deadlocka, te će jedna od dvije transakcije dobiti grešku ORA-00060: deadlock detected while waiting for resource.

Nakon što ta transakcija (koja je dobila grešku) napravi ROLLBACK, druga transakcija će nastaviti sa radom.

2.4. Distribuirane transakcije

Sustav distribuiranih baza podataka (ili jednostavnije - distribuirana baza podataka) je sustav od dvije ili više baza podataka koje bi aplikacijama (korisničkim programima) trebale izgledati kao jedna jedinstvena baza podataka. Date je u svojoj knjizi [6] postavio "temeljni princip za distribuirane baze podataka", a to je: "Za korisnika, distribuirani sustav (baza podataka) treba izgledati potpuno isto kao ne-distribuirani sustav". Na temelju tog principa, Date u knjizi daje 12 zahtjeva koje distribuirana baza mora zadovoljiti. Činjenica je da je 100%-tno zadovoljenje svih tih zahtjeva gotovo

nemoguće, ali ti zahtjevi služe kao ideal prema kojemu bi trebale težiti konkretne implementacije distribuiranih baza podataka.

Ne ulazeći detaljnije u definiranje svakog od ovih zahtjeva, može se reći da ih Oracle baza podataka zadovoljava u velikoj mjeri. U zadovoljavanju tih zahtjeva veliku ulogu igra nepostojanje "pravog" globalnog rječnika podataka u Oracle bazi, jer svaka Oracle baza ima svoj lokalni rječnik podataka. S druge strane, lokalna baza ipak mora sadržavati neke informacije o bazama sa kojima komunicira. Oracle baza (ali i druge baze) to radi tako da lokalni rječnik sadrži podatke o udaljenim objektima baze.

Za čuvanje informacija o udaljenim objektima, Oracle baza podataka koristi tzv. "database link". Može se reći da je database link objekt baze podataka koji definira jednosmjernu vezu baze podataka na drugu bazu podataka. Postoji više vrsta database linkova. Jedna je podjela na globalne (koji pripadaju bazi) i privatne database linkove (koji pripadaju određenoj shemi baze), a druga podjela dijeli database link-ove prema načinu na koji se korisnik prijavljuje na udaljenu bazu (connected user, fixed user ili current user link). Slijedi primjer definiranja privatnog database linka koji je po drugoj podjeli "fixed user link" (pretpostavimo da smo database link kreirali kao objekt sheme "shemaX" unutar baze "bazaA"):

```
CREATE DATABASE LINK neki_link
   CONNECT TO shemaY IDENTIFIED BY zaporka USING "bazaB";
```

Ponekad nužno moramo raditi sa distribuiranom bazom podataka i distribuiranim transakcijama. Distribuirana transakcija koristi tzv. dvofazni commit protokol, koji ima faze Prepare i Commit (u Oracle realizaciji on se, zapravo, sastoji od tri faze – treća faza je Forget).

Ovako možemo ukratko prikazati zbivanja kod uspješne distribuirane transakcije, koja završava sa COMMIT i kod koje nije bilo nikakvih problema:

1. Klijentska aplikacija šalje DML naredbe (ili/i pozive udaljenih procedura) po distribuiranoj bazi i na kraju završava sa COMMIT.

2. Ovdje počinje prva faza. Globalni koordinator (baza na koju je direktno vezana klijentska aplikacija) određuje koja će baza biti tzv. commit point site (u tome pomažu i ostale baze, lokalni koordinatori).

3. Globalni koordinator svim bazama, osim commit point site bazi, šalje naredbu da se pripreme.

4. Sve baze javljaju potvrdan odgovor (Prepared).

5. Ovdje počinje druga faza. Globalni koordinator javlja commit point site bazi da izvrši lokalni COMMIT.

6. Commit point site baza izvršava lokalni COMMIT i obavještava globalnog koordinatora.

7. Globalni i lokalni koordinatori javljaju svim ostalim bazama da naprave lokalni COMMIT.

8. Sve baze rade lokalni COMMIT i obavještavaju koordinatore.

9. Ovdje počinje treća faza. Globalni koordinator obavještava commit point site bazu da zaboravi distribuiranu transakciju.

10. Commit point site baza briše svoje podatke o distribuiranoj transakciji i obavještava globalnog koordinatora.

11. Globalni koordinator briše svoje podatke o distribuiranoj transakciji.

Nažalost, kod dvofaznog commit protokola postoji period u kojem je transakcija osjetljiva na pad (nekog) servera baze ili veze između baza.

Greške koje se mogu desiti jesu:

- palo je računalo na kojem radi Oracle baza;
- prekinula se veza između dvije ili više baza koje sudjeluju u distribuiranoj transakciji;
- desio se neki softverski problem.

U tom slučaju (tj. ako se desi greška u osjetljivoj fazi) transakcija postaje in-doubt. U fazi pripreme, baze stavljaju distribuirani lokot na sve modificirane tablice. Taj lokot sprečava čak i čitanje podataka! Ako to traje kratko, nije problem. No ako transakcija postane in-doubt, može se desiti da duže vrijeme drži zaključane podatke (čak i za čitanje). Najbolje je ostaviti da baza sama razriješi in-doubt transakciju. No ako je zadovoljen jedan od ova dva uvjeta, trebali bismo ručno razriješiti in-doubt transakciju:

1. in-doubt transakcija je zaključala kritične podatke ili undo segmente;

2. pad računala, prekid veze, ili softverski problem ne mogu se riješiti u kratkom vremenu.

3. Pristup Oracle bazi podataka preko JDBC-a

3.1. JDBC driveri

Java Database Connectivity (JDBC) je aplikacijsko programsko sučelje (API) za programski jezik Java, koje definira kako klijent može pristupiti i koristiti bazu podataka. JDBC je dio Java Standard Edition (JSE) platforme, koja je sada vlasništvo Oracle korporacije. Postoji i JDBC-to-ODBC bridge, koji omogućava da se iz Jave pristupa bilo kojem ODBC izvoru podataka.

JDBC se prvi put pojavio 1997. godine, kada je Sun Microsystem izbacio drugu verziju Jave, tj. Java Development Kit (JDK) 1.1. Od tada nadalje, JDBC je dio Java Standard Edition. Trenutačno je zadnja verzija JDBC 4.2, koja je uključena u Java SE 8.

JDBC je od početka podržavao tzv. result set (klasa ResultSet), pomoću kojeg u JDBC-u klijent pristupa podacima dobivenim iz baze. Kroz specifikaciju JSR 114, uveden je i tzv. row set, koji je nadograđen nad result setom, i koji pruža dodatne mogućnosti manipulacije podacima kroz JDBC.

Često se u aplikacijama ne koristi JDBC direktno, nego se koriste različiti object-relational mapping (ORM) alati / programski okviri, kao što su npr. Hibernate, TopLink (koji podržavaju JPA specifikaciju) i drugi. No, svi ti alati / programski okviri rade (i) nad JDBC nižim slojem. Ponekad, kad je aplikacija jednostavnija, ili se žele bolje performanse, ili specijalni načini pristupa, koristimo JDBC direktno.

Postoje različite podjele JDBC drivera. Uobičajena podjela (neovisna o proizvođaču) je ovakva:

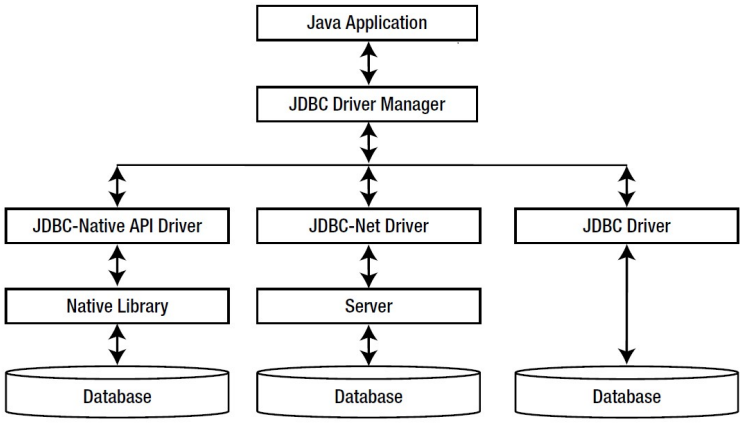
Type 1 driver: poziva nativni kod lokalnog ODBC drivera, koji dalje komunicira s bazom podataka;

Type 2 driver: poziva nativnu biblioteku (library) određenog proizvođača, koja se (biblioteka) isto nalazi na klijentskoj strani; kod u biblioteci dalje komunicira s bazom;

Type 3 driver: čisti-Java (pure-Java) driver koji komunicira s middleware serverom, koji dalje komunicira s bazom;

Type 4 driver: čisti-Java driver, direktno komunicira s bazom podataka koristeći nativni protokol baze.

Slika 5. prikazuje vrste JDBC drivera, pri čemu su gore navedeni tip 1 i tip 2 prikazani zajedno, jer se u oba slučaja koristi nativna biblioteka kao posrednik između JDBC drivera i baze podataka.



Slika 5. Vrste JDBC drivera; Izvor: [20]

3.2. Result set

U 1. primjeru Java programa vidi se uobičajen način rada pomoću result seta (klasa ResultSet). Prvo se otvara konekcija nad bazom podataka (ili se konekcija uzima iz connection poola, što je prikazano u 2. primjeru), nad tom konekcijom se kreira naredba, a onda se naredba izvršava. U ovom slučaju je to naredba SELECT, pa se rezultat upita vraća kroz result set.

Primijetimo da je u primjeru korištena nova mogućnost koja postoji od Jave 7, tzv. Automatic Resource Management, koji koristi try-with-resources, čime se izbjegava potreba da se u finally bloku eksplicitno zatvaraju (redom) result set, naredba i konekcija (obrnuto od otvaranja). Naravno, zatvaranje tih resursa dešava se i sada, ali to sada ne moramo eksplicitno navesti.

U primjeru se u petlji jednostavno čita svaki dobiveni redak, te se prikazuje na ekranu, bez ikakvog mijenjanja.

```
/* 1.Primjer: ResultSet sa SELECT naredbom */
import java.sql.*;
import oracle.jdbc.OracleDriver;
```

```
public class P1ResultSet {

public static void main (String[] args) throws SQLException {
  String url = "jdbc:oracle:thin:@localhost:1521:ORCL";
  String shema = "obuka";
  String lozinka = "obuka";
  String query = "select naziv from m_artikli order by naziv";
  try (Connection conn = DriverManager.getConnection(url, shema, lozinka);
       Statement stmt = conn.createStatement();
       ResultSet rs = stmt.executeQuery(query);
      ) // try-with-resources, postoji od Jave 7
  {
    while (rs.next())
      System.out.println(rs.getString(1));
  }
}

}
```

3.3. Konekcije, connection pool, proxy user

Konekcije baze podataka u današnjim web i mobilnim aplikacijama rijetko se kreiraju i zatvaraju kod svake upotrebe. Razlog je taj što kreiranje konekcije traje neko vrijeme, koje nije zanemarivo (reda je npr. stotinjak milisekundi, ili više). Zbog toga se, za razliku od 1. primjera, konekcije često koriste kroz connection pool. Nakon upotrebe, konekcije se stvarno ne zatvaraju, već se vraćaju u connection pool (zatvaraju se logički, a ne i fizički).

2. primjer Java programa pokazuje korištenje connection poola (u konkretnom slučaju je to Oracle Universal Connection Pool, koji je raspoloživ od Oracle baze 11g). U primjeru se kreira connection pool koji ima maksimalno 4 konekcije, a inicijalno se kreiraju 3 konekcije.

U primjeru se koristi i tzv. proxy user. Naime, problem koji se može pojaviti kod connection poola jeste taj da svaki korisnik baze podataka dobiva svoj connection pool. To obično nije problem, jer današnje web aplikacije najčešće rade kroz samo jednog korisnika baze podataka (iako to nije baš najbolje za sigurnost baze podataka). Ako u nekoj aplikaciji želimo zadržati (npr. zbog sigurnosti) odnos jedna osoba = jedan korisnik na bazi, tada svakako želimo izbjeći kreiranje connection poola za svakog korisnika. Uobičajeno se to radi kroz proxy usera. On nam služi kao "vlasnik" connection poola, a "pravi" korisnici baze ulaze preko proxy usera u bazu.

Ovako izgleda kreiranje proxy usera (imena npr. uproxy) i određenog konkretnog usera (imena npr. u1) na Oracle bazi:

```
CREATE USER uproxy IDENTIFIED BY uproxy_pwd;
GRANT CREATE SESSION TO uproxy;

CREATE USER u1 IDENTIFIED BY u1_pwd;
GRANT ... TO u1;
ALTER USER u1 GRANT CONNECT THROUGH uproxy;
```

Primijetimo da (barem u Oracle bazi) konekcija baze podataka i sesija baze podataka nisu istovrsni pojmovi! Jedna konekcija baze podataka može imati jednu ili više sesija baze podataka (zapravo, konekcija u određenom trenutku može imati i nula sesija). Istina, najčešće se kroz jednu konekciju koristi samo jedna sesija. No, baš u 2. primjeru to nije tako. Vidimo da se npr. na početku otvore tri konekcije i tri sesije za proxy usera, uproxy. Međutim, kada kroz takvu konekciju uđemo kao pravi korisnik u1 (kroz proxy usera), vidimo da broj konekcija ostaje isti, a broj sesija se povećava za jedan, tj. kroz jednu konekciju sada idu dvije sesije, za korisnike uproxy i u1. Istina, korisnička sesija uproxy je u ovom slučaju nepristupačna, ali u nekim drugim slučajevima zaista možemo imati dvije aktivne sesije kroz istu konekciju.

```
/* 2.Primjer: Universal Connection Pool i proxy user */
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;
import oracle.ucp.jdbc.ValidConnection;
import oracle.ucp.jdbc.JDBCConnectionPoolStatistics;
import oracle.jdbc.OracleConnection;
import java.util.Properties;

public class P2UCPProxy {

public static void main(String args[]) throws SQLException, Exception {
  try {
    //Create pool-enabled data source instance.
    PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();

    //set the connection properties on the data source.
    pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
    pds.setURL("jdbc:oracle:thin:@localhost:1521:ORCL");
    pds.setUser("uproxy");
    pds.setPassword("uproxy");

    //Override any pool properties.
    pds.setInitialPoolSize(3);
    pds.setMaxPoolSize(4);

    System.out.println("Prije otvaranja logicke konekcije\n");
    // printStatistics(pds); // NullPointerException
    Thread.sleep (5_000);
```

```
//Get a database connection from the datasource.
    Connection conn1 = pds.getConnection();

    System.out.println("Connection obtained from UniversalConnectionPool");
    printStatistics(pds);
    Thread.sleep (5_000);
    // vide se tri konekcija uproxy

    Properties prop1 = new Properties();
    prop1.put(OracleConnection.PROXY_USER_NAME, "u1");
    prop1.put(OracleConnection.PROXY_USER_PASSWORD, "u1");
    ((OracleConnection)conn1).openProxySession
            (OracleConnection.PROXYTYPE_USER_NAME, prop1);
    System.out.println("Pridružili smo 1.konekciju u1");
    printStatistics(pds);
    Thread.sleep(5_000);
    // vide se tri sesije uproxy i jedna u1, ali samo su tri konekcije

    Connection conn2 = pds.getConnection();
    ((OracleConnection)conn2).openProxySession
            (OracleConnection.PROXYTYPE_USER_NAME, prop1);
    System.out.println("Otvorili smo 2.konekciju, za u1");
    printStatistics(pds);
    Thread.sleep(5_000);
    // vide se tri sesije uproxy i dvije u1, ali samo su tri konekcije

    Connection conn3 = pds.getConnection();
    ((OracleConnection)conn3).openProxySession
            (OracleConnection.PROXYTYPE_USER_NAME, prop1);
    System.out.println("Otvorili smo 3.konekciju, za u1");
    printStatistics(pds);
    Thread.sleep(5_000);
    // vide se tri sesije uproxy i tri u1, ali samo su tri konekcije

    Connection conn4 = pds.getConnection();
    ((OracleConnection)conn4).openProxySession
            (OracleConnection.PROXYTYPE_USER_NAME, prop1);
    System.out.println("Otvorili smo 4.konekciju, za u1");
    printStatistics(pds);
    Thread.sleep(5_000);
    // vide se ČETIRI sesije uproxy i četiri u1, ali samo su četiri konekcije

    ((ValidConnection) conn4).setInvalid();
    conn4.close();
    conn4 = null;
    System.out.println("Zatvorili smo 4.konekciju");
    printStatistics(pds);
    Thread.sleep(5_000);
    // ostaju tri konekcije, sve kao uproxy

    ((ValidConnection) conn3).setInvalid();
    conn3.close();
    conn3 = null;
    System.out.println("Zatvorili smo 3.konekciju");
    printStatistics(pds);
    Thread.sleep(5_000);
    // ostaju dvije konekcije, sve kao uproxy
```

```
((ValidConnection) conn2).setInvalid();
    conn2.close();
    conn2 = null;
    System.out.println("Zatvorili smo 2.konekciju");
    printStatistics(pds);
    Thread.sleep(5_000);
    // ostaje jedna konekcija, kao uproxy

    ((ValidConnection) conn1).setInvalid();
    conn1.close();
    conn1 = null;
    System.out.println("Zatvorili smo 1.konekciju");
    printStatistics(pds);
    Thread.sleep(5_000);
    // nema više konekcija

  } catch (Exception e) {
    System.out.println(e.toString());
  }
}

public static void printStatistics(final PoolDataSource pds) {
  JDBCConnectionPoolStatistics statistics = pds.getStatistics();
  System.out.println
    ("Available Conn: " + statistics.getAvailableConnectionsCount());
  System.out.println
  ("Borrowed Conn: "  + statistics.getBorrowedConnectionsCount());
  System.out.println
    ("Closed Conn: "    + statistics.getConnectionsClosedCount());
  System.out.println("");
}
} // class P2UCPProxy
```

3.4. Updatable result set

Result set ne mora služiti samo za čitanje, već i za izmjene (updatable result set). Podrazumijevani (default) result set je onaj kojeg možemo čitati samo prema naprijed, tj. kod kojega se kurzor može pomicati samo prema naprijed (forward only), i koji se ne može mijenjati (read only). Postoje tri tipa result set kurzora:

1. TYPE_FORWARD_ONLY (default): kurzor se može pomicati samo naprijed;

2. TYPE_SCROLL_INSENSITIVE: može se micati naprijed ili nazad, ili na određenu poziciju; no, takav kurzor ne vidi promjene na bazi – otuda riječ INSENSITIVE;

3. TYPE_SCROLL_SENSITIVE: može se micati naprijed ili nazad, ili na određenu poziciju; dodatno, promjene na bazi (ali ne sve) reflektiraju se na result set, tj. kurzor ih vidi - otuda riječ SENSITIVE; rijetki su JDBC driveri koji implementiraju ovu mogućnost.

Dalje, što se tiče ažuriranja, postoje dva tipa result seta:

1. CONCUR_READ_ONLY (default);

2. CONCUR_UPDATABLE: result set može se mijenjati.

Kako je već rečeno, ne reflektiraju se sve promjene na bazi na result set, čak i ako je sensitive. Tablica 1. prikazuje koje se promjene (ne)reflektiraju na određeni tip result seta.:

Tablica 1. Vidljivost internih i eksternih DML naredbi kod tri vrste result seta; Izvor: [13]

Result Set Type	Visibility of	Internal Delete	Internal Update	Internal Insert	External Delete	External Update	External Insert
Forward-only		No	Yes	No	No	No	No
Scroll-insensitive		Yes	Yes	No	No	No	No
Scroll-sensitive		Yes	Yes	No	No	Yes	No

U 3. primjeru Java programa koristi se result set koji je TYPE_SCROLL_SENSITIVE i CONCUR_UPDATABLE. Primijetimo dvije važne stvari:

1. Jako je bitno da se na početku transakcije kaže conn.setAutoCommit(false); ako to ne kažemo, JDBC default ponašanje je da nakon svake DML naredbe radi automatski COMMIT;

2. Kad napravimo rs.updateRow(); izmjena retka (UPDATE; isto vrijedi za DELETE i INSERT) odmah se šalje na bazu, što kao posljedicu ima i to da je za drugu sesiju baze podataka taj redak odmah zaključan, i ostaje zaključan do kraja transakcije, tj. do COMMIT ili ROLLBACK; dakle, updatable result set nema mogućnost optimističkog zaključavanja, što je standard za web aplikacije.

```
/* 3.Primjer: ResultSet sa UPDATE */
import java.sql.*;
import oracle.jdbc.OracleDriver;

public class P3UpdateResultSet {

public static void main (String[] args) throws SQLException {
  String url = "jdbc:oracle:thin:@localhost:1521:ORCL";
  String shema = "obuka";
  String lozinka = "obuka";
  String query = "select sifra, naziv from m_artikli order by naziv";

  try (Connection conn = DriverManager.getConnection(url, shema, lozinka);
       Statement stmt =
         conn.createStatement
           (ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
       ResultSet rs = stmt.executeQuery(query);
      )
  {
    conn.setAutoCommit(false);
    while (rs.next()) {
      System.out.println("Prije UPDATE " + rs.getString("naziv"));
      rs.updateString("naziv", "XXX");
      rs.updateRow(); // nakon ovoga, "vanjski UPDATE čeka"
      System.out.println("Nakon UPDATE " + rs.getString("naziv"));
      Thread.sleep(5_000);
    }

    conn.rollback();
  } catch (Exception e) {
    System.out.println(e.toString());
  }
}
} // class P3UpdateResultSet
```

3.5. Row set

JDBC row set nije nova stvar. Zapravo, pojavio se već u JDBC 2.0 (kada su se pojavili npr. i scrollable result set i updatable result set), ali kao opcionalni paket. Implementacija za row set standardizirana je kroz JDBC RowSet Implementations Specification (u JSR-114), kao ne-opcionalni paket, od Java SE 5.0 dalje. Row set se naslanja na result set, tj. RowSet sučelje (interface) nasljeđuje ResultSet sučelje. Neke prednosti row seta u odnosu na result set su sljedeće:

1. programiranje sa row setom je jednostavnije; kada se koristi result set, mora se eksplicitno raditi s konekcijama (Connection) i naredbama (Statement), koji su kod row seta skriveni od programera;

2. row set je serijabilan (Serializable; result set nije), pa može biti poslan preko mreže, ili snimljen na disk za kasniju upotrebu;

3. result set mora uvijek biti spojen na izvor podataka, dok row set može biti i odspojen; row set se može spojiti na bazu podataka npr. samo onda kada čita ili piše podatke u bazu;

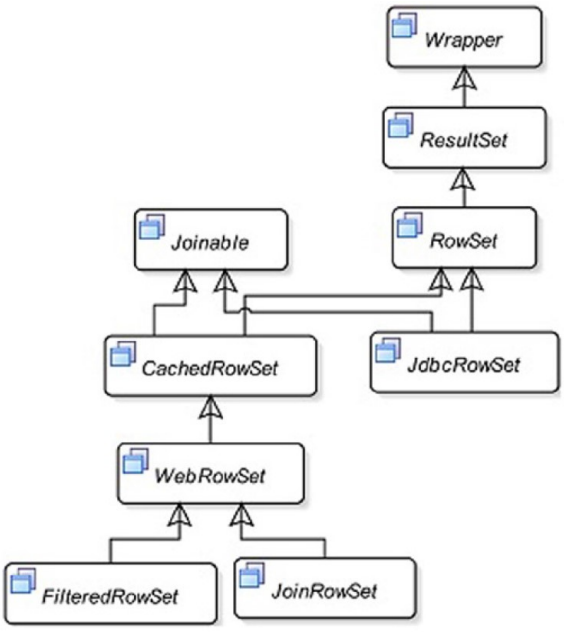
4. row set je default scrollable i updatable;

5. result set koristi bazu podataka kao izvor podataka; row set može koristiti i druge izvore podataka koji vraćaju podatke u tabularnom obliku;

6. serijabilnost row seta i mogućnost da radi i kada nije spojen na izvor podataka, čine row set vrlo korisnim za web ili mobilne aplikacije; klijent ne mora imati niti JDBC driver, jer može pokupiti podatke (ili ih ažurirati) sa srednjeg sloja, koristeći odgovarajući row set.

Row set može imati i nedostataka. Budući da se kod nekih row set klasa podaci keširaju u memoriji klijenta, može se javiti veliko zauzeće memorije (ako se na klijent učita velika količina podataka). Dalje, javlja se veća vjerojatnost nekonzistencije između podataka na klijentu i podataka na izvoru.

Slika 6. prikazuje sučelja koja definiraju row set.



Slika 6. Hijerarhija sučelja (interface) vezanih za row set; Izvor: [20]

Osnovna je podjela na CachedRowSet i JdbcRowSet. Tablica 2. prikazuje glavne razlike između njih (može se primijetiti da JdbcRowSet ima većinu osobina kao i ResultSet).

Tablica 2. Usporedba JDBC i cached row seta; Izvor: [17]

RowSet Type	Serializable	Connected to Database	Movable Across JVMs	Synchronization of data to database	Presence of JDBC Drivers
JDBC	Yes	Yes	No	No	Yes
Cached	Yes	No	Yes	Yes	No

3.6. Cached row set

Kako je prikazano na slici 3.2., CachedRowSet ima i podklase (zapravo, sučelja):

1. WebRowSet: omogućava čitanje i pisanje podataka / metapodataka kao XML dokumenta;
2. FilteredRowSet: omogućava filtriranje podataka na klijentskoj strani;

JoinRowSet: omogućava spajanje (join) dva ili više row seta u jedan row set, na klijentskoj strani.

Sljedeći Java program (4.primjer) pokazuje korištenje sučelja CachedRowSet (tj. klase OracleCachedRowSet). Nakon čitanja podataka sa baze (nakon crs.execute();), klijent se automatski odspaja. Dok je klijent odspojen, nad lokalnim podacima radi se UPDATE (crs.updateRow();), DELETE (crs.deleteRow();) i INSERT (crs.insertRow();). Nakon što se promjene pošalju na bazu (crs.acceptChanges(); - pritom se klijent automatski konektira na bazu), stvarno se ažurira baza podataka.

Kako je označeno u programskom kodu (kao komentar), može se desiti sljedeće:

1. ako netko u međuvremenu na bazi mijenja ili izbriše (i napravi COMMIT) redak koji se lokalno mijenjao, lokalna izmjena neće imati efekta, ali se na klijentu neće javiti greška;

2. ako netko u međuvremenu na bazi izbriše (i napravi COMMIT) redak koji se lokalno brisao, na klijentu se neće javiti greška;

3. ako netko u međuvremenu na bazi unese (i napravi COMMIT) redak koji se lokalno unio, na bazi će se možda javiti greška - ako je zbog toga npr. narušen unique key.

```
/* 4.Primjer: Cached Row Set sa UPDATE */
import java.sql.*;
import oracle.jdbc.rowset.*;

public class P4CachedRowSetTest {

public static void main (String[] args) throws SQLException {
  String url = "jdbc:oracle:thin:@localhost:1521:ORCL";
  String shema = "obuka";
  String lozinka = "obuka";
  String query = "select sifra, naziv from m_artikli order by naziv";

  try {
    OracleCachedRowSet crs = new OracleCachedRowSet();
    crs.setUrl(url);
    crs.setUsername(shema);
    crs.setPassword(lozinka);
    crs.setCommand(query);
    crs.execute();
    crs.next(); // idemo na 1.redak

    System.out.println("Prije UPDATE " + crs.getString("naziv"));
    Thread.sleep(5_000);

    crs.updateString("naziv", "XXX");
    crs.updateRow();
    System.out.println("Nakon UPDATE " + crs.getString("naziv"));
    Thread.sleep(5_000);
    // ako se izvana promijeni redak i da commit, izmjena se neće napraviti

    crs.absolute(3);
    System.out.println("Prije DELETE " + crs.getString("naziv"));
    Thread.sleep(5_000);

    crs.deleteRow(); // brišemo 3. redak
    System.out.println("Nakon DELETE ");
    Thread.sleep(5_000);
    // ako se izvana izbriše redak i da commit, ne desi se greška

    crs.moveToInsertRow();
    crs.updateString("sifra", "9");
    crs.updateString("naziv", "A9");
    crs.insertRow();
    crs.moveToCurrentRow();
    System.out.println("Nakon INSERT i prije accept");
    Thread.sleep(5_000);

    // ako se izvana unese redak sa istim PK / UK i da commit, desi se greška

    crs.acceptChanges(); // odmah je commit
    System.out.println("Nakon accept");
    Thread.sleep(5_000);

  } catch (Exception e) {
    System.out.println(e.toString());
  }
}

}
```

Kako je već rečeno, kod cached row seta mogu se javiti konflikti između lokalnih ažuriranja i ažuriranja koje je netko drugi u međuvremenu napravio u bazi podataka (ili drugom izvoru podataka). Kada se detektira konflikt kod izvršavanja metode acceptChanges(), dolazi do SyncProviderException iznimke. Tada se može koristiti synchronization resolver objekt (instanca od SyncResolver) za rješavanje konflikta, kako prikazuje sljedeći isječak koda (iz [20]):

```
catch (SyncProviderException spe) {
  // When acceptChanges() detects some conflicts
  SyncResolver resolver = spe.getSyncResolver();

  // Print the details about the conflicts
  printConflicts(resolver, cachedRs);
}
```

SyncResolver objekt omogućava navigaciju kroz sve konflikte, te omogućava da izmijenimo retke u row setu u skladu s time, kao što prikazuje sljedeći isječak koda (iz [20]):

```
public static void printConflicts
  (SyncResolver resolver, CachedRowSet cachedRs)
{
  try {
    while (resolver.nextConflict()) {
      int status = resolver.getStatus();
      String operation = "None";
      if (status == INSERT_ROW_CONFLICT)
        operation = "insert";
      else if (status == UPDATE_ROW_CONFLICT)
        operation = "update";
      else if (status == DELETE_ROW_CONFLICT)
        operation = "delete";

      // Get person_id from the database
      Object oldPersonId = resolver.getConflictValue("person_id");
      // Get person ID from the cached rowset
      int row = resolver.getRow();
      cachedRs.absolute(row);
      Object newPersonId = cachedRs.getObject("person_id");
      // Use setResolvedValue() method to set resolved value for a column
      // resolver.setResolvedValue(columnName,resolvedValue);
      System.out.println("Conflict detected in row #"
        + row + " during " + operation + " operation."
        + " person_id in database is " + oldPersonId
        + " and person_id in rowset is " + newPersonId);
    }
  } catch (SQLException e) {
    e.printStackTrace();
  }

}
```

4. Utjecaj razvoja mikroprocesora na programiranje

4.1. Mooreov zakon

Iako i dalje vrijedi Mooreov zakon (naravno, to nije zakon niti u matematičkom, niti u fizikalnom smislu, to je statistička prognoza), koji tvrdi da se broj tranzistora na mikroprocesorskom čipu udvostručuje otprilike svake dvije godine, danas se kaže: "vrijeme jednostavnog povećanja brzine programa je prošlo". Radni takt procesora praktički je prestao rasti oko 2005. godine. Razlog za to je prije svega veliko povećanje potrošnje struje procesora na velikim brzinama. Zbog toga su se proizvođači okrenuli drugačijem načinu povećanja performansi procesora. Umjesto povećanja brzine, povećali su broj CPU-a na jednom mikroprocesorskom čipu, tj. počeli su proizvoditi višejezgrene procesore.

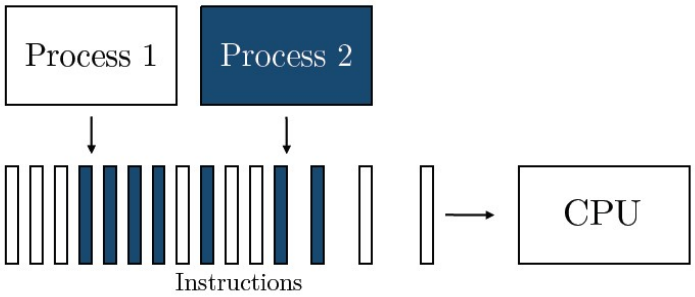
No, dok smo kod jednojezgrenih procesora povećanjem takta procesora dobili linearno povećanje brzine programa, kod višejezgrenih procesora program najčešće moramo pisati drugačije da bismo iskoristili raspoložive jezgre, tj. moramo preći na konkurentno programiranje. Nažalost, konkurentne programe nije lako pisati (barem ne imperativne, za razliku od deklarativnih). Međutim, (neki) današnji procesori mogu znatno pomoći u pisanju konkurentnih programa, naročito oni koji podržavaju tzv. hardversku transakcijsku memoriju (HTM).

4.2. Konkurentno programiranje i operacijski sustavi

Ideja o konkurentnom računanju (concurrent computation) razvila se baš na području operacijskih sustava. Prvi operacijski sustavi, napravljeni 50-tih godina prošlog stoljeća, bili su sekvencijalni, tj. računalni programi izvršavali su se na njima slijedno, jedan iza drugoga. Vrlo brzo se uvidjelo da je to vrlo neracionalan način korištenja računala (u to vrijeme vrlo rijetkih i skupih), pa su 60-tih godina napravljeni brojni operacijski sustavi koji su omogućavali konkurentno korištenje računalnih resursa.

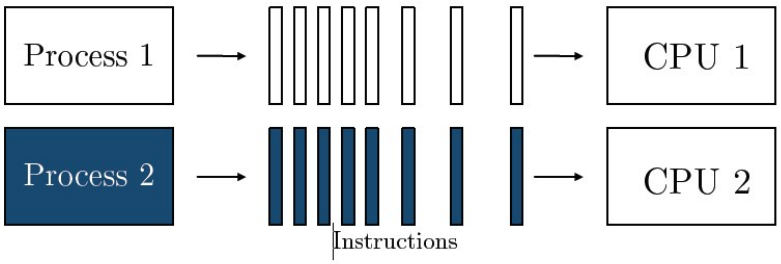
Tadašnja velika računala, kao i donedavno uobičajena osobna računala, imala su samo jedan CPU. Zbog toga se programi na njima nisu zaista izvršavali paralelno (ako zanemarimo specijalne programe, koji su se zaista izvršavali paralelno sa glavnim programima, npr. na specijaliziranim procesorima ulazno-izlaznih jedinica), već kvazi-paralelno. Uobičajeno se takav kvazi-paralelan način rada naziva *višezadaćnost* (*multitasking)*. *Procesi,* kako se nazivaju programi u izvršavanju (može se reći i da su

procesi instance programa u izvršavanju) u višezadaćnom radu izvršavaju se slijedno na istom CPU, ali se zbog isprepletenog izvršavanja djelića različitih procesa (tj. niza instrukcija), čini kao da se procesi izvršavaju paralelno. Na slici 7. vidi se prikaz višezadaćnosti.



Slika 7. Višezadaćnost (multitasking): instrukcije se međusobno isprepliću; Izvor: [15]

Vrlo brzo su se pojavila velika računala sa dva ili više CPU-a, a danas je uobičajeno da i najjeftinija prijenosna računala imaju dva CPU-a, u obliku dvije jezgre smještene u isti mikroprocesor (bolje rečeno mikroprocesorski čip). Ovakva računala omogućavaju da se dva procesa (ili više, ovisno o broju CPU-a) zaista izvršavaju paralelno, što se uobičajeno naziva multiprocesorski rad ili multiprocesiranje (multiprocessing), kako prikazuje slika 8.



Slika 8. Multiprocesiranje (multiprocessing): instrukcije se izvršavaju paralelno; Izvor: [15]

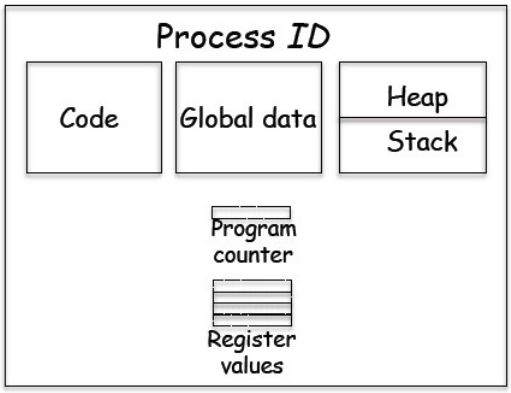
I multiprocesiranje i višezadaćnost su primjeri konkurentnog izvršavanja programa. Postoji i *distribuirano izvršavanje*, koje je po svom ponašanju dosta slično konkurentnom izvršavanju. Razlika je u tome što se konkurentno izvršavanje zbiva na jednom računalu (koje može imati više CPU-a, pa i stotine i tisuće), a distribuirano izvršavanje zbiva se na dva ili više računala koja su spojena mrežom. No, postoje i definicije po kojima se distribuirano izvršavanje odvija kroz dva ili više procesa (a ne

dretvi) operacijskog sustava, bez obzira da li se procesi nalaze na različitim računalima, pa bi po tome distribuirano izvršavanje bilo jedna vrsta konkurentnog izvršavanja.

Ne ulazeći u detalje zbivanja u operacijskim sustavima (detalje se može vidjeti npr. u [2]), može se reći da proces operacijskog sustava ima sljedeće najvažnije elemente [15]:

- identifikator procesa: jedinstveni ID procesa;
- stanje procesa: tekuća aktivnost procesa;
- kontekst procesa: vrijednost programskog brojača i vrijednost svih registara CPU-a;
- memorija: tekst programa, globalni podaci, stog (stack) i gomila (heap).

Slika 9. prikazuje najvažnije elemente procesa operacijskog sustava:

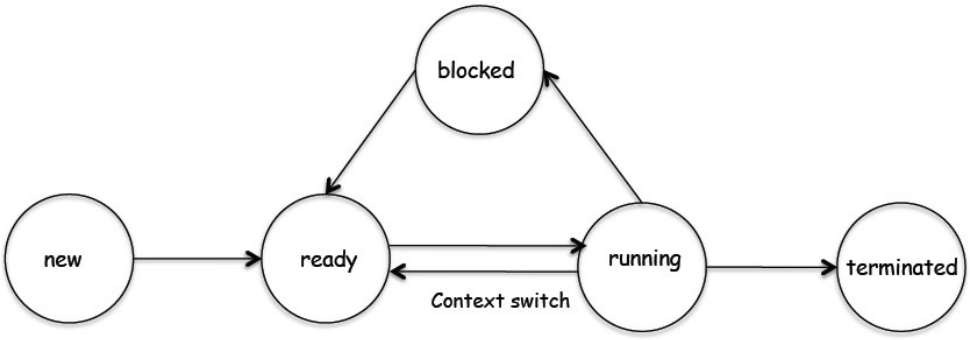


Slika 9. Najvažniji elementi procesa operacijskog sustava; Izvor: [14]

Operacijski sustav koristi poseban program - *raspoređivač* (*scheduler*), koji određuje kojem procesu treba dodijeliti (neki) procesor. Procesi mogu biti u ova tri generalna stanja [15]:

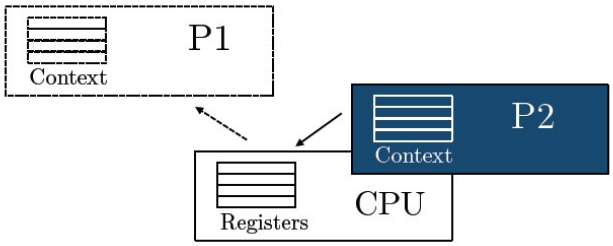
- pokrenut (running): instrukcije procesa se izvršavaju na (nekom) procesoru;
- spreman (ready): proces je spreman za izvršavanje, ali trenutačno mu nije dodijeljen procesor;
- blokiran (blocked): proces čeka da se desi neki događaj (npr. da se završi čitanje podataka sa diska ili iz memorije, koji su potrebni za nastavak rada procesa).

Slika 10. prikazuje tranziciju stanja procesa operacijskog sustava (dodana su i terminalna stanja *new* i *terminated*):



Slika 10. Tranzicija stanja procesa operacijskog sustava; Izvor: [14]

Zamjena (swapping) procesa koji se izvršavaju na CPU-u naziva se *zamjena konteksta* (*context switch*). Pretpostavimo da je proces P1 u stanju *pokrenut* i treba biti zamijenjen procesom P2 koji mora biti u stanju *spreman*. Program raspoređivač postavlja stanje procesa P1 na *spreman* i snima njegov kontekst u memoriju, kako bi ga poslije mogao "probuditi" i omogućiti da nastavi sa istog mjesta na kojem je stao. Raspoređivač dalje koristi kontekst procesa P2 kako bi postavio vrijednosti registara CPU-a i postavlja proces P2 u stanje *pokrenut.* Ta zbivanja prikazuje slika 11



Slika 11. Zamjena konteksta (context switch): proces P1 se odstranjuje iz procesora i zamjenjuje ga proces P2; Izvor: [15]

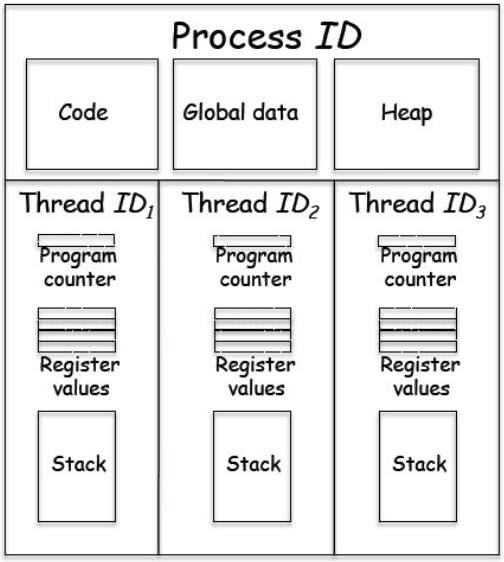
U prethodnom opisu proces P1 je iz stanja *pokrenut* prešao u stanje *spreman*, a proces P2 obrnuto. No, moguće je da proces pređe iz stanja *pokrenut* u stanje *blokiran*, jer čeka da se desi neki događaj (npr. da završi čitanje podataka sa diska ili iz memorije, ili da drugi proces napravi neki niz instrukcija). Blokirani proces ne može direktno preći u stanje *pokrenut*, nego prvo mora preći u stanje *spreman*, a onda ga raspoređivač može staviti u stanje *pokrenut*.

Kod paralelnog ili (kvazi-paralelnog) izvršavanja procesa možemo u načelu razlikovati dva slučaja. U prvom slučaju procesi su nezavisni jedan od drugoga, tj. ne komuniciraju međusobno. Inače, komunikacija između dva procesa radi se preko

razmjene poruka (rjeđe) ili preko čitanja/pisanja u zajednički dio memorije (češće). Zapravo, i nezavisni procesi mogu u nekom smislu biti zavisni, tj. ovise o zajedničkim resursima kao što su procesori (koji ih izvršavaju), ulazno-izlazne jedinice i sl. Programiranje nezavisnih konkurentnih procesa zapravo se ne razlikuje od programiranja sekvencijalnih procesa. U drugom slučaju procesi su zavisni, tj. međusobno komuniciraju. Tada je potrebno koristiti mehanizme sinkronizacije kako bi se procesi izvršavali na ispravan način - to je "pravo" konkurentno programiranje.

No, nisu samo procesi oni koji se mogu izvršavati paralelno (ili kvazi-paralelno). Ako je dobro da se procesi mogu izvršavati paralelno, onda to može biti dobro i za manje dijelove programa nego što su procesi, tj. dobro je da se paralelno (ili kvazi-paralelno) mogu izvršavati dijelovi istog programa! Dijelovi programa koji se mogu izvršavati paralelno zovu se *dretve* ili *niti* (*threads*). Može se reći: "dretve su lagani procesi". Procesi koji su sastavljeni od više dretvi zovu se *višedretveni* (*multithreaded*).

Dretve jednog procesa dijele adresni prostor tog procesa, tj. jedna dretva vidi podatke druge dretve. Zapravo, dretve dijele globalnu memoriju (programski kod i globalne podatke) i *gomilu* (*heap*), ali imaju vlastiti *stog* (*stack*) i kontekst dretve, kako prikazuje slika 12.



Slika 12. Najvažniji elementi dretvi operacijskog sustava (prikazane su tri dretve koje pripadaju istom procesu); Izvor: [14]

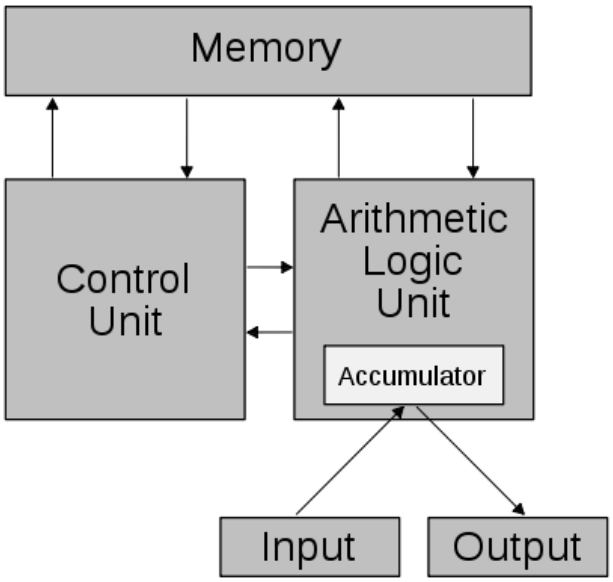
Zamjena konteksta dretvi u pravilu se izvodi nekoliko puta brže od zamjene konteksta procesa. U [9] se navodi da je zamjena konteksta dretve puno efikasnija od zamjene konteksta procesa, koja tipično traje stotine do tisuće procesorskih taktova. Zbog toga svi današnji moderni operacijski sustavi nisu samo višeprocesni, nego i

višedretveni. No i tu se očekuju poboljšanja, jer kako je navedeno u [4] "Postojeći raspoređivači nisu pripremljeni za stotine CPU-a i tisuće dretvi koje su u stanju *pokrenut* (*running*)."

Dretve se mogu dodjeljivati procesorima na isti način na koji se mogu dodjeljivati procesi. Npr. u računalnom sustavu sa četiri CPU-a, sustav može u određenom trenutku paralelno izvršavati četiri različita procesa, ali može izvršavati i četiri dretve istog procesa, ili bilo koju kombinaciju. Istina, zbog optimizacije sustav može odabrati da je bolje rasporediti određene dretve na određene procesore.

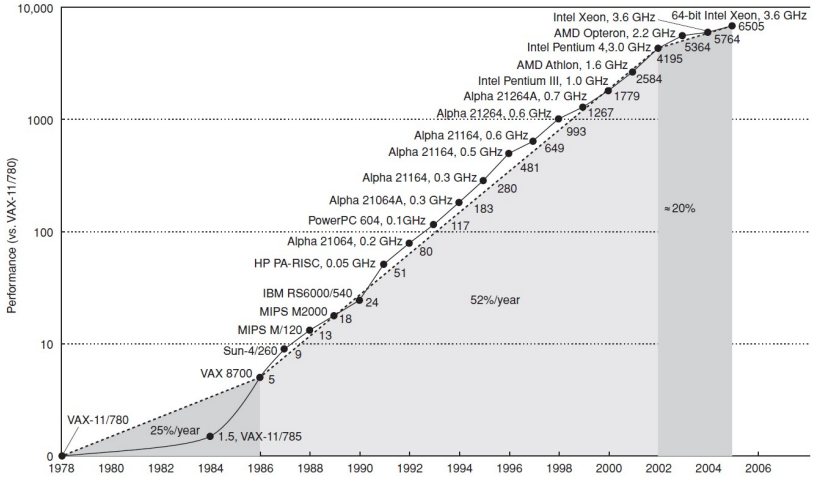
4.3. Razvoj mikroprocesora

Materijal [4], znakovitog naslova "Not Your Father's Von Neumann Machine: A Crash Course in Modern Hardware", navodi da je Von Neumannov model računala (prikazan na slici 13.) u današnje vrijeme samo korisna apstrakcija, koja u mnogim detaljima odudara od stvarnih današnjih računala.



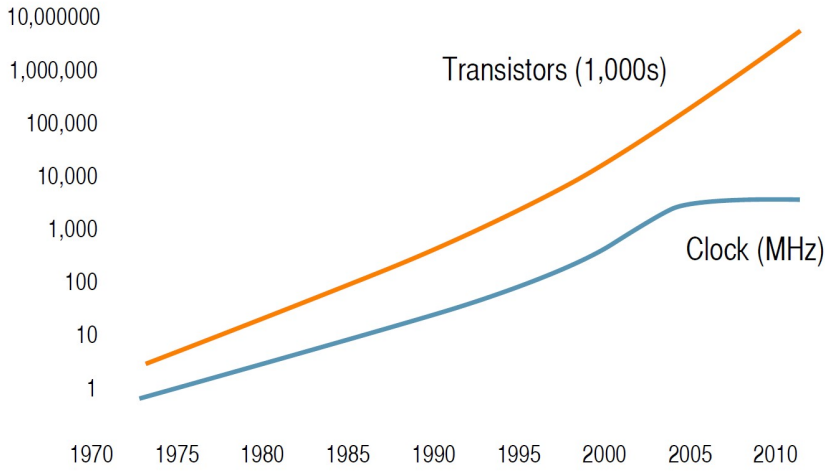
Slika 13. Von Neumannov model računala; Izvor: [4]

U knjizi [9], autori daju grafikon (slika 14.) koji prikazuje rast performansi procesora 1978.-2005. godine. Vidljiva su tri razdoblja: razdoblje CISC računala do 1986., u kojem je prosječno godišnje povećanje performansi 25%; razdoblje velikog povećanja radnog takta procesora, gdje je povećanje performansi 52% godišnje; razdoblje višejezgrenih procesora, gdje se radni taktovi procesora više nisu povećavali:



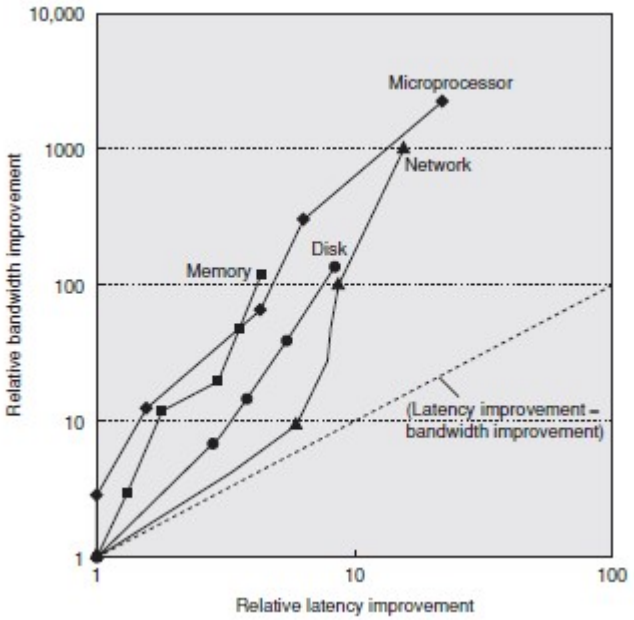
Slika 14. Rast performansi procesora od 1978. do 2005.; Izvor: [9]

No, to ne znači da je (već) prestao vrijediti *Mooreov zakon*, jer broj tranzistora po procesoru i dalje eksponencijalno raste (iako mnogi vjeruju da to neće biti dugo tako), kako prikazuje slika 15. No, radni takt procesora praktički je prestao rasti oko 2005. Razlog za to je prije svega veliko povećanje potrošnje struje procesora na velikim brzinama. Zbog toga su se proizvođači okrenuli drugačijem načinu povećanja performansi procesora. Umjesto povećanja brzine, povećali su broj CPU-a na jednom mikroprocesorskom čipu, tj. počeli su proizvoditi višejezgrene procesore. No, dok smo kod jednojezgrenih procesora povećanjem takta procesora dobili linearno povećanje brzine programa, kod višejezgrenih procesora program najčešće moramo pisati drugačije da bismo iskoristili raspoložive jezgre, tj. moramo preći na konkurentno programiranje.



Slika 15. Povećanje broja tranzistora i radnog takta procesora od 1973. do 2010.; Izvor: [18]

U [9] se daje zanimljiv dijagram (slika 16.) koji pokazuje kako se *širina pojasa* (*bandwidth*) ili *propusnost* glavnih elemenata računala (procesora, memorije, diska, mreže) povećavala daleko brže od brzine pristupa (latency).



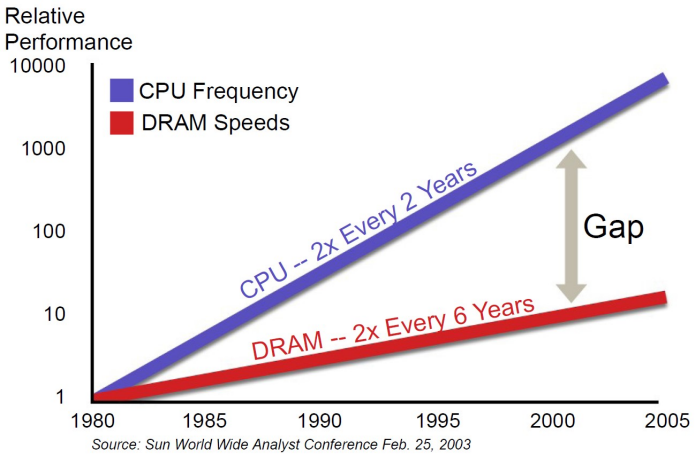
Slika 16. Log-log dijagram kretanja odnosa propusnosti i brzine pristupa kod glavnih elemenata računala; Izvor: [9]

Tablica 3. prikazuje razlike u veličini, brzini pristupa, propusnosti i nekim drugim karakteristikama različitih vrsta memorije, tj. procesorskih registara, priručne memorije (cache), glavne memorije i diskova (konkretni podaci vrijede za oko 2005. godinu). Vidi se da je glavna memorija 10 i više puta sporija od registara i priručne memorije:

Tablica 3. Razlike u veličini, brzini pristupa i drugim karakteristikama različitih vrsta memorije; Izvor: [9]

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 512 GB	> 1 TB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25–0.5	0.5–25	50–250	5,000,000
Bandwidth (MB/sec)	50,000–500,000	5000–20,000	2500–10,000	50–500
Managed by	compiler	hardware	operating system	operating system/ operator
Backed by	cache	main memory	disk	CD or tape

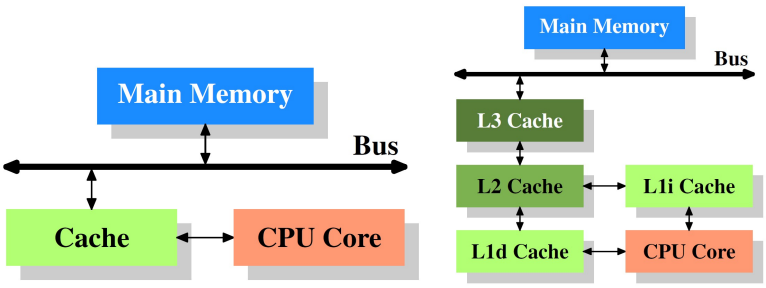
Slika 17. pokazuje kako se eksponencijalno povećava jaz između performansi CPU-a i performansi glavne memorije (konkretno, DRAM memorije):



Slika 17. Povećanje jaza između performansi CPU-a i performansi DRAM-a;

Naravno, moguće je napraviti glavnu memoriju koja bi bila brza skoro kao registri procesora. Problem je što bi takva memorija bila puno skuplja. Naime, glavna memorija standardno koristi tzv. DRAM (Dynamic RAM) memoriju, koja za svaki bit memorije treba jedan tranzistor i jedan kondenzator. Daleko brža SRAM (Static RAM) memorija za svaki bit memorije treba šest tranzistora. Ako treba birati između sustava sa većom količinom sporije glavne memorije i sustava sa manjom količinom brže glavne memorije, u pravilu je bolje izabrati prvo, jer je magnetski disk daleko sporiji od najsporije glavne memorije. No, još je bolje napraviti hijerarhiju memorija, tj. koristiti bržu i manju SRAM priručnu memoriju (ili više razina te memorije) zajedno sa većom i sporijom DRAM glavnom memorijom. Budući da programi često koriste programske instrukcije koje se nalaze blizu jedna drugoj, a to često vrijedi i za podatke, priručne memorije značajno ublažavaju sporost pristupa glavnoj memoriji.

Slika 18. prikazuje dva primjera korištenja priručne memorije, jedan jednostavniji i jedan složeniji (oba primjera prikazuju jednojezgreni procesor).

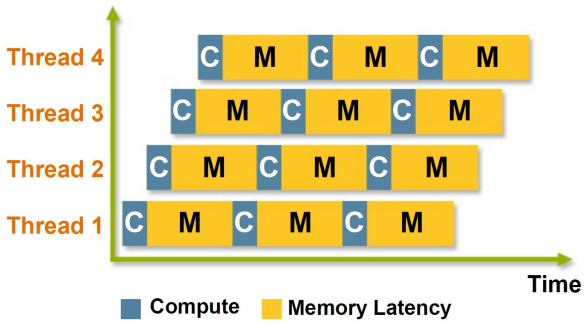


Slika 18. Dva primjera korištenja priručne memorije; Izvor: [5]

Osim uvođenja priručne memorije u mikroprocesorske čipove, dizajneri su od 1985. nalazili i druge tehnike za rješavanje jaza između brzine registara i brzine glavne memorije, kako bi u konačnici povećali brzinu izvođenja programa.

Jedan od načina koji je jako pridonosio povećanju performansi procesora sve dok je bilo moguće povećavati radni takt procesora (a to je postalo problematično zbog prevelikog zagrijavanja procesora) je tzv. *paralelizam na razini instrukcija*, *ILP* (Instruction-Level Parallelism). ILP obuhvaća više mehanizama, od kojih je najvažniji *pipelining* ("cjevovod instrukcija"), gdje se u procesor paralelno dovodi više instrukcija (istog programa) koje se paralelno obrađuju, ali tako da se za svaku instrukciju istovremeno radi različita faza obrade. Na taj način, ako se npr. jedna prosječna instrukcija obrađuje u pet faza kroz pet radnih taktova, paralelnom obradom pet instrukcija moguće je teoretski svesti prosječno vrijeme obrade instrukcije na samo jedan takt. Kod tzv. *superskalarne arhitekture* u jednom taktu može se obraditi i više instrukcija. Problem su, međutim, npr. instrukcije grananja, tj. sve instrukcije koje narušavaju sekvencijalni tok programa. Na rješavanje takvih problema kod ILP-a dizajneri procesora su trošili sve više i više tranzistora u procesoru.

Dosjetili su se da bi ILP mogli koristiti i na drugi način. Umjesto da paralelno obrađuju (u različitim fazama) instrukcije istog programa, mogli bi paralelno obrađivati instrukcije različitih programa, tj. instrukcije različitih dretvi (threads). Treba naglasiti da procesorske dretve (ili *hardverske dretve*) mogu odgovarati i dretvama i procesima operacijskog sustava. Takvo obrađivanje instrukcija naziva se *multithreading*.Postoji *temporal multithreading*, kod kojeg se u jednom taktu obrađuje maksimalno jedna instrukcija samo jedne dretve, i *simultaneous multithreading* (Intel koristi ime *Hyper-Threading Technology*), kod kojeg se u jednom taktu mogu istovremeno obrađivati instrukcije više dretvi. Slika 19. prikazuje *temporal multithreading*, i to tzv. *fine-grained temporal multithreading* varijantu, kod koje se u svakom taktu može izmjenjivati instrukcija druge dretve (za razliku od *coarse-grained* varijante).

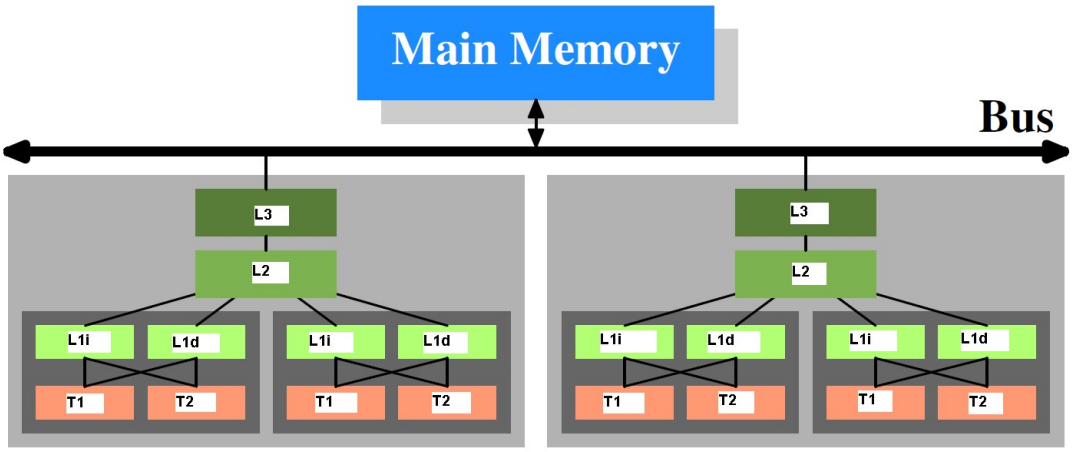


Slika 19. Multithreading (fine-grained temporal multithreading); Izvor: [4]

Multithreading (kao niti ILP) ne može činiti čuda. Naime, za razliku od višejezgrenih procesora, kod multithreadinga su duplirani samo neki elementi CPU-a. Npr. kod Intelove implementacije duplirani su samo procesorski registri. Dretve dijele i istu priručnu memoriju. Zbog toga procesor koji podržava dvije procesorske dretve nikako ne može imati 2 puta bolje performanse od istovrsnog procesora koji podržava samo jednu dretvu, već u praksi maksimalno do oko 1,3 puta (a i to vrlo rijetko). No, multithreading je ipak koristan, a koristi se i kod višejezgrenih procesora.

Kada je postalo jasno da se više ne može povećavati radni takt procesora (zbog eksponencijalnog povećanja potrošnje, pa onda i zagrijavanja procesora), a broj tranzistora po procesoru se i dalje eksponencijalno povećava, bilo je razumno početi smještati dva ili više CPU-a (jezgri) u jedan procesorski čip.

U računalo se onda može ugrađivati i nekoliko višejezgrenih procesora, pa se dobije arhitektura npr. kao na slici 20. Na njoj je prikazan sustav sa dva dvojezgrena procesora, a svaka jezgra ima dvije procesorske dretve (T1 i T2). Dretve jedne jezgre dijele priručne memorije L1i i L1d. Jezgre dijele priručne memorije L2 i L3. Dva procesora nemaju zajedničku priručnu memoriju, te pristupaju glavnoj memoriji na uobičajen način – to je tzv. *centralna djeljiva memorija*. Takva se arhitektura glavne memorije ponekad naziva *UMA* (*Uniform Memory Access*), ali se češće naziva *SMP* (*Symmetric Multi-Processors*) arhitektura, iako se naziv SMP ponekad koristi za UMA i NUMA arhitekturu zajedno.



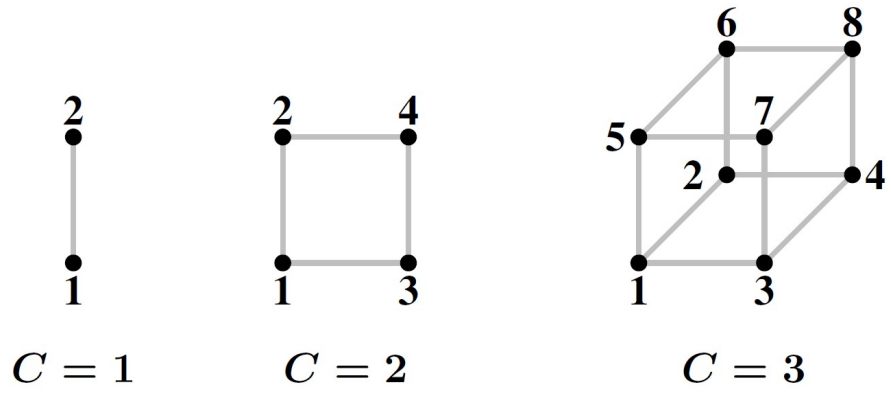
Slika 20. Sustav sa dva dvojezgrena procesora (svaka jezgra podržava dvije dretve); Izvor: [5]

Osim arhitekture centralne djeljive memorije, postoji i arhitektura *distribuirane djeljive memorije*. Iako je i takva memorija djeljiva između svih procesora (pa neki kažu da je i to simetrična arhitektura, SMP), određeni dijelovi memorije su bliži određenim procesorima.

Takva se arhitektura uobičajeno naziva NUMA (Non-Uniform Memory Access). Procesor ima najbrži pristup svom "lokalnom" dijelu glavne memorije, a brzina pristupa udaljenim dijelovima memorije ovisi o udaljenosti procesora od onoga procesora kojemu "pripada" taj dio memorije.

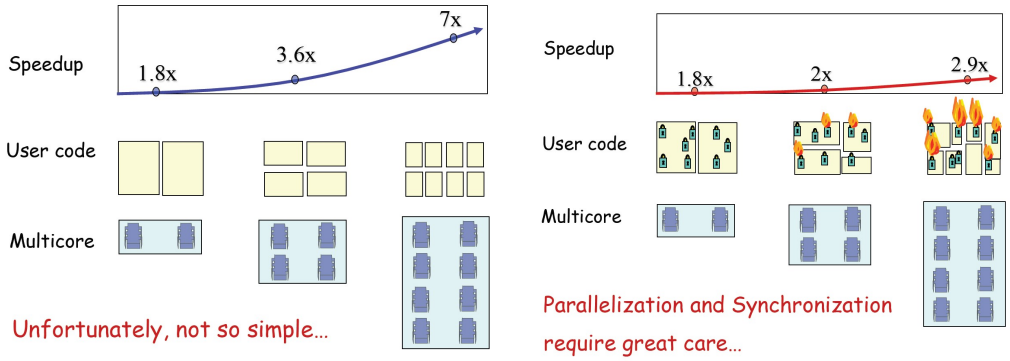
U NUMA arhitekturi procesori nisu međusobno povezani (samo) preko memorijske sabirnice, nego su direktno povezani sa određenim brojem drugih procesora (tzv. Hyper Link; AMD koristi varijantu HyperTransport, tehnologiju licenciranu od nekadašnje firme Digital). Najčešće su povezani u obliku "hiperkocke", kao što prikazuje slika 21.

Npr., kod desne hiperkocke (sa 8 procesora, C = 3), procesor 1 najbrže pristupa vlastitoj memoriji, a nakon toga memorijama procesora 2, 3 i 5 (sa kojima je direktno povezan), a najudaljeniji mu je procesor 8, pa je pristup njegovoj memoriji najsporiji.



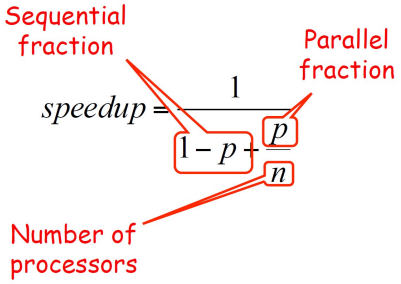
Slika 21. Povezivanje procesora (i pripadajućih dijelova glavne memorije) u hiperkocke; Izvor: [5]

Nažalost, kod višejezgrenih procesora (i višeprocesorskih sustava općenito) rast performansi sustava rijetko raste proporcionalno sa povećanjem broja jezgri, već je rast općenito manji. Za razliku od toga, povećanje radnog takta kod jednojezgrenih i jednoprocesorskih sustava u pravilu je povećavalo performanse skoro linearno. Zapravo, u nekim slučajevima se proporcionalno povećanje performansi može postići i kod višejezgrenih / višeprocesorskih sustava, npr. onda kada takvi sustavi služe za podršku baza podataka i aplikacijskih servera, gdje su pojedini procesi (ili dretve) međusobno nezavisni. No, kada pokušamo paralelizirati (razbiti u dretve) jedan program, najčešće dobijemo rezultat prikazan na desnoj strani slike 22., a ne onaj prikazan na lijevoj strani.



Slika 22. Povećanje broja jezgri obično ne rezultira (skoro) linearnim povećanjem performansi (lijeva strana slike), već puno manjim od toga (desna strana slike); Izvor: [14]

Razlog za to objašnjava *Amdahlov zakon* [14]. Ako je u nekom programu proporcija onih dijelova programa koji se mogu paralelno izvršavati jednaka *p* (što znači da je proporcija dijelova koji se ne mogu paralelno izvršavati jednaka *1 – p*), onda se povećanjem broja CPU-a može dobiti ovo povećanje:



Npr. ako imamo 10 procesora i ako je moguće paralelizirati 90% programa, onda je maksimalno povećanje brzine 5,26 puta, što je skoro dva puta manje od broja procesora. Ako je moguće paralelizirati 99% programa, onda je povećanje 9,17 puta.

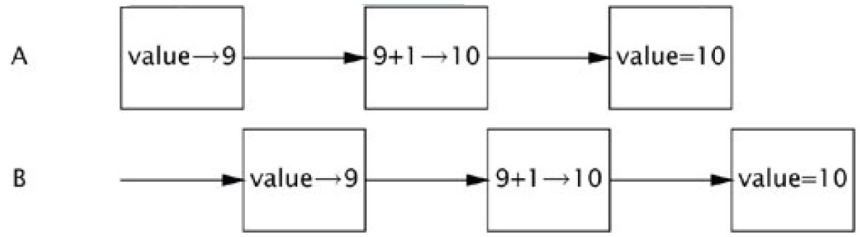
4.4. Sinkronizacijski algoritmi i mehanizmi

Pretpostavimo da smo napisali sljedeći Java programski kod [8] za računanje sekvence cijelih brojeva:

```
@NotThreadSafe
public class UnsafeSequence {
  private int value;
  /** Returns a unique value. */
  public int getNext() {
    return value++;
  }
}
```

Prethodni kod nije dobar u konkurentnom radu. Npr., ako dvije dretve A i B izvode metodu *getNext()*, moguć je sljedeći tok izvođenja (slika 23.), koji će dati pogrešan rezultat – obje dretve dobile su istu sekvencu 10, umjesto 10 i 11. Problem je u tome što (Java) naredba *value++* kod izvođenja nije *atomarna*, već se razlaže na (uobičajeno) tri interne naredbe:

temp = value; temp = temp + 1; value = temp.



Slika 23. Poziv metode getNext() u dretvama A i B dao je (pogrešno) istu sekvencu; Izvor: [8]

Kao i kod sekvencijalnih programa, tako je i kod konkurentnih programa najvažnija osobina programa *korektnost*. Međutim, u konkurentnim programima se korektnost daleko teže postiže / provjerava. Prethodni program nije korektan, jer njegova točnost ovisi o međusobnom redoslijedu izvođenja naredbi dva (ili više) programa. Taj se problem uobičajeno naziva *race conditions*.

Da bismo riješili taj problem, dretve (ili procese, ali u nastavku ćemo navoditi samo dretve, iako može biti oboje) moramo sinkronizirati [8]. *Sinkronizacija* se zasniva na ideji da dretve komuniciraju jedna sa drugom kako bi se "dogovorile" o sekvenci akcija. U prethodnom primjeru trebale bi se "dogovoriti" da u isto vrijeme samo jedna drži resurs (tj. da ima ekskluzivan pristup do njega).

Dretve mogu komunicirati na dva načina:

- korištenjem djeljive memorije (shared memory): dretve komuniciraju čitajući i pišući u zajednički dio memorije; ova tehnika je dominanta i bit će korištena u nastavku;
- slanjem poruka (message-passing): dretve međusobno komuniciraju porukama.

Nažalost, potreba za ekskluzivnim držanjem resursa može dovesti do različitih problema. Najveći problem je *deadlock*, kada dvije dretve (ili više njih) ostaju blokirane zato što obje trebaju (i) resurs koji je ekskluzivno zauzela druga dretva. Osim deadlocka, problem je i *livelock*, kada dretva naizgled radi, ali se ne dešava ništa korisno. Problem je i kada se resursi ne dodjeljuju dretvama na pravedan (fer) način, a pogotovo kada neka dretva konstantno ostaje bez resursa - to je tzv. *gladovanje* (*starvation*).

Međusobno isključivanje (*mutual exclusion*) je oblik sinkronizacije koji onemogućava istovremeno korištenje djeljivog resursa. S tim je povezan pojam *kritične sekcije* (*critical section*), što je naziv za dio programa koji pristupa djeljivom resursu. Pseudokod koji prikazuje osnovnu logiku rada sa kritičnom sekcijom je sljedeći:

```
while true loop
  entry protocol
  critical section
  exit protocol
  non-critical section
end
```

Sinkronizacijski mehanizmi temeljeni na ideji protokola ulaz / izlaz (iz kritične sekcije) nazivaju se *lokoti* (*locks*). Lokoti se mogu realizirati na različite načine. Dobro su poznata [10] dva algoritma za implementaciju lokota - Petersonov algoritam za dvije dretve, te Lamportov Bakery (= pekara) algoritam za *n* dretvi:

```
class Peterson implements Lock {
  // thread-local index, 0 or 1
  private volatile boolean[] flag = new boolean[2];
  private volatile int victim;

  public void lock() {
    int i = ThreadID.get();
    int j = 1 - i;
    flag[i] = true; // I'm interested
    victim = i; // you go first
    while (flag[j] && victim == i) {}; // wait
  }

  public void unlock() {
    int i = ThreadID.get();
    flag[i] = false; // I'm not interested
  }
}
```

```
class Bakery implements Lock {
  boolean[] flag;
  Label[] label;

  public Bakery (int n) {
    flag = new boolean[n];
    label = new Label[n];
    for (int i = 0; i < n; i++) {
      flag[i] = false; label[i] = 0;
    }
  }

  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ...,label[n-1]) + 1;
    while ((?k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
  }

  public void unlock() {
    flag[ThreadID.get()] = false;
  }
}
```

U [10] je pokazano da je Petersonov algoritam *deadlock-free* i *starvation-free*. Lamportov algoritam je isto *deadlock-free,* ali zadovoljava i svojstvo *prvi-došao-prvi-obrađen* (*first-come-first-served*), što je jače nego *starvation-free*. Može se primijetiti (označeno crveno) da su oba algoritma temeljena na *radnom čekanju* (*busy waiting*), što se često naziva i *obrtanje* (*spinning*), a to je potencijalna slabost tih algoritama. Spinning je naročito neefikasan kod multitaskinga. Spinning ima smisla samo tamo [8] gdje je vrijeme čekanja tipično vrlo kratko, pa bi zamjena konteksta (context switch) bila skuplja od njega. *Spin locks* se često koriste kod jezgre (kernela) operacijskih sustava. Spinning može biti prihvatljiv i kod višeprocesorskih sustava, kada se radno čitanje odvija samo iz priručne memorije CPU-a i ne utječe na ostale CPU-e. To je tzv. *local spinning*.

No, nije samo radno čekanje (potencijalni) problem. Lamportov algoritam nije praktičan stoga što ima potrebu čitanja i pisanja u *n* različitih lokacija za *n* dretvi. Kako je pokazano u [10], ne postoji bolji algoritam koji se temelji na čitanju i pisanju, a da treba manje od *n* lokacija. Taj je rezultat krucijalno važan, jer pokazuje da je kod multiprocesorskih sustava potrebno uvesti sinkronizacijske mehanizme koji su jači od *čitaj-piši* (*read-write*) mehanizama, i koristiti ih za izradu algoritama za međusobno isključivanje dretvi.

Kako se navodi u [10], lokote bi trebalo implementirati sa sljedećim *atomarnim osnovnim operacijama* (*atomic primitives*) za sinkronizaciju, koje su moćnije od operacija atomarnog čitanja-pisanja (pseudokod je pisan u Eiffelu):

```
test-and-set (x, value)
do
  temp := x
  x := value
  result := temp
end

compare-and-swap (x, old, new)
do
  if x = old then
    x := new; result := true
  else
    result := false
  end
end
```

Treba napomenuti da su navedene operacije u praksi realizirane na razini procesora, tako da je navedeni pseudokod samo prikaz logike tih operacija. Operacija *test-and-set* (*TAS*) bila je osnovna operacija za sinkronizaciju u mikroprocesorima 1990-tih godina, dok praktički svaki mikroprocesor dizajniran poslije 2000. godine podržava jaču operaciju *compare-and-swap* (*CAS*), ili njoj ekvivalentnu. No, CAS (i *CASD*, *Compare-and-Swap-Double*) nisu novost – bile su dio IBM 370 arhitekture od 1970!

Jedan od najpoznatijih današnjih algoritama za implementaciju lokota je MCSLock. Njegove varijante osnova su za sinkronizaciju koju koriste današnji JVM-ovi (Java Virtual Machines).

U nastavku su prikazane metode *lock()* i *unlock()* MCSLock algoritma. Vidi se (označeno crveno) da obje metode koriste radno čekanje (ali to je *local spinning*), te da *lock()* koristi operaciju *getAndSet*, a *unlock()* operaciju *compareAndSet* (što je uobičajeni Java naziv za hardversku operaciju *compare-and-swap*):

```
public class MCSLock implements Lock {
  ...
  public void lock() {
    QNode qnode = myNode.get();
    QNode pred = tail.getAndSet(qnode);
    if (pred != null) {
      qnode.locked = true;
      pred.next = qnode;
      // wait until predecessor gives up the lock
      while (qnode.locked) {}
    }
  }
```

```
  public void unlock() {
    QNode qnode = myNode.get();
    if (qnode.next == null) {
      if (tail.compareAndSet(qnode, null))
        return;
      // wait until predecessor fills in its next field
      while (qnode.next == null) {}
    }
    qnode.next.locked = false;
    qnode.next = null;
  }
}
```

Zanimljiv je pojam *konsenzus objekt* (*consensus object*), uveden u [10]. Ne ulazeći u detalje, tj. matematičke definicije i dokaze navedene u [10], mogu se navesti neki zanimljivi rezultati. Konsenzus objekt je objekt čija klasa ima metodu *decide()*. Cilj je napraviti rješenja bez čekanja (wait-free solutions) za tzv. *problem konsenzusa* (*consensus problem*), kod kojeg se grupa procesa pokušava usaglasiti oko zajedničke vrijednosti. *Broj konsenzusa* (*consensus number*) je maksimalni broj procesa za koje određena primitivna operacija (koju koristi konsenzus objekt) može riješiti problem konsenzusa.

Zanimljivi su sljedeći rezultati (teoremi) iz [10]:

- *atomarni registri* imaju broj konsenzusa 1;
- *FIFO redovi* (queues) imaju broj konsenzusa 2; korolar je da je nemoguće napraviti implementaciju bez čekanja za redove, stogove (stacks), skupove (sets) ili liste samo od skupova atomarnih registara;
- tzv. *Common2 RMW* (Read–Modify–Write) operacije, koje su standardno realizirane u procesorima, imaju broj konsenzusa 2; to su npr. operacije *getAndSet(v)*, *getAndIncrement()*, *getAndAdd(k)* i slične;
- registri koji koriste *compareAndSet()* i *get()* operacije imaju beskonačni broj konsenzusa.

Kako se navodi u [10], strojevi (računala) koja imaju operacije kao što je *compareAndSet()* znače na području asinkronog računanja (asynchronous computation) ono što Turingovi strojevi znače na području sekvencijalnog računanja (sequential computation): svaki konkurentni objekt koji se može potencijalno implementirati, može se implementirati na način bez čekanja (wait-free manner) na takvim strojevima. Po autorima u [10]: "***compareAndSet()* is the king of all wild things**".

Osim navedenih nižih osnovnih operacija (primitives) za sinkronizaciju, postoje i više osnovne operacije, kao što su *semafori* i *monitori*. Semafore je izumio E.W. Dijkstra

1965. godine. Oni su vrlo važna viša osnovna operacija za sinkronizaciju, ali ne baš dovoljno visoka [14]. Opći semafor je objekt koji se sastoji od varijable *count* i dvije operacije *wait* i *signal*. Ako dretva (ili proces) zove *wait* dok je *count > 0*, tada se *count* smanjuje za 1, a inače dretva čeka da *count* postane pozitivan. Ako dretva zove *signal*, *count* se povećava za 1. Testiranje varijable *count*, te njeno dekrementiranje i inkrementiranje, mora biti izvedeno atomarnim operacijama niže razine (kao što su one prije navedene). Postoje i *binarni semafori*, koji mogu imati vrijednost 0 ili 1. Semafori nisu baš pogodni za konkurentno programiranje, jer pružaju preveliku fleksibilnost: teško je odrediti da li je korištenje semafora u nekom dijelu programa izvršeno korektno, pa često treba pregledati cijeli program; operacije *wait* i *signal* se često mogu greškom izostaviti, ili staviti na pogrešno mjesto; lako je uvesti deadlock u program.

Zbog toga su Per Brinch-Hansen i Tony Hoare 1973./1974. godine izumili *monitore*. Treba napomenuti da se monitori često implementiraju na temelju semafora (iako mogu i na drugim temeljima), ali može se i obrnuto - implementirati semafore na temelju monitora [14], što je važno za teoretska razmatranja, ali u praksi nije iskoristivo. Monitori se temelje na objektno-orijentiranim principima (iako njegovi izumitelji u to vrijeme nisu tako zvali; napomena: prvi OOPL Simula 67 napravljen je još 1967.). U terminima objektno orijentirane paradigme može se reći da:

- *klasa monitor* (*monitor class*) ima atribute koji su isključivo privatni, a njene metode (funkcije i procedure) se izvode sa međusobnim isključivanjem (mutual exclusion);
- monitor je objekt (instanca) klase monitor.

Intuitivno, atributi klase monitor odgovaraju djeljivim varijablama (shared variables), tj. dretve im mogu pristupati samo preko monitor metoda. Tijela rutina odgovaraju kritičnim sekcijama, jer u isto vrijeme može biti aktivna najviše jedna metoda monitora.

Prednosti monitora [14] jesu:

- *strukturni pristup*: programer ne mora pamtiti da iza operacije *wait* mora staviti *signal* i sl.;
- *raspodjela odgovornosti* (*separation of concerns)*: međusobno isključivanje se dobije automatski, a za *sinkronizaciju na temelju uvjeta* (*condition synchronization*) se koriste *varijable uvjeta* (*condition variables*).

Nedostaci monitora [14] jesu:

- *performanse*: lakši su za programiranje, ali ponekad sporiji od semafora;
- *pravila signalizacije* (*signaling disciplines*): mogući su izvor konfuzije; jedno od dva

pravila za signalizaciju, tzv. *Signal and Continue*, može biti problematično kod korištenja, kad se sinkronizacijski uvjet promijeni prije nego čekajuća dretva uđe u monitor (upravo to pravilo koristi Java; drugo pravilo je *Signal and Wait*);

- *ugnježđeni pozivi monitora* (*nested monitor calls*): ako metoda r1 monitora M1 poziva r2 od M2, a r2 sadrži operaciju *wait*, da li se međusobno isključivanje treba odnositi na M1 i M2, ili samo M2?

Kako se navodi u [8], zaključavanje, bez obzira na varijantu, ima i mana. Npr. moderni JVM-ovi mogu optimizirati baratanje lokotima, ali kada više dretvi istovremeno traže isti lokot, JVM najčešće treba "pomoć" operacijskog sustava, pri čemu dretve najčešće bivaju suspendirane (dok se lokot ne otključa). Zapravo, "pametni" JVM-ovi mogu koristiti statističke podatke (profiling data) za odluku da li je bolje suspendirati dretvu, ili koristiti spin zaključavanje (spin locking).

Zaključavanje ima i druge mane. Ako je dretva koja drži lokot odgođena na duže vrijeme, nijedna dretva koja treba taj lokot ne može napredovati. Taj problem se uvećava kada lokot čeka dretva višeg prioriteta od one koja drži lokot – to se naziva *inverzija prioriteta* (*priority inversion*). Ako je dretva koja drži lokot permanentno blokirana (npr. zbog beskonačne petlje, deadlocka, livelocka i dr.), nijedna dretva koja čeka taj lokot neće moći napredovati. No, najveću manu zaključavanja autori u [10] vide u tome što **"nitko stvarno ne zna kako organizirati i održavati veliki sustav temeljen na zaključavanju"**.

Srećom, danas postoji već spomenuta CAS operacija za sinkronizaciju (ili njoj ekvivalentne), koja je po svojoj osnovi optimistička, tj. nije blokirajuća (za razliku od lokota koji su, iako se danas grade na temelju CAS operacije, pesimistički, tj. blokirajući). Sljedeći primjer koda [8] prikazuje brojač koji je realiziran pomoću CAS operacije na *neblokirajući (nonblocking) način* - nema lokota:

```
@ThreadSafe
public class CasCounter {
  private SimulatedCAS value;

  public int getValue() {
    return value.get();
  }

  public int increment() {
    int v;
    do {
      v = value.get();
    } while (v != value.compareAndSwap(v, v + 1));
    return v + 1;
  }
}
```

Kako je već rečeno, CAS operacija je vrlo važna za sadašnjost i budućnost konkurentnog programiranja. Trošak korištenja CAS operacije kod današnjih procesora varira, od oko 10 do oko 150 procesorskih taktova, pri čemu se stalno radi na ubrzavanju. Korištenje CAS-a je jako doprinijelo implementaciji neblokirajućih (bez lokota) konkurentnih algoritama i struktura podataka. Npr. U Javi 5, a naročito 6, implementirani su (pomoću CAS operacije) algoritmi i strukture podataka koji se u konkurentnom radu ponašaju dramatično brže od onih iz Java verzije 4 i ranijih verzija.

U [10] autori navode i mane CAS operacije: algoritme koji koriste CAS (ili ekvivalentne operacije) vrlo je teško smisliti i često su vrlo neintuitivni. Osnovna teškoća sa svim današnjim sinkronizacijskim operacijama (pa i CAS) je da one rade na samo jednoj riječi memorije, što tjera na korištenje kompleksnih i neprirodnih algoritama. Zapravo, postoji operacija *CASD* (*Compare-and-Swap-Double*)*,* ali ona ažurira dvije susjedne riječi, a još nije realizirana operacija *DCAS* koja bi ažurirale dvije memorijske riječi na nezavisnim lokacijama. Pogotovo nije realizirana nekakva operacija *multiCompareAndSet().* No, čak i da postoji, niti ona ne bi u potpunosti riješila problem sinkronizacije.

4.5. Transakcijska memorija

Problem sa svim dosadašnjim sinkronizacijskim mehanizmima i operacijama (i onima koje postoje i navedenim nepostojećim operacijama), bez obzira da li rade ili ne rade zaključavanje, je da se ne mogu lagano komponirati, što ima veliki negativan utjecaj na modularnost konkurentnih programa. Zato je izmišljena *transakcijska memorija* (TM), a njena realizacija može biti softverska (STM), hardverska (HTM) ili hibridna. Transakcija [10] je sekvenca koraka koje izvršava jedna dretva. Transakcije moraju biti *serijabilne* (*serializable*), što znači da mora izgledati kao da se izvršavaju sekvencijalno (jedna iza druge) i onda kada se izvršavaju paralelno. Serijabilnost je na neki način teža varijanta *linearizabilnosti* (*linearizability*). Linearizabilnost definira atomarnost individualnog objekta. Serijabilnost definira atomarnost cijele transakcije, tj. bloka programskog koda koji može uključivati poziv metoda različitih objekata. Ispravno implementirane, transakcije nemaju problem deadlocka ili livelocka.

Postoje različite softverske implementacije transakcijske memorije. Jednu implementaciju za Javu daju autori u [10] (inače, jedan autor je prvi dao ideju hardverske transakcijske memorije, a drugi autor je (su)inventor softverske transakcijske memorije). Npr. programski jezik Clojure podržava STM na razini jezika.

U [10] je dat prikaz i hardverske transakcijske memorije (HTM). Prikazano je kako se standardna hardverska arhitektura može prilagoditi za podršku kratkotrajnih i malih (po veličini) transakcija. Napomenimo da danas postoje više mikroprocesora koji podržavaju HTM. Prvi je (2005.) bio Vega 1 procesor firme Azul Systems, a 2011. mu se pridružio BlueGene/Q firme IBM. Temeljna ideja za podršku HTM-a je u tome da današnji mikroprocesori podržavaju *protokole usklađivanja priručne memorije* (*cache-coherence protocols*), pa time već podržavaju većinu toga što je potrebno za realizaciju HTM-a. Oni već detektiraju i rješavaju sinkronizacijske konflikte između dretvi pisaca (writers), kao i između dretvi čitatelja (readers) i dretvi pisaca, te stavljaju u međuspremnik (buffer) neke probne izmjene (umjesto da ih direktno upisuju u glavnu memoriju). Treba dodati u hardver samo još neke detalje.

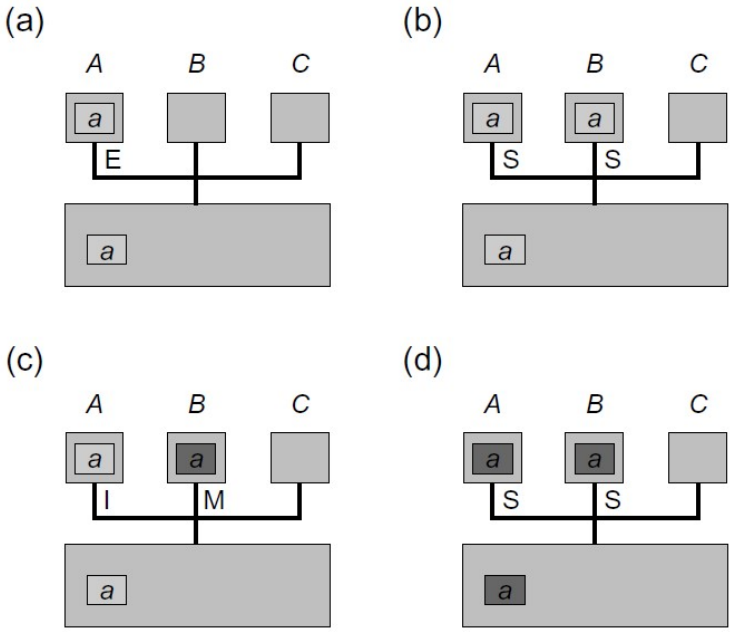
Svaki procesor ima svoju priručnu memoriju. Priručna memorija sastoji se od grupa memorijskih riječi. Grupa se naziva *linija* (*line*) i ima *oznaku* (*tag*), koja pokazuje stanje linije. Koristi se tzv. *MESI protokol*, u kojem je svaka linija u jednom od sljedeća četiri stanja (MESI = početna slova tih stanja):

- **M**odified: linija je modificirana i eventualno treba biti upisana natrag u (glavnu) memoriju;
  nijedan drugi procesor nema tu liniju u svom međuspremniku;
- **E**xclusive: linija nije modificirana, i nijedan drugi procesor ju nema u međuspremniku;
- **S**hared: linija nije modificirana, ali drugi procesori ju mogu imati u međuspremniku;
- **I**nvalid: linija ne sadrži suvisle podatke.

MESI protokol detektira sinkronizacijske konflikte između individualnih čitanja i pisanja, te osigurava da se procesori usklade oko stanja (djeljive) memorije. Kada procesor čita ili piše u memoriju, emitira (broadcasts) zahtjev na sabirnicu (bus), a ostali procesori to slušaju (ponekad se to zove *snooping*). Pojednostavljeno se dešava sljedeće:

- kada procesor daje zahtjev za učitavanje linije u ekskluzivnom modu, ostali procesori invalidiraju svoju kopiju; svaki procesor sa modificiranom kopijom mora upisati svoju kopiju u memoriju, prije nego se nastavi učitavanje;
- kada procesor daje zahtjev za učitavanje linije u djeljivom modu, svi procesori koji imaju ekskluzivnu kopiju moraju joj promijeniti stanje u *Shared*; ostatak je kao prije;
- ako priručna memorija postane puna, mora se izbaciti neka linija; ako je ta linija *Shared* ili *Exclusive*, zanemari se; ako je *Modified*, mora se upisati u memoriju.

Slika 24. prikazuje jedan slijed događaja. Prvo je (a) procesor A učitao (neku) liniju u ekskluzivnom modu. Onda je (b) procesor B isto učitao tu liniju, pa je ona sada u djeljivom modu. Zatim je (c) procesor B mijenjao tu liniju, pa je kod njega ona u statusu *Modified*, a kod procesora A je u statusu Invalid. Na kraju je (d) procesor B upisao tu liniju u memoriju, procesor A je osvježio svoju liniju, i oba procesora opet imaju status linije *Shared*:



Slika 24. MESI protokol; Izvor: [10]

Kako navode autori u [10], za podršku HTM-a treba zadržati MESI protokol kakav jeste, samo treba dodati jedan transakcijski bit u svaku oznaku linije priručne memorije (cache line's tag). Uobičajeno je taj bit postavljen na 0. Kada se u priručnoj memoriji mijenja vrijednost kao posljedica transakcije, bit se postavlja na 1. Treba samo osigurati da se modificirane transakcijske linije ne upisuju natrag u memoriju i da invalidacija linije abortira transakciju. Mana ove sheme je zajednička mana skoro svih HTM shema. Prvo, veličina transakcije ograničena je veličinom priručne memorije (zato npr. IBM BlueGene/Q mikroprocesor ima čak 32 MB L2 memorije, koja se koristi za transakcije). Drugo, većina operacijskih sustava briše priručnu memoriju kada se dretva pasivizira, tako da trajanje transakcije može biti limitirano dužinom "vremenskog kvanta" operacijskog sustava. Po tome bi slijedilo da je HTM najbolji za kratkotrajne i male transakcije, a ostale transakcije trebale bi koristiti STM, ili kombinaciju HTM-a i STM-a.

Iako se često kaže da se transakcijska memorija (bilo softverska ili hardverska) ponaša slično kao transakcije u bazi podataka, postoji jedna značajna razlika. Za razliku od transakcija u bazi podataka, kod kojih se radi zaključavanje redaka tablica, kod transakcijske memorije nema zaključavanja, tako da se nikad ne može desiti deadlock. Transakcijska memorija se ponaša optimistički, tj. polazi od toga da problema u ažuriranju neće biti. Ako se na kraju ipak pokaže da ima problema, transakcija radi rollback. Napomenimo da deadlock u bazi podataka nije problem, jer svi značajni sustavi za upravljanje bazama podataka automatski detektiraju deadlock i poništavaju jednu od sesija koja se nalazi u deadlock konfliktu, te time omogućavaju drugoj sesiji da nastavi sa radom.

Naglasimo da već spomenuti mikroprocesor IBM BlueGene/Q ima jedno značajno ograničenje. Naime, njegova realizacija (hardverske) transakcijske memorije ograničena je na jedan mikroprocesorski čip, tj. na procesorske jezgre unutar jednog čipa. Kod spajanja više mikroprocesorskih čipova, transakcijska memorija između njih ne radi. Istina, to ne stvara veliki problem kod IBM superračunala Sequoia (završen 2012. godine, kada je bio najbrži superkompjuter u svijetu; ove godine (2019.) je 22. na TOP500 listi; sljedeće godine bit će demontiran), koje se sastoji od 100 000 čipova BlueGene/Q, jer je ono namijenjeno za specijalizirane primjene, kod kojih taj nedostatak nije važan. Zanimljivo je napomenuti da IBM BlueGene/Q ima 18 jezgri, od kojih je 16 namijenjeno za rad aplikacijskih programa, jedna je namijenjena za operacijski sustav, a jedna je jezgra pričuvna (spare) i može zamijeniti bilo koju od preostalih 17 jezgara koja se pokvari. Stručnjaci IBM-a predviđali su da će se kod navedenog superračunala svaka 3 tjedna pokvariti (u prosjeku) jedna jezgra.

Intel u nekim svojim mikroprocesorima, počevši od od Haswell arhitekture (2013. godine), također podržava hardversku transakcijsku memoriju. Zanimljivi su po tome što imaju dva načina podržavanja transakcijske memorije. Jedan način, tzv. Hardware Lock Elision, omogućava da se programi koji koriste zaključavanje vrlo lako prerade tako da rade sa transakcijskom memorijom. Suština je u tome da će procesor "lagati" dretve da su dobile lokot, a da stvarno nikakvog zaključavanja neće biti. Novonapisani programi mogu koristiti napredniji način Restricted Transactional Memory, tj. mogu koristiti eksplicitne naredbe XBEGIN, XEND ili XABORT za pokretanje transakcije, normalno završavanje ili abortiranje. Kod pokretanja nove transakcije (sa XBEGIN) dretva će moći navesti i "fallback" rutinu koja će se automatski izvršiti ako transakcija ne uspije.

5. Konkurentno programiranje u Javi

5.1. Konkurentno programiranje u Javi verzije od 1 do 4

Svaki Java program ima barem jednu dretvu, onu koja izvršava metodu main(). Kada želimo napraviti svoju dretvu, imamo u Javi dva načina; jedan način je da napravimo klasu koja nasljeđuje klasu **Thread** i da nadjačamo (override) metodu **run()**, kao što prikazuje sljedeći primjer [14], u kojem se kreiraju dvije klase, Worker1 i Worker2:

```
class Worker1 extends Thread {
  public void run() {
    // implement doTask1() here
  }
}
class Worker2 extends Thread {
  public void run() {
    // implement doTask2() here
  }
}
```

Da bi se kreirale dretva, potrebno je kreirati objekt (instancu) klase (u ovom slučaju klase Worker1 i/ili Worker2) i pozvati metodu start() nad tim objektom. Time će se automatski kreirati dretva i pozvati metoda **run()** u njoj. U nastavku se prikazuje implementacija metode compute() (iz neke treće klase čiji ostali detalji nisu prikazani), koja kreira dvije dretve, tako da dva posla (tasks) mogu biti izvedena paralelno (ako postoje dva slobodna procesora koja ih izvode):

```
void compute() {
  Worker1 worker1 = new Thread();
  Worker2 worker2 = new Thread();
  worker1.start();
  worker2.start();
}
```

Sada klase Worker1 i Worker2 proširimo sa sljedećim atributom i metodom:

```
private int result;
public void getResult() {
  return result;
}
```

Pretpostavimo da dretve snimaju rezultat izračuna u tu varijablu, a metodom getResult() ih možemo čitati. Sada želimo dobiti rezultat oba izračuna u metodi compute():

```
    return worker1.getResult() + worker2.getResult();
```

Očito, moramo čekati da obje dretve završe prije nego dobijemo taj rezultat. To se postiže naredbom **join()** koja, kad se poziva nad dretvom-izvršiteljem, uzrokuje da dretva-pozivatelj čeka dok dretva-izvršitelj ne završi. U ovom slučaju programski kod dretve-pozivatelja izgleda ovako:

```
int compute() {
  worker1.start();
  worker2.start();
  worker1.join();
  worker2.join();
  return worker1.getResult() + worker2.getResult();
}
```

Naravno, u ovom slučaju nije važno da li prvo pozivamo join() nad worker1 ili worker2.

Kako je prije navedeno, postoji i drugi način za kreiranje dretvi u Javi. Prethodno prikazani način (kreiranje klase koja nasljeđuje klasu Thread), iako izgleda logičan, manje se koristi jer je manje fleksibilan. Problem je u tome što u Javi (za sada) postoji samo jednostruko nasljeđivanje klasa. Bez višestrukog nasljeđivanja klasa, ako naša klasa "potroši" jednostruko nasljeđivanje za klasu Thread, ne može naslijediti od neke druge klase. Zbog toga se češće koristi drugi način kreiranja dretve. Prvo se napiše "pomoćna" klasa koja nasljeđuje sučelje (interface) **Runnable**, pri čemu mora implementirati metodu run(). Onda se objekt te "pomoćne" klase šalje konstruktoru koji kreira objekt klase Thread (tj. dretvu) u kodu naše "glavne" klase.

Do sada prikazani rad sa dretvama u Javi izgleda prilično jednostavno. No, problemi nastaju kada se dvije dretve (ili više njih) upliću jedna drugoj u posao, npr. tako da modificiraju isti objekt. To može stvoriti netočne rezultate i naziva se **race condition**, npr. u ovom slučaju [14]:

```
class Counter {
  private volatile int value = 0;

  public int getValue() {
    return value;
  }

  public void setValue(int someValue) {
    value = someValue;
  }

  public void increment() {
    value++; -- napomena: niti sama naredba value++ nije atomarna
  }
}
```

Pretpostavimo da neka metoda u nekoj drugoj klasi kreira objekt klase Counter i sadrži sljedeći kod:

```
x.setValue(0);
x.increment();
int i = x.getValue();
```

Pitanje je: koju vrijednost ima varijabla i na kraju ovih naredbi? Ako je riječ o jednodretvenom programu, onda je vrijednost 1. No u konkurentnom radu brojač može biti modificiran od drugih dretvi, tako da rezultat ovisi o ispreplitanju naredbi ove dretve sa naredbama neke druge dretve. Npr. ako druga dretva konkurentno izvodi naredbu **x.setValue(2);** varijabla i na kraju navedenih naredbi prve dretve može imati vrijednost 1, 2 ili 3, tj. rezultat ovisi o slučajnom redoslijedu izvođenja naredbi dvije dretve (zapravo, dodatni problem je i u tome što sama naredba value++ nije atomarna, već se u stvarnosti razlaže na tri interne naredbe: temp = value; temp = temp + 1; value = temp). Naravno, to nije ono što se želi. Taj se problem rješava pomoću sinkronizacije koja se zove međusobno isključivanje (mutual exclusion), za što Java ima jednostavno rješenje još od verzije Java 1. Svaki objekt u Javi ima lokot (lock; nasljeđuje se automatski od superklase **Object**), kojega istovremeno može držati (zaključati) samo jedna dretva.

Objekt koji će služiti kao lokot može se kreirati npr. ovako:

```
Object lock = new Object();
```

Dretva koja će tražiti lokot (tj. zaključati lokot) to radi pomoću naredbe **synchronized**, koja označava početak tzv. synchronized bloka (inače synchronized i **volatile** su jedine Java ključne riječi vezane za konkurentnost; ostalo su metode klase Object ili metode klasa specijaliziranih za konkurentnost):

```
synchronized(lock) {
  // critical section
}
```

Kada dretva dođe do početka tog bloka, pokuša zaključati lokot objekta koji je naveden kao argument naredbe synchronized. Ako je lokot zaključan od neke druge dretve, polazna dretva čeka dok on ne postane otključan. Nakon toga ga polazna dretva drži zaključanim sve do kraja tog bloka. Problem iz prethodnog primjera mogli bismo riješiti pomoću synchronized npr. ovako (u prvoj, odnosno drugoj dretvi; naravno, moramo biti sigurni da varijable lock u oba koda referenciraju isti objekt):

```
// prva dretva
synchronized(lock) {
  x.setValue(0); x.increment(); int i = x.getValue();
}
// druga dretva
synchronized(lock) {
  x.setValue(2);
}
```

Osim bloka, i cijela metoda (funkcija / procedura) može imati synchronized na početku:

```
synchronized type method(args) {
  // body
}
```

što je, zapravo, isto kao i ovo:

```
type method(args) {
  synchronized(this) {
    // body
  }
}
```

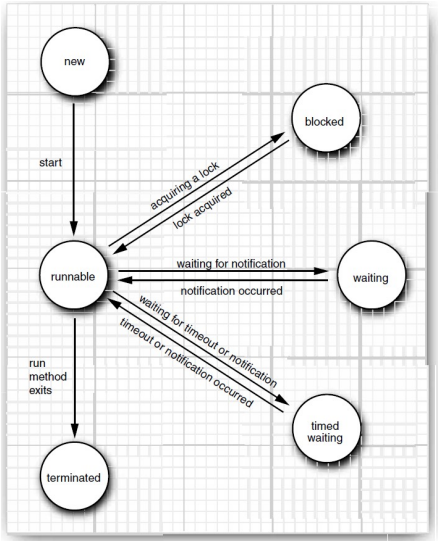
Prethodno je slično konceptu monitora, ali dizajneri Jave su napravili određena odstupanja od tog koncepta. Svaki objekt u Javi ima unutarnji (intrinsic) lokot i unutarnju kondiciju (condition). Ako je metoda deklarirana pomoću ključne riječi synchronized, ona djeluje kao monitor. Međutim, Java objekt se razlikuje od monitora [11] u tri važne stvari, kompromitirajući time sigurnost dretve (thread safety) :

- atributi ne moraju biti privatni;
- metode ne moraju biti sinkronizirane;
- unutarnji lokot je pristupačan klijentima.

Zbog toga je jedan od inventora monitora, Per Brinch Hansen, 1999. godine izjavio [3]: "Zaprepašćuje me da je ovaj nesigurni Java paralelizam shvaćen tako ozbiljno od programske zajednice, četvrt stoljeća nakon invencije monitora i programskog jezika Concurrent Pascal."

Kao što vrijedi za monitor, tako vrijedi i za Java klase - zaštita pristupa djeljivim varijablama nije jedini razlog zašto dretve moraju biti međusobno sinkronizirane. Često puta treba odgoditi izvođenje metode (ili dijela metode) u nekoj dretvi, dok se ne zadovolji određeni uvjet (a taj uvjet nije samo otključavanje određenog lokota). To se zove **sinkronizacija na temelju uvjeta** (condition synchronization), koja se u Javi implementira pomoću naredbi **wait / notifyAll / notify**, koje se pozivaju nad sinkroniziranim objektima.

Slika 25. prikazuje promjenu stanja Java dretvi (napomenimo da zbivanja koja implicitno sadrže naredbe notify() / notifyAll() nisu prikazana potpuno detaljno). Java dretva prelazi u stanje Waiting ne samo nakon naredbe wait() i join(), već i kod čekanja na objekte klase **Lock** i **Condition** iz java.util.concurrent librarya, uvedenog u Java 5 verziji.



Slika 25. Dijagram promjene stanja Java dretvi; Izvor: [11]

5.2. Konkurentno programiranje u Javi verzije od 5 do 8 (kratak pregled)

Konkurentnost u Javi od verzije 1 do verzije 4 praktički se nije mijenjala, osim što su se rješavali bugovi i sl. Suštinu "alata" za konkurentno programiranje činili su: klasa **Object**, tj. njen unutarnji (intrinsic) lokot (atribut te klase) i njene metode **wait(...) / notify / notifyAll**, Java ključne riječi **synchronized** i **volatile**, klasa **Thread** i sučelje **Runnable**.

U Javi verzije 5, koja se pojavila 2004. godine, uvedeno je dosta novina na drugim područjima (npr. generičke klase, bolje kolekcije i dr.), ali i na području konkurentnog programiranja. JSR 166, koji se odnosi na konkurentnost, temeljen je uglavnom na *paketu* edu.oswego.cs.dl.util.concurrent, kojega je napravio Doug Lea. Kroz novi paket java.util.concurrent uvedene su sljedeće nove mogućnosti:

- Executors (thread pools, scheduling);
- Futures;
- Concurrent Collections;
- Locks (ReentrantLock, ReadWriteLock...);
- Conditions;
- Synchronizers (Semaphores, Barriers...);
- Atomic variables;
- System enhancements.

U Java verziji 6, koja se pojavila godinu i pol nakon verzije 5, nije se pojavilo ništa revolucionarno, ali su se na području konkurentnog programiranja (kao i na drugim područjima) "iznutra" poboljšale biblioteke, bilo da su se riješili bugovi ili poboljšale performanse. U Java verziji 7, koja je izašla 2011. godine, u području konkurentnog programiranja novost je ForkJoin framework. U Java verziji 8, koja je izašla u ljeto 2014. godine, u području konkurentnog programiranja novost je Stream API, koji je podržan lambda izrazima i default metodama (koje su i same po sebi jako značajne).

U ovoj točki prikazat će se (ukratko) samo jedna mogućnosti uvedena u Javi 5 (sa poboljšanjima u Javi 6), i to **ReentrantLock** (u sljedećoj točki prikazat će se Java 5 executori). Do Java 5 verzije, jedini mehanizmi za koordinaciju pristupa djeljivim podacima bili su synchronized (koji koristi unutarnji lokot) i volatile. Java 5 donijela je i ReentrantLock, što je (klasa za) eksplicitan lokot. Kako navode autori u [8], on nije zamjena za implicitan, unutarnji lokot, već alternativa koju je bolje koristiti u nekim (ali ne svim) slučajevima. Klasa ReentrantLock implementira sučelje **Lock**, koje ima ove metode:

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long timeout, TimeUnit unit)throws InterruptedException;
  void unlock();
  Condition newCondition();
}
```

Zaključavanja pomoću objekata klase ReentrantLock ima istu semantiku kao i synchronized, ali ima i dodatne mogućnosti. Najčešći oblik korištenja je sljedeći [8]:

```
Lock lock = new ReentrantLock();
...
lock.lock();
try {
  // update object state
  // catch exceptions and restore invariants if necessary
} finally {
  lock.unlock();
}
```

Za razliku od implicitnog zaključavanja i otključavanja kod synchronized, ovdje se zaključavanje i otključavanje mora raditi eksplicitno, a vrlo je važno da se otključavanje stavi u finally blok, inače lokot može ostati stalno zaključan. Moglo bi se reći da je to korak nazad u odnosu na synchronized, jer je sad moguće (dodatno) pogriješiti. No, mogućnost da se zaključavanje i otključavanje lokota napravi u različitim dijelovima koda ponekad je nužna (iako je takav kod manje čitljiv), a sa synchronized se to nije

moglo izvesti. Preporuča se korištenje dosadašnje synchronized varijante ako nam ne trebaju mogućnosti klase ReentrantLock.

5.3. Paralelno programiranje pomoću executora (Java 5 i dalje)

U Javi 5 su kroz paket java.util.concurrent uvedeni **executori** (tj. sučelja Executor, ExecutorService, Callable, Future, klase Executors, ThreadPoolExecutor, FutureTask i dr.). Executori, za razliku od direktnog rada s klasom Thread, pomažu da se programeri koncentriraju na kreiranje zadataka koji će se poslati executoru na izvršavanje, a optimizaciju izvršavanja rade executori, koji koriste **thread pool**. Executorima se daje zadatak koji je instanca klase (najčešće anonimne) koja implementira sučelje **Runnable** ili **Callable**.

Pretpostavimo da imamo ovakav zadatak: naći koliko ima prim (prostih) brojeva među prvih N (npr. 10 000 000) prirodnih brojeva (N zadajemo). Naravno, cilj nam je izvršiti zadatak u što kraćem vremenu. Zbog toga želimo zadatak riješiti kroz paralelno programiranje, tako da maksimalno koristimo procesorske resurse.

Glavno pitanje je: **kako podijeliti zadatak na podzadatke?** Povezano pitanje je: koliko Java dretvi kreirati? Možda onoliko koliko ima podzadataka? To najčešće ne bi bilo dobro! Za (približno) određivanje broja Java dretvi može se koristiti jednostavna formula:

broj_Java_dretvi = broj_HW_dretvi / (1 – koeficijent_blokiranja)

Koeficijent blokiranja (blocking coefficient) je broj između 0 i 1 i nije ga lako odrediti. Računski intenzivni problemi imaju koeficijent koji se približava nuli, pa je tada preporučeni broj Java dretvi jednak ili nešto malo veći od broja HW dretvi.

No, sada treba odrediti kako podijeliti problem, tj. koliko ćemo imati podzadataka. Svaki podzadatak radit će konkurentno, pa ih svakako treba biti barem onoliko koliko ima Java dretvi. U općenitom slučaju može se primijeniti relativno jednostavna tehnika: broj podzadataka treba biti dovoljno velik da se iskoriste postojeće Java dretve, tj. **ne smije se desiti da neke Java dretve ostanu dugo neiskorištene.**

Dakle, broj podzadataka svakako mora biti veći od broja Java dretvi. Naravno, nije lako odrediti (a ponekad niti moguće) koliki točno treba biti taj broj – najčešće treba eksperimentirati. Uglavnom se pokazuje da se kod početnog povećanja broja podzadataka dobije značajno povećanje performansi, a s daljnjim povećanjem,

povećanje performansi je sve manje (performanse se mogu i smanjiti).

Slijedi Java program, koji se sastoji od klase PrimesExecutor, a ulazni parametri su:

- **number**: gornja granica do koje se traže prim brojevi;
- **poolSize**: broj Java dretvi;
- **numberOfParts**: broj podzadataka.

```
import java.util.List;
import java.util.ArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Callable;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class PrimesExecutor {
  public static void main(final String[] args) {
    if (args.length != 3) {
      System.out.println("Usage: number poolSize numberOfParts");
      return;
    }

    final long number = Long.parseLong(args[0]);
    final int poolSize = Integer.parseInt(args[1]);

    long numberOfParts = Long.parseLong(args[2]);
    // svaki podzadatak radi barem 10 brojeva
    if (numberOfParts > number) numberOfParts = number / 10;

    final long numberOfPrimes;
    final long startTime = System.nanoTime();

    if (number <= 1)
      numberOfPrimes = 0;
    else {
      PrimesExecutor task = new PrimesExecutor();
      numberOfPrimes = task.countPrimes(number, poolSize, numberOfParts);
    }

    final long endTime = System.nanoTime();
    System.out.printf
      ("Number of primes under %d is %d\n", number, numberOfPrimes);
    System.out.println
      ("Time (seconds) taken is " + (endTime - startTime) / 1.0e9);
  }

  private long countPrimes
    (final long number, final int poolSize, final long numberOfParts)
  {
    long count = 0;

    try {
      final List<Callable<Long>> partitions = new ArrayList<>();
      // zaokruživanje na više
      final long chunksPerPartition =
        (number + numberOfParts - 1) / numberOfParts;
```

```
      for(long i = 0; i < numberOfParts; i++) {
        final long lower = i * chunksPerPartition + 1;
        long upperTemp = (i + 1) * chunksPerPartition;

        // zadnji podzadatak može imati manje brojeva
        if (upperTemp > number) upperTemp = number;

        // lokalna varijabla referencirana iz inner class mora biti final
        // zato smo koristili upperTemp
        final long upper = upperTemp;
        partitions.add(new Callable<Long>() {
          public Long call() {
            return countPrimesInRange(lower, upper);
          }
        });
      }

      final ExecutorService executorPool =
        Executors.newFixedThreadPool(poolSize);
      final List<Future<Long>> resultFromParts =
        executorPool.invokeAll(partitions, 10000, TimeUnit.SECONDS);

      executorPool.shutdown();
      for(final Future<Long> result : resultFromParts) {
        count += result.get();
      }
    } catch(Exception ex) { throw new RuntimeException(ex); }

    return count;
  }


  private static long countPrimesInRange(long lower, long upper) {
    long total = 0;

    // Provjeravaju se i parni brojevi, zbog usporedbe sa Streams rješenjem
    for(long i = lower; i <= upper; i++) {
      if (isPrime(i)) total++;
    }

    // 1 nije prim, ali 2 je prim (a za 2 isPrime daje false),
    // pa je ukupan broj točan.
    return total;
  }

  private static boolean isPrime(final long number) {
    if (number % 2 == 0) return false;

    for(long i = 3; i * i <= number; i = i + 2) {
      if (number % i == 0) return false;
    }
    return true;
  }
}
```

5.4. Paralelno programiranje pomoću Fork Join frameworka (Java 7 i dalje)

Stara rimska poslovica glasi: "Divide et impera" ("Podijeli pa vladaj" ili "Zavadi pa vladaj", engl. "Divide and Conquer").

Implementaciju ExecutorService sučelja u slučajuForkJoin frameworka radi klasa **ForkJoinPool**. Tipično se ForkJoinPool instanci šalje samo jedan zadatak (task), a onda ForkJoinPool instanca i zadatak zajedno primjenjuju tehniku Divide and Conquer.

Broj Java dretvi u poolu se može zadati eksplicitno ili implicitno:

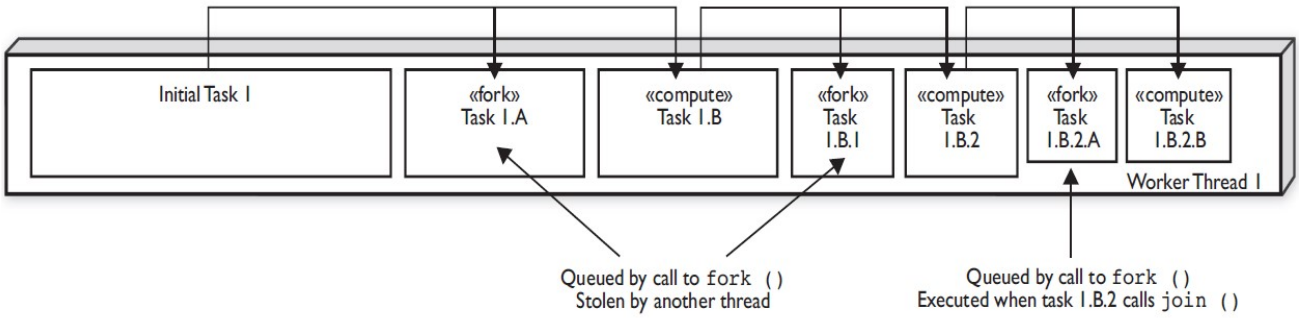
- broj Java dretvi se zadaje eksplicitno (u ovom slučaju 8):
  ForkJoinPool fjPool = new ForkJoinPool(8);
- ili implicitno, jer framework koristi metodu Runtime.availableProcessors()
  (na računalu sa 8 HW dretvi, bit će 8 Java dretvi):
  ForkJoinPool fjPool = new ForkJoinPool();

Zadatak (task) treba biti instanca podklase apstraktne klase **ForkJoinTask**, preciznije podklasa apstraktne podklase klase ForkJoinTask, a najčešće su to **RecursiveTask** (u primjeru) ili **RecursiveAction**. ForkJoinTask ima puno metoda, ali najvažnije su: **compute(), fork(), join().**

Slijedi pseudo kod za **compute():**

```
if (podzadatakJeDovoljnoMali()) {
  rijesiPodzadatak();
} else {
  MojForkJoinTask lijevaPolovica = ...
  MojForkJoinTask desnaPolovica = ...
  lijevaPolovica.fork();  -- stavi u queue
  desnaPolovica.compute();-- radi(rekurzivno)
  lijevaPolovica.join();  -- čekaj završetak
}
```

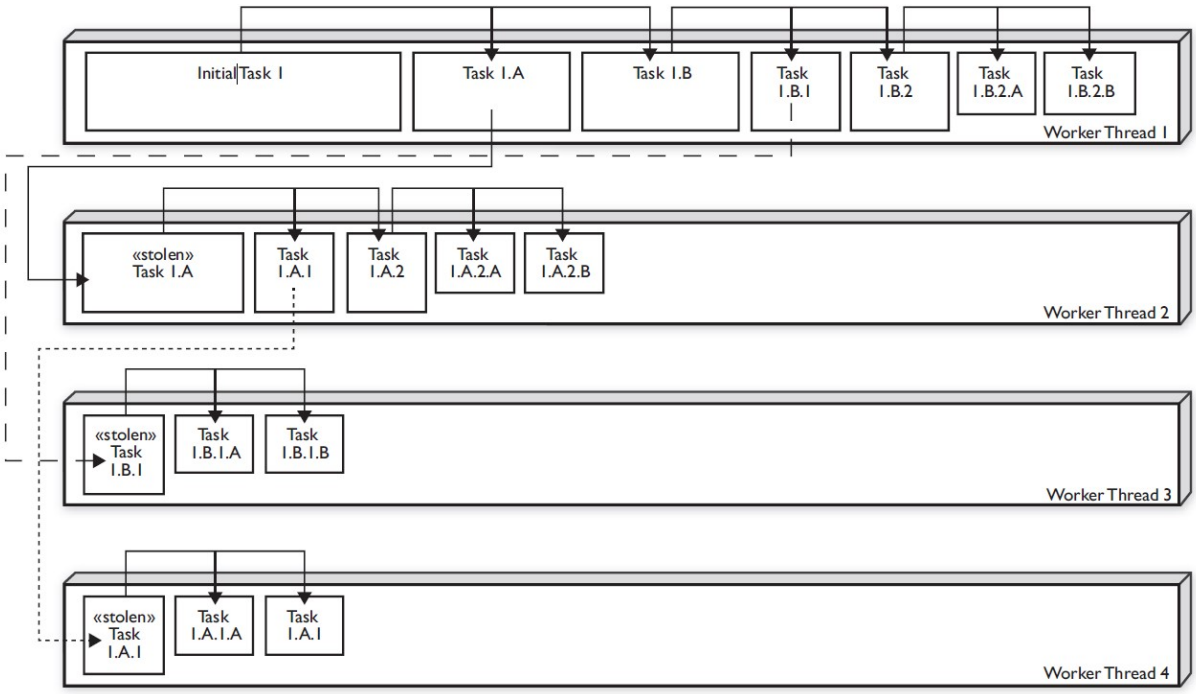
Za razliku od većine ExecutorService implementacija, kod ForkJoin frameworka **svaka Java dretva u ForkJoinPool-u ima svoj red (queue)** podzadataka na kojima radi. Metoda fork() stavlja ForkJoinTask u red tekuće Java dretve. Inicijalno je zauzeta samo jedna Java dretva - kada joj pošaljemo (cijeli) zadatak. Dretva tada počinje dijeliti zadatak u dva podzadatka, pa prvi podzadatak (lijevi) stavi u red, a drugi podzadatak (desni) pokuša izvršiti (i tako rekurzivno), kao što prikazuje slika 26.



Slika 26. ForkJoin - podjela zadatka na podzadatke; Izvor: [21]

Ključna značajka ForkJoin frameworka je **Work Stealing** (krađa posla). Java dretve kradu posao (podzadatak) drugoj Java dretvi iz njenog reda podzadataka, i stavljaju ga u svoj red (nešto što većina nas ne bi nikad napravila ☺).

Vrlo je važan redoslijed stavljanja podzadataka u red. Podzadaci koji se prvi stavljaju u red trebaju predstavljati veći dio posla. Npr. na početku imamo jedan zadatak, koji pokriva 100% posla. Njega dijelimo u dva podzadatka, svaki po (otprilike) 50% posla. Prvi stavljamo u red, a drugi obrađujemo, pri čemu drugi opet dijelimo u polovice, itd. One Java dretve koje nemaju posla (tj. njihov red je prazan), kradu posao drugim Java dretvama, i to tako da uzmu posao (podzadatak) koji je najstariji u redu, a to je istovremeno i najveći posao iz reda druge Java dretve. Slika 27. prikazuje ta događanja, s četiri Java dretve.



Slika 27. ForkJoin – krađa podzadataka od strane drugih Java dretvi; Izvor: [21]

Dijeljenje na podzadatke se, za razliku od primjera sa običnim executorima, radi implicitno – **ne zadaje se broj podzadataka, već se određuje kada je podzadatak dovoljno mali**. Nažalost, ne postoji opća metoda kojom se ispravno određuje veličina tog malog podzadatka – to se radi eksperimentalno. Podzadatak na kojem se poziva metoda join() može tada biti već gotov, jer ga je možda ukrala druga Java dretva, i već napravila. Ili može biti ukraden, ali ga druga dretva upravo radi, pa tada treba čekati da završi. Treći je slučaj da podzadatak nije ukraden i tada ga radi tekuća dretva.

Poziv join() metode u compute() metodi treba biti jedna od zadnjih radnji, iza poziva fork() metode i rekurzivnog poziva compute() metode. Budući da je to jako važno, postoji i metoda **invokeAll(a2, a1);** koja može zamijeniti niz naredbi **a1.fork(); a2.compute(); a1.join();**

Slijedi Java program, koji se sastoji od klase **PrimesForkJoin**, koji ima ove ulazne parametre:

- **number**: gornja granica do koje se traže prim brojevi;
- **poolSize**: broj Java dretvi;
- **threshold**: broj koji definira kada je podzadatak dovoljni mali (pa se više ne dijeli na dva).

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.ExecutionException;

public class PrimesForkJoin extends RecursiveTask<Long> {
  private static int threshold;
  private long start;
  private long end;

  private PrimesForkJoin(final long theStart, final long theEnd) {
    start = theStart;
    end = theEnd;
  }

  public static void main(final String[] args) {
    if (args.length < 1 || args.length > 3) {
      System.out.println
       ("Usage: number poolSize threshold  OR  number poolSize  OR  number");
      return;
    }
    final long number = Long.parseLong(args[0]);
    final long numberOfPrimes;
    final long startTime = System.nanoTime();

    if (number <= 1)
      numberOfPrimes = 0;
    else {
      PrimesForkJoin task = new PrimesForkJoin(1, number);
```

```
      ForkJoinPool fjPool;
      if (args.length == 2 || args.length == 3) {
        fjPool = new ForkJoinPool(Integer.parseInt(args[1]));
      } else {
        fjPool = new ForkJoinPool();
      }

      if (args.length == 3) {
        threshold = Integer.parseInt(args[2]);
      } else {
        threshold = 100;
      }

      numberOfPrimes = fjPool.invoke(task);
    }

    final long endTime = System.nanoTime();
    System.out.printf
      ("Number of primes under %d is %d\n", number, numberOfPrimes);
    System.out.println
      ("Time (seconds) taken is " + (endTime - startTime) / 1.0e9);
  }

  protected Long compute() {
    if (end - start <= threshold) {
      return countPrimesInRange(start, end);
    } else {
      long halfWay = ((end - start) / 2) + start;
      PrimesForkJoin t1 = new PrimesForkJoin(start, halfWay);
      PrimesForkJoin t2 = new PrimesForkJoin(halfWay + 1, end);
      t1.fork();
      long count2 = t2.compute();
      long count1 = t1.join();
      return count1 + count2;
    }
  }

  private static long countPrimesInRange ... kao prije ...

  private static boolean isPrime ... kao prije ...
}
```

5.5. Paralelno programiranje pomoću paralelnih Streamsa (Java 8 i dalje)

Na temelju onoga što čitamo i čujemo, mogli bismo zaključiti da je najvažnija nova mogućnost u Javi 8 **lambda izraz** (ili kraće, **lambda**). Inače, lambda izraz je (u Javi) naziv za metodu bez imena.

U pravilu je ta metoda funkcija, a ne procedura. Zato možemo reći i da **lambda izraz je anonimna funkcija**, koja se može javiti kao parametar (ili povratna vrijednost) druge funkcije (koja je, onda, funkcija višeg reda). U Javi 8 pojavile su se i tzv. **default metode** u Java sučeljima (interfaces). One, zapravo, predstavljaju uvođenje **višestrukog nasljeđivanja implementacije** u Javu. Međutim, lambda izrazi i default metode su, na neki način, posljedica uvođenja treće važne mogućnosti u Javi 8, a to je

Stream API, koji nadograđuju dosadašnje Java kolekcije.

Lambda izrazi (ili lambda funkcije), imaju teoretsko porijeklo u **lambda računu** (lambda calculus), kojega je sredinom 30-ih godina prošlog stoljeća kreirao **Alonzo Church**. Poznata je **Church-Turingova hipoteza** (inače, **Alen Turing** je kod Alonza Churcha radio doktorat), koja se ne može matematički dokazati, već se smatra intuitivno prihvatljivom. Ona (pojednostavljeno, te u današnjoj terminologiji) kaže da sve što se efektivno može izračunati, može se izračunati pomoću lambda računa ili **Turingovog stroja**. 50-ih godina se formalno dokazalo da postoje i neki drugi sustavi koji su njima ekvivalentni: Gödelove rekurzivne funkcije, Postov sustav, Markovljevi algoritmi i dr. Bez obzira na teoretsku ekvivalentnost, Turingovi strojevi su po svom ponašanju bliži imperativnoj programskoj paradigmi, dok je lambda račun teoretska osnova za funkcijske programske jezike, od kojih je najstariji **Lisp** (nastao 1959.).

Java 8 lambda (izrazi) temelje se na tzv. **funkcijskim sučeljima** (functional interface), koji su postojali od početka. Funkcijska sučelja su ona sučelja koja imaju točno jednu (jednu i samo jednu) apstraktnu funkciju. No, od Jave 8 funkcijska sučelja mogu imati i statičke metode i default metode (koje nisu postojale prije Jave 8). Npr. kad Java kompajler naiđe na ovakvu naredbu (lambda izraz je desno od znaka jednakosti; ovo je samo jedna od brojnih varijanti pisanja lambda izraza):

StringToIntMapper mapper = (String str) -> str.length();

kompajler provjerava da li postoji odgovarajuće sučelje StringToIntMapper, koje ima samo jednu apstraktnu funkciju. Pretpostavimo da postoji takvo sučelje:

```
// anotacija @FunctionalInterface uvedena je u Javi 8
@FunctionalInterface interface StringToIntMapper {
  int map(String str);
}
```

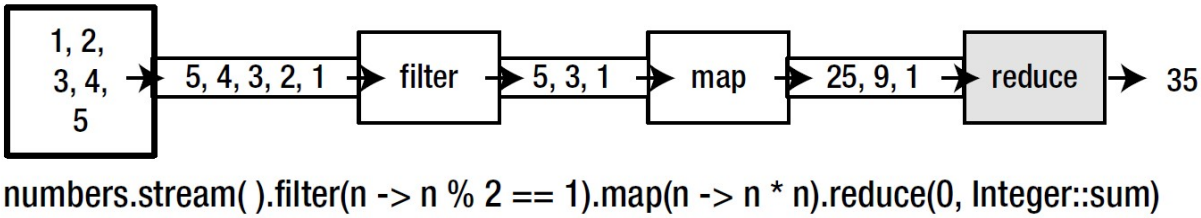
Kompajler tada napravi nešto što ima jednak efekt kao sljedeća anonimna klasa:

```
StringToIntMapper mapper =
  new StringToIntMapper() {
    @Override
    public int map(String str) {
      return str.length();
    }
  };
```

Kako je već rečeno, default metode su u Javi 8 trebale prvenstveno zbog uvođenja Stream API-a. Kod uvođenja Streamsa, bilo je potrebno nadograđivati brojna postojeća Java sučelja, tj. dodavati im nove metode. Međutim, kad dodajemo nove metode u

sučelje, moramo mijenjati sve klase koje ga (direktno ili indirektno) nasljeđuju, što može značiti izmjenu milijuna redaka programskog koda. Default metode su riješile taj problem. Sučelje sada može imati i implementaciju metode, a ne samo deklaraciju.

Pojava masivno višejezgrenih procesora traži bolji i lakši način izrade paralelnih programa od dosadašnjih načina. Jedan (dobar) način je da se koristi funkcijsko programiranje, a naročito paralelne kolekcije. Java nema paralelne kolekcije (ima samo paralelni niz, uveden u Javi 8), ali su zato uvedeni Streamsi, kako bi se postojeće kolekcije "zaogrnule" u (paralelne) Streamse i mogle (indirektno) paralelizirati. Za razliku od dosadašnjih kolekcija, koje sadrže sve svoje elemente u memoriji, Streamse možemo shvatiti kao "vremenske kolekcije", čiji se elementi (kojih teoretski može biti i beskonačan broj) stvaraju po potrebi. Način rada Java Streamsa prikazuje slika 28.



Slika 28. Način rada Java Streamsa; dretvi; Izvor: [19]

Usporedba rada s kolekcijama na klasičan način (korištenjem for petlje), pomoću lambda izraza bez Streamsa, i pomoću paralelnih Streamsa:

```
// 1. "klasična" obrada "klasične" kolekcije - for petlja
private static void checkBalance(List<Account> accList) {
  for (Account a : accList)
    if (a.balance() < a.threshold) a.alert();
}

// 2. primjena lambda izraza za obradu "klasične" kolekcije
private static void checkBalance(List<Account> accList) {
  accList.forEach(
   (Account a) -> { if (a.balance() < a.threshold) a.alert(); }
  );
}

// 3. primjena Streamsa za paralelizaciju "klasične" kolekcije
private static void checkBalance(List<Account> accList) {
  accList.parallelStream().forEach(
   (Account a) -> { if (a.balance() < a.threshold) a.alert(); }
  );
}
```

Paralelne verzije Streamsa interno se temelje na ForkJoin frameworku.

Slijedi Java program, koji se sastoji od klase **PrimesStream**, koji ima ove ulazne parametre:

- **number**: gornja granica do koje se traže prim brojevi;
- **isParallel**: unese se 0 za serijsko izvršavanje, 1 za paralelno izvršavanje.

```
import java.util.stream.*;
import java.util.concurrent.TimeUnit;

public class PrimesStream {
  public static void main(final String[] args) {
    if (args.length != 2) {
      System.out.println("Usage: number 1/0 (parallel/not parallel)");
      return;
    }

    final long number = Long.parseLong(args[0]);
    final boolean isParallel =
     (Integer.parseInt(args[1]) == 1) ? true : false;
    final long numberOfPrimes;
    final long startTime = System.nanoTime();
   if (number <= 1)
      numberOfPrimes = 0;
    else
      numberOfPrimes = countPrimes(number, isParallel);

    final long endTime = System.nanoTime();
    System.out.printf
      ("Number of primes under %d is %d\n", number, numberOfPrimes);
    System.out.println
      ("Time (seconds) taken is " + (endTime - startTime) / 1.0e9);
  }

  protected static long countPrimes
    (final long number, final boolean isParallel)
  {
    long count;

    if (isParallel)
      count = LongStream.rangeClosed(1L, number)
                        .parallel()
                        .filter(n -> isPrime(n)) // lambda expression
                        .count();
    else
      count = LongStream.rangeClosed(1L, number)
                        .filter(PrimesStream::isPrime) // method reference
                        .count();

    // 1 nije prim, ali 2 je prim (a za 2 isPrime daje false),
    // pa je ukupan broj točan.
    return (count);
  }

  private static boolean isPrime ... kao prije ...
}
```

Slijede rezultati usporedbe rješenja pomoću Java 5/6 Executora, Java 7 ForkJoina i Java 8 paralelnog Streamsa na računalu s mikroprocesorkim čipom i7-4702MQ na 2.2 GHz (korištena Java 1.8.0_111):

```
PrimesExecutor 10000000  1 1000  Time (seconds) taken is 8.69
PrimesExecutor 10000000  2 1000  Time (seconds) taken is 4.57
PrimesExecutor 10000000  4 1000  Time (seconds) taken is 2.51
PrimesExecutor 10000000  8 1000  Time (seconds) taken is 1.76
PrimesExecutor 10000000 16 1000  Time (seconds) taken is 1.72

PrimesForkJoin 10000000  1 100   Time (seconds) taken is 8.89
PrimesForkJoin 10000000  2 100   Time (seconds) taken is 4.64
PrimesForkJoin 10000000  4 100   Time (seconds) taken is 2.59
PrimesForkJoin 10000000  8 100   Time (seconds) taken is 1.85
PrimesForkJoin 10000000 16 100   Time (seconds) taken is 1.79

PrimesStream   10000000 0        Time (seconds) taken is 8.92
PrimesStream   10000000 1        Time (seconds) taken is 1.92

... Number of primes under 10000000 is 664579
```

6. Testiranje konkurentnih transakcija u Oracle bazi podataka pomoću Java paralelnog programiranja

6.1. Objašnjenje problema

Često se u bazi podataka izvode transakcije kod kojih integritet podataka ovisi i o ostalim transakcijama koje se istovremeno izvode. Rad tih (konkurentnih) transakcija treba testirati u višekorisničkom radu, ili simulirati višekorisnički rad. Za testiranje rada npr. sto konkurentnih transakcija, vjerojatno nećemo moći koristiti 100 testera (ljudi), već odgovarajuće alate za testiranje. Možemo koristiti specijalne alate za tu namjenu, ili možemo sami napraviti testove na relativno jednostavan način.

U ovom dijelu će se prikazati testiranje (složenog) poslovnog pravila u Oracle bazi, koji smo prikazali na CASE 16 (2004. godine): "Kako spriječiti 'začarani krug' (rješavanje određenog tipa poslovnih pravila u Oracle bazi podataka)".

U 6.2 prikazuju se Oracle specifične i ANSI standardne varijante SQL hijerarhijskih upita u Oracle bazu.

U 6.3. i 6.4. prikazat ćemo dio teksta iz navedenog rada s CASE 16.

U 6.3. prikazat ćemo rješenje koje sprečava pojavu petlje (u hijerarhijskoj strukturi podataka) u jednokorisničkom radu, ali to rješenje nije dobro za višekorisnički rad (vrlo lako se nađe primjer koji to pokazuje).

U 6.4. prikazuju se dva rješenja koja bi trebala biti dobra i za višekorisnički rad. Jedno koristi tzv. autonomne transakcije, a drugo (bolje, jer javlja manje lažnih grešaka) koristi (naš) trik – simulaciju "ROLLBACK TO SAVEPOINT" u okidaču baze podataka. Iako navedena rješenja intuitivno izgledaju pouzdano (u smislu da ne dozvoljavaju grešku, tj. pojavu petlje), nije dat formalni dokaz. Prema tome, jako je važno da ta rješenja dobro testiramo.

U 6.5. prikazat će se testiranje pomoću Java Executora.

6.2. Varijante SQL hijerarhijskih upita u Oracle bazi podataka

Oracle baza je dugo godina imala specifičnu varijantu sintakse za hijerarhijske upite, pomoću CONNECT BY klauzule. Slijede dva primjera – jedan kod kojeg naredba javlja grešku ako već postoji petlja u podacima, i drugi primjer, koji ne javlja grešku:

```
-- varijanta koja puca kod petlje
SELECT empno, ename, mgr, LEVEL
  FROM emp
 START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr;


-- varijanta koja NE puca kod petlje
SELECT empno, ename, mgr, LEVEL, CONNECT_BY_ISCYCLE iscycle
  FROM emp
 START WITH mgr IS NULL
CONNECT BY NOCYCLE PRIOR empno = mgr;
```

Oracle ANSI standard za hijerarhijski upit koristi tzv. **recursive common table expressions** (recursive CTE). Oracle baza podržava recursive CTE od Oracle verzije 11.2 (2009. godine) – od tada WITH klauzula (u SELECT naredbi) može biti rekurzivna.

```
-- varijanta koja puca kod petlje
WITH each_level (empno, ename, mgr, rlevel) AS
  (SELECT empno, ename, mgr, 1 rlevel -- kao START WITH
     FROM emp
    WHERE mgr IS NULL
   UNION ALL -- kao CONNECT BY
   SELECT emp.empno, emp.ename, emp.mgr, rlevel + 1
     FROM emp, each_level
    WHERE emp.mgr = each_level.empno
  )
SELECT * FROM each_level;


-- varijanta koja NE puca kod petlje
WITH each_level (empno, ename, mgr, rlevel) AS
  (SELECT empno, ename, mgr, 1 rlevel -- kao START WITH
     FROM emp
    WHERE mgr IS NULL
   UNION ALL -- kao CONNECT BY
   SELECT emp.empno, emp.ename, emp.mgr, rlevel + 1
     FROM emp, each_level
    WHERE emp.mgr = each_level.empno
  )
 CYCLE mgr SET iscycle TO 'y' DEFAULT 'n'
SELECT * FROM each_level; -- prikazuje i stupac iscycle
```

6.3. Sprečavanje pojave petlje u hijerarhijskoj strukturi podataka u jednokorisničkom radu

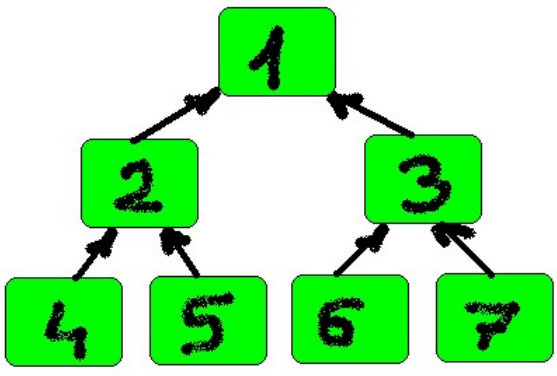
Često se želi zabraniti "začarani krug", tj. pojavu (zatvorene) petlje u jednostablastim ili višestablastim rekurzivnim strukturama podataka. Jednostablasta rekurzivna struktura je ona u kojoj točno jedan objekt nema "roditelja" (on je "vrh" stabla), a svi ostali imaju jednog "roditelja". Višestablasta rekurzivna struktura je sastavljena od više stabala, tj. barem dva objekta nemaju "roditelja", a svi ostali objekti imaju točno jednog "roditelja". Jedan primjer (jedno)stablaste strukture je dat poznatom Oracle tablicom djelatnika – **emp** (employes). Najjednostavniji opis tablice **emp** je:

```
CREATE TABLE emp (
   empno NUMBER (4) NOT NULL PRIMARY KEY,
   ename VARCHAR2 (20) NOT NULL,
   mgr   NUMBER (4) REFERENCES emp (empno)
)
/
```

Želi se da baza spriječi petlju u tablici **emp**, odnosno želi se onemogućiti da se dobiju podaci u kojima bi neki djelatnik bio menadžer drugom djelatniku, a drugi djelatnik bi (direktno ili indirektno) bio menadžer prvom djelatniku.

Prvo se tablicu **emp** puni sa 7 redaka. Djelatnik sa brojem 1 bit će "glavni šef", djelatnici sa brojevima 2 i 3 bit će "šefovi" (podređeni "glavnom šefu"), djelatnici 4 i 5, odnosno 6 i 7, bit će podređeni "šefu 2", odnosno "šefu 3" (slika 29.).

```
INSERT INTO emp (empno, ename, mgr) VALUES (1, 'EMP 1', NULL);
INSERT INTO emp (empno, ename, mgr) VALUES (2, 'EMP 2', 1);
INSERT INTO emp (empno, ename, mgr) VALUES (3, 'EMP 3', 1);
INSERT INTO emp (empno, ename, mgr) VALUES (4, 'EMP 4', 2);
INSERT INTO emp (empno, ename, mgr) VALUES (5, 'EMP 5', 2);
INSERT INTO emp (empno, ename, mgr) VALUES (6, 'EMP 6', 3);
INSERT INTO emp (empno, ename, mgr) VALUES (7, 'EMP 7', 3);
```



Slika 29. Grafički prikaz početnih podataka u tablici EMP

Ovaj zahtjev je, očito, relativno lako definirati, a dosta je čest u praksi. Međutim, nije ga lako realizirati (isključivo) u bazi podataka, tj. bez pomoći klijentske strane. Ovdje se prikazuje rješenje tog zahtjeva u Oracle bazi podataka, bez pomoći programa na klijentu ili aplikacijskom serveru (dakle, rješenje je u cijelosti na strani baze).

Rješenje u jednokorisničkom radu je relativno jednostavno. Zapravo, rješenje bi bilo vrlo jednostavno kad ne bi dolazilo do jednog problema - problema mutirajućih tablica (mutating tables). Mutirajuća tablica je ona tablica koja se trenutačno modificira pomoću naredbe INSERT, UPDATE ili DELETE, ili ona tablica koja bi trebala biti ažurirana zbog efekta DELETE CASCADE deklarativnog ograničenja. Oracle ne dozvoljava da se mutirajuće tablice čitaju (niti ažuriraju) u **row** okidačima baze (database triggers), tj. okidačima koji se okidaju za svaki redak tablice, jer bi se kao rezultat (čitanja) mogla dobiti neka neočekivana vrijednost. Pojednostavljeno rečeno, Oracle ne dozvoljava da se čita tablicu dok traje proces njene izmjene u istoj sesiji baze.

Međutim, za razliku od row okidača, čitanje se može raditi u **statement** okidačima, tj. okidačima koji se okidaju jedanput za svaku naredbu INSERT, UPDATE ili DELETE. Klasično rješenje problema mutirajućih tablica jeste da se u **row** okidaču zapamti (npr. u PL/SQL memorijsku tablicu) koji su redovi ažurirani, a onda se u **after statement** okidačima čita PL/SQL tablica i primjenjuje se provjera poslovnog pravila na retke koji su zapamćeni u PL/SQL tablici. Obično se želi da okidači sadrže što manje programskog koda, tako da okidači najčešće samo pozivaju pohranjene (a najčešće i pakirane) procedure ili funkcije.

Okidač **bus_emp** (**before update statement** nad tablicom **emp** - okida se jedanput prije naredbe UPDATE) poziva (pakiranu) proceduru za čišćenje PL/SQL:

```
CREATE OR REPLACE TRIGGER bus_emp
   BEFORE UPDATE ON emp
BEGIN
   emp_closed_loop.clear_plsql_tab;
END;
/
```

Okidač **bir_emp** (**before insert row** - okida se jedanput za svaki uneseni redak) provjerava da li su u stupcima **empno** i **mgr** različite vrijednosti (inače javlja grešku):

```
CREATE OR REPLACE TRIGGER bir_emp
   BEFORE INSERT ON emp
   FOR EACH ROW
BEGIN
   IF :NEW.empno = :NEW.mgr THEN
      RAISE_APPLICATION_ERROR
        (-20002, 'Djelatnik ne može biti nadređen samome sebi!');
   END IF;
END;
/
```

Okidač **bur_emp** (**before update row**) zabranjuje mijenjanje šifra djelatnika, zabranjuje da djelatnik bude nadređen samome sebi i poziva proceduru koja pamti redak u PL/SQL tablicu:

```
CREATE OR REPLACE TRIGGER bur_emp
   BEFORE UPDATE ON emp
   FOR EACH ROW
BEGIN
   IF :NEW.empno <> :OLD.empno THEN
      RAISE_APPLICATION_ERROR (-20001, 'EMPNO se ne može mijenati');
   END IF;

   IF :NEW.empno = :NEW.mgr THEN
      RAISE_APPLICATION_ERROR
         (-20002, 'Djelatnik ne može biti nadređen samome sebi!');
   END IF;


IF :NEW.mgr IS NOT NULL
      AND :NEW.mgr <> NVL (:OLD.mgr, 0)
   THEN
      emp_closed_loop.write_plsql_tab (
         p_empno => :OLD.empno,
         p_mgr   => :NEW.mgr);
   END IF;
END;
/
```

Okidač **aus_emp** (**after update statement** nad tablicom **emp** - okida se jedanput nakon naredbe UPDATE) poziva (pakiranu) proceduru za provjeru poslovnog pravila (ta je procedura krucijalni dio programskog koda):

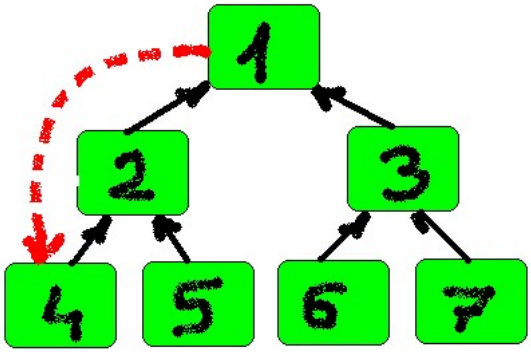
```
CREATE OR REPLACE TRIGGER aus_emp
   AFTER UPDATE ON emp
BEGIN
   emp_closed_loop.test;
END;
/
```

Slijedi paket "emp_closed_loop", sa tri (već navedene) procedure:

```
CREATE OR REPLACE PACKAGE emp_closed_loop IS
   PROCEDURE clear_plsql_tab;
   PROCEDURE write_plsql_tab (
      p_empno emp.empno%TYPE,
      p_mgr   emp.mgr%TYPE);
   PROCEDURE test;
END;
/


CREATE OR REPLACE PACKAGE BODY emp_closed_loop IS

   TYPE rec_t IS RECORD (
      empno emp.empno%TYPE,
      mgr   emp.mgr%TYPE);
   TYPE plsql_tab_t IS TABLE OF rec_t INDEX BY BINARY_INTEGER;
   m_plsql_tab plsql_tab_t;
   m_rows      BINARY_INTEGER;

   PROCEDURE clear_plsql_tab IS
   BEGIN
      m_rows := 0;
   END;


   PROCEDURE write_plsql_tab (
      p_empno emp.empno%TYPE,
      p_mgr   emp.mgr%TYPE)
   IS
   BEGIN
      m_rows := m_rows + 1;
      m_plsql_tab (m_rows).empno := p_empno;
      m_plsql_tab (m_rows).mgr   := p_mgr;
   END;
   PROCEDURE test IS
      l_mgr   emp.mgr%TYPE;
      l_empno emp.empno%TYPE;
   BEGIN
      FOR i IN 1..m_rows LOOP
         l_empno := m_plsql_tab (i).empno;
         l_mgr   := m_plsql_tab (i).mgr;
         WHILE l_mgr IS NOT NULL LOOP
            SELECT mgr INTO l_mgr
              FROM emp
             WHERE empno = l_mgr;
            IF l_mgr = l_empno THEN
               RAISE_APPLICATION_ERROR
                  (-20003, 'Greška - zatvorena petlja!');
            END IF;
         END LOOP;
      END LOOP;
   END;

END emp_closed_loop;
/
```

Vidljivo je da procedura **test** čita tablicu **emp**, pa tu proceduru nije moguće pozvati u row okidaču **bur_emp**, već se to može napraviti samo u statement okidaču **aus_emp**. Procedura **test** mogla se napisati i drugačije (konciznije), tako da se koristi klauzula CONNECT BY naredbe SELECT. Međutim, u nastavku će se dograđivati postojeća verzija bez klauzule CONNECT BY. Sada se može testirati rješenje. Ako se nad početnim podacima primijeni sljedeća UPDATE naredbu, dobija se poruka o grešci (i to je u redu):

```
UPDATE emp SET mgr = 4 WHERE empno = 1;
ERROR at line 1: ORA-20003: Greška - zatvorena petlja! …
```

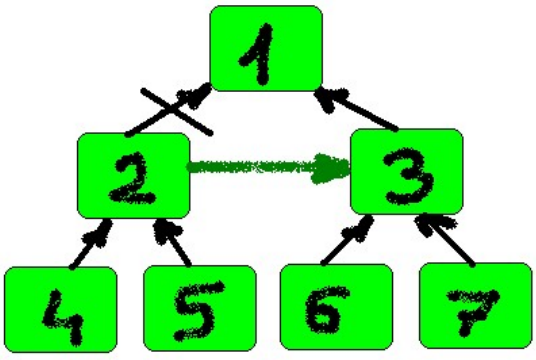


Slika 30. Rješenje radi dobro u jednokorisničkom radu – ne dopušta petlju

Nažalost, navedeno rješenje radi dobro samo u jednokorisničkom radu!

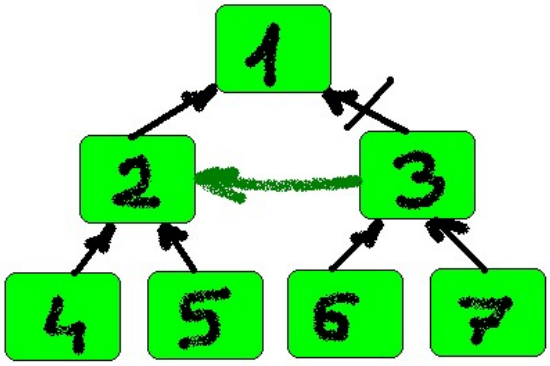
U višekorisničkome radu (ili, što je isto, u jednokorisničkom radu u kojem korisnik ima više sesija baze), može se desiti greška, kao u sljedećem primjeru:

```
-- 1. SESIJA
UPDATE emp SET mgr = 3 WHERE empno = 2;
```



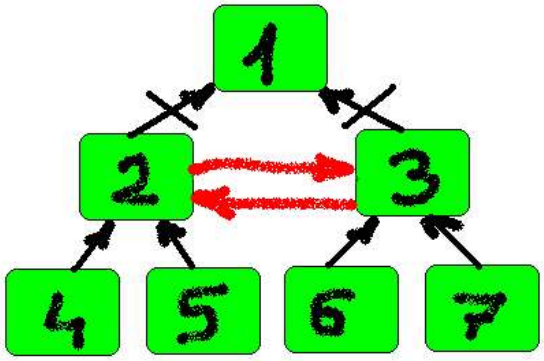
Slika 31. Naredba u prvoj sesiji je prošla - to JE u redu

```
-- 2. SESIJA
UPDATE emp SET mgr = 2 WHERE empno = 3;
```



Slika 32. Prošla je i naredba u drugoj sesiji – to NIJE u redu

Dakle, obje naredbe su prošle i dobila se zatvorena petlja!



Slika 33. Dobila se zatvorena petlja

6.4. Sprečavanje pojave petlje u hijerarhijskoj strukturi podataka u višekorisničkom radu

Glavna ideja za sprečavanje pojave petlje u višekorisničkom radu je da se, istovremeno dok se provjerava da li je došlo do petlje, gleda da li je tekući redak (tj. redak koji se trenutačno provjerava) zaključan. Ako je zaključan, može se pretpostaviti da bi moglo doći do zatvorene petlje, te javiti grešku. Međutim, kako provjeriti da li je redak (koji provjeravamo) zaključan?

Ako se za tu namjenu koristi SELECT FOR UPDATE, zaključat će se redak sve do kraja transakcije, zato što se u okidaču Oracle baze ne može koristiti naredbu ROLLBACK TO SAVEPOINT (napomena: ovo ograničenje, kao ni ograničenje vezano za mutirajuće tablice, nije mana Oracle baze, već prednost, jer ta ograničenja sprečavaju da dođe do programskih grešaka koje bi se vrlo teško mogle otkriti). No, ako redak ostane zaključan sve do kraja transakcije, to će spriječiti druge da rade sa takvim retkom, što je neprihvatljivo.

Budući da od verzije 8i Oracle baza podržava autonomnu transakciju, može se razmišljati da se ona primijeni za rješenje problema zaključavanja. Naime, u autonomnoj transakciji može se koristiti ROLLBACK (zapravo, autonomna transakcija i mora na kraju imati ROLLBACK ili COMMIT). U tijelo paketa **emp_closed_loop** dodat će se nova autonomna procedura **test_lock**:

```
PROCEDURE test_lock (p_mgr emp.mgr%TYPE) IS
   PRAGMA AUTONOMOUS_TRANSACTION;
   l_dummy NUMBER;
BEGIN
   SELECT 1 INTO l_dummy
     FROM emp
    WHERE empno = p_mgr FOR UPDATE NOWAIT;

   ROLLBACK;
EXCEPTION
   WHEN OTHERS THEN
      IF SQLCODE = -54 THEN
         RAISE_APPLICATION_ERROR
           (-20004, 'Greška - moguća zatvorena petlja!');
      ELSE
         RAISE;
      END IF;
END;
```

Nova procedura pozvat će se iz mijenjane procedure **test**:

```
PROCEDURE test IS
   l_mgr   emp.mgr%TYPE;
   l_empno emp.empno%TYPE;
BEGIN
```

```
   FOR i IN 1..m_rows LOOP
      l_empno := m_plsql_tab (i).empno;
      l_mgr   := m_plsql_tab (i).mgr;

      WHILE l_mgr IS NOT NULL LOOP
         test_lock (l_mgr);
         SELECT mgr INTO l_mgr
           FROM emp
          WHERE empno = l_mgr;
         IF l_mgr = l_empno THEN
            RAISE_APPLICATION_ERROR
               (-20003, 'Greška - zatvorena petlja!');
         END IF;
      END LOOP;
   END LOOP;
END;
```

Naredbe koje su uzrokovale grešku u prethodnoj točci sada neće uspjeti, jer će baza upozoriti da bi moglo doći do petlje ("moguća greška" a ne "sigurna greška", jer to što je redak zaključan ne znači da bi do greške sigurno došlo):

```
-- 1. SESIJA
UPDATE emp SET mgr = 3 WHERE empno = 2;


-- 2. SESIJA
UPDATE emp SET mgr = 2 WHERE empno = 3;
ERROR at line 1:
ORA-20004: Greška - moguća zatvorena petlja! …
```

Nažalost, rješenje sa autonomnom transakcijom može javiti dosta lažnih grešaka (false negative), zato što su autonomnoj transakciji (baš zato što je autonomna, tj. nezavisna od "glavne" transakcije) zaključani oni redovi koje je zaključala "glavna" transakcija. Zato autonomna procedura "zaključuje" da je došlo do zatvorene petlje i onda kad je očito da nije došlo do zatvorene petlje. Evo takvog slučaja, u kojem autonomna transakcija "pogrešno zaključuje":

```
-- dvije UPDATE naredbe u istoj sesiji
UPDATE emp SET mgr = 2 WHERE empno = 6;

UPDATE emp SET mgr = 6 WHERE empno = 7;
ERROR at line 1:
ORA-20004: Greška - moguća zatvorena petlja! …
```

Iako bi bilo sasvim u redu da djelatnik broj 6 postane (direktno) nadređen djelatniku 7, druga naredba UPDATE javlja grešku zato jer autonomna procedura **test_lock** nalazi da je djelatnik 6 zaključan (zaključala ga je "glavna" transakcija, kroz prvu naredbu

UPDATE).

Međutim, našli smo (2002. godine) da je u Oracle bazi moguće simulirati SAVEPOINT / ROLLBACK TO SAVEPOINT naredbe u okidaču baze, primjenom trik rješenja. Rješenje se temelji na sljedećem: ako se poziva udaljena procedura (pomoću database linka) i ako se u njoj desi neobrađena greška, njeni se efekti u cijelosti poništavaju (za razliku od lokalne procedure). Istina, udaljena procedura nije potrebna, ali zato se radi kvazi-udaljena procedura, koristeći "lokalni" database link (link baze na sebe samu):

```
CREATE DATABASE LINK local_db_link
   CONNECT TO scott IDENTIFIED BY tiger
   USING 'local_alias'; -- alias na lokalnu bazu
```

Ova simulacija se još u nečemu ponaša kao "pravi" ROLLBACK TO SAVEPOINT. Naime, ako druga transakcija pokuša zaključati redak koji je već zaključala prva transakcija, i ako prva transakcija otključa taj redak sa ROLLBACK TO SAVEPOINT, redak i dalje ostaje zaključan za drugu transakciju (međutim, neka treća transakcija bi sad mogla bez problema zaključati otključani redak).

Sada se može mijenjati paket **emp_closed_loop**. U odnosu na prethodnu verziju, paket sada ima proceduru **test_lock** navedenu (i) u specifikaciji, zato jer se procedura **test_lock** poziva iz procedure **test** kao udaljena procedura. Procedura **test_lock** koristi naredbu RAISE_APPLICATION_ERROR (-20999, ...) (koju procedura **test** ignorira, tj. ne smatra ju greškom), da bi otključala redak koji je prethodno zaključala (sa SELECT ... FOR UPDATE):

```
CREATE OR REPLACE PACKAGE emp_closed_loop IS
   PROCEDURE clear_plsql_tab;
   PROCEDURE write_plsql_tab (
      p_empno emp.empno%TYPE,
      p_mgr   emp.mgr%TYPE);
   PROCEDURE test;
   PROCEDURE test_lock (p_mgr emp.mgr%TYPE);
END emp_closed_loop;
/
```

```
CREATE OR REPLACE PACKAGE BODY emp_closed_loop IS
... kao prije, osim procedura TEST i TEST_LOCK ...

   PROCEDURE test IS
      l_mgr   emp.mgr%TYPE;
      l_empno emp.empno%TYPE;
   BEGIN
      FOR i IN 1..m_rows LOOP
         l_empno := m_plsql_tab (i).empno;
         l_mgr   := m_plsql_tab (i).mgr;

         WHILE l_mgr IS NOT NULL LOOP
            BEGIN
               emp_closed_loop.test_lock@local_db_link (l_mgr);
            EXCEPTION
               WHEN OTHERS THEN
                  IF SQLCODE = -20999 THEN NULL;
                  ELSE RAISE;
                  END IF;
            END;

            SELECT mgr INTO l_mgr
              FROM emp
             WHERE empno = l_mgr;

            IF l_mgr = l_empno THEN
               RAISE_APPLICATION_ERROR
                  (-20004, 'Greška - zatvorena petlja!');
            END IF;
         END LOOP;
      END LOOP;
   END;

   PROCEDURE test_lock (p_mgr emp.mgr%TYPE) IS
      l_dummy NUMBER;
   BEGIN
      SELECT 1 INTO l_dummy
        FROM emp
       WHERE empno = p_mgr FOR UPDATE NOWAIT;

      RAISE_APPLICATION_ERROR (-20999, 'Nije važno');
   EXCEPTION
      WHEN OTHERS THEN
      -- resource busy and acquire with NOWAIT specified
         IF SQLCODE = -54 THEN
            RAISE_APPLICATION_ERROR
               (-20004,'Greška - moguća zatvorena petlja!');
         ELSE
            RAISE;
         END IF;
   END;

END emp_closed_loop;
/
```

No, mora se reći da i ovo rješenje ponekad može javiti "lažnu uzbunu", tj. javiti da je (možda) došlo do petlje, iako do toga nije došlo, kao npr. u primjeru:

```
-- 1.SESIJA
UPDATE emp SET mgr = 6 WHERE empno = 2;

-- 2.SESIJA
UPDATE emp SET mgr = 5 WHERE empno = 7;
ERROR at line 1:
ORA-20004: Greška - moguća zatvorena petlja!…
```

Nažalost, ovakva "lažna uzbuna" ne može se spriječiti, jer sesija baze ne može točno "znati" što druge sesije baze rade!

6.5. Testiranje konkurentnih transakcija pomoću Java Executora

Za potrebe testiranja, tablicu EMP ćemo napuniti sa 1000 redaka, na ovaj način:

- 1 korijenski redak (šifra 0)
- njegovih 9 podređenih
- njihovih 90 podređenih (po 10 za svakog)
- njihovih 900 podređenih (po 10 za svakog).

```
-- 0
insert into emp (empno, ename, mgr) values (0, 'EMP 0', null);
-- 1-9
declare
   ename varchar2(20);
begin
   for j in 1..9 loop
      ename := 'EMP ' || j;
      insert into emp (empno, ename, mgr) values (j, ename, 0);
   end loop;
end;
/
```

```
-- 10-99
declare
   empno number(4);
   ename varchar2(20);
begin
   for i in 1..9 loop
      for j in 0..9 loop
         empno := i * 10 + j;
         ename := 'EMP ' || empno;
         insert into emp (empno, ename, mgr) values (empno, ename, i);
      end loop;
   end loop;
end;
/

-- 100-999
declare
   empno number(4);
   ename varchar2(20);
begin
   for i in 10..99 loop
      for j in 0..9 loop
         empno := i * 10 + j;
         ename := 'EMP ' || empno;
         insert into emp (empno, ename, mgr) values (empno, ename, i);
      end loop;
   end loop;
end;
/

commit;
```

Testiranje ćemo raditi pomoću Java programa (u nastavku) koji ima sljedeće ulazne parametre:

- brDretvi: broj Java dretvi (default je 10)
- cekanje: vrijeme namjernog čekanja u pojedinoj dretvi (default je 1 sekunda), kako bi se simuliralo kašnjenje kod rada korisnika
- brIteracija: broj ponavljanja testa (default je 1).

Glavna metoda **main** poziva metodu **testiraj**, u kojoj se kreira objekt (connectionTask) anonimne klase (podklasa od Runnable). U (nadjačanoj) metodi **run** dvaput se poziva metoda **slucajni_broj**, koja generira slučajne emp i mgr (brojeve između 200 i 300). Na temelju toga se radi UPDATE jednog retka i COMMIT. Kreira se fiskni broj executora sa **newFixedThreadPool(brDretvi)** i svakom se daje njegov zadatak sa **executorService.submit(connectionTask)**.

```
import java.sql.*;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import static java.util.concurrent.TimeUnit.MINUTES;

public class TestEmp {

static int brDretvi = 10;
static int cekanje = 1; // 1 sekunda
static int brIteracija = 1;

public static void main(String[] args)  {
   if (args.length > 0) {
      brDretvi = Integer.parseInt(args[0]);
   };

   if (args.length > 1) {
      cekanje = Integer.parseInt(args[1]) * 1000;
   };

   if (args.length > 2) {
      brIteracija = Integer.parseInt(args[2]);
   };

   for (int i = 1; i <= brIteracija; i++) {
      testiraj();
   };
}; // procedure main
```

```
private static void testiraj() {
   String url = "jdbc:oracle:thin:emp/emp@localhost:1521:ORCL";

   Runnable connectionTask = new Runnable() {

      public void run() {
         try (Connection con = DriverManager.getConnection(url);) {
           con.setAutoCommit(false);
           String query;
           PreparedStatement stm;

           // ORA-01436: CONNECT BY loop in user data
           query = "select distinct 0 from emp connect by prior empno = mgr";
           stm = con.prepareStatement(query);
           stm.executeQuery();

           // s donjim postavkama dolazi do zatvorene petlje
           // kod nesigurne verzije paketa
           int mgr   = slucajni_broj (200, 300);

           // nije pogodno: slucajni_broj (0, 999);
           int empno = slucajni_broj (200, 300);

           query = "update emp set mgr = ? where empno = ?";
           stm = con.prepareStatement(query);
           stm.setInt(1, mgr);
           stm.setInt(2, empno);
           stm.execute();
           Thread.sleep(cekanje);

           con.commit();
           System.out.println("Empno:" + empno + " Mgr:" + mgr);
         } catch (Exception e) {
           // System.out.println(e.toString().substring(34, 67));
           System.out.println(e.toString());
         }
       } // procedure run

   }; // new Runnable
```

```
   try {
      ExecutorService executorService =
         Executors.newFixedThreadPool(brDretvi);
      for (int j = 1; j <= brDretvi; j++) {
         executorService.submit(connectionTask);
      }
      executorService.shutdown();
      executorService.awaitTermination(30, MINUTES);
   } catch (Exception e) {
      System.out.println(e.toString());
   }

}; // procedure testiraj

private static int slucajni_broj (int min_p, int max_p) {
   /*
   Generira slučajni cijeli broj od min (uključujući) do max (uključujući).
   Math.random vraća vrijednost od 0 (uključujući) do 1 (isključujući).
   Zato se množi s max_p i zbraja min_p. Vraća se u cijeli broj.
   */
   int broj = min_p + (int) (Math.random() * (max_p - min_p + 1));
   return broj;
} // function slucajni_broj

} // class TestEmp
```

Testiranje možemo raditi npr. ovako – pokreće se 100 Java dretvi (time i 100 paralelnih transakcija), s vremenom čekanja od 2 sekunde u transakciji, u 5 iteracija testiranja:

```
java –cp ”*”; TestEmp 100 2 5
```

Pokazuje se da nesigurna varijanta paketa **emp_closed_loop** vrlo brzo dovodi do greške. Sigurne (dvije) varijante tog paketa ne dovode do greške niti nakon puno ponavljanja. Naravno, to nije matematički dokaz da su to zaista sigurna varijante. Kao i uvijek, testiranjem se može dokazati da program ne radi dobro, ali se testiranjem ne može dokazati da program (uvijek) radi dobro.

7. Zaključak

Često aplikacijski programeri gledaju na sustave za upravljanje bazama podataka (SUBP) kao na "crnu kutiju". Nemaju dovoljno vremena (ili motivacije) za dublje poznavanje s mogućnostima konkretnog SUBP-a, drže da su svi ono isti (ili vrlo slični), žele pisati generički kod (neovisan o SUBP-u) itd. Naravno, postoje slučajevi kada je pisanje generičkog koda (vrlo) opravdano, npr. kada je riječ o aplikacijama koje zaista moraju raditi na većem broju različitih SUBP-a. No često se potreba za neovisnošću od konkretnog SUBP-a postavlja kao zahtjev, iako se u stvarnosti radi samo s jednim sustavom. Nažalost, različiti SUBP sustavi prilično se razlikuju, naročito po upravljanju transakcijama, zaključavanju redaka i sl.

Zbog toga treba vrlo pažljivo raditi rješenja za podršku poslovnih pravila u konkretnom SUBP-u. U ovom radu prikazali smo jedno takvo (naše) rješenje, koje sprečava pojavu petlje (u hijerarhijskoj strukturi podataka) u višekorisničkom radu. To je rješenja pisano pomoću Oracle PL/SQL pohranjenih procedura. Ono je značajno kompleksnije od rješenja koje bi se koristilo u jednokorisničkom radu. Zbog toga bismo ga trebali jako dobro testirati. Naravno, još bi bolje bilo kada bismo mogli dati matematički dokaz da je to rješenje zaista korektno. Vjerujemo da školovanom matematičaru ne bi bilo naročito teško naći takav dokaz za naše rješenje (ako je ono zaista korektno). Nažalost, mi informatičari najčešće nemamo dovoljno matematičkog znanja za dokazivanje korektnosti programa (a i za matematičare su ti dokazi ponekad nepraktično glomazni). Zato se najčešće zadovoljavamo testiranjima, iako znamo da se testiranjem može dokazati da program ne radi dobro, ali se testiranjem ne može dokazati da program (uvijek) radi dobro.

Testiranje konkretnog rješenja (koje sprečava pojavu petlje u višekorisničkom radu) tražilo bi veliki broj testera (osoba) - nekoliko desetaka, koji bi testiranje trebali paralelno raditi duže vrijeme (nekoliko sati). Poželjno je (ako je moguće) zamijeniti toliki broj ljudi nekim (polu)automatiziranim rješenjem. U radu smo to napravili kroz relativno jednostavan program, koji se temelji na Java executorima (postoje još od Jave 5). Kroz taj Java program lako je pokrenuti npr. stotinjak (ili više) paralelnih Java dretvi, koje pokreću konkurentne sesije baze podataka.

Vjerojatno postoje i bolji načini testiranja konkurentnih transakcija, npr. pomoću gotovih komercijalnih alata namijenjenih tome. No nije loše napraviti vlastite, relativno jednostavne programe, koje lako možemo prilagoditi našim potrebama.

Literatura

1. BERNSTEIN, P.A. i NEWCOMER, E. (2009.) *Principles of transaction processing*. 2. izdanje. Elsevier / Morgan Kaufmann Publishers.

2. BUDIN, L., GOLUB, M., JAKOBOVIĆ, D. i JELENKOVIĆ, L. (2010.) *Operacijski sustavi*, Zagreb: Element.

3. HANSEN B.P. (1998.) *Java insecure Parallelism.* Dostupno na: http://brinch-hansen.net/papers/1999b.pdf (Pristupljeno: 1. studeni 2019.)

4. CLIFF C. i GOETZ B. (2009.) *Not Your Father's Von Neumann Machine - A Crash Course in Modern Hardware*. Dostupno na: https://www.infoq.com/presentations/click-crash-course-modern-hardware/ (Pristupljeno: 1. studeni 2019.)

5. DREPPER U. (2007.) *What Every Programmer Should Know About Memory.* Dostupno na: http://www.akkadia.org/drepper/cpumemory.pdf (Pristupljeno: 1. studeni 2019.)

6. DATE, C.J. (2004.): *An introduction to Database Systems.* 8. izdanje. Addison-Wesley.

7. ECKEL B. (2006) *Thinking in Java.* 4.izdanje. Prentice Hall.

8. GOETZ B. et al (2006.) *Java Concurrency in Practice.* Addison-Wesley.

9. HENNESSY, J.L. i PATTERSON D.A. (2007.) *Computer Architecture - A Quantitative Approach.* 4.izdanje. Elsevier / Morgan Kaufmann Publishers.

10. HERLIHY, M. i SHAVIT, N. (2008.) *The Art of Multiprocessor Programming*. Elsevier / Morgan Kaufmann Publishers.

11. HORSTMANN, C.S. i CORNELL, G. (2008.) *Core Java: Volume 1 – Fundamentals*. 8. izdanje. Prentice Hall / Sun Microsystems.

12. KYTE, T. (2009.) *Expert Oracle Database Architecture*. Apress.

13. MENNON, R., M. (2005.) *Expert Oracle JDBC Programming*. Apress.

14. MEYER, B. i NANZ S. (2010.) *Concepts of Concurrent Computation (ETH Zuerich - Chair of Software Engineering).* Dostupno na: http://se.inf.ethz.ch/old/teaching/2010-S/0268/index.html#lectures_slides (Pristupljeno: 1. studeni 2019.)

15. NANZ S. et al (2010.) *A Comparative Study of the Usability of Two Object-oriented Concurrent Programming Languages (ETH Zuerich & University of Toronto).* Dostupno na:http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.420.8217&rep=rep1&type=pdf (Pristupljeno: 1. studeni 2019.)

16. Oracle priručnik (2013.) *Oracle Database Development Guide 12c Release 1.*

17. Oracle priručnik (2013.) *Oracle Database JDBC Developer's Guide 12c Release 1.*

18. REINHOLD M. (2011.) *Divide and Conquer Parallelism with the Fork/Join Framework.* Dostupno na: http://www.oracle.com/us/technologies/java/fork-join-framework-428206.pdf (Pristupljeno: 1. studeni 2019.)

19. SHARAN, K. (2014.) *Beginning Java 8 Language Features - Lambda Expressions, Inner Classes, Threads, I/O, Collections and Streams.* Apress.

20. SHARAN, K. (2014.) *Java 8 APIs, Extensions and Libraries*. Apress.

21. SIERRA K. i BATES B. (2015.) *OCA/OCP Java SE 7 Programmer I & II Study Guide*. McGraw-Hill Education.

Popis slika

Popis tablica

Sažetak

U bazama podataka često rješavamo složena poslovna pravila. Naročito su kompleksna ona rješenja koja moraju dobro raditi u višekorisničkoj okolini. U ovom radu prikazali smo jedno takvo (naše) rješenje (pisano pomoću Oracle PL/SQL pohranjenih procedura), koje sprečava pojavu petlje (u hijerarhijskoj strukturi podataka) u višekorisničkoj okolini.

Za testiranje tog rješenja trebali bismo imati veliki broj testera (osoba), koji bi testiranje trebali paralelno raditi duže vrijeme (nekoliko sati). Zbog toga smo napravili Java paralelni program, koji se temelji na Java executorima (postoje još od Jave 5). Kroz taj Java program lako je pokrenuti npr. stotinjak (ili više) paralelnih Java dretvi, koje pokreću konkurentne sesije baze podataka.

Ključne riječi: *poslovna pravila u bazama podataka, transakcije u bazama podataka, Java paralelno programiranje, testiranje konkurentnih transakcija pomoću Java paralelnih programa*

Summary

In databases, we often handle complex business rules. Particularly complex are those solutions that need to work well in a multiuser environment. In this paper, we have presented one such (our) solution (written using Oracle PL / SQL stored procedures), which prevents the appearance of a loop (in a hierarchical data structure) in a multiuser environment.

To test this solution, we should have a large number of testers (persons), who should work in parallel for a long time (several hours). Because of this, we created a Java parallel program, based on Java executors (they have existed since Java 5). It is easy to run through this Java program e.g. one hundred (or more) parallel Java threads that run concurrent database sessions.

Keywords: *database business rules, database transactions, Java parallel programming, testing of concurrent transactions using Java parallel programs*