KONKURENTNO PROGRAMIRANJE – ORACLE BAZA, JAVA, EIFFEL

Zlatko Sirotić
Istra informatički inženjering d.o.o., Pula
e-mail: zlatko.sirotic@iii.hr

SAŽETAK

U radu se govori o osnovama konkurentnog programiranja. Konkurentno programiranje nije novost, vuče korijene još iz 60-tih godina prošlog stoljeća, kada se uglavnom primjenjivalo kod operacijskih sustava. Prvo se daje prikaz konkurentnosti u Oracle bazi - naime, PL/SQL i JVM u bazi izravno ne podržavaju konkurentno programiranje, ali konkurentnost se neminovno javlja na razini rada sa transakcijama. Zatim se ukratko prikazuje konkurentnost u operacijskim sustavima, hardverska podrška konkurentnosti, pa sinkronizacijski algoritmi i mehanizmi. Zatim se prikazuje konkurentno programiranje u Javi verzije 1 - 4. Posebno se navode nove mogućnosti u Java verzijama 5, 6, 7. Na kraju se daje drugačiji pristup konkurentnom programiranju u OOPL jeziku Eiffel - SCOOP (Simple Concurrent Object-Oriented Programming).

The paper discusses the basics of concurrent programming. Concurrent programming is not new, it has roots in 1960s when it was mainly applied in operating systems. First, concurrency in the Oracle database is described - although PL/SQL and the JVM in the database do not directly support concurrent programming, concurrency inevitably occurs at the level of transactions. Concurrency in the operating system, hardware support for concurrency, and synchronization algorithms and primitives are briefly discussed later. Concurrent programming in Java versions 1 - 4 is described after that. Specifically new features of Java 5, 6, 7 versions are outlined. Finally, the different approach of concurrent programming in Eiffel OOPL is presented - SCOOP (Simple Concurrent Object-Oriented Programming).

UVOD

Gotovo sva današnja računala imaju više CPU-a, manja računala u obliku višejezgrenih (engl. multi-core) procesora, a veća i snažnija računala obično sadrže više procesora (isto višejezgrenih). Prednost je takvih računala da mogu paralelno izvršavati dva ili više (u ovisnosti od broja CPU-a) nezavisnih programa, ali mogu paralelno izvršavati i dijelove istog programa. U potonjem slučaju, ako su dijelovi programa nezavisni jedan od drugoga, tada programeri i dalje mogu pisati kod kao da se on odvija u jednom dijelu.

No, najčešće su dijelovi programa međusobno zavisni, jer čitaju / pišu u isto memorijsko područje ili koriste neki drugi dijeljeni resurs, pa je moguće da dođe do tzv. *race condition,* gdje rezultat izračuna ovisi o redoslijedu izvršenja programskih instrukcija iz različitih dijelova programa. Zbog toga je potrebna sinkronizacija dijelova programa. Sinkronizacija traži posebne programske tehnike, koje svoje porijeklo imaju još u 60-tim godinama prošlog stoljeća. Te se programske tehnike obično zovu imenom *konkurentno programiranje*.

U radu se u 1. točki prikazuje konkurentnost u Oracle bazi. Iako to nije konkurentno programiranje kao u programskim jezicima npr. Java, C++, Eiffel, ipak postoje neki slični problemi konkurentnosti u Oracle bazi i tim jezicima, npr. problemi zaključavanja, deadlocka (potpuni zastoj; u nastavku ćemo koristiti engleski termin, zbog jasnoće) i sl. S druge strane, neka najnovija softverska ili/i hardverska rješenja konkurentnog programiranja nastala su po ugledu na baze podataka – tzv. transakcijske memorije.

U 2. točki prikazuje se konkurentnost na svom izvoru, tamo gdje je prvobitno i nastala – u operacijskim sustavima. U 3. točki daje se prikaz hardverske podrške za konkurentnost (naročito u dizajnu procesora). U 4. točki prikazuju se neki sinkronizacijski algoritmi i mehanizmi. U 5. točki prikazuju se neke "stare" tehnike konkurentnog programiranja u Javi verzije 1 do 4, a u 6. točki neke novije tehnike, koje je omogućila verzija 5, a u verzijama 6 i 7 su još poboljšane.

Nažalost, konkurentno programiranje teže je od uobičajenog, sekvencijalnog programiranja, ali sa razvojem višejezgrenih procesora konkurentno programiranje postaje nužnost u "svakodnevnom programiranju". Postoje pokušaji (vrijeme će pokazati da li su uspješni) da se konkurentno programiranje učini jednostavnijim. Jedan drugačiji pristup konkurentnom programiranju u odnosu na onaj u Javi (a slično Javi ima i C#, pa i C++), koji bi trebao olakšati konkurentno programiranje, prikazan je u 7. točki – Simple Concurrent Object Oriented Programming (SCOOP), model koji je realiziran u OOPL jeziku Eiffel.

1. KONKURENTNO PROGRAMIRANJE U ORACLE BAZI

Uz programiranje operacijskih sustava i programiranje superračunala, programiranje sustava za upravljanje bazama podataka (SUBP) je treće područje koje je i do sada značajno koristilo tehnike konkurentnog programiranja. No, tehnike konkurentnog programiranja prisutne su, iako u blažoj verziji (tj. na jednostavniji način), i kod korištenja SUBP-a, npr. kod programiranja u jeziku Oracle PL/SQL.

Poznato je da u Oracle bazi sesija (baze) nikada ne vidi promjene (tj. efekte DML naredbi) koje je napravila neka druga sesija, sve dok ta druga sesija ne napravi COMMIT. U Oracle bazi to je realizirano pomoću UNDO tablespacea, diskovne strukture u kojoj sesije čitaju prethodno stanje podataka, a ne tekuće stanje koje je zapisano u tablicama podataka, ali promjene još nisu COMMIT-irane.

No, sesija ipak ovisi o drugoj sesiji, jer su redci koje je ažurirala (unijela / mijenjala / brisala) druga sesija nedostupni (tj. zaključani) prvoj sesiji. Redci se uobičajeno ne mogu otključati do kraja tekuće transakcije u sesiji koja ih je zaključala, tj. dok ona ne daje COMMIT ili ROLLBACK. Redci se mogu otključati i sa ROLLBACK TO SAVEPOINT, ali treba podsjetiti da sesiji koja pokuša pristupiti zaključanim redcima prije nego druga sesija napravi ROLLBACK TO SAVEPOINT, ti redci i daje ostaju zaključani (do COMMIT ili ROLLBACK).

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

- 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;

- 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 redak za druge sesije ne postoji).

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).

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. Npr. pretpostavimo da imamo tri bankovna računa, svaki sa početnim iznosom od 1000 kuna. Transakcija T1 koja skida 100 kuna sa računa broj 1 i stavlja ih na račun broj 2, mogla bi izgledati ovako (zanemarimo što programski kod nije optimalan):

```
SELECT iznos INTO v_iznos_d FROM racuni WHERE broj = 1;
v_iznos_d := v_iznos_d - 100;
UPDATE racuni SET iznos = v_iznos_d WHERE broj = 1;

SELECT iznos INTO v_iznos_p FROM racuni WHERE broj = 2;
v_iznos_p := v_iznos_p + 100;
UPDATE racuni SET iznos = v_iznos_p WHERE broj = 2;
```

No, pretpostavimo da u isto vrijeme transakcija T2 skida 200 kuna sa računa broj 3 i stavlja ih isto na račun broj 2:

```
SELECT iznos INTO v_iznos_d FROM racuni WHERE broj = 3;
v_iznos_d := v_iznos_d - 200;
UPDATE racuni SET iznos = v_iznos_d WHERE broj = 3;

SELECT iznos INTO v_iznos_p FROM racuni WHERE broj = 2;
v_iznos_p := v_iznos_p + 200;
UPDATE racuni SET iznos = v_iznos_p WHERE broj = 2;
```

Može se desiti npr. sljedeći redoslijed radnji:

```
T1:
SELECT iznos INTO v_iznos_d FROM racuni WHERE broj = 1;
v_iznos_d := v_iznos_d - 100;
UPDATE racuni SET iznos = v_iznos_d WHERE broj = 1;

SELECT iznos INTO v_iznos_p FROM racuni WHERE broj = 2;
v_iznos_p := v_iznos_p + 100;
UPDATE racuni SET iznos = v_iznos_p WHERE broj = 2;

T2:
SELECT iznos INTO v_iznos_d FROM racuni WHERE broj = 3;
v_iznos_d := v_iznos_d - 200;
UPDATE racuni SET iznos = v_iznos_d WHERE broj = 3;

SELECT iznos INTO v_iznos_p FROM racuni WHERE broj = 2; -- čita staro stanje!
v_iznos_p := v_iznos_p + 200;
UPDATE racuni SET iznos = v_iznos_p WHERE broj = 2; -- redak je zaključan od T1

T1:
COMMIT; -- sada je T2 napravila UPDATE

T2:
COMMIT;
```

Kakvo je sada stanje računa. Stanje računa 1 i 3 (davatelji) je točno – račun 1 ima 900 kuna, račun 3 ima 800 kuna. Račun 2 (primatelj) trebao bi imati 1300 kuna, no ima 1200, jer je druga transakcija kod SELECT-a na računu broj 2 vidjela stanje od 1000 kuna.

Jedan od načina da se ovaj kod napiše ispravno, je – zaključavanje oba računa unaprijed, pomoću naredbi SELECT FOR UPDATE. Npr. transakcija T1 mogla bi izgledati ovako:

```
SELECT iznos INTO v_iznos_d FROM racuni WHERE broj = 1 FOR UPDATE;
SELECT iznos INTO v_iznos_p FROM racuni WHERE broj = 2 FOR UPDATE;
v_iznos_d := v_iznos_d - 100;
v_iznos_p := v_iznos_p + 100;
UPDATE racuni SET iznos = v_iznos_d WHERE broj = 1;
UPDATE racuni SET iznos = v_iznos_p WHERE broj = 2;
```

No, sada se može desiti deadlock - ako dvije transakcije pokušaju zaključati dva retka, ali u suprotnom redoslijedu. Npr. u slučaju da transakcija T1 uspješno zaključa račun broj 1, transakcija T2 uspješno zaključa račun 2, a obje žele zaključati i suprotni račun, desit će se 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.

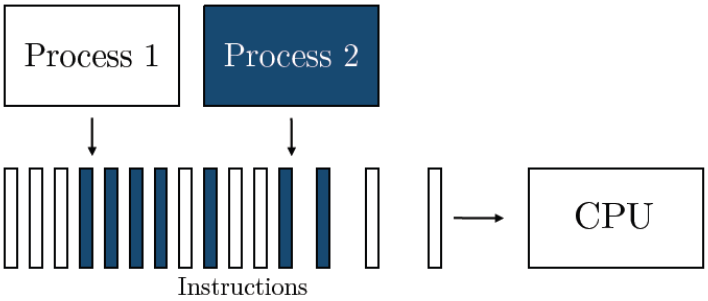
Za kraj, prikažimo kako se u Oracle bazi može napraviti paralelno izvršavanje dijelova programa pomoću jobova. Izvorno smo ovu metodu koristili kod distribuiranih baza, gdje smo na jednoj bazi ("centralna baza") pokretali jobove koji su pozivali udaljene procedure na drugim (udaljenim) bazama. Udaljene procedure paralelno obrađuju lokalne podatke, a nakon završetka svake procedure job javlja centralnoj bazi da je procedura završila. Kada centralna baza dobije odgovor od svih jobova, sekvencijalno (u jednoj transakciji) prikuplja rezultate udaljenih baza. Navedena metoda može se koristiti i na jednoj bazi, ali je korisna samo ako ta baza ima više raspoloživih (slobodnih) CPU-a. Poželjno bi bilo da je broj slobodnih CPU-a veći ili jednak broju jobova, tako da sve procedure mogu raditi paralelno. Ovo je osnovna logika te metode:

```
...FOR cur IN (SELECT baza FROM udaljene_baze) LOOP
      DBMS_ALERT.REGISTER (cur.baza);

      naredba_l :=
      'BEGIN ' ||
      '   udaljena_procedura@' || cur.baza || ';' ||
      '   DBMS_ALERT.SIGNAL (''' || cur.baza || ''', '' OK, '');' ||
      'EXCEPTION' ||
      '   WHEN OTHERS THEN' ||
      '      DECLARE' ||
      '         l_greska VARCHAR2 (20) := SUBSTR (SQLCODE, 1, 20);' ||
      '      BEGIN' ||
      '         ROLLBACK;' ||
      '         DBMS_ALERT.SIGNAL (''' || cur.baza || ''', l_greska);' ||
      '      END;' ||
      'END;';

      DBMS_JOB.SUBMIT (pid_l, naredba_l, SYSDATE, NULL, FALSE);
      br_jobova_l := br_jobova_l + 1;
   END LOOP;

   COMMIT; -- nužan za pokretanje JOB-ova

   /*
   Čeka na završetak svih JOB-ova ili istek vremena (1800 sekundi)
   za svaki prolaz u FOR petlji.
   */
   FOR i IN 1..br_jobova_l LOOP
      DBMS_ALERT.WAITANY (alert_l, poruka_l, status_l, 1800);

      IF status_l = 1 THEN -- isteklo je vrijeme, a alert se nije okinuo
         RAISE jobovima_isteklo_vrijeme;
      END IF;

      IF poruka_l <> ' OK, ' THEN -- JOB je puknuo
         poruka_za_gresku_l := alert_l || ' - ' || poruka_l;
         RAISE puknuo_job;
      END IF;

      poruka_za_gresku_l := poruka_za_gresku_l || alert_l || poruka_l;
   END LOOP;

   DBMS_ALERT.REMOVEALL;

   /*
   INSERT sa svih udaljenih baza
   - serijski (ali radi brzo), kako bi bila jedna transakcija
   */
   FOR cur IN (SELECT baza FROM udaljene_baze)
   LOOP ...
   END LOOP;
   COMMIT;
```

2. KONKURENTNO PROGRAMIRANJE I OPERACIJSKI SUSTAVI

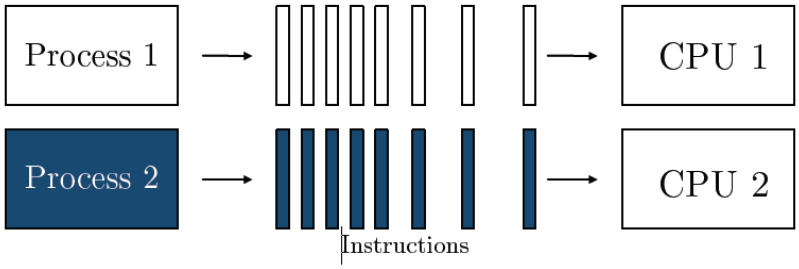
Ideja o konkurentnom računanju (concurrent computation) razvila se baš na području operacijskih sustava. Prvi operacijski sustavi, napravljeni 50-tih godina prošlog stoljeća, bili su sekvencijalni, tj. računalni programi izvršavali su se na njima slijedno, jedan iza drugoga. Vrlo brzo se uvidjelo da 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 2.1 vidi se prikaz višezadaćnosti:



Slika 2.1: Višezadaćnost (multitasking): instrukcije se međusobno isprepliću; Izvor: [14]

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 2.2:



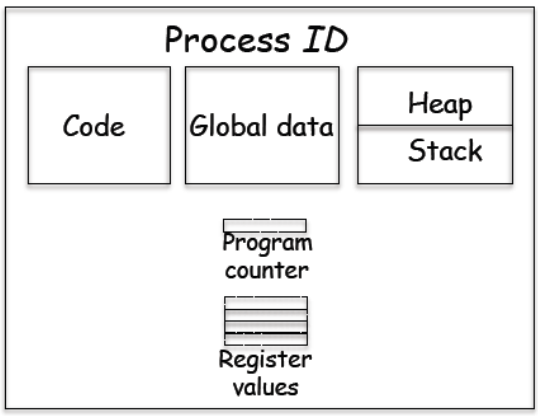
Slika 2.2: Multiprocesiranje (multiprocessing): instrukcije se izvršavaju paralelno; Izvor: [14]

I multiprocesiranje i višezadaćnost su primjeri konkurentnog izvršavanja. 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.

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 [14]:

- 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 2.3 prikazuje najvažnije elemente procesa operacijskog sustava:

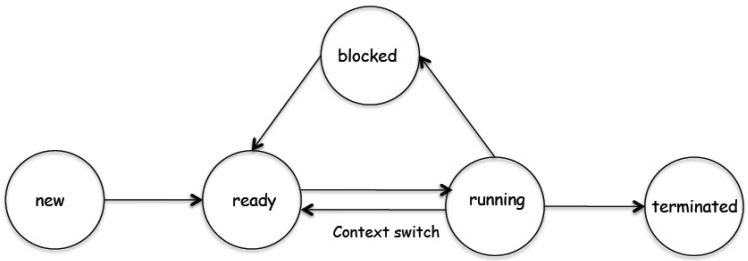


Slika 2.3: Najvažniji elementi procesa operacijskog sustava; Izvor: [13]

Operacijski sustav koristi poseban program - *raspoređivač* (*scheduler*), koji određuje kojem procesu treba dodijeliti (neki) procesor. Procesi mogu biti u ova tri generalna stanja [14]:

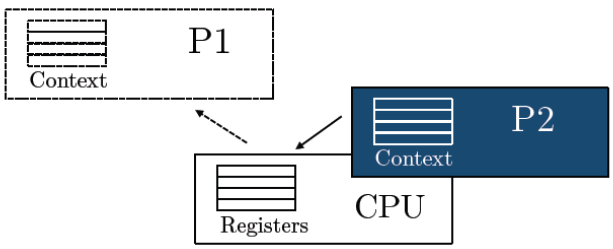
- Pokrenut (running): instrukcije procesa se izvršavaju na (nekom) procesoru;
- Spreman (ready): proces je spreman za izvršavanje, ali trenutačno mu nije dodijeljen procesor;
- Blokiran (blocked): proces čeka da se desi neki događaj (npr. da se završi čitanje podataka sa diska ili iz memorije, koji su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

Slika 2.4 prikazuje tranziciju stanja procesa operacijskog sustava (dodana su i terminalna stanja *new* i *terminated*):



Slika 2.4: Tranzicija stanja procesa operacijskog sustava; Izvor: [13]

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 2.3:



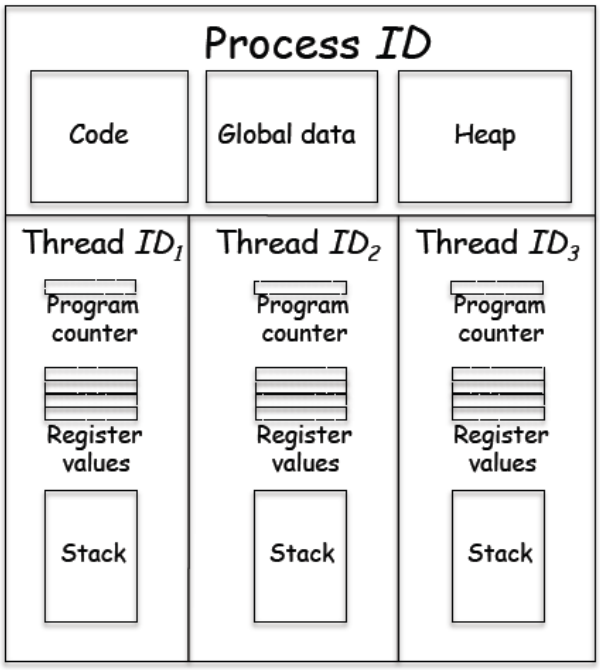
Slika 2.5: Zamjena konteksta (context switch): proces P1 se odstranjuje iz procesora i zamjenjuje ga proces P2; Izvor: [14]

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 2.6:



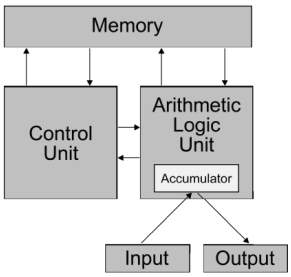
Slika 2.6: Najvažniji elementi dretvi operacijskog sustava (prikazane su tri dretve koje pripadaju istom procesu); Izvor: [13]

Zamjena konteksta dretvi u pravilu se izvodi nekoliko puta brže od zamjene konteksta procesa. U [7] se navodi da je zamjena konteksta dretve puno efikasnija od zamjene konteksta procesa, koja tipično traje stotine do tisuće ciklusa procesa. Zbog toga svi današnji moderni operacijski sustavi nisu samo višeprocesni, nego i višedretveni. No i tu se očekuju poboljšanja, jer kako je navedeno u [3] "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.

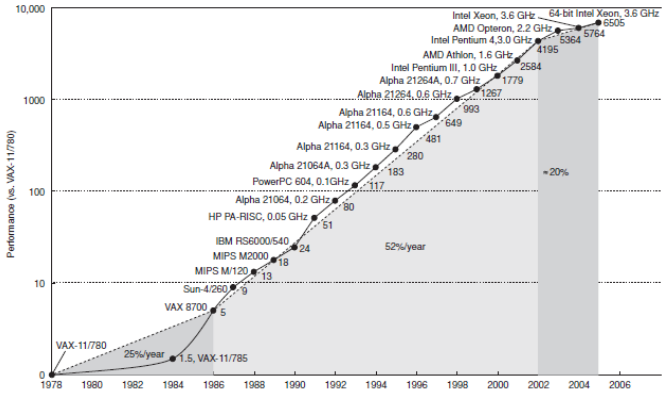
3. UTJECAJ RAZVOJA MIKROPROCESORA NA KONKURENTNO PROGRAMIRANJE

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 3.1) u današnje vrijeme samo korisna apstrakcija, koja u mnogim detaljima odudara od stvarnih današnjih računala:



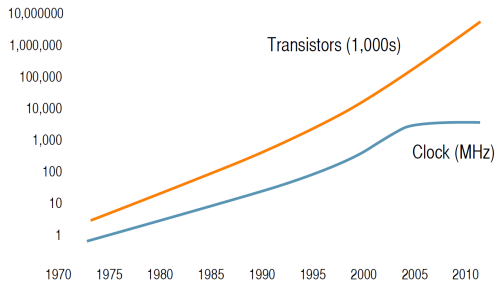
Slika 3.1: Von Neumannov model računala; Izvor: [4]

U knjizi [7], autori daju grafikon (slika 3.2) 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:



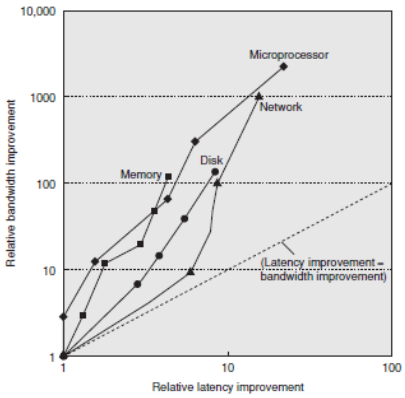
Slika 3.2: Rast performansi procesora od 1978. do 2005.; Izvor: [7]

No, to ne znači da je prestao vrijediti *Mooreov zakon*, jer broj tranzistora po procesoru i dalje eksponencijalno raste, kako pokazuje slika 3.3. 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 3.3: Povećanje broja tranzistora i radnog takta procesora od 1973. do 2010.; Izvor: [15]

U [7] se daje zanimljiv grafikon (slika 3.4) koji pokazuje kako se *širina pojasa* (*bandwidth*) ili *propusnost* glavnih elemenata računala (procesora, memorije, diska, mreže) povećavala daleko brže od brzine pristupa (latency).



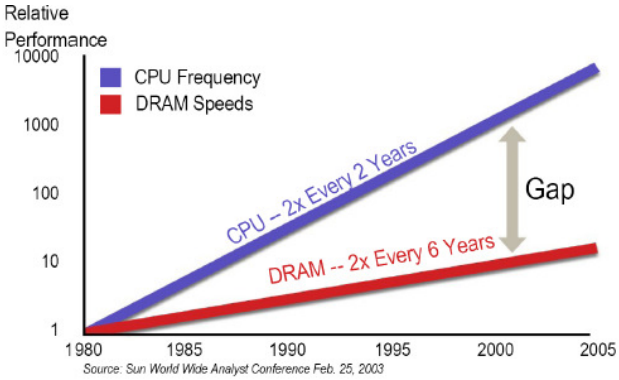
Slika 3.4: Log-log dijagram kretanja odnosa propusnosti i brzine pristupa kod glavnih elemenata računala; Izvor: [7]

Tablica 3.1 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 3.1: Razlike u veličini, brzini pristupa i drugim karakteristikama različitih vrsta memorije; Izvor: [7]

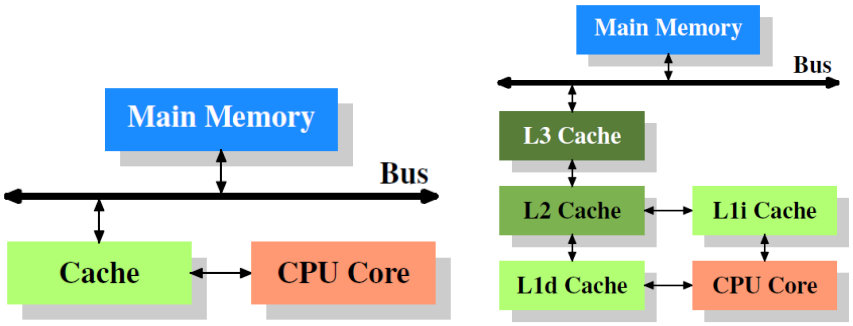
Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 512 GB	> 1 TB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25–0.5	0.5–25	50–250	5,000,000
Bandwidth (MB/sec)	50,000–500,000	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 3.5 pokazuje kako se eksponencijalno povećava jaz između performansi CPU-a i performansi glavne memorije (DRAM):



Slika 3.5: Povećanje jaza između performansi CPU-a i performansi DRAM-a; Izvor: [4]

Naravno, nije nemoguće 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 3.6 prikazuje dva primjera korištenja priručne memorije, jedan jednostavniji i jedan složeniji (oba primjera prikazuju jednojezgreni procesor):

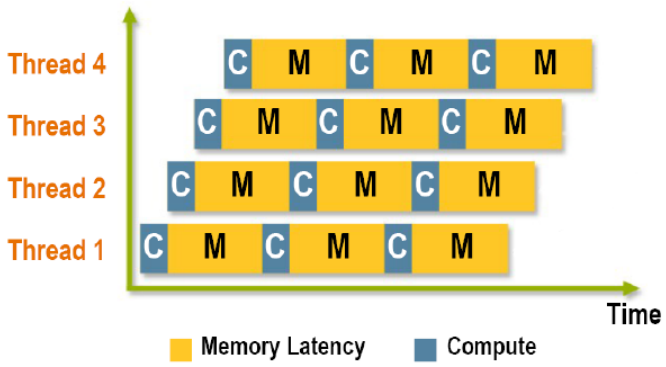


Slika 3.6: Dva primjera korištenja priručne memorije; lijevo je jednostavniji sustav sa jednom razinom priručne memorije; desno je složeniji sustav sa tri razine priručne memorije, a prva razina ima odvojenu memoriju za podatke (L1d) i za instrukcije programa (L1i); Izvor: [5]

Osim uvođenja priručne memorije u mikroprocesorske čipove, dizajneri su od 1985. nalazili i druga rješenja 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 istovremeno moguće je teoretski svesti prosječno vrijeme obrade instrukcije na samo jedan takt. 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.

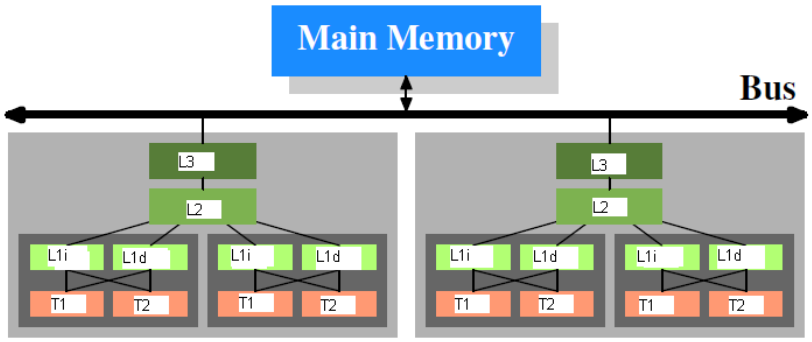
Dosjetili su se da bi ILP mogli koristiti i na drugi način. Umjesto da paralelno obrađuju (u različitim fazama) instrukcije istog programa, mogli bi paralelno obrađivati instrukcije različitih programa, tj. instrukcije različitih dretvi (threads). Treba naglasiti da procesorske dretve (ili *hardverske dretve*) mogu odgovarati i dretvama i procesima operacijskog sustava. Takvo obrađivanje instrukcija naziva se *multithreading* (Intel koristi ime *Hyper-Threading Technology*), a prikazuje ga slika 3.7:



Slika 3.7: Multithreading kod procesora; Izvor: [4]

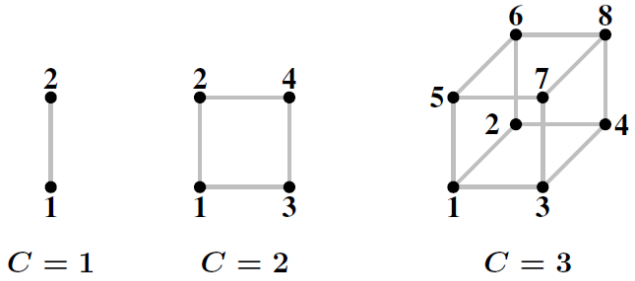
Multithreading (kao niti ILP) ne može činiti čuda. Naime, za razliku od višejezgrenih procesora, kod multithreadinga su duplirani samo neki elementi CPU-a. Npr. kod Intelove implementacije duplirani su samo procesorski registri. Dretve dijele i istu priručnu memoriju. Zbog toga procesor koji podržava 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 3.8. 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 3.8: Sustav sa dva dvojezgrena procesora (svaka jezgra podržava dvije dretve); Izvor: [5]

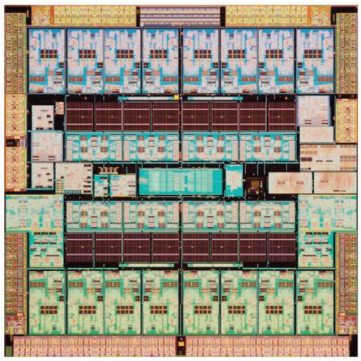
Osim arhitekture centralne djeljive memorije, postoji i arhitektura *distribuirane djeljive memorije*. Iako je i takva memorija djeljiva između svih procesora (pa neki kažu da je i to simetrična arhitektura, SMP), određeni dijelovi memorije su bliži određenim procesorima. Takva se arhitektura uobičajeno 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 pokazuje slika 3.9. 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, C = 1), a najudaljeniji mu je procesor 8, pa je pristup njegovoj memoriji najsporiji (C = 3).



Slika 3.9: Povezivanje procesora (i pripadajućih dijelova glavne memorije) u hiperkocke ; Izvor: [5]

Danas na tržištu postoji nekoliko procesora koji su višejezgreni, podržavaju višedretvenost i mogu se međusobno povezivati. Npr. Oracle (prije Sun) SPARC T3 procesor, poznat i kao Niagara 3 (prikazan na slici 3.10) ima 16 jezgri, svaka jezgra podržava 8 dretvi, a moguće je spojiti 4 takva procesora, čime se

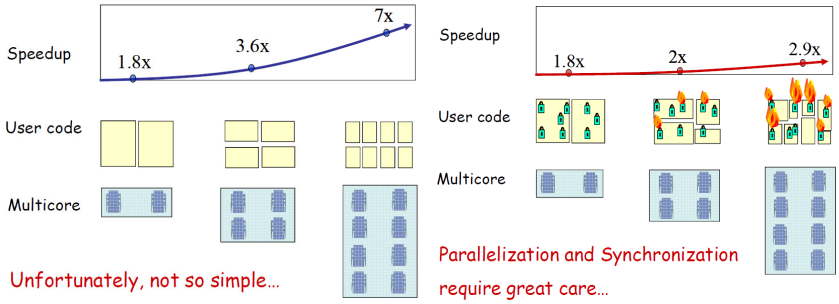
dobije 16 * 8 * 4 = 512 dretvi.



Slika 3.10: Oracle SPARC T3 procesor ima 16 jezgri, a svaka podržava 8 dretvi; Izvor: [15]

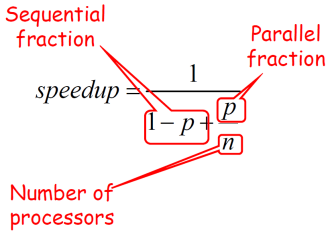
No, trenutačni lider na području standardnih (nespecijaliziranih) procesora je Vega 3 firme Azul Systems, koji je prvenstveno ciljan za tržište Java servera. Procesor ima 54 jezgre, a moguće je spojiti 16 procesora (koristi se UMA arhitektura), čime se dobije sustav sa 864 jezgre. Ono što čini ovaj procesor zanimljivim je i podrška za tzv. *hardversku transakcijsku memoriju* (*Hardware Transactional Memory*). Osim Vega procesora (verzije 1, 2 i 3), Azul Systems je prilagodio Sun JVM koji podržava heap veličine i preko 500 GB, sa maksimalnim pauzama za *GC* (*Garbage Collector*) od samo 10-20 ms, a to zahvaljujući i podrški procesora Vega za GC [3].

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 3.10, a ne onaj prikazan na lijevoj strani:



Slika 3.11 Povećanje broja jezgri obično ne rezultira (skoro) linearnim povećanjem performansi (lijeva strana slike), već puno manjim od toga (desna strana slike); Izvor: [13]

Razlog za to objašnjava *Amdahlov zakon* [13]. 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:



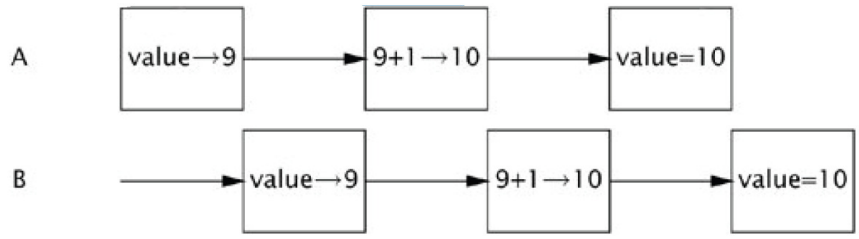
Npr. ako imamo 10 procesora i ako je moguće paralelizirati 90% programa, onda je maksimalno povećanje brzine 5,26 puta, što je skoro dva puta manje od broja procesora. Ako je moguće paralelizirati 99% programa, onda je povećanje 9,17 puta.

4. SINKRONIZACIJSKI ALGORITMI I MEHANIZMI

Pretpostavimo da smo napisali sljedeći programski kod [10] 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, 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 4.1: Poziv metode getNext() u dretvama A i B dao je (pogrešno) istu sekvencu; Izvor: [10]

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 [10]. *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 najgora varijanta je 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 [8] 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 [8] 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 [10] 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 [8], 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 [8], lokote bi trebalo implementirati sa sljedećim *atomarnim osnovnim operacijama* (*atomic primitives*) za sinkronizaciju, koje su kompleksnije 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 ekivalentnu. 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 [8]. Ne ulazeći u detalje, tj. matematičke definicije i dokaze navedene u [8], mogu se navesti neki zanimljivi rezultati. Konsenzus objekt je objekt čija klasa ima samo 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 (konsenzus objekt) može implementirati problem konsenzusa.

Zanimljivi su sljedeći rezultati (teoremi) iz [8]:

- *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 [8], 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 riječima Mauricea Sendaka: "***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 [13]. 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 [13], 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 [13] 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 [13] 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 odnosti na M1 i M2, ili samo M2?

Kako se navodi u [10], 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 [8] 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 [10] 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 ciklusa, 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 [8] autori navode i mane CAS operacije: algoritme koji koriste CAS (ili ekvivalentne operacije) vrlo je teško smisliti i često su vrlo neintuitivni. Zapravo, 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.

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 [8] 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 [8] (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. Java za sada ne podržava STM niti na razini jezika, niti na razini biblioteka.

U [8] 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 barem dva mikroprocesora koja podržavaju HTM - već spomenuti Vega procesor firme Azul Systems, a nedavno (kolovoz 2011.) mu se pridružio i BlueGene/Q firme IBM. Otkazan je razvoj procesora Oracle (prije Sun) Rock, koji je isto trebao podržavati HTM. 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), 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.

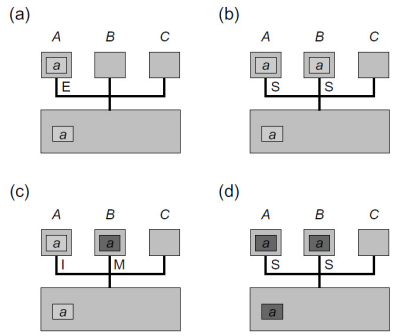
MESI protokol detektira sinkronizacijske konflikte između individualnih čitanja i pisanja, te osigurava da se procesori usklade oko stanja (djeljive) memorije. Kada procesor čita ili piše u memoriju, emitira (broadcasts) zahtjev na sabirnicu (bus), a ostali procesori to slušaju (ponekad se to zove *snooping*). 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*, jednostavno se zanemari; ako je *Modified*, mora se upisati u memoriju.

Slika 4.2 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 4.2: MESI protokol; Izvor: [8]

Kako navode autori u [8], za podršku HTM-a treba zadržati MESI protokol kakav jeste, samo treba dodati jedan transakcijski bit za 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.

5. KONKURENTNO PROGRAMIRANJE U JAVI VERZIJE 1 - 4

Kako navode autori u [10], iako mi sami možda nećemo kreirati dretve u našim Java programima, ipak ne možemo izbjeći višedretveni rad. Naime, svaka Java aplikacija koristi dretve. Kada se starta JVM, on kreira posebne dretve, npr. za GC (garbage collection), uz *main* dretvu. Kada koristimo Java AWT ili Swing framework, oni kreiraju posebnu dretvu za upravljanje korisničkim sučeljem. Kada koristimo servlete ili RMI, oni kreiraju pričuvu (pool) dretvi. Zato, kada koristimo te frameworke, moramo do neke mjere biti upoznati sa konkurentnošću u Javi. Naime, svaki takav framework uvodi u našu aplikaciju konkurentnost na implicitan način, te moramo znati napraviti da mješavina našeg koda i frameworkovog koda bude sigurna u višedretvenom radu.

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-poslužitelj 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 ponašanja, tj. klasa (a ne samo sučelja). A, "Višestruko nasljeđivanje klasa je korisno, koliko god mi šutjeli o tome :)" (odnosno - koliko god nas pokušavali uvjeriti u suprotno; uzgred, izgleda da se neki oblik višestrukog nasljeđivanja ponašanja priprema u Javi 8). 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()*. Primjer [13]:

```
public class RunThread implements Runnable {
  String id;
  public RunThread(String id) {
    this.id = id;
  }
  public void run() {
    // do something when executed
    System.out.println("This is thread " + id);
  }
}
```

Onda se objekt te "pomoćne" klase šalje konstruktoru koji kreira objekt klase *Thread* (tj. dretvu) u kodu naše "glavne" klase:

```
Thread mt = new Thread(new RunThread("mt"));
mt.start(); ...
```

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 je 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*, koji je prikazan u 4. točki, 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. Kondicijskim varijablama se pristupa pomoću naredbi *wait / notifyAll / notify*, što će biti prikazano u nastavku. Međutim, Java objekt se razlikuje od monitora [9] 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: "Zaprepašćuje me da je ovaj nesigurni Java paralelizam shvaćen tako ozbiljno od programske zajednice, četvrt stoljeća nakon invencije monitora i programskog jezika Concurrent Pascal." [Java's Insecure Parallelism, ACM SIGPLAN Notices 34:38–45, April 1999.]

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.

Jedan primjer problema koji traži sinkronizaciju na temelju uvjeta je tzv. *problem proizvođač-potrošač* (*producer-consumer problem*), koji je,u različitim varijantama, čest u praksi. Može se apstraktno opisati na ovaj način [14]:

- *Proizvođač*: u svakoj iteraciji (beskonačne) petlje, proizvodi podatke koje potrošač konzumira;
- *Potrošač*: u svakoj iteraciji (beskonačne) petlje, troši podatke koje je proizveo proizvođač.

Proizvođači i potrošači komuniciraju preko djeljivog međuspremnika (buffer) koji implementira red (queue), za koji smatramo (u ovom primjeru) da je neograničen, pa će nam biti važno samo da li je prazan. U primjeru [14] se prikazuje samo dio klase, a ne kompletan programski kod. Prvo pretpostavimo da imamo klasu *Buffer* (koja implementira neograničeni red) i da imamo jedan objekt te klase:

```
Buffer buffer = new Buffer();
```

Proizvođači dodaju podatke na kraj reda, koristeći metodu (reda) *void put(int item)*, a potrošači skidaju podatak sa reda koristeći metodu *int get()*. Broj podataka koji se nalaze u redu može se dobiti pomoću metode *int size()*. Pretpostavimo sada da u klasi *Consumer* imamo sljedeću metodu:

```
public void consume() {
  int value;
  synchronized(buffer) {
    value = buffer.get(); // incorrect: buffer could be empty
  }
}
```

Metoda nije dobra, jer ne provjerava da li je red (tj. međuspremnik) prazan, što može prouzročiti grešku kod izvođenja (runtime error). Dretva treba čekati dok red bude neprazan i tek tada pročitati podatak iz njega. Čekanje se u Javi može napraviti pomoću metode *wait()*, koja se može pozvati samo nad objektom koji je prethodno zaključan, tj. samo unutar *synchronized* bloka. *Wait()* tada blokira tekuću dretvu i otpušta lokot koji je dretva držala. Slijedi primjer [14]:

```
public void consume() throws InterruptedException {
  int value;
  synchronized(buffer) {
    while (buffer.size() == 0) {
      buffer.wait();
    }
    value = buffer.get();
  }
}
```

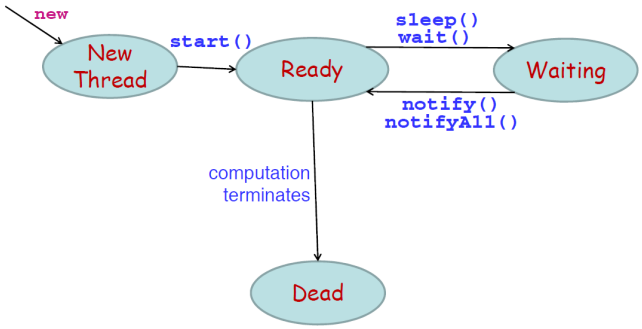
Sada lokot može preuzeti (zaključati) neka druga dretva, koja može mijenjati stanje navedene kondicije. Da obavijesti prvu dretvu o promjeni kondicije, druga dretva poziva *notify(),* što odblokira prvu dretvu, koja čeka na taj lokot, ali druga dretva ne otključa lokot odmah, već tek na kraju svog *synchronized bloka* (unutar kojega je i pozvala *notify()*). U 4. točki smo naveli da kod monitora dva osnovna tzv. *pravila signalizacije* (*signaling disciplines*) mogu biti *Signal and Wait* ili *Signal and Continue* (Java koristi samo ovo drugo). Primjer u kojem proizvođač obavještava potrošača o promjeni kondicije;

```
public void produce() {
  int value = random.produceValue();
  synchronized(buffer) {
    buffer.put(value);
    buffer.notify();
  }
}
```

Proizvođač je proizveo neki slučajni broj, zaključao red, spremio podatak u njega i pozvao *notify()*, čime je odblokirao jednu (bilo koju!) dretvu koja je čekala na taj lokot. Na kraju *synchronized bloka* (što je u ovom slučaju bilo odmah iza) je i otključao taj lokot. No, odblokirana dretva ne može biti sigurna da je taj uvjet valjan i kada ona dođe na red, jer je u međuvremenu neka treća dretva mogla potrošiti podatak i red je možda opet prazan. Stoga je dretva potrošač morala ispitivati uvjet u *while* petlji. Važno je i to da *notify()* uvijek odblokira samo jednu (bilo koju!) dretvu koja čeka na određeni lokot. Stoga je u praksi puno sigurnije pozivati *notifyAll(),* koja odblokira sve dretve koje čekaju na određeni lokot.

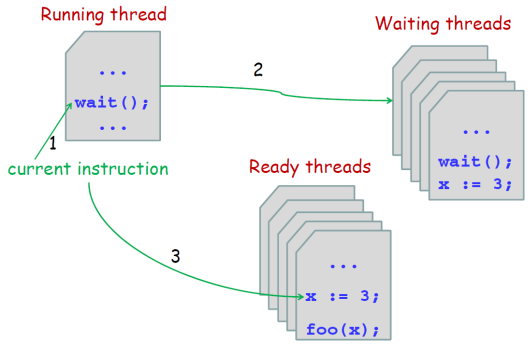
Metode *wait()*, *notify()* / *notifyAll()* rade interno sa tzv. *unutarnjim redovima kondicija* (*intrinsic condition queues*). Vidjet ćemo da u Javi 5 postoji njihova generalizacija, koja omogućava da programeri eksplicitno rade sa kondicijama.

Na slici 5.1 prikazan je dijagram promjene stanja Java dretvi. Vidi se (i) da dretva prelazi iz stanja *Ready* (češće se to stanje naziva *Runnable*) u stanje *Waiting* nakon što pozove *wait()*, a obrnuti prijelaz se zbiva kada **druga dretva** (što se iz slike ne vidi) pozove *notify() / notifyAll()*:



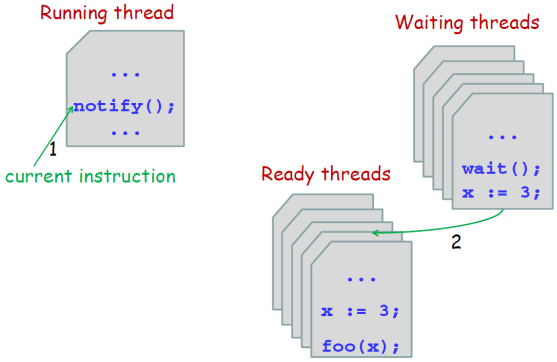
Slika 5.1: Dijagram promjene stanja Java dretvi; Izvor: [13]

Preciznije se zbivanje može pratiti kroz sljedeće dvije slike. Na slici 5.2 prikazana su tri koraka kod prijelaza dretve iz stanja *Running* (u kojem se dretva izvršava u procesoru) u stanje *Waiting*. Prvo dretva poziva metodu *wait()* (1), naravno, uvijek unutar *synchronized bloka*, čime otključa lokot i prelazi u stanje *Waiting* (2). Sada neka dretva koja je *Ready* može preći u stanje *Running*, jer se procesor oslobodio.



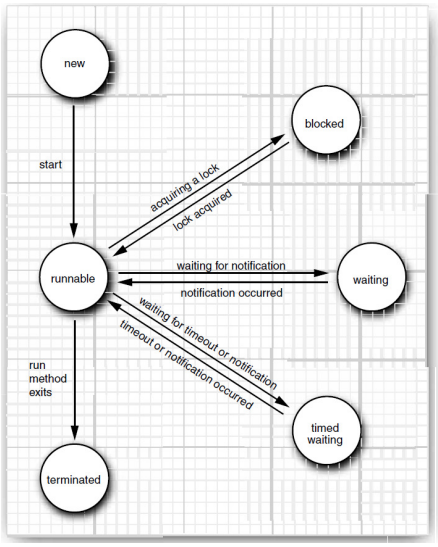
Slika 5.2: Promjena stanja nakon wait() operacije; Izvor: [13]

Slika 5.3 prikazuje nastavak zbivanja - poziva se *notify()* ili *notifyAll()*. Nakon što neka druga dretva pozove *notify()* ili *notifyAll()* (1) (naravno, uvijek unutar *synchronized bloka*), jedna dretva (ako je *notify()*) ili sve dretve (ako je *notifyAll()*) koje čekaju na lokot koji ima ta druga dretva, prelazi (2) iz stanja *Waiting* u Ready. Naravno, rekli smo da dretva koja poziva *notify()* / *notifyAll()* ne otključava lokot odmah, već tek kod izlaska iz *synchronized bloka* (to je signalizacijsko pravilo *Signal and Continue*). Mora se primijetiti da ova slika ne prikazuje metodu *notify()* u svim njenim detaljima.



Slika 5.3: Promjena stanja nakon notify() operacije; Izvor: [13]

Slika 5.4 prikazuje stanja Java dretvi na malo drugačiji način. Ovdje je stanje *Waiting* detaljnije razloženo u tri posebna stanja: *Blocked, Waiting* i *Timed waiting* (a *Ready* se naziva *Runnable*). No, niti ovdje zbivanja koja implicitno imaju naredbe *notify()* / *notifyAll()* nisu prikazana potpuno detaljno. Napomenimo da dretva prelazi u stanje *Waiting ne* samo nakon naredbe *wait()* (i *join()*), već i kod čekanja na objekte klase *Lock* ili *Condition* iz *java.util.concurrent* librarya, uvedenog u Java 5 verziji.



Slika 5.4: Dijagram promjene stanja Java dretvi; Izvor: [9]

Spomenimo da uz osnovnu varijantu metode *wait()* postoje i varijante *wait(long millis)* i *wait(long millis, int nanos)* koje uzrokuju da dretva čeka na obavijest od druge dretve ili na istek definiranog vremena. Java metode *wait()* / *notify* / *notifyAll* pripadaju klasi Object, od koje ih nasljeđuju sve druge klase. Inače, Java metode *notify* / *notifyAll* se u izvornoj terminologiji monitora zovu *signal* / *signal_all.*

Na kraju, prikažimo kako može nastati deadlock, kada se dvije dretve (ili grupa dretvi) blokiraju zauvijek, jer jedna dretva čeka lokot koji ima druga, i obrnuto. Primjer iz [14]:

```
public class C extends Thread {
  private Object a;
  private Object b;
  public C(Object x, Object y) {
    a = x;
    b = y;
  }
  public void run() {
    synchronized(a) {
      synchronized(b) {
        ...
      }
    }
  }
}
```

Pretpostavimo sada da se izvodi sljedeći kod, gdje su *a1* i *b1* objekti tipa *Object*:

```
C t1 = new C(a1, b1);
C t2 = new C(b1, a1);
t1.start();
t2.start();
```

Budući da su argumenti *a1* i *b1* međusobno permutirani kod kreiranja dretvi *t1* i *t2*, može doći do takve sekvence poziva u kojem dretva *t1* zaključa objekt *a1*, dretva *t2* zaključa objekt *b1*, i onda su obje dretve blokirane. Za razliku od sustava za upravljanje bazom podataka (što smo vidjeli u 1. točki), Java sustav ovdje neće detektirati (i onda riješiti) deadlock. Zato programer mora paziti da do deadlocka ne dođe. U Javi 5 postoji mogućnost da se program lakše napiše na način da ne dođe do deadlocka.

6. KONKURENTNO PROGRAMIRANJE U JAVI VERZIJE 5, 6, 7

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 u*vedene 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 u ljeto ove godine (2011.) u području konkurentnog programiranja novost je Fork/Join Framework.

U ovoj točki prikazat će se (ukratko) samo neke mogućnosti uvedene u Javi 5 (sa poboljšanjima u Javi 6), i to *ReentrantLock*, *ReadWriteLock*, *Conditions* i *Atomic variables*.

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 [10], 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 [10]:

```
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 treba ova mogućnost ili dodatne mogućnosti klase *ReentrantLock, koje su* navedene u nastavku.

ReentrantLock ima ove dodatne mogućnosti:

- pomoću metode *tryLock()* moguće je vidjeti da li je lokot slobodan; ako jeste, zaključa se, a ako nije, metoda odmah vraća exception; ovo je slično kao SQL naredba SELECT ... FOR UPDATE NOWAIT;

- pomoću metode *tryLock(long timeout,TimeUnit unit)* moguće je vidjeti da li je lokot slobodan; ako jeste, zaključa se, a ako nije, metoda vraća exception nakon isteka zadanog vremena; ovo je slično kao SQL naredba SELECT ... FOR UPDATE WAIT timeout;

- pomoću metode *lockInterruptibly()* dretva može (pokušati) zaključati lokot na taj način da aktivnost koja se može prekinuti (cancellable activities) može prekinuti dretvu dok čeka na lokot;

- *metoda newCondition()* omogućava definiranje *kondicija* (*condition*) uz lokot.

Sljedeći primjer [10] prikazuje korištenje metode *tryLock(long timeout,TimeUnit unit)*:

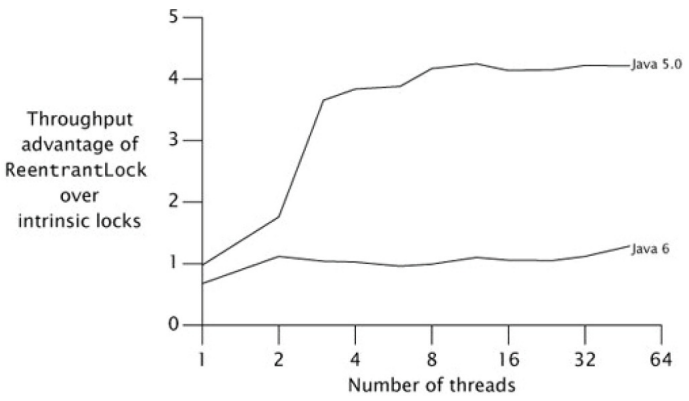
```
public boolean trySendOnSharedLine
  (String message, long timeout, TimeUnit unit)throws InterruptedException
{
  long nanosToLock = unit.toNanos(timeout)
  if (!lock.tryLock(nanosToLock, NANOSECONDS)) return false;
  try {
    return sendOnSharedLine(message);
  } finally {
    lock.unlock();
  }
}
```

Sljedeći primjer [10] prikazuje korištenje metode *lockInterruptibly()*:

```
public boolean sendOnSharedLine(String message)
  throws InterruptedException
{
  lock.lockInterruptibly();
  try {
    return cancellableSendOnSharedLine(message);
  } finally {
    lock.unlock();
  }
}
```

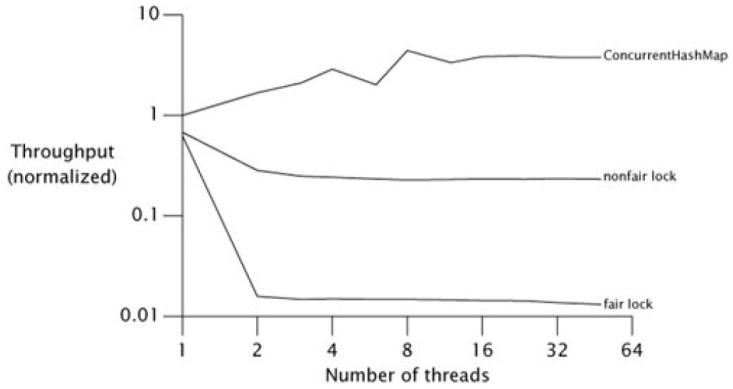
Treba napomenuti da zapravo i metoda *tryLock(long timeout,TimeUnit unit)* ima mogućnost prekidanja, a ne samo mogućnost čekanja (određeno vrijeme) da se lokot otključa. Prikazane metode omogućavaju programiranje na način da se izbjegne deadlock.

Slika 6.1 prikazuje kako je u Java 5 verziji *ReentrantLock* bio značajno bolje propusnosti (kod većeg broja dretvi) od *synchronized* varijante, ali je u verziji Java 6 to izjednačeno:



Slika 6.1: Propusnosti ReentrantLock-a u odnosu na propusnost unutarnjeg lokota; Izvor: [10]

Kod kreiranja *ReentrantLock* lokota mogu se definirati dvije varijante: fer i ne-fer lokot. Fer lokoti osiguravaju da dretve dobivaju lokote po redoslijedu prispijeća zahtjeva. Default su ne-fer lokoti, isto kao kod *synchronized* varijante. Ako nam nije nužno potreban fer lokot, bolje je koristiti ne-fer lokot, zbog puno veće propusnosti, što prikazuje slika 6.2:



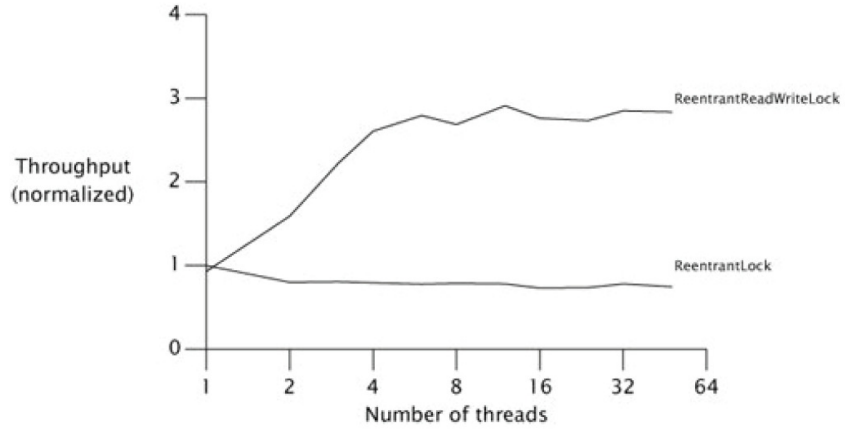
Slika 6.2: Propusnost fer i ne-fer ReentrantLock lokota; Izvor: [10]

Osim dvije krivulje za fer i ne-fer lokote, vidi se i krivulja koja prikazuje korištenje *ConcurrentHashMap* kolekcije. Uglavnom je bolje koristiti konkurentne kolekcije, koje često koriste *neblokirajuće algoritme* (tj. sinkronizaciju bez lokota), nego vlastita rješenja.

Osim *ReentrantLock* lokota, postoje i lokoti klase *ReentrantReadWriteLock*, koja implementira sučelje *ReadWriteLock*:

```
public interface ReadWriteLock {
  Lock readLock();
  Lock writeLock();
}
```

Strategija zaključavanja koju implementiraju ovakvi lokoti je: istovremeno može raditi više čitatelja koji blokiraju pisce, ali može raditi samo jedan pisac, koji blokira čitatelje i (druge) pisce. Slika 6.3 prikazuje propusnost koju ima *ReentrantReadWriteLock* u odnosu na *ReentrantLock*:



Slika 6.3: Propusnost "običnih" i read-write reentrant lokota; Izvor: [10]

U Java 5 verziji pojavili su se i *objekti-kondicije* (*condition objects*). Kao što su eksplicitni lokoti generalizacija unutarnjih lokota, tako su i objekti-kondicije generalizacija *unutarnjih redova kondicija* (*intrinsic condition queues*). *Kondicija* (*condition*) se povezuje sa *Lock* objektom na taj način da se pozove *Lock.newCondition* na već kreiranom lokotu (*Lock* objektu). Za razliku od unutarnjih lokota i njihovih redova kondicija, gdje je uz jedan unutarnji lokot vezan samo jedan red kondicija, kod eksplicitnih lokota može se vezati više kondicija za jedan lokot, ako postoji potreba. Kondicije imaju metode *await, signal, signalAll*, koje su ekvivalentne metodama *wait, notify, notifyAll* kod unutarnjih redova kondicija.

Ovako izgleda sučelje Condition:

```
public interface Condition {
  void await() throws InterruptedException;
  boolean await(long time, TimeUnit unit) throws InterruptedException;
  long awaitNanos(long nanosTimeout) throws InterruptedException;
  void awaitUninterruptibly();
  boolean awaitUntil(Date deadline) throws InterruptedException;
  void signal();
  void signalAll();
}
```

Primjer korištenja kondicija za implementaciju ograničenog međuspremnika (vidi se da se na jedan lokot vežu dvije kondicije, te da se koriste nove metode *await* i *signal*):

```
class BoundedBuffer {
  Lock lock = new ReentrantLock();
  Condition notFull = lock.newCondition();  // Povezivanje jednog lokota
  Condition notEmpty = lock.newCondition(); // i dvije kondicije
  Object[] items = new Object[100];
  int putptr, takeptr, count;

  public void put(Object x)throws IE {
    lock.lock();
    try {
      while (count == items.length) notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }

  public Object take() throws IE {
    lock.lock();
    try {
      while (count == 0) notEmpty.await();
      Object x = items[takeptr];
      if (++takeptr == items.length) takeptr = 0;
      --count;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

Vidjeli smo neka (a ima ih još dosta) poboljšanja u zaključavanju u Javi 5. No, kako kažu autori u [10], mnoge klase u paketu *java.util.concurrent*, kao što su *Semaphore* i *ConcurrentLinkedQueue*, pružaju bolje performanse i skalabilnost u odnosu na stare klase (koje su koristile *synchronized*), ne zato što koriste nove vrste lokota, već zato što uopće ne koriste lokote - koriste *atomarne varijable* (*atomic variables*) i *neblokirajuću sinkronizaciju* (sinkronizacija bez lokota). Neblokirajući algoritmi značajno su kompleksniji od blokirajućih, ali pružaju bolje performanse i skalabilnost, nemaju problema npr. sa deadlockom, pa su najčešći predmet novijih istraživanja na području konkurentnih algoritama.

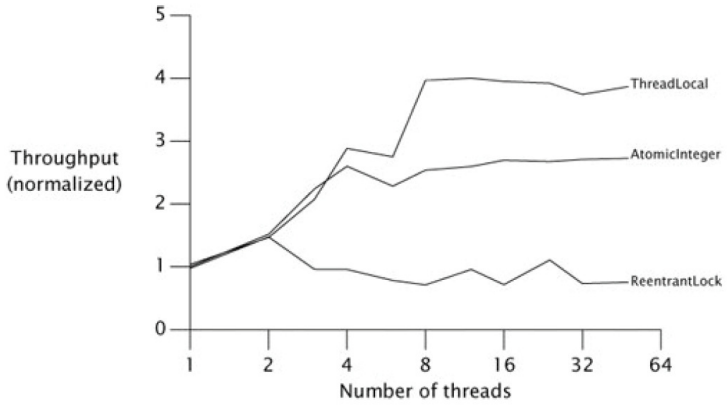
No, kako naglašava autor u [6], pisanje neblokirajućih konkurentnih algoritama je posao za eksperte. Navodi da onaj tko misli da zna pisati takve algoritme treba proći tzv. *Goetzov test* (Brian Goetz je dugogodišnji stručnjak za konkurentno programiranje u Javi, trenutačno Java Language Architect u firmi Oracle): "Ako znate pisati JVM visokih performansi za moderne mikroprocesore, tada ste kvalificirani *da razmišljate o tome* da li smijete izbjeći lokote za sinkronizaciju (tj. pisati neblokirajuće algoritme)".

Atomarne varijable su na neki način generalizacija *volatilnih varijabli* (*volatile variables*) koje su postojale i prije Jave 5. Postoji dvanaest klasa atomarnih varijabli, podijeljenih u četiri grupe, a najvažnija je grupa *skalarnih varijabli* (*scalars*), koju čine klase *AtomicInteger, AtomicLong, AtomicBoolean* i *AtomicReference*. Atomarne varijable su, kao i lokoti u novim verzijama Jave, implementirane uglavnom uz pomoć CAS (compare-and-swap) operacije, koja je opisana u 4. točki. Jedino kad određeni hardver ne podržava tu operaciju (što je rijetkost, jer svi procesori već desetak godina podržavaju CAS ili ekvivalent), onda JVM umjesto CAS obično koristi *spinn lock* (zaključavanje pomoću radnog čekanja). JVM-ovi su, kao i operacijski sustavi, koristili CAS (ako je postojao na određenom hardveru) i prije Jave 5, ali tek od Jave 5 mogu Java klase (uključujući i one koje mi pišemo) koristiti CAS operaciju.

U nastavku se prikazuju dvije realizacije [10] generatora pseudoslučajnih brojeva - pomoću klase *ReentrantLock* i pomoću klase *AtomicInteger* i operacije CAS [10]:

```
public class ReentrantLockPseudoRandom extends PseudoRandom {
  private final Lock lock = new ReentrantLock(false);
  private int seed;
  ReentrantLockPseudoRandom(int seed) {this.seed = seed;}
  public int nextInt(int n) {
    lock.lock();
    try {
      int s = seed;
      seed = calculateNext(s);
      int remainder = s % n;
      return remainder > 0 ? remainder : remainder + n;
    } finally {lock.unlock();}
  }
}

public class AtomicPseudoRandom extends PseudoRandom {
  private AtomicInteger seed;
  AtomicPseudoRandom(int seed) {this.seed = new AtomicInteger(seed);}
  public int nextInt(int n) {
    while (true) {
      int s = seed.get();
      int nextSeed = calculateNext(s);
      if (seed.compareAndSet(s, nextSeed)) {
        int remainder = s % n;
        return remainder > 0 ? remainder : remainder + n;
      }
    }
  }
}
```

Na slici 6.4 prikazuje se propusnost oba rješenja. Prikazana je i krivulja za tzv. *ThreadLocal* varijable (imaju najbolje performanse), a to su varijable koje imaju svoju posebnu instancu za svaku dretvu, na temelju *thread local storage* mehanizma - to je nešto slično kao kada u bazi podataka jedna sesija ne vidi mijenjane, ali nekomitirane, podatke druge sesije, jer čita svoju vlastitu verziju u UNDO tablespaceu.



Slika 6.4: Propusnost ReentrantLock i AtomicInteger kod srednjeg opterećenja; Izvor: [10]

7. EIFFEL OOPL

Eiffel je 1985. godine dizajnirao (a 1986. je napravljen prvi compiler) Bertrand Meyer, jedan od najvećih autoriteta na području OOPL-a. Prvo izdanje njegove knjige OOSC (1988.), značajno je djelo informatičke literature (aktualno je 2. izdanje iz 1997.). Ta je knjiga upoznala javnost sa OO jezikom Eiffel (tada verzije 2). Eiffel je od početka je podržavao *višestruko nasljeđivanje*, *generičke klase*, *obradu iznimaka*, *garbage collection* i metodu *Design by Contract* (DBC), a kasnije su mu dodani *agenti* (vjerojatno će nešto slično imati Java 8 pod imenom *closures* ili *lambda expressions*; C# ih već ima), *nasljeđivanje implementacije* (uz nasljeđivanje tipa) i metoda za konkurentno programiranje *Simple Concurrent Object-Oriented Programming* (SCOOP). U široj je javnosti daleko manje poznat nego C++ i Java, ali ga mnogi autoriteti smatraju danas najboljim OOPL jezikom. Eiffel je od 2005. godine ECMA standardiziran, a od 2006. ISO standardiziran.

Slijedi primjer Eiffel klase. Napomenimo da konstruktor metoda u Eiffelu ne mora imati isto ime kao klasa u kojoj se nalazi, ali smo namjerno zadržali isto ime, kao što npr. mora biti u Javi:

```
class ZIVOTINJA
create zivotinja
feature {ANY}
   ime      : STRING
   visina_cm: DOUBLE
   zivotinja (p_ime: STRING; p_visina_cm: REAL) is
      do
         ime       := p_ime
         visina_cm := p_visina_cm
      end
   prikazi_podatke is
      do
         print ("  Ime: ");           print (ime)
         print (", Visina (cm): ");   print (visina_cm)
         print (", Visina (inch): "); print (visina_inch)
      end
   visina_inch: REAL is
      do
         Result := visina_cm / 2.54
      end
end
```

Vidimo da se u Eiffelu ne mora koristiti znak za odvajanje naredbi " ; ", ali je preporuka da se koristi ako imamo više naredbi u istom retku (zbog čitljivosti). Primijetimo da u funkciji ne postoji ključna riječ **return**, već Eiffel ima posebnu predefiniranu varijablu **Result**. Također, primijetimo da Eiffel (a to ima i PL/SQL) omogućava tzv. uniformni pristup (uniform access), tj. poziv atributa ne razlikuje se od poziva funkcije bez parametara. Npr. u proceduri *prikazi_podatke* funkciju pozivamo sa *visina_inch*, dok npr. Java traži da se koriste zagrade i kad metoda nema parametara, pa se mora pisati *visina_inch()*. Uniformni pristup se smatra značajnom mogućnošću OOPL jezika.

Eiffel ima vrlo jednostavno i vrlo fleksibilno definiranje pristupa, jer za svaki atribut/metodu možemo definirati koja mu klasa može pristupati. Npr, ako ne želimo dati niti jednoj klasi pristup atributu *visina_cm* i funkciji *visina_inch*, pristup konstruktoru *zivotinja* želimo dozvoliti samo klasama TEST1 i TEST2, a pristup atributu *ime* i proceduri *prikazi_podatke* želimo dozvoliti svim klasama, napisat ćemo:

```
feature {NONE} visina_cm ...; visina_inch ...;
feature {TEST1, TEST2} zivotinja ...;
feature {ANY} ime ...; prikazi_podatke ...;
```

Važno je naglasiti da u Eiffelu dozvola pristupa atributu znači dozvolu čitanja atributa, ne i dozvolu pisanja. Vrijednosti atributa može mijenjati samo metoda iz klase u kojoj je atribut definiran (ili iz njene podklase), pomoću *set* procedura. Zbog toga u Eiffelu ne treba raditi *get* funkcije, dok Java mora imati *get* funkcije ako želimo atribute zaštiti od upisa izvan klase. Zanimljivo je da Eiffel može sakriti atribute nekog objekta čak i od drugih objekata (instanci) iste klase, pomoću {NONE} ili, što je isto, pomoću {}. Eiffel ne poznaje sakrivanje atributa/metoda od podklase (nema nešto kao Java **private**), jer Meyer drži da to nije u skladu sa OO modelom.

Eiffel podržava jednostruko i *višestruko nasljeđivanje* (klasa). No, postoje dvije vrste nasljeđivanja - *nasljeđivanja tipa* (zove se i *semantičko nasljeđivanje*, ili nasljeđivanje sučelja) i *nasljeđivanja implementacije* (zove se i *sintaktičko nasljeđivanje*, ili nasljeđivanje programskog koda). Prva vrsta nasljeđivanja je "pravo" nasljeđivanje, u kojem podklasa predstavlja podtip. Druga vrsta nasljeđivanja je "praktično" nasljeđivanje, gdje se želi iskoristiti neki postojeći programski kod, ali se ne želi koristiti polimorfizam. Eiffel je dobio nasljeđivanje implementacije naknadno, po uzoru na jezik C++.

Neki drže višestruko nasljeđivanje vrlo važnim svojstvom OOPL jezika. Npr. Meyer drži da je u mnogim konkretnim situacijama potrebno da klasa može naslijediti dvije ili više klasa i duhovito kaže da je odgovor na pitanje "Da li moja klasa C treba naslijediti klasu A ili klasu B (budući da moj jezik podržava samo jednostruko nasljeđivanje)?" često isto tako težak kao i odgovor na pitanje "Da li da izaberem mamu ili tatu?". Višestruko nasljeđivanje je naročito korisno u slučaju kada postoje dva (ili više) jednako važna kriterija za kreiranje hijerarhije klasa (jednostruko nasljeđivanje dopušta samo jednu hijerarhiju) i u slučaju kada podklasa od jedne nadklase nasljeđuje tip, a od druge nadklase nasljeđuje implementaciju, tzv. mix-in nasljeđivanje (primjer: klasa ARRAYED_STACK nasljeđuje od apstraktne klase STACK i klase ARRAY). Slijedi Eiffel (nepotpuni) primjer klase C koja nasljeđuje od klasa A i B, pri čemu klasa C nadjačava jedan atribut i jednu metodu iz klase A i jednu metodu iz klase B:

```
class C
inherit
   A redefine atribut_iz_a, metoda_iz_a end
   B redefine metoda_iz_b end
feature
   ...
end
```

Kod višestrukog nasljeđivanja najčešće se navode dva glavna problema. Jedan je problem kada roditeljske klase imaju atribute/metode istog imena, ali različitog značenja. Eiffel za to ima vrlo jednostavno rješenje – preimenovanje (barem jednog) atributa/metode pomoću rename. Drugi je problem kada imamo tzv. ponavljajuće (repeated) nasljeđivanje, npr. u primjeru gdje su klase B i C djeca klase A, a klasa D je dijete i od B i od C, pa izlazi da je klasa D dva puta (indirektno) dijete od klase A (to se naziva i dijamantnim nasljeđivanjem, zbog sličnosti crteža takvih klasa sa skicom dijamanta). Ako atributi/metode klase A nisu nadjačani u klasama B i C, Eiffel za to ima najjednostavnije moguće rješenje – ne treba ništa napraviti, jer duplih atributa/metoda niti nema. Ako su pak atributi/metode iz klase A redefinirani u klasi B i/ili C, tada postoje dva slučaja – da je kod nadjačavanja u klasama B/C zadržano isto ime atributa/metode kao u klasi A, ili da je ime promijenjeno. Prvi slučaj (ista imena) se najčešće svodi na drugi, preimenovanjem jednog atributa/metode (a rjeđe se koristi "eliminacija" jednog atributa/metode, pomoću **undefine**). U drugom slučaju (različita imena) Eiffel traži da se eksplicitno izabere (pomoću **select**) željeni atribut/metoda, što je potrebno da bi se razriješila dilema izbora prave metode kod dinamičkog pozivanja metoda (zbog polimorfičnog pridruživanja).

Eiffel šalje parametre referentnog tipa kao reference, a **expanded** parametre šalje kao vrijednosti. Bez obzira kako se parametri šalju, Eiffel ne dozvoljava da se oni u metodi mijenjaju, niti pomoću naredbe pridruživanja "parametar := nesto", niti pomoću naredbe kreiranja objekta "create parametar;". Ali, iako Eiffel metoda ne može mijenjati objekt predstavljen parametrom, može mijenjati njegove atribute (bilo direktno, bilo pozivom drugih metoda). Zanimljivo je pitanje *promjene signature*, tj. pitanje da li parametri u nadjačanoj metodi mogu imati drugačiji tip od parametara u metodi iz nadklase i (ako mogu) kakav mora biti taj tip. Postoje tri (glavne) mogućnosti:

1. tip parametra se ne može mijenjati - *no variance;*
2. može se mijenjati tako da bude podtip u odnosu na bazni – *covariance;*
3. može se mijenjati tako da bude nadtip - *contravariance*.

Eiffel podržava covariance i za parametre metoda i za povratnu (return) vrijednost funkcije i za atribute. Npr. Java je do verzije 1.4 imala u potpunosti no variance pristup, ali od verzije 1.5 (ili 5.0) podržava covariance za povratne vrijednosti funkcije (i PL/SQL se ponaša kao Java 1.5).

Eiffel je imao *generičke klase* od početka, dok ih Java dobila u verziji 1.5. Možemo reći da generičke klase zapravo nisu prave klase, nego predlošci za klase, jer imaju tzv. generičke parametre. Npr. u Eiffelu ovako izgleda generička klasa STACK [G] (generički parametar nazvan je G, a može se zvati i drugačije):

```
class STACK [G]
feature
   element_na_vrhu: G is
      do ... end
   ...
end
```

Generička klasa se koristi kao klijent neke (druge) klase, u kojoj se (drugoj klasi) atribut, varijabla ili parametar deklarira pomoću generičke klase, tako da se generički parametar zamijeni sa nekom (trećom) klasom. Npr. klasa STACK_KLIJENT može sadržavati dva atributa definirana na sljedeći način (generički parametar G zamijenjen je klasom ZIVOTINJA, odnosno klasom REAL):

```
atribut1: STACK [ZIVOTINJA]
atribut2: STACK [REAL]
```

Eiffel ima (kao i Java) i tzv. ograničenu generičnost (constrained genericity), gdje se generički parametar ograničava nekom određenom klasom, pomoću operatora ->. U slučaju ograničene generičnosti, klasa koja zamjenjuje generički parametar mora biti ili ista kao klasa koja ograničava generički parametar, ili podklasa te klase. Npr. ako bismo ovako definirali (ograničenu) generičku klasu STACK:

```
class STACK [G -> ZIVOTINJA] ... end
```

tada bismo imali sljedeće:

```
atribut1: STACK [BAKTERIJA]      -- greška, nije podklasa od ZIVOTINJA
atribut2: STACK [ZIVOTINJA]      -- OK, može biti ista klasa
atribut3: STACK [KUCNI_LJUBIMAC] -- OK, to je podklasa od ZIVOTINJA
```

Design By Contract (DBC) je metoda čiji je autor također Meyer. Stoga nije čudno da Eiffel u potpunosti podržava DBC. Za sada niti jedan drugi OOPL ne podržava DBC u potpunosti, barem ne na razini jezika. DBC je opširno opisan u [12]. Pojednostavljeno rečeno, DBC se zasniva na ideji da svaka metoda (procedura ili funkcija), uz "standardni" programski kod, treba imati još dva dodatna dijela - *pretkondiciju* (precondition) i *postkondiciju* (postcondition). Klasa treba imati još jedan dodatni dio - *invarijantu* (invariant). Ugovor (contract) se zasniva na tome da metoda "traži" od svog pozivatelja (neke druge metode) da zadovolji uvjete definirane u pretkondiciji (plus uvjete definirane u invarijanti), a ona (pozvana metoda) se tada "obvezuje" da će na kraju zadovoljiti uvjete definirane u postkondiciji (plus uvjete definirane u invarijanti). Ideja je na neki način upravo suprotna od tzv. defanzivnog programiranja, koje zagovora da se u svim mogućim trenucima pokušava što više toga provjeriti.

Eiffel za specifikaciju DBC elemenata koristi ključne riječi **require** (odnosno **require else** u nadjačanoj metodi, kod nasljeđivanja) za označavanje pretkondicije, **ensure** (odnosno **ensure then** u nadjačanoj metodi) za postkondicije i **invariant** za invarijante klasa. Navedene naredbe su najvažnije za DBC podršku, ali Eiffel ih ima još. Naredba **check** služi za provjeru nekog uvjeta u bilo kom trenutku. Za provjeru programskih petlji postoje dvije naredbe: **variant** provjerava da li se cjelobrojni izraz smanjuje kod svakog prolaza u petlji, a **invariant** (opet ista riječ kao za označavanje invarijante klase, ali je u kontekstu petlje značenje drugačije) provjerava da li je određeni uvjet u petlji zadovoljen u svakom prolazu.

U Eiffelu metoda ne može obraditi iznimku koja se desila zbog nezadovoljavanja pretkondicije – jednostavno, pozvana metoda nema što raditi ako se druga metoda (pozivatelj) ne drži ugovora! Dakle, za pojavu iznimke u vrijeme izvršavanja programskog koda pretkondicije "krivac" je metoda-pozivatelj, dok je za pojavu iznimke u vrijeme izvršavanja programskog koda postkondicije "krivac" metoda-izvršavatelj.

Nadjačane metode ne mogu imati bilo kakve pretkondicije ili postkondicije, već nadjačana metoda mora imati jednaku ili slabiju pretkondiciju (tj. može zahtijevati od metode koja ju je pozvala ili isto što i metoda nadklase, ili manje od toga) i mora imati jednaku ili jaču postkondiciju (tj. mora osigurati barem ono što je osiguravala metoda nadklase, ili više od toga). Zato se za definiranje pretkondicije u nadjačanoj metodi koristi oblik **require else** (umjesto **require**), a za definiranje postkondicije u nadjačanoj metodi koristi se **ensure then**.

Može se postaviti pitanje utjecaja izvršenja pretkondicija, postkondicija i invarijanti na brzinu izvođenja programa. Nažalost, utjecaj postoji, a naročito je skupa provjera invarijanti. Stoga se u Eiffel-u može odrediti nekoliko stupnjeva provjere. Najslabija provjera (ali sa najmanjim negativnim utjecajem na brzinu izvođenja) je provjera samo pretkondicija (ta se provjera obično ostavlja i nakon završetka faze testiranja, tj. ostaje u produkcijskom kodu). Sljedeći stupanj uključuje provjeru postkondicija, slijedi provjera invarijanti, zatim provjera petlji i na kraju provjera pomoću check naredbi. Najveći stupanj provjere se obično koristi samo kod testiranja, jer se takvi programi izvršavaju i do 2-3 puta sporije u odnosu na programe koji nemaju nikakvih provjera.

Treba naglasiti da su pretkondicije, postkondicije i invarijante korisne čak i kada nisu realizirane kao programski kod, nego samo kao komentar (najčešće zato što je nešto teško ili nemoguće izraziti – Eiffel nema operatore univerzalnog i egzistencijalnog kvantifikatora iz predikatnog računa), jer poboljšavaju dokumentiranost izvornog koda. Zbog DBC podrške, može se reći da je Eiffel i specifikacijski (a ne samo programski) jezik.

Slijedi Eiffel programski kod za realizaciju stoga (OOSC2 str. 390.-391.). Zapravo, to nije kompletan programski kod, već samo tzv. kratki oblik klase, koji ne prikazuje izvršni kod metoda (zato to nije class, već class interface). Kratki oblik klase ne prikazuje niti skrivene implementacijske detalje, tako da ne vidimo da klasa ima jedan skriveni atribut tipa niz (implement: ARRAY [G]) koji služi za implementaciju stoga:

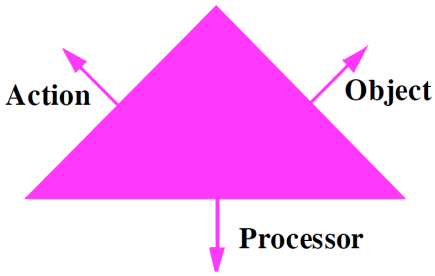
```
class interface STACK [G]

creation make

feature - inicijalizacija
-- stog od n elemenata
make (n: INTEGER) is
   require non_negative_capacity: n >= 0
   ensure  capacity_set: capacity = n
end

feature -- pristup
-- maksimalni broj elemenata
capacity: INTEGER
-- broj elemenata stoga
el_count: INTEGER
-- element na vrhu stoga

item: G is
  require not_empty: not empty
end

feature - statusi
-- da li je stog prazan?
empty: BOOLEAN is
   ensure empty_definition: Result = (el_count = 0)
end

-- da li je stog pun?
full: BOOLEAN is
  ensure full_definition: Result = (el_count = capacity)
end

feature - promjena
-- dodaj x na vrh stoga
put (x: G) is
  require not_full: not full
  ensure not_empty: not empty
         added_to_top: item = x
         one_more_item: el_count = old el_count + 1
end

-- makni element sa vrha
remove is
   require not_empty: not empty
   ensure not_full: not full
          one_fewer:  el_count = old el_count - 1
end

invariant
   count_non_negative: 0 <= el_count
   count_bounded: el_count <= capacity
   empty_if_no_elements: empty = (el_count = 0)

end -- class interface STACK [G]
```

8. KONKURENTNO PROGRAMIRANJE U EIFFELU – SCOOP METODA

Metoda SCOOP (Simple Concurrent Object-Oriented Programming) ima prilično davne korijene. Njen kreator, Bertrand Meyer (koji je i kreator jezika Eiffel), prikazao je osnovne ideje te metode još 1990. godine na TOOLS Europe, a preradio ih je 1993. Detaljan prikaz dao je 1997. u [12]. Kasnije je SCOOP metoda eksperimentalno realizirana i poboljšavana na ETH Zurich, te se koristi naročito za nastavne i znanstvene potrebe. Formalni prikaz te metode zaokružio je 2007. godine P. Nienaltowski u doktorskoj disertaciji kod prof. Meyera. Nedavno je (lipanj 2011.) firma Eiffel Software (www.eiffel.com) uključila SCOOP metodu u svoj proizvod EiffelStudio v.6.8. Moglo bi se reći da je SCOOP zaseban (mini) jezik za konkurentno programiranje, jer eksperimentalne implementacije postoje i izvan jezika Eiffel, npr. postoje i za jezik Java. SCOOP se i dalje razvija, npr. rade se istraživanja na sljedećim područjima:

- prevencija i detekcija deadlocka;
- uvođenje softverske transakcijske memorije;
- distribuirani SCOOP.

Kod SCOOP metode vrlo je važan pojam *procesor*, ali se pod tim pojmom ne misli na fizički procesor, već na apstraktni procesor, koji može biti i fizički procesor (u nastavku će se za njega koristiti termin CPU), proces operacijskog sustava, dretva operacijskog sustava i sl. Da se smanji zabuna, u nastavku će se koristiti oblik (*apstraktni) procesor* (a ne samo *procesor*, kao što se koristi u [12], [13] i [14]). Za razliku od CPU-a, čiji je broj ograničen, možemo držati da je broj (apstraktnih) procesora praktički neograničen. Kako se navodi u [12], (apstraktni) procesor je jedna od tri sile računanja kod OOPL programiranja: neka akcija (ili metoda, tj. funkcija ili procedura) izvodi se na određenom (apstraktnom) procesoru nad određenim objektom (instancom klase), kako prikazuje slika 8.1:



Slika 8.1: Tri sile kod računanja (The three forces of computation); Izvor: [12]

Temeljna ideja vezana za (apstraktne) procesore u SCOOP metodi je: svaki objekt je dodijeljen samo jednom (apstraktnom) procesoru, koji se naziva *rukovatelj (handler) objektom*. S druge strane, jedan (apstraktni) procesor može biti rukovatelj za više objekata. Kad se kreira novi objekt, runtime sistem (na temelju programskih uputa) odlučuje da li će mu se dodijeliti neki rukovatelj od postojećih, ili će se kreirati novi (apstraktni) procesor kao rukovatelj za novokreirani objekt. Ta veza ostaje do kraja - objekt je uvijek vezan za samo jednog rukovatelja, a to u konačnici znači da se nad istim objektom istovremeno može izvoditi samo jedna metoda, jer metode određenog objekta može izvoditi samo rukovatelj tog objekta.

Kada se u nekoj metodi, koja se izvodi nad nekim objektom, poziva metoda nad objektom kojim rukuje drugi rukovatelj (tj. koji nije isti kao rukovatelj prvog objekta), taj se poziv zove *asinkroni poziv* ili *odvojeni (separate) poziv*. U tom slučaju rukovatelj prvog objekta može nastaviti rad, ne čekajući da pozvana metoda završi. Za razliku od toga, poziv metode nad drugim objektom koji ima istog rukovatelja kao i prvi objekt, je *sinkroni poziv* ili *neodvojeni poziv* – to je standardna situacija iz sekvencijalnog programiranja, gdje rukovatelj mora čekati da jedna metoda završi prije nego nastavi izvršavati drugu (naravno, zanemarujemo fizičke detalje CPU-a, mogućnost instrukcijskog paralelizma i sl.).

Ostalo je otvoreno pitanje na koji način runtime sistem određuje kojeg će rukovatelja dodijeliti objektu. U odnosu na nekonkurentni Eiffel, SCOOP metoda uvodi samo još jednu ključnu riječ - **separate**. Uz uobičajenu deklaraciju varijable (napomena: Eiffel izvorno koristi termin *entitet* za atribute, lokalne varijable i argumente metoda)

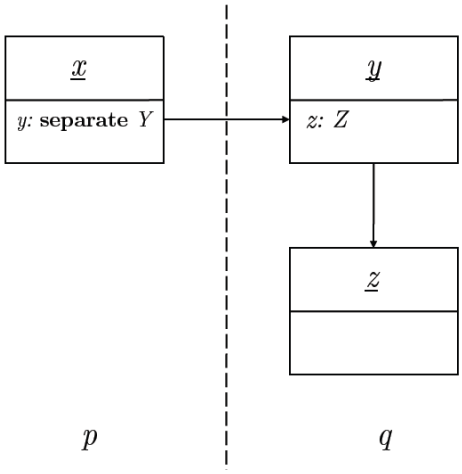
x : X

koja označava da je varijabla *x* referenca na objekt tipa (tj. klase) *X*, sada se može pisati i

x : separate X

čime se izražava da kod izvođenja programa *x* može biti referenca na objekt koji ima drugog rukovatelja u odnosu na objekt u kojem se ta referenca nalazi. Takva se referenca zove *odvojena referenca*, a objekt na koji pokazuje zove se *odvojeni objekt*.

Slika 8.2 prikazuje tri objekta: *x*, *y* i *z*. Objektu *x* je rukovatelj (apstraktni) procesor *p*, a objektima *y* i *z* rukovatelj je (apstraktni) procesor *q*. Objekt *x* ima atribut *y*, koji predstavlja udaljenu referencu, jer ta referenca pokazuje na objekt *y*, koji ima drugog rukovatelja. Za razliku od toga, objekt *y* ima atribut *z* koji je ne-odvojena referenca na ne-odvojeni objekt *z*, jer objekti *y* i *z* imaju istog rukovatelja.



Slika 8.2: Odvojena referenca: u objektu *x*, atribut *y* referencira objekt na drugom (apstraktnom) procesoru; Izvor: [14]

Primjer [12] pojasnit će do sada navedeno. Prtpostavimo da u klasi *WORKER*, metoda *task* nešto radi i stavlja rezultat u atribut *output*:

```
class WORKER
  feature
    output: INTEGER
    do task (input: INTEGER) do ... end
end
```

Klasa *MANAGER* ima dva atributa - reference tipa *WORKER,* ali jedna referenca je odvojena, a druga je ne-odvojena. U nekoj metodi te klase pozivaju se metode *task* nad odvojenim objektom *worker1* i nad ne-odvojenim objektom *worker2*, a onda se rezultati izvođenja zbrajaju:

```
class MANAGER
  feature
    worker1 : separate WORKER
    worker2 : WORKER

  -- in some routine:
    do
      . . .
      worker1.do task (input1)
      worker2.do task (input2)
      result := worker1.output + worker2.output
    end
end
```

Zbivanja kod izvođenja ovog programskog koda mogu se ovako opisati: procedura *task* nad odvojenim objektom *worker1* poziva se asinkrono, tj. program ne čeka da ona završi, već odmah prelazi na sljedeću proceduru *task* nad objektom *worker2*, koju izvodi sinkrono. Kada je ta druga procedura gotova, izvršava se sljedeća naredba, koja zbraja dva rezultata. Tek u ovom trenutku mora se čekati da procedura *task* nad objektom *worker1* završi, jer je sada potreban rezultat te procedure. To se čekanje obavlja automatski od strane SCOOP mehanizma, i zove se *wait-by-necessity* (čekanje po potrebi). U ovom primjeru je rad sa odvojenim objektima bio lagan, jer je sistem sam znao da li je poziv sinkron ili asinkron i kada treba čekati na rezultat.

U nastavku će biti prikazano kako se u SCOOP-u radi međusobno isključivanje (mutual exclusion) u slučajevima kada se odvojeni objekti međusobno upliću jedan drugome u rad, tj. kod tzv. *race condition*. Primjer rada sa brojačima je ekvivalentan Java primjeru iz 5. točke (oboje je iz [12]):

```
class COUNTER
feature
  value : INTEGER

  set value (a value: INTEGER)
    do
      value := a value
    end

  increment
    do
      value := value + 1
    end
end
```

Pretpostavimo da postoji varijabla *x* deklarirana kao *separate COUNTER* i pratimo sljedeći niz naredbi:

```
x.set value (0)
x.increment
i := x.value
```

Istovremeno se u nekom odvojenom (apstraktnom) procesoru odvija naredba:

```
x.set value (2)
```

Kao i u ekvivalentnom Java primjeru u 5. točki, zbog mogućeg ispreplitanja ovih naredbi u paralelnom radu, vrijednost varijable *i* iz prvog koda može biti 1, 2 ili 3. U Javi se to moglo spriječiti sinkronizacijom pomoću (npr.) *synchronized*. U Eiffelu se koristi jednostavno pravilo, koje se zove *pravilo odvojenih argumenata* (*separate argument rule*): runtime sistem automatski zaključava (apstraktne) procesore koji rukuju odvojenim objektima koji su (objekti) poslani kao argumenti metode. Ako je (apstraktni) procesor zaključan, ne može ga se koristiti, pa se ne može pozvati niti jedna metoda nad objektom kojemu je rukovatelj zaključani (apstraktni) procesor. Izmijenimo prethodni kod (tri instrukcije) na sljedeći način, tako da ih stavimo u proceduru koja će imati odvojeni objekt kao argument:

```
compute (x: separate COUNTER)
  do
    x.set value (0)
    x.increment
    i := x.value
  end
```

Pogledajmo sada poziv *compute (x)* i pretpostavimo da je (apstraktni) procesor *p* rukovatelj za *x*. Kako je prije rečeno, budući da je *x* odvojeni argument te procedure, procesor *p* mora biti zaključan. Tekući procesor, koji izvodi proceduru *compute*, automatski čeka dok runtime sistem ne zaključa procesor *p*. Kada je *p* zaključan, tijelo procedure *compute* može se izvršavati bez problema (ne postoje višestruki lokoti na jedan procesor), pa će varijabla *i* na kraju procedure uvijek ima vrijednost 1. SCOOP forsira takvo ponašanje, tj. svi pozivi nad odvojenim objektima moraju se uokviriti u procedure kojima se odvojeni objekt šalje kao argument.

Npr. ovo je neispravno (kompajler javlja grešku):

```
x : separate X
compute
  do x.f end
```

a ovo je ispravno:

```
x : separate X
compute (x1: separate X)
  do x1.f end
```

Prikažimo sada u SCOOP-u *sinkronizaciju na temelju uvjeta* (*condition synchronization*), koja se u Javi implementirala npr. pomoću naredbi *wait / notifyAll / notify*, na istom primjeru proizvođača i potrošača kao u 5. točki. Pretpostavimo da imamo generičku klasu *BUFFER[T]* koja implementira neograničeni red (unbounded queue):

buffer: separate BUFFER[INTEGER]

i sljedeću proceduru u klasi potrošača:

```
consume (a buffer: separate BUFFER[INTEGER])
  require
    not (a buffer.count = 0)
  local
    value: INTEGER
  do
    value := a buffer.get
  end
```

Primijetimo da smo koristili pretkondiciju kako bismo osigurali da *buffer* nije prazan kada iz njega čitamo. Pitanje je što će se desiti ako je *buffer* prazan? U sekvencijalnom slučaju desila bi se iznimka (exception). Međutim, u konkurentnom radu pretkondicije na odvojene objekte imaju drugu semantiku - rukovatelj odvojenog objekta se otključa, čeka se dok se ne zadovolji zahtjev, a onda se rukovatelj odvojenog objekta opet zaključa. Pretkondicija se u konkurentnom radu pretvara u *uvjet za čekanje* (*wait condition*). To se ponašanje može iskazati kao *pravilo čekanja* (*wait rule*): "Poziv metode koja ima odvojene argumente izvršit će se onda kada su svi odgovarajući rukovatelji slobodni (nisu zaključani) i kada su zadovoljene sve pretkondicije. Rukovatelji su ekskluzivno zaključani za vrijeme trajanja metode."

Kao i u Javi, tako se i u SCOOP metodi može desiti deadlock. Primjer [12]:

```
class C
  creation
    make

  feature
    a : separate A
    b : separate A

    make (x : separate A, y : separate A)
      do
        a := x
        b := y
      end

    f do g (a) end
    g (x : separate A) do h (b) end
    h (y : separate A) do ... end
end
```

Pretpostavimo sada da se izvršava sljedeći kod, gdje su objekti *c1* i *c2* tipa *separate C*, objekti *a* i *b* su tipa *separate A*, objekt *a* ima rukovatelja *p*, objekt *b* ima rukovatelja *q*:

```
create c1.make (a, b)
create c2.make (b, a)
c1.f
c2.f
```

Budući da su kod inicijalizacija objekata *c1* i *c2* argumenti permutirani, moguća je sekvenca poziva kod koje je u jednom slučaju zaključan rukovatelj *p*, a čeka se da se oslobodi rukovatelj *q*, i obrnuto. Nastao je deadlock.

Deadlock se trenutačno ne može automatski detektirati u SCOOP-u, pa je programerova odgovornost da radi programski kod slobodan od deadlocka (deadlock-free). No, kako se navodi u [12], radi se na implementaciji sheme koja prevenira deadlock, a ona je temeljena na redoslijedu zaključavanja koji prevenira *cikličko zaključavanje* (*cyclical locking*).

ZAKLJUČAK

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.

Vrlo je važno postići visoku paralelnost programa, a razlog za to objašnjava tzv. Amdahlov zakon. 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.

Nažalost, konkurentne programe nije lako pisati. Kako je naglasio autor u [6] (na kraju poglavlja o konkurentnosti): "Nakon rada kroz ovo poglavlje, možete primijetiti da rad sa dretvama u Javi izgleda vrlo kompleksno i teško za korektnu upotrebu. Dodatno, izgleda malo neproduktivno - iako dretve rade paralelno, morate investirati veliki trud da implementirate tehnike koje ih sprečavaju da na loš način utječu jedna na drugu. Ako ste ikada pisali programe u zbirnom jeziku (assembleru), kad pišete programe sa dretvama, imate sličan osjećaj: svaki mali detalj je važan, i nemate sigurnosnu mrežu u obliku provjere od strane kompajlera."

Nije lako pisati tzv. blokirajuće algoritme, koji koriste različite vrste lokota za (blokirajuću) sinkronizaciju. Svi se ti lokoti danas uglavnom zasnivaju na mikroprocesorskoj funkciji (operaciji) C*ompare And Swap* (*CAS*), ili njoj ekivalentnoj. No, CAS (i *CASD*, *Compare-and-Swap-Double*) nisu novost - bile su dio IBM 370 arhitekture od 1970. godine, iako je tek 1991. matematički dokazano da "registri koji koriste *compareAndSet()* i *get()* operacije imaju beskonačni broj konsenzusa", što bi vrlo pojednostavljeno (do banalizacije) značilo da je CAS operacija "jako dobra za konkurentno programiranje". No, korištenje lokota (tj. zaključavanje), bez obzira što se za implementaciju lokota koristi operacija CAS, ima dosta mana. Najveću manu zaključavanja autori u [8] vide u tome što **"nitko stvarno ne zna kako organizirati i održavati veliki sustav temeljen na zaključavanju"**.

Još je teže pisati neblokirajuće konkurentne algoritme, koji ne koriste zaključavanje. I oni se interno temelje (uglavnom) na CAS operaciji, Dakle, CAS operacija je vrlo važna za sadašnjost i budućnost konkurentnog programiranja. No CAS operacija ima i mane: neblokirajuće algoritme koji koriste CAS (ili ekvivalentne operacije) vrlo je teško smisliti i često su vrlo neintuitivni. Zapravo, 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. Problem sa većinom dosadašnjim sinkronizacijskih mehanizama i operacija, 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 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. Ispravno implementirane, transakcije nemaju problem deadlocka ili livelocka, ali najvažnije je da je pomoću njih lakše pisati konkurentne programe. Postoje različite softverske implementacije transakcijske memorije. Npr. programski jezik Clojure podržava STM na razini jezika, a neki jezici podržavaju STM na razini biblioteka.

Što se tiče HTM-a, danas postoje barem dva mikroprocesora koja podržavaju HTM - Vega procesor firme Azul Systems (već 5-6 godina), a nedavno (kolovoz 2011.) mu se pridružio i 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.

Postoje i neka softverska rješenja konkurentnog programiranja koja (za sada) ne koriste STM, ali izgledaju naprednija nego npr. rješenja u Javi. Jedno takvo rješenje je Simple Concurrent Object Oriented Programming (SCOOP), model koji je realiziran u OOPL jeziku Eiffel (iako postoje eksperimentalne realizacije i u drugim jezicima, pa i u Javi). Suština te metode je da objektima ekskluzivno rukuju njihovi rukovatelji - apstraktni procesi, i ne zaključavaju se objekti, već procesi. Uobičajena semantika Eiffel pretkondicije se u SCOOP-u mijenja – neuspjeh zadovoljenja pretkondicije ne uzrokuje exception, već čekanje na zadovoljavanje uvjeta. Vrijeme će pokazati da li je taj model uspješan.

LITERATURA

1. Bloch, J. (2008): Effective Java (2.izdanje), Addison-Wesley / Sun Microsystems

2. Budin, L., Golub, M., Jakobović, D., Jelenković, L. (2010): Operacijski sustavi, Element, Zagreb

3. Cliff C. (2010): Azul's Experiences with Hardware / Software Co-Design (prezentacija), Azul Systems, blogs.azulsystems.com/cliff (kolovoz 2011.)

4. Cliff C., Göetz B. (2009): Not Your Father's Von Neumann Machine - A Crash Course in Modern Hardware (prezentacija), JavaOne 2009, http://www.azulsystems.com/events/javaone_2009/session/2009_J1_HardwareCrashCourse.pdf (kolovoz 2011.)

5. Drepper U. (2007): What Every Programmer Should Know About Memory (white paper), Red Hat Inc., http://www.akkadia.org/drepper/cpumemory.pdf (kolovoz 2011.)

6. Eckel B. (2006): Thinking in Java (4.izdanje), Prentice Hall

7. Hennessy, J.L., Patterson D.A. (2007): Computer Architecture - A Quantitative Approach (4.izdanje), Elsevier / Morgan Kaufmann Publishers

8. Herlihy, M., Shavit, N. (2008): The Art of Multiprocessor Programming, Elsevier / Morgan Kaufmann Publishers

9. Horstmann, C.S., Cornell, G. (2008): Core Java: Volume 1 - Fundamentals (8.izdanje), Prentice Hall / Sun Microsystems

10. Göetz B. i ostali (2006): Java Concurrency in Practice, Addison-Wesley

11. Lea D. (1999): Concurrent Programming in Java - Design Principles and Patterns (2.izdanje), Addison-Wesley

12. Meyer, B. (1997): Object-Oriented Software Construction, Prentice Hall

13. Meyer, B., Nanz S. (2010): Concepts of Concurrent Computation (materijali sa kolegija), ETH Zuerich - Chair of Software Engineering, http://se.inf.ethz.ch/old/teaching/2010-S/0268/index.html#lectures_slides (kolovoz 2011.)

14. Nanz S. i dr. (2010): A Comparative Study of the Usability of Two Object-oriented Concurrent Programming Languages, ETH Zuerich & University of Toronto, http://se.inf.ethz.ch/people/nanz/publications/nanz-et-al_arxiv1011.6047.pdf (kolovoz 2011.)

15. Reinhold M. (2011): Divide and Conquer Parallelism with the Fork/Join Framework (prezentacija), Oracle Java Platform Group, http://www.oracle.com/us/technologies/java/fork-join-framework-428206.pdf www (kolovoz 2011.)

Oracle priručnici za bazu 11g Release 2 (2009):

16. Oracle Database Concepts

17. Oracle Database PL/SQL Packages and Types Reference