JAVA PARALELIZACIJA

Zlatko Sirotić, univ.spec.inf.
ISTRA TECH d.o.o.
Pula

ISTRA TECH =
Istra informatički inženjering (staro ime)
Poduzeće osnovano prije 25 godina

iii *maginarna jednadžba*

Istra informatički inženjering

$$iii = i^{i^{i}}$$

Autor je (bar neko vrijeme) radio i sa ovim jezicima / alatima:

- "Assembler" za Texas Instruments TI-58 kalkulator (1981.)
- Fortran, BASIC (1982.)
- Pascal, Assembler za Zilog Z80 (ZX Spectrum) (1983.)
- Cobol, dBASE III, Prolog (1984.)
- ADS (Application Development System (1985. – 1999.)
- **Oracle Database, Designer, Forms, Reports (1995. -)**
- Eiffel, C, C++ (1998. -)
- **Java (2003. -)**
- Scala (2012. -)
- **Oracle ADF (2013. -)**

Neki autorovi radovi zadnjih godina

- HrOUG 2014: Nasljeđivanje je dobro, naročito višestruko - Eiffel, C++, Scala, Java 8
- CASE 2014: Trebaju li nam distribuirane baze u vrijeme oblaka?
- JavaCro 2014: Da li postoji samo jedna "ispravna" arhitektura web poslovnih aplikacija
- HrOUG 2013: Transakcije i Oracle - baza, Forms, ADF
- CASE 2013: Što poslije Pascala? Pa … Scala!
- HrOUG 2012: Ima neka loša veza (priča o in-doubt distribuiranim transakcijama)
- CASE 2012b: Konkurentno programiranje u Javi i Eiffelu
- CASE 2012a: Utjecaj razvoja µP na programiranje

Uvod

- Gotovo sva današnja računala imaju **više CPU-a**, u obliku **višejezgrenih procesora**, ili imaju **više procesora**.
- Takva računala mogu **paralelno izvršavati** dva (ili više) **nezavisna programa ili dijelove istog programa**.
- U drugom slučaju, ako su dijelovi programa nezavisni, programeri mogu pisati kod kao da nema paralelnosti.
- No, **najčešće su dijelovi programa međusobno zavisni**, jer čitaju / pišu u isto memorijsko područje, pa je moguće da 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 se obično zovu imenom **konkurentno programiranje**.

Teme

- Konkurentno programiranje i operacijski sustavi
- Utjecaj razvoja mikroprocesora na konkurentno programiranje
- Sinkronizacijski algoritmi i mehanizmi
- Konkurentno programiranje u Javi verzije 1 – 4
- Konkurentno programiranje u Javi verzije 5 – 7
- Java 8 - najvažnije nove mogućnosti: Streams, lambda, default metode
- Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka, na jednom jednostavnom primjeru

Konkurentno programiranje i operacijski sustavi

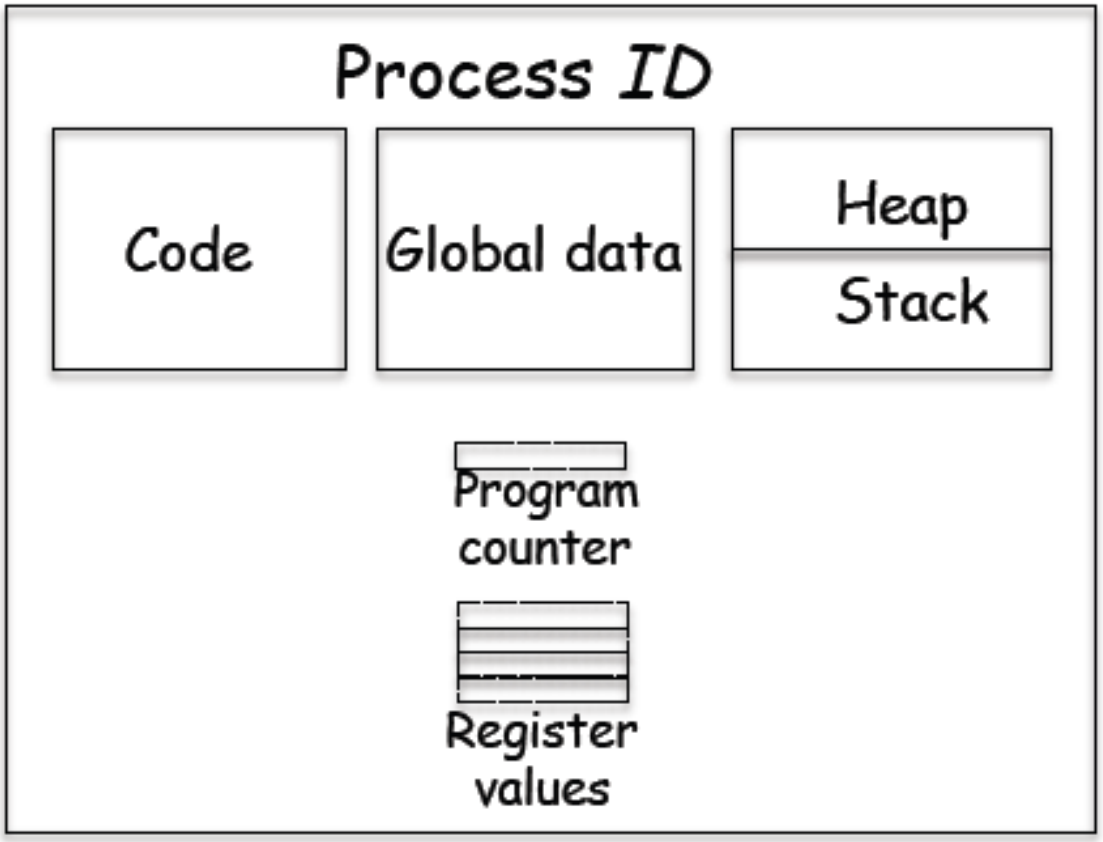
- Prvi operacijski sustavi, napravljeni 50-tih godina prošlog stoljeća, bili su **sekvencijalni**, tj. računalni programi izvršavali su se na njima jedan iza drugoga.
- Vrlo brzo se uvidjelo da je to vrlo neracionalan način korištenja računala, pa su 60-tih godina napravljeni brojni operacijski sustavi 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, već **kvazi-paralelno**. Uobičajeno se takav kvazi-paralelan način rada naziva **višezadaćnost (multitasking)**.

Konkurentno programiranje i operacijski sustavi

- 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).
- Ovakva računala omogućavaju da se dva (ili više) procesa zaista izvršavaju paralelno, što se uobičajeno naziva **multiprocesiranje (multiprocessing)**.
- I multiprocesiranje i višezadaćnost su **primjeri konkurentnog izvršavanja**.
- Postoji i **distribuirano izvršavanje**. Kod njega se, za razliku od konkurentnog izvršavanja, procesiranje zbiva na **dva ili više računala** koja su spojena mrežom.

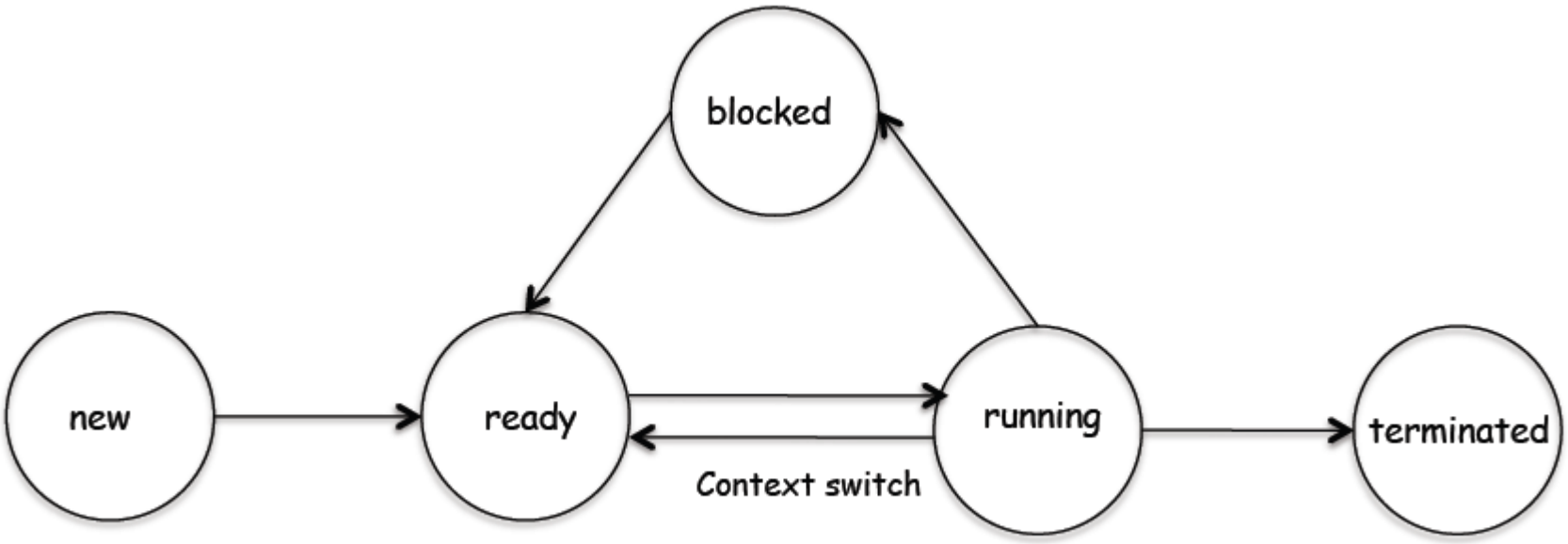
Konkurentno programiranje i operacijski sustavi

- Proces operacijskog sustava ima sljedeće elemente:



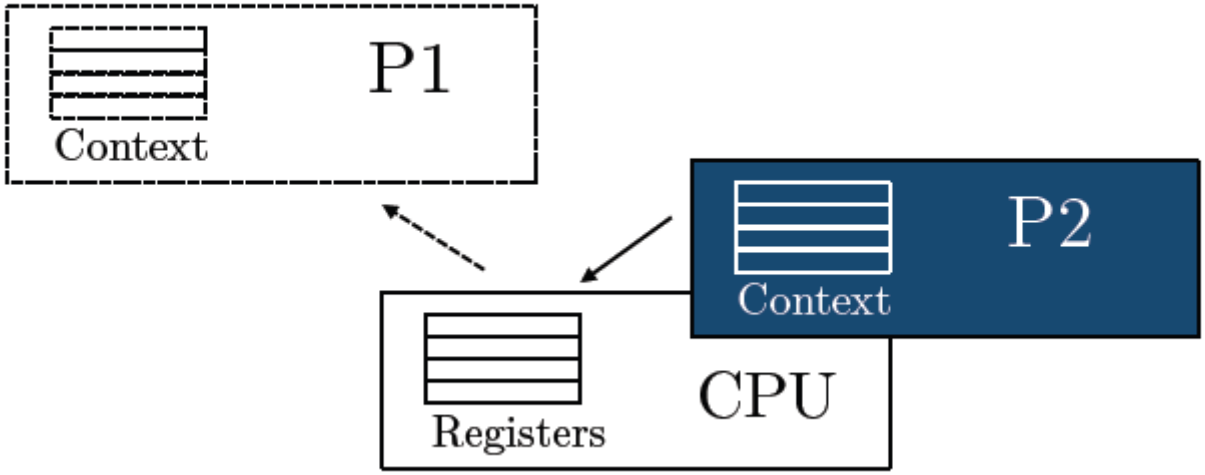
Konkurentno programiranje i operacijski sustavi

- OS koristi poseban program - **raspoređivač (scheduler)**, koji određuje kojem procesu treba dodijeliti (neki) procesor.
- Tranzicija stanja procesa operacijskog sustava:



Konkurentno programiranje i operacijski sustavi

- Zamjena (swapping) procesa koji se izvršavaju na CPU-u naziva se **zamjena konteksta (context switch)**.
- Proces P1 treba biti zamijenjen procesom P2. 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:

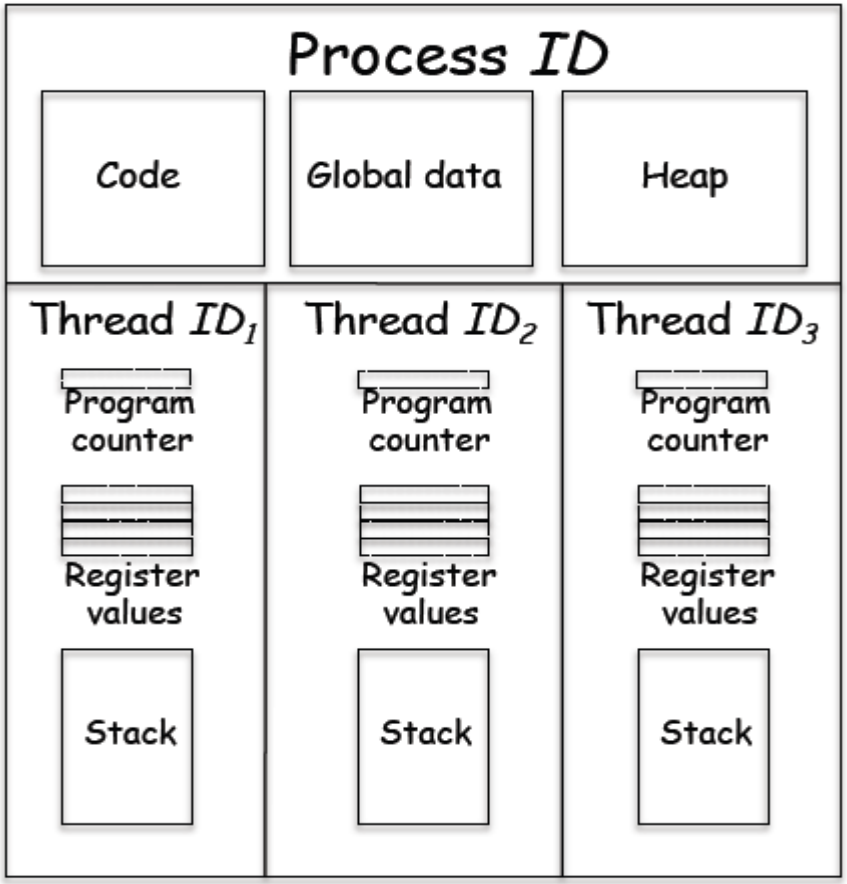


Konkurentno programiranje i operacijski sustavi

- Dobro je da se paralelno mogu izvršavati **i dijelovi istog programa** – **dretve** ili **niti (threads)**.
- 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.
- Zamjena konteksta dretvi u pravilu se izvodi **nekoliko puta brže** od zamjene konteksta procesa. Zbog toga svi današnji moderni OS nisu samo višeprocesni, nego i višedretveni.
- Dretve se mogu dodjeljivati procesorima isto kao i 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.

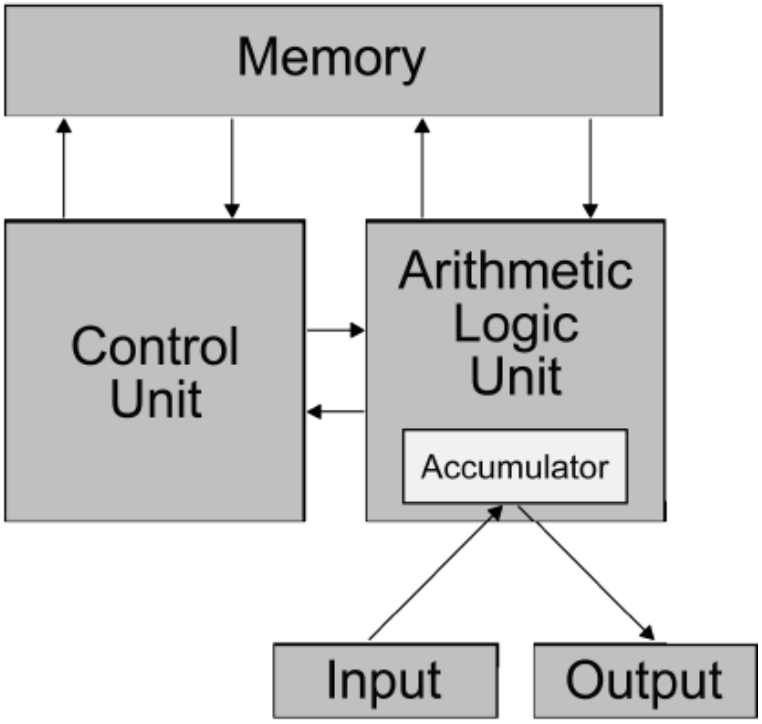
Konkurentno programiranje i operacijski sustavi

- Dretve dijele globalnu memoriju (programski kod i globalne podatke) i gomilu, ali imaju vlastiti stog i kontekst dretve:



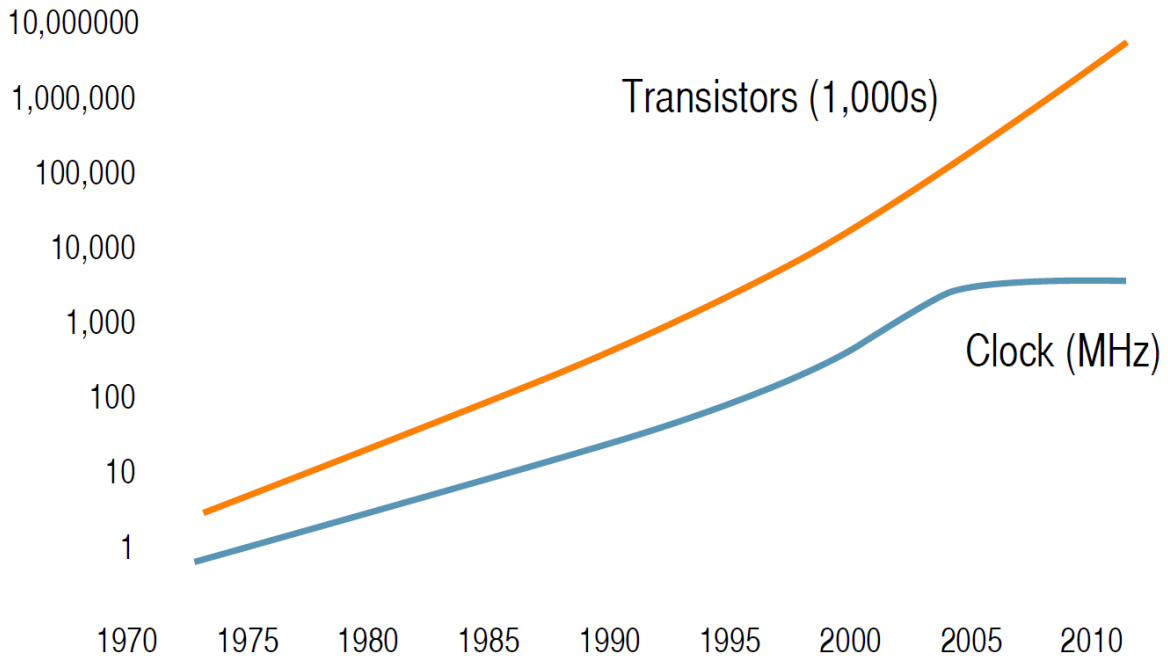
Utjecaj razvoja mikroprocesora na konkurentno programiranje

- Materijal "Not Your Father's Von Neumann Machine: A Crash Course in Modern Hardware kaže da je **Von Neumannov model računala** u današnje vrijeme **samo korisna apstrakcija**, koja odudara od stvarnih današnjih računala:



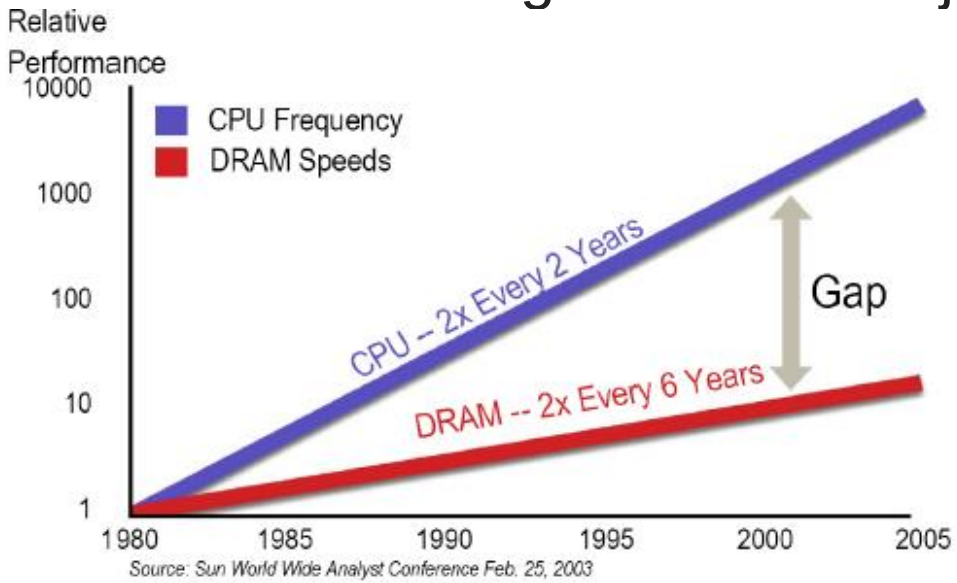
Utjecaj razvoja mikroprocesora na konkurentno programiranje

- **Mooreov zakon**: Broj tranzistora na mikroprocesoru udvostručuje se otprilike svake dvije godine.
- Mooreov zakon i dalje vrijedi. No, **radni takt procesora prestao je rasti** oko 2005. Razlog za to je veliko **povećanje potrošnje struje** na velikim brzinama.



Utjecaj razvoja mikroprocesora na konkurentno programiranje

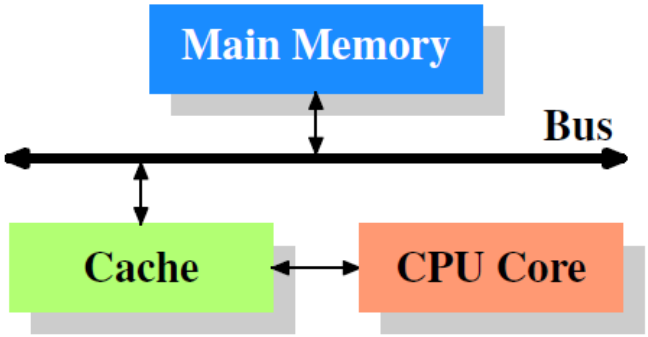
- Slika pokazuje kako se **eksponencijalno povećava jaz** između performansi CPU-a i glavne memorije (DRAM):

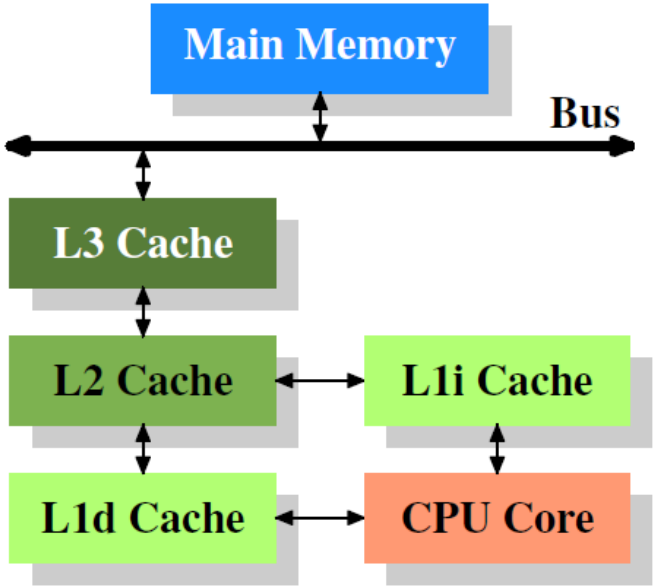


- Moguće je napraviti RAM (SRAM) koji bi bio brz skoro kao registri procesora, ali bi bio puno skuplji. A, **bolje je imati više sporije glavne memorije** nego manje brže glavne memorije, jer je magnetski disk puno sporiji.

Utjecaj razvoja mikroprocesora na konkurentno programiranje

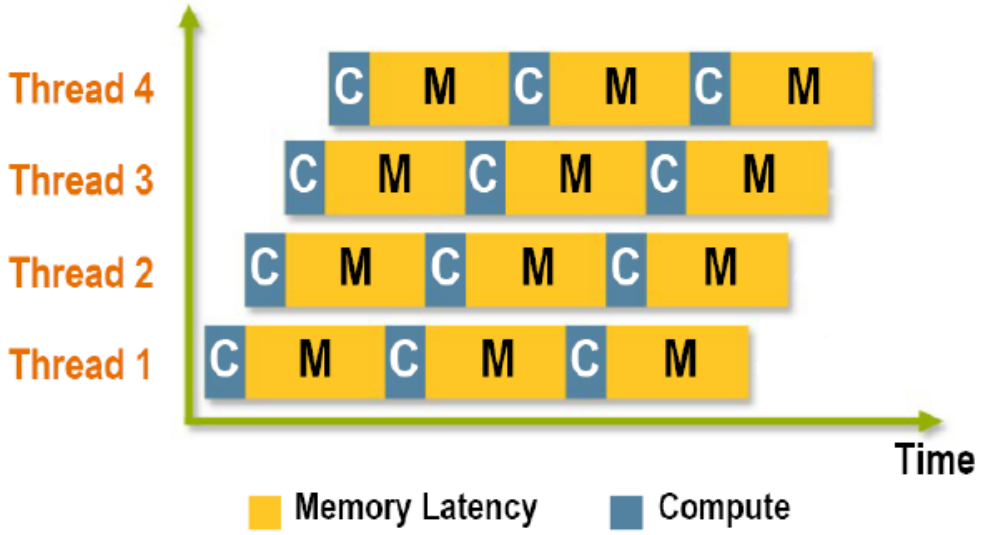
- Još je bolje napraviti **hijerarhiju memorija**, tj. koristiti bržu i manju **SRAM priručnu memoriju** (ili više razina te memorije) zajedno sa većom i sporijom DRAM glavnom memorijom. Dva primjera korištenja priručne memorije (oba prikazuju jednojezgreni procesor):





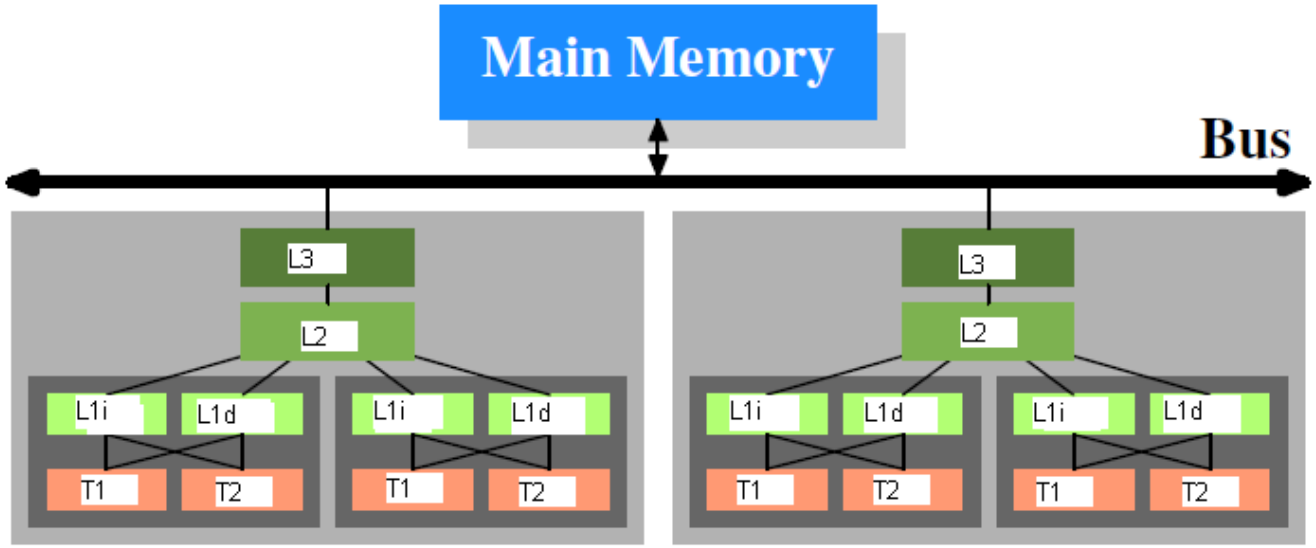
Utjecaj razvoja mikroprocesora na konkurentno programiranje

- Jedan od načina za povećanje performansi procesora je **paralelizam na razini instrukcija**, ILP (Instruction-Level Parallelism). Kasnije su se dosjetili: umjesto da paralelno obrađuju instrukcije istog programa, mogli bi paralelno obrađivati **instrukcije različitih programa** ili dretvi - **multithreading** (Intel koristi ime Hyper-Threading Technology, HTT):



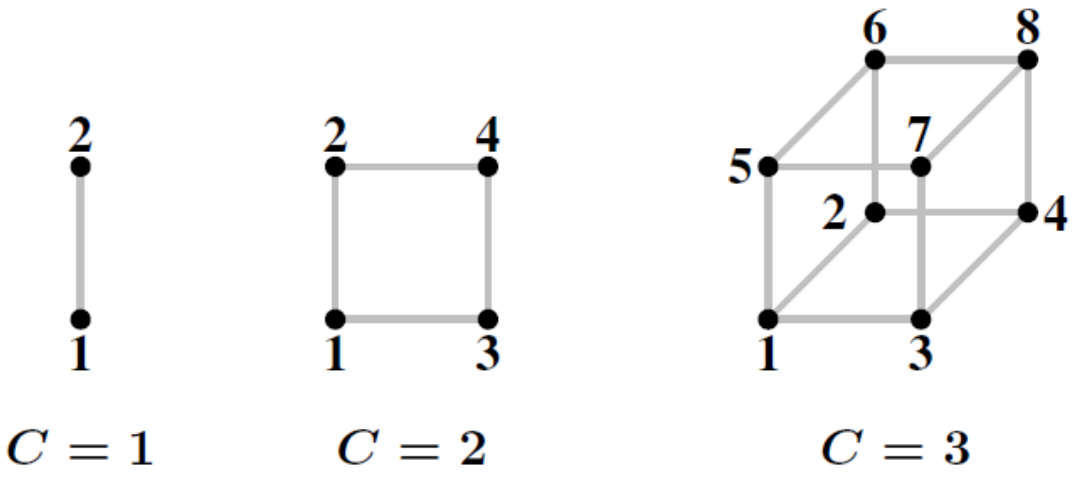
Utjecaj razvoja mikroprocesora na konkurentno programiranje

- Kod multithreadinga su duplirani samo neki elementi CPU-a, npr. procesorski registri. Zbog toga su **performanse do oko 1,3 puta veće**. Povećanje broja tranzistora počelo se koristiti za smještanje **više CPU-a (jezgri)** u jedan procesorski čip. Primjer sustava sa dva dvojezgrena procesora (svaka jezgra podržava dvije dretve):



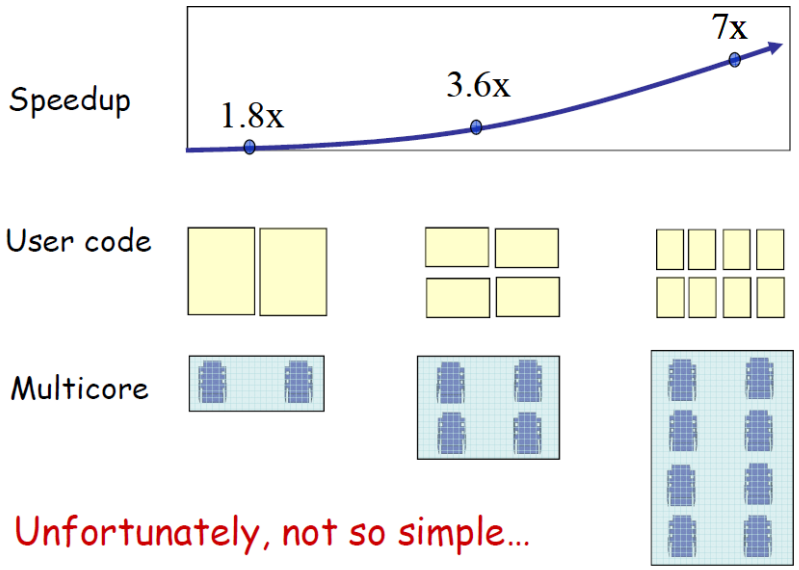
Utjecaj razvoja mikroprocesora na konkurentno programiranje

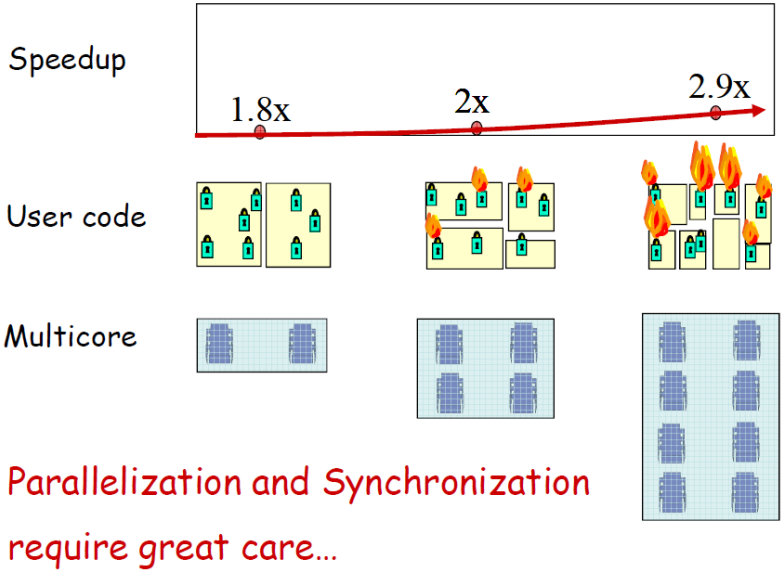
- Osim arhitekture centralne djeljive memorije (prethodna slika), postoji i arhitektura distribuirane DM. Neki kažu da je i to **SMP** (Symmetric Multi-Processors), ali najčešće se naziva **NUMA** (Non-Uniform Memory Access). Procesori nisu međusobno povezani (samo) preko memorijske sabirnice, nego su direktno ili indirektno povezani sa drugim procesorima (tzv. **Hyper Link**):



Utjecaj razvoja mikroprocesora na konkurentno programiranje

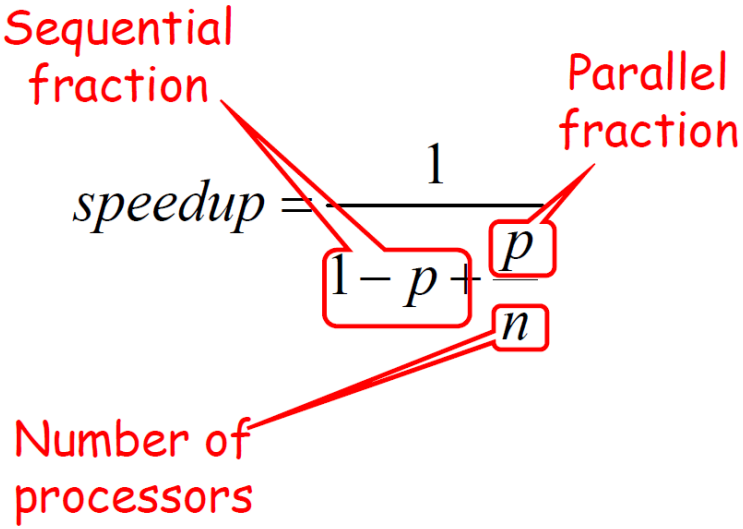
- **Povećanje radnog takta** u pravilu je davalo skoro **linearno povećanje performansi**.
- Nažalost, kod višejezgrenih / višeprocesorskih sustava rast performansi rijetko raste proporcionalno sa povećanjem broja jezgri, već je **rast općenito manji**:





Utjecaj razvoja mikroprocesora na konkurentno programiranje

- Razlog za to objašnjava **Amdahlov zakon**. Ako je u nekom programu **proporcija dijelova programa koji se mogu paralelno izvršavati** jednaka **p**, onda se povećanjem broja CPU-a može dobiti ovo povećanje:



- Npr. ako imamo **10 procesora** i **p = 90%** programa, onda je maksimalno povećanje brzine **5,26 puta**.

Sinkronizacijski algoritmi i mehanizmi

- **Lokoti**, tj. blokirajuća sinkronizacija
- **Neblokirajući sinkronizacijski mehanizmi**, bez lokota (lock-free)
- **Softverska, hardverska i hibridna transakcijska memorija** (STM, HTM, hibridna TM)

Pritom se koriste sljedeće **vrste lokota**:

- test-and-set (TAS) lokoti
- test-and-test-and-set (TATAS) lokoti
- lokoti temeljeni na redovima (queue-based locks)
- hijerarhijski lokoti
- lokoti tipa čitatelj-pisac (reader-writer locks)

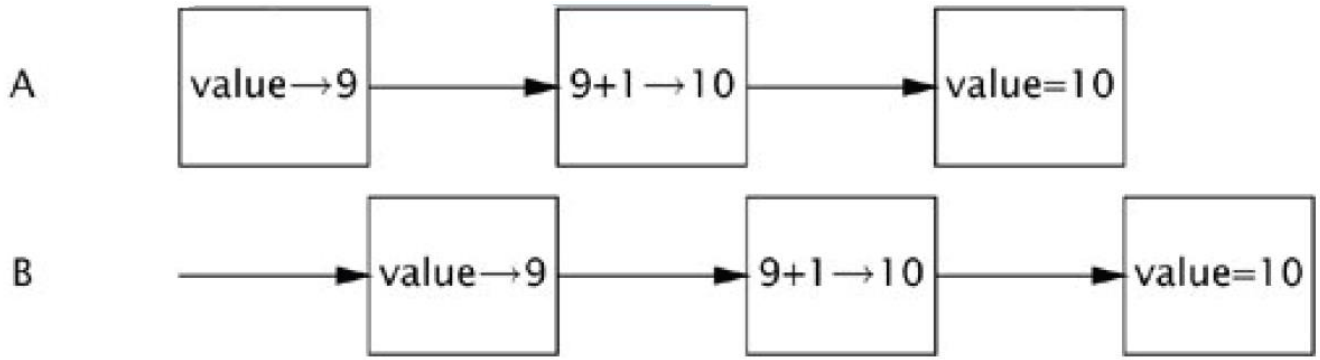
Sinkronizacijski algoritmi i mehanizmi

- Pretpostavimo da smo napisali sljedeći programski kod za računanje sekvence cijelih brojeva:

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

Sinkronizacijski algoritmi i mehanizmi

- 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 isti broj 10**:



- Problem je u tome što **naredba value++ kod izvođenja nije atomarna**, već se razlaže na (uobičajeno) tri interne naredbe: temp = value; temp = temp + 1; value = temp.

Sinkronizacijski algoritmi i mehanizmi

- Prethodni program nije korektan, jer njegova točnost ovisi o međusobnom redoslijedu izvođenja naredbi dva (ili više) programa. Taj se problem uobičajeno **race conditions**.
- Da bismo riješili taj problem, **dretve moramo sinkronizirati**. Sinkronizacija se zasniva na ideji da **dretve komuniciraju** jedna sa drugom kako bi se "dogovorile" o sekvenci akcija.
- 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.

Sinkronizacijski algoritmi i mehanizmi

- 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): dio programa koji pristupa djeljivom resursu.

Sinkronizacijski algoritmi i mehanizmi

- 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 dva algoritma za implementaciju lokota: **Petersonov algoritam** za dvije dretve, te **Lamportov Bakery (= pekara) algoritam** za n dretvi.
- Petersonov algoritam je **deadlock-free** i **starvation-free**. Lamportov algoritam je deadlock-free, ali i **prvi-došao-prvi-obrađen**, što je jače nego starvation-free.
- No, oba algoritma temeljena na **radnom čekanju** (busy waiting), što se često naziva i **obrtanje (spinning)**.

Sinkronizacijski algoritmi i mehanizmi

- **Spinning je naročito neefikasan kod multitaskinga**, pa ima smisla samo tamo gdje je vrijeme čekanja tipično vrlo kratko, pa bi zamjena konteksta bila skuplja od njega.
- **Spin locks** se često koriste kod jezgre (kernela) OS-a. 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, Lamportov algoritam nije praktičan i stoga što ima **potrebu čitanja i pisanja u n različitih lokacija za n dretvi, a bolji algoritam ne postoji**.
- To pokazuje da je potrebno **uvesti sinkronizacijske mehanizme koji su jači od čitaj-piši** (read-write), i koristiti ih za izradu algoritama za međusobno isključivanje dretvi.

Sinkronizacijski algoritmi i mehanizmi

- Lokote bi trebalo implementirati sa sljedećim **atomarnim osnovnim operacijama** (atomic primitives) za sinkronizaciju, koje su kompleksnije od operacija atomarnog čitanja-pisanja. **One su realizirane na razini procesora** (ovo je pseudokod):

```
test-and-set (x, value) --stara operacija
do temp := x; x := value; result := temp end
compare-and-swap (x, old, new) --nova operacija
do
   if x = old then x := new; result := true
   else result := false
   end
end
```

Sinkronizacijski algoritmi i mehanizmi

- Operacija **test-and-set** (**TAS**) bila je osnovna operacija za sinkronizaciju **u mikroprocesorima 1990-tih godina**.
- 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!**
- Zanimljiv je pojam **broj konsenzusa** (consensus number), a to je maksimalni broj procesa za koje određena primitivna operacija (konsenzus objekt) može implementirati problem konsenzusa.
- Atomarni registri imaju broj konsenzusa 1, TAS ima ima broj konsenzusa 2, **a CAS ima beskonačni broj konsenzusa**. **CAS je osnova (današnjeg) konkurentnog programiranja**.

Sinkronizacijski algoritmi i mehanizmi

- Zaključavanje, bez obzira na varijantu, ima i mana. Npr., kada više dretvi istovremeno traže isti lokot, JVM najčešće treba "pomoć" OS-a, pri čemu dretve najčešće bivaju suspendirane.
- Ako je dretva koja drži lokot odgođena na duže vrijeme, nijedna dretva koja treba taj lokot ne može napredovati.
- Najgore je: **"Nitko stvarno ne zna kako organizirati i održavati veliki sustav temeljen na zaključavanju"**.
- Srećom, CAS operacija za sinkronizaciju 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). Zato se pomoću nje mogu graditi **neblokirajući (nonblocking) algoritmi** – bez lokota.

Sinkronizacijski algoritmi i mehanizmi

- 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 konkurentnih algoritama i struktura podataka. Npr. **U Javi 5, a naročito 6**, implementirani su (pomoću CAS) algoritmi i strukture podataka koji se u konkurentnom radu ponašaju dramatično **brže od Java 4** i prethodnih verzija.
- No, algoritme koji koriste CAS vrlo je teško smisliti i često su vrlo neintuitivni. Osnovna teškoća sa svim današnjim sinkronizacijskim operacijama (pa i CAS) je da **one rade na samo jednoj riječi memorije, što tjera na korištenje kompleksnih i neprirodnih algoritama**.

Sinkronizacijski algoritmi i mehanizmi

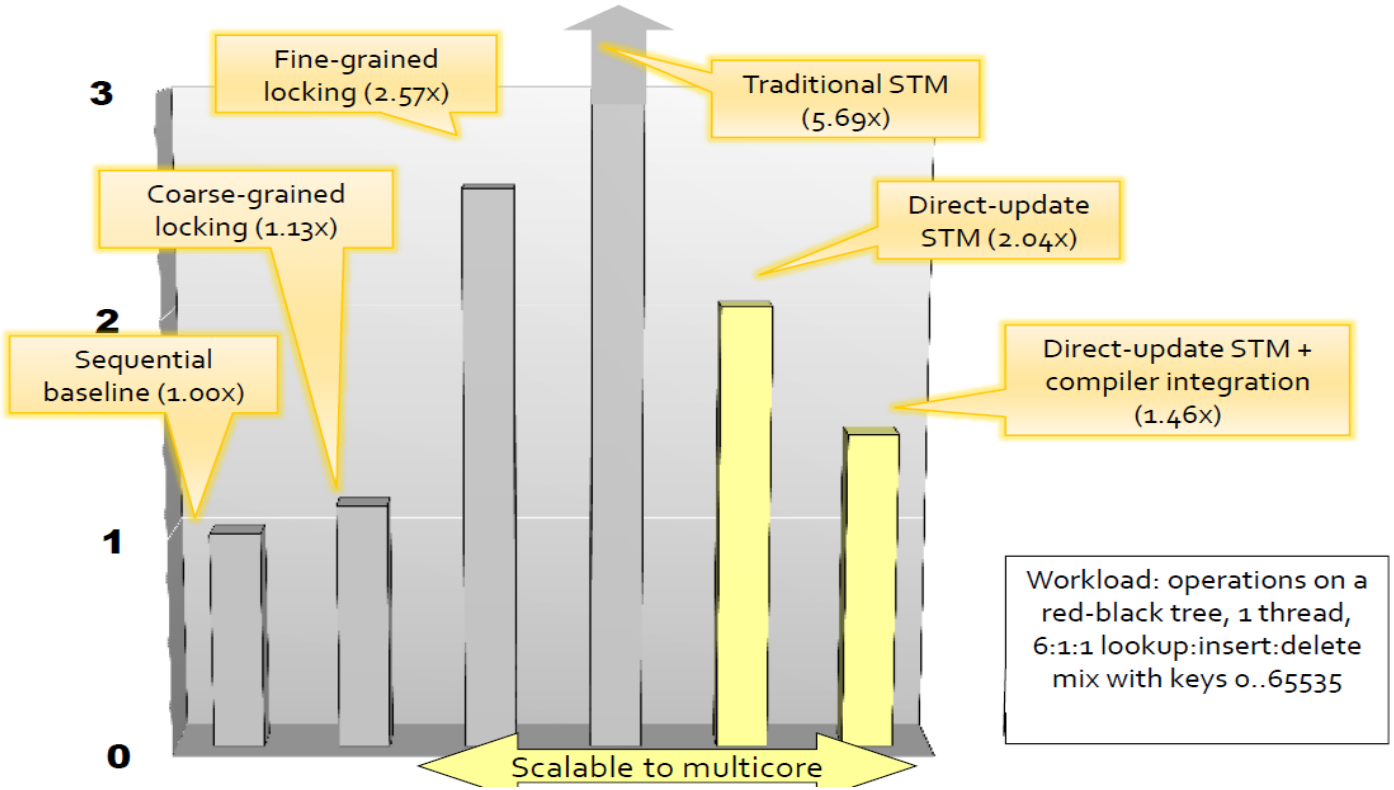
- Zato je izmišljena **transakcijska memorija** (TM), a njena realizacija može biti softverska (STM), hardverska (HTM) ili hibridna.
- Transakcija je sekvenca koraka koje izvršava jedna dretva. Transakcije moraju biti **serijabilne** (serializable), što znači da mora izgledati kao da se izvršavaju sekvencijalno i onda kada se izvršavaju paralelno.
- Serijabilnost je na jača varijanta **linearizabilnosti** (linearizability). Linearizabilnost definira atomarnost individualnog objekta. **Serijabilnost definira atomarnost cijele transakcije**.
- Ispravno implementirane, **transakcije nemaju problem deadlocka ili livelocka**.

Sinkronizacijski algoritmi i mehanizmi

- Često se kaže da se transakcijska memorija ponaša slično kao transakcije u bazi podataka.
- No, za razliku od transakcija u bazi podataka, kod kojih se radi zaključavanje redaka tablica, **kod transakcijske memorije nema zaključavanja**, tako da se nikad ne može desiti deadlock.
- **Transakcijska memorija se ponaša optimistički**, tj. polazi od toga da problema u ažuriranju neće biti. Ako se na kraju ipak pokaže da ima problema, transakcija radi rollback.
- Napomenimo da **deadlock u bazi podataka nije problem**, jer DBMS detektira deadlock i poništava jednu od sesija koja se nalazi u deadlock konfliktu.

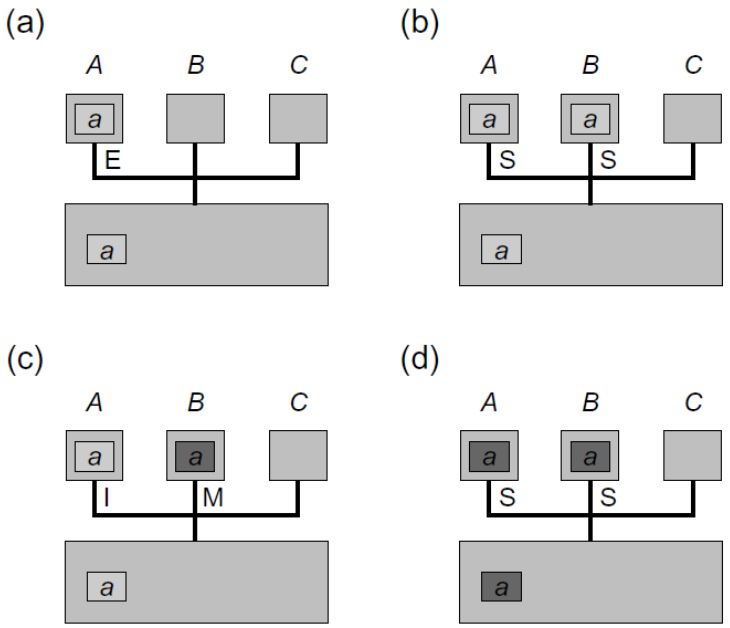
Sinkronizacijski algoritmi i mehanizmi

- **STM** na razini jezika podržava npr. jezik **Clojure**, a na razini biblioteke jezik **Scala**.
- Poboljšanja kod današnjih realizacija STM-a:



Sinkronizacijski algoritmi i mehanizmi

- Danas barem tri mikroprocesora podržavaju **HTM**:
  Vega firme Azul Systems (već 10 godina)
  BlueGene/Q firme IBM (4 godine)
  Intel Haswell (2 godine).
- HTM koristi tzv. **MESI protokol**:
  - **Modified**: linija cache-a je modificirana;
  - **Exclusive**: linija nije modificirana, i nijedan drugi procesor ju nema;
  - **Shared**: linija nije modificirana, ali drugi procesori ju mogu imati;
  - **Invalid**: linija ne sadrži suvisle podatke.

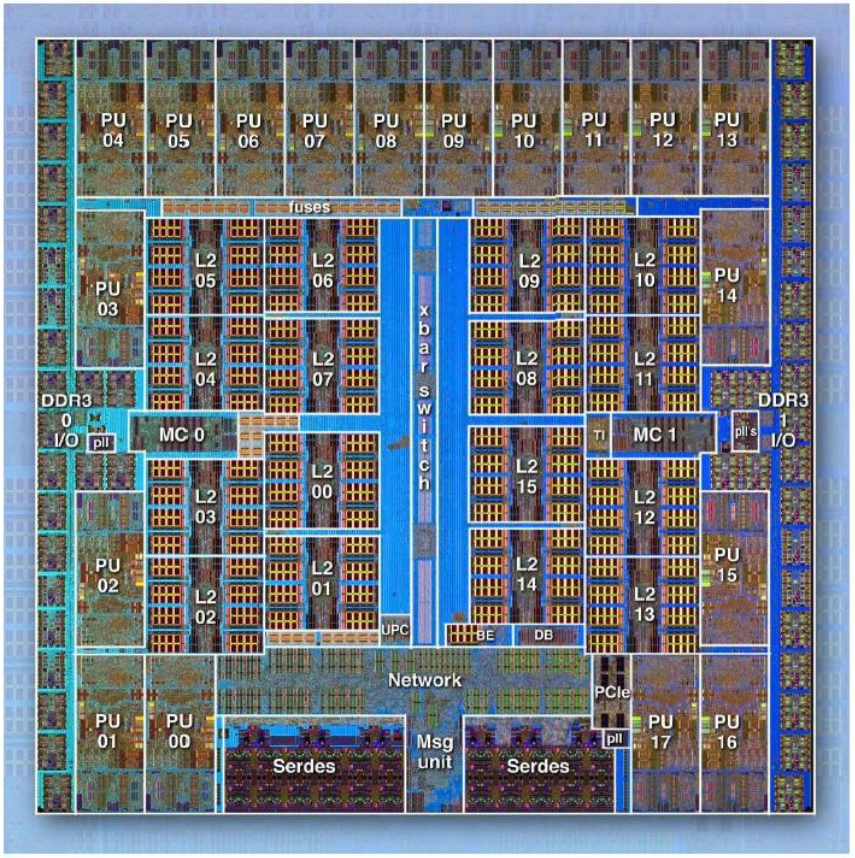


Sinkronizacijski algoritmi i mehanizmi

- Procesor IBM BlueGene/Q ima hardversku transakcijsku memoriju
- Ima 18 jezgri
  - jedna je namijenjena za OS,
  - jedna je rezerva
- Superračunalo Sequoia

ima 100 000 procesora,

tj.1 800 000 jezgri !



Sinkronizacijski algoritmi i mehanizmi

- IBM BlueGene/Q mikroprocesor je ipak prvenstveno namijenjen za izradu superračunala, pa je za "obično" programiranje značajniji **Intel mikroprocesor Haswell arhitekture koji isto podržava HTM** (ne u svim verzijama).
- Time se **HTM uvodi u masovniju primjenu** i vjerojatno će značajno olakšati pisanje nekih konkurentnih programa.
- Za razliku od mikroprocesora IBM BlueGene/Q, Intelovi mikroprocesori podržavaju transakcijsku memoriju i između mikroprocesorskih čipova, a ne samo između jezgri jednog čipa.
- Intelovi Haswell mikroprocesori zanimljivi su i po tome što imaju **dva načina podržavanja transakcijske memorije.**

Sinkronizacijski algoritmi i mehanizmi

- **Hardware Lock Elision** način omogućava da se **programi koji koriste zaključavanje vrlo lako prerade tako da koriste HTM**. Suština je u tome da procesor "laže" dretve da su dobile lokot, a da stvarno nikakvog zaključavanja nema. Takvi programi rade i na "starim" procesorima bez HTM-a, ali sa zaključavanjem.
- **Novonapisani programi** mogu koristiti napredniji način **Restricted Transactional Memory**, tj. Mogu koristiti eksplicitne naredbe **XBEGIN**, **XEND** ili **XABORT** za pokretanje transakcije, normalno završavanje ili abortiranje. Kod pokretanja nove transakcije (sa XBEGIN) dretva može navesti i "fallback" rutinu koja se automatski izvršava ako transakcija ne uspije.

HTM poezija

Među programerskim pukom
zavladala silna euforija
– uskoro stiže nam trkom
hardverska transakcijska memorija.

Konkurentno programiranje u Javi verzije 1 – 4

- **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 GUI-em.
- Kada koristimo **servlete ili RMI**, oni kreiraju pričuvu (pool) dretvi. Zato, kada koristimo te frameworke, moramo biti upoznati sa konkurentnošću u Javi.
- 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**.

Konkurentno programiranje u Javi verzije 1 – 4

- 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:

```
class Worker1 extends Thread {
  public void run() { // impl.doTask1() here}
}

class Worker2 extends Thread {
  public void run() {// impl.doTask2() here}
}
```

Konkurentno programiranje u Javi verzije 1 – 4

- U nastavku se prikazuje implementacija metode compute() iz neke treće klase, koja kreira dvije dretve, tako da **dva posla (tasks) mogu biti izvedena paralelno** (ako postoje dva slobodna CPU-a koja ih izvode):

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

Konkurentno programiranje u Javi verzije 1 – 4

- Pretpostavimo da želimo dobiti rezultat obje dretve:

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

- Očito, **moramo čekati da obje dretve završe** prije nego dobijemo taj rezultat. To se postiže sa **join()**:

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

Konkurentno programiranje u Javi verzije 1 – 4

- 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).
- **"Višestruko nasljeđivanje klasa je korisno, koliko god mi šutjeli o tome :)"** (odnosno - koliko god nas pokušavali uvjeriti u suprotno).
- Bez višestrukog nasljeđivanja klasa, ako naša klasa "potroši" jednostruko nasljeđivanje za klasu Thread, **ne može naslijediti od neke druge klase**.

Konkurentno programiranje u Javi verzije 1 – 4

- Prvo se napiše **"pomoćna" klasa koja nasljeđuje sučelje Runnable**, koja mora **implementirati metodu run()**:

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

- A onda se objekt te "pomoćne" klase šalje konstruktoru:

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

Konkurentno programiranje u Javi verzije 1 – 4

- Problemi nastaju kada se dvije dretve upliću jedna drugoj u posao, npr. tako da modificiraju isti objekt. To može stvoriti netočne rezultate i naziva se **race condition**:

```
class Counter {
  private volatile int value = 0;
  public int getValue() {return value;}
  public void setValue(int someValue) {
    value = someValue;
  }
  public void increment() {value++}
}
```

Konkurentno programiranje u Javi verzije 1 – 4

- Pretpostavimo da neka metoda u drugoj klasi radi:

  x.setValue(0); x.increment(); int i = x.getValue();
- **Koju vrijednost ima varijabla i na kraju ovih naredbi?** Za jednodretveni program, odgovor je 1.
- U konkurentnom radu brojač može biti modificiran od drugih dretvi, tako da rezultat ovisi o ispreplitanju naredbi ove dretve sa naredbama neke druge dretve.
- Taj se problem rješava pomoću sinkronizacije koja se zove **međusobno isključivanje** (mutual exclusion).
- **Svaki objekt u Javi ima lokot (lock)**. Nasljeđuje se automatski od superklase Object. Lokot istovremeno **može držati (zaključati) samo jedna dretva**.

Konkurentno programiranje u Javi verzije 1 – 4

- Objekt koji će služiti kao lokot može se kreirati ovako:

  Object lock = new Object();

- Dretva koja traži lokot to radi pomoću naredbe **synchronized**, koja označava početak **synchronized bloka**:

  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**.

Konkurentno programiranje u Javi verzije 1 – 4

- Problem iz prethodnog primjera mogli bismo riješiti pomoću synchronized ovako:

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

Konkurentno programiranje u Javi verzije 1 – 4

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

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

što je isto kao i ovo:

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

Konkurentno programiranje u Javi verzije 1 – 4

- 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 čest u praksi (u različitim varijantama).

Konkurentno programiranje u Javi verzije 1 – 4

- **Proizvođač** u svakoj iteraciji (beskonačne) petlje **proizvodi podatke** koje potrošač konzumira. **Potrošač** u svakoj iteraciji petlje **troši podatke** koje je proizveo proizvođač.
- **Proizvođači i potrošači komuniciraju preko djeljivog međuspremnika (buffer)** koji implementira red (queue). U ovom primjeru smatramo da je red neograničen.
- 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()**.

Konkurentno programiranje u Javi verzije 1 – 4

❖ Pretpostavimo da u klasi Consumer imamo sljedeće:

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

❖ 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.

Konkurentno programiranje u Javi verzije 1 – 4

- 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.

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

Konkurentno programiranje u Javi verzije 1 – 4

- Sada lokot može zaključati neka druga dretva, koja može mijenjati stanje navedene kondicije. Da obavijesti prvu dretvu o promjeni kondicije, **druga dretva poziva notify()**.
- To 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():

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

Konkurentno programiranje u Javi verzije 1 – 4

- Odblokirana dretva ne može biti sigurna da je 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 dretva potrošač mora ispitivati uvjet u while petlji. Važno je i to što **notify() uvijek odblokira samo jednu dretvu** koja čeka na određeni lokot (bilo koju!). 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). **U Javi 5 postoji njihova generalizacija**, koja omogućava da programeri eksplicitno rade sa kondicijama.

Konkurentno programiranje u Javi verzije 1 – 4

- Na kraju, prikažimo kako može nastati **deadlock**, kada se dvije dretve (ili grupa dretvi) blokiraju zauvijek:

```
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) {...}
      }
   }
}
```

Konkurentno programiranje u Javi verzije 1 – 4

- Sada se izvodi npr. sljedeći kod, gdje su a1 i b1 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. **Java sustav 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 tako da ne dođe do deadlocka.

Konkurentno programiranje u Javi verzije 5 – 7

- U Javi verzije 5, koja se pojavila 2004. godine, uvedeno je dosta novina na drugim područjima (generičke klase, bolje kolekcije i dr.), ali i na području konkurentnog programiranja.
- Kroz novi paket **java.util.concurrent** uvedene su novosti:

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

Konkurentno programiranje u Javi verzije 5 – 7

- U **Java verziji 6**, koja je izašla 2006., nije se pojavilo ništa revolucionarno, uglavnom su se "iznutra" poboljšale biblioteke, tj. **riješili bugovi ili poboljšale performanse**.
- U **Java verziji 7**, koja je izašla 2011., u području konkurentnog programiranja najveća novost je **Fork/Join Framework**.
- U **Java verziji 8**, koja je izašla 2014., u području konkurentnog programiranja najveća novost je **Streams** (**lambda izrazi** i **default metode** su, na neki način, posljedica uvođenja Streams-a, ali su korisne i zasebno).
- Ukratko će se prikazati samo neke mogućnosti uvedene u Javi 5 (sa poboljšanjima u Javi 6), i to **ReentrantLock, ReadWriteLock, Conditions i Atomic variables.**

Konkurentno programiranje u Javi verzije 5 – 7

- Do Java 5 verzije, jedini mehanizmi za koordinaciju pristupa djeljivim podacima bili su synchronized i volatile. Java 5 uvela je klasu **ReentrantLock** koja implementira sučelje Lock:

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

Konkurentno programiranje u Javi verzije 5 – 7

- Zaključavanja pomoću objekata klase ReentrantLock ima istu semantiku kao i synchronized, ali ima i dodatne mogućnosti. Najčešći oblik korištenja:

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

Konkurentno programiranje u Javi verzije 5 – 7

- 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();}
}
```

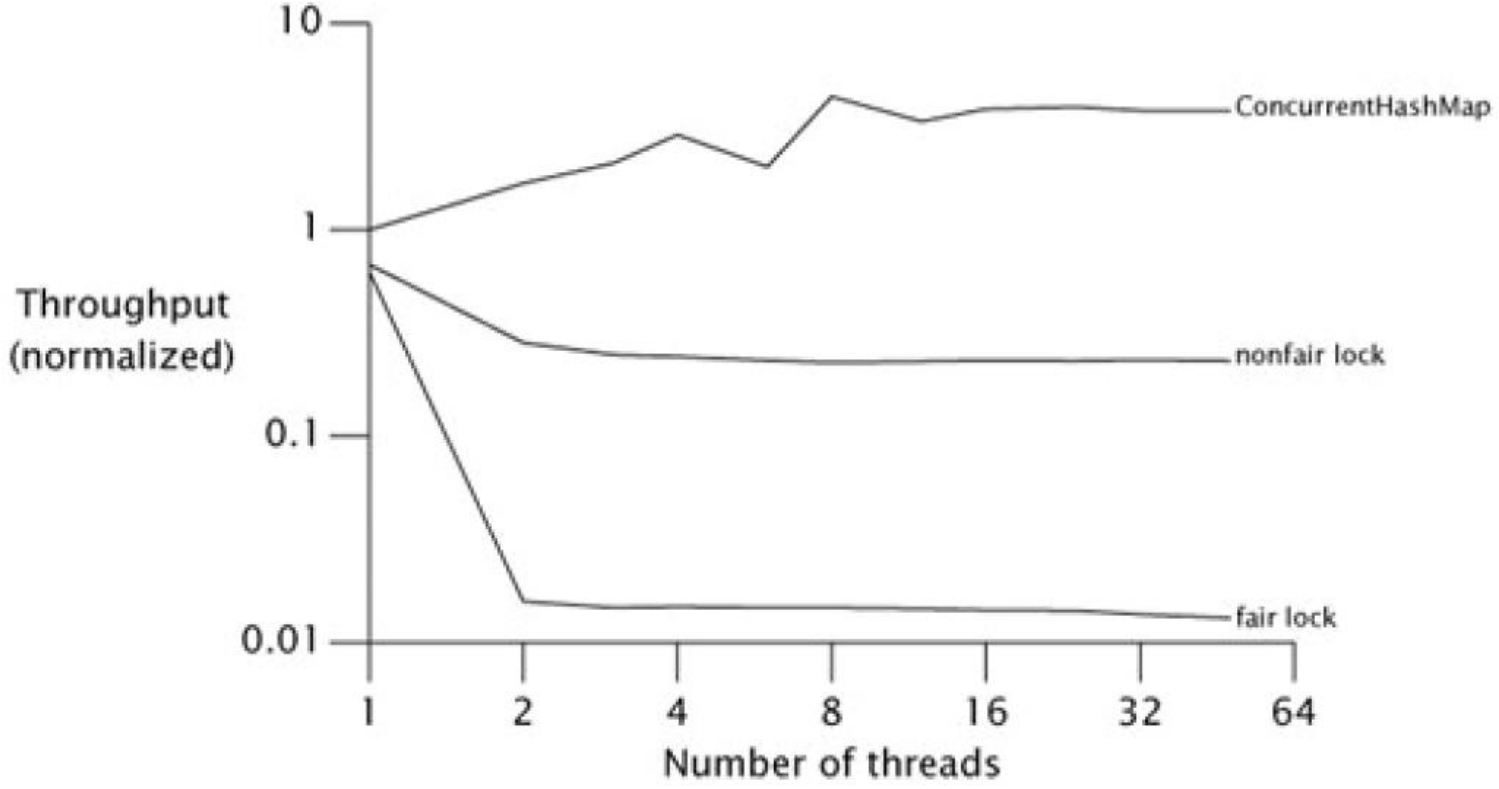
Konkurentno programiranje u Javi verzije 5 – 7

❖ Korištenje metode **lockInterruptibly**():

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

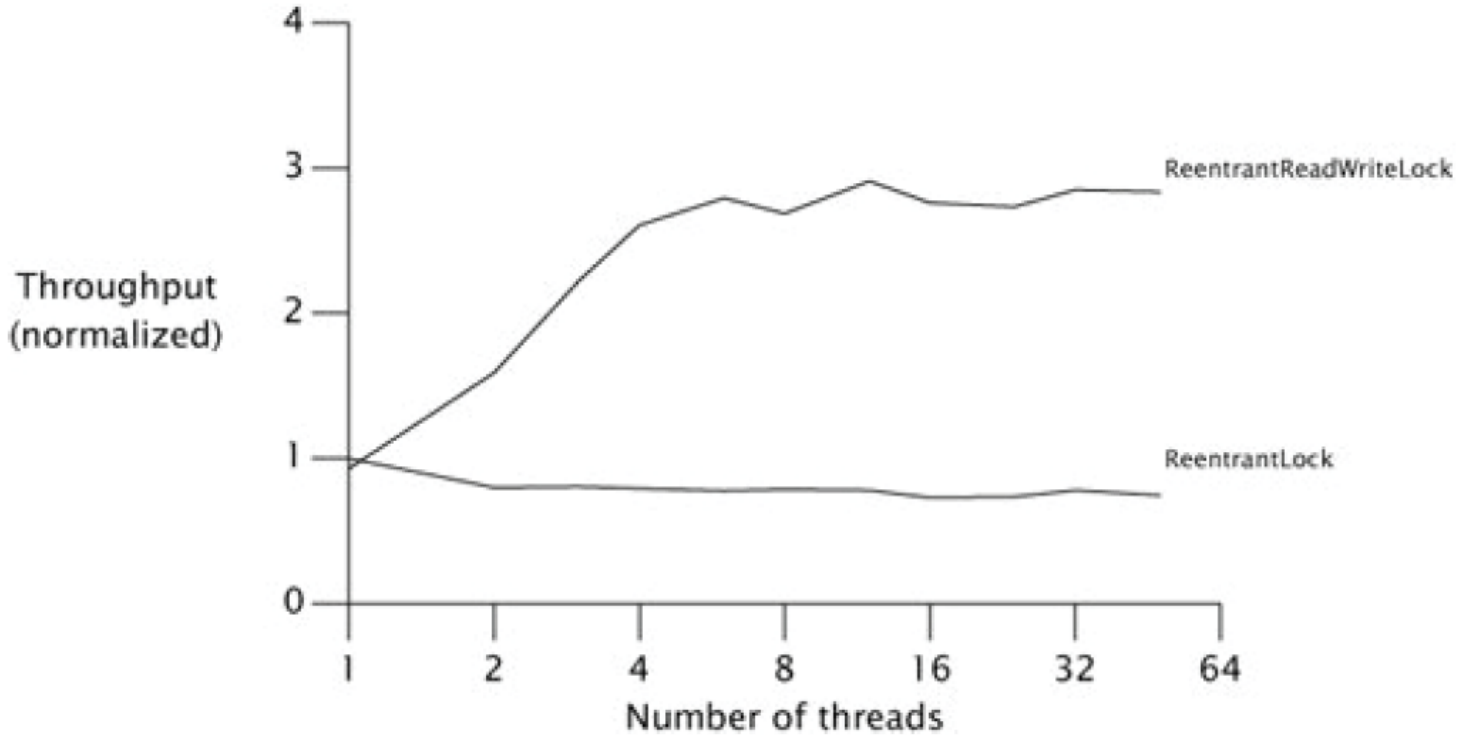
Konkurentno programiranje u Javi verzije 5 – 7

- **ReentrantLock** lokoti mogu biti **fer** i **ne-fer** (default) **lokoti**. Fer lokoti osiguravaju da dretve dobivaju lokote po redoslijedu prispijeća zahtjeva.



Konkurentno programiranje u Javi verzije 5 – 7

- Postoje i lokoti klase **ReentrantReadWriteLock**: istovremeno može raditi više čitatelja koji blokiraju pisce, ali može raditi samo jedan pisac, koji blokira čitatelje i (druge) pisce.



Konkurentno programiranje u Javi verzije 5 – 7

- 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.

Konkurentno programiranje u Javi verzije 5 – 7

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

```
class BoundedBuffer {
   Lock lock = new ReentrantLock();
   //Povezivanje jednog lokota i dvije kondicije
   Condition notFull = lock.newCondition();
   Condition notEmpty = lock.newCondition();

     Object[] items = new Object[100];
   int putptr, takeptr, count;
```

Konkurentno programiranje u Javi verzije 5 – 7

```
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();}
}
```

Konkurentno programiranje u Javi verzije 5 – 7

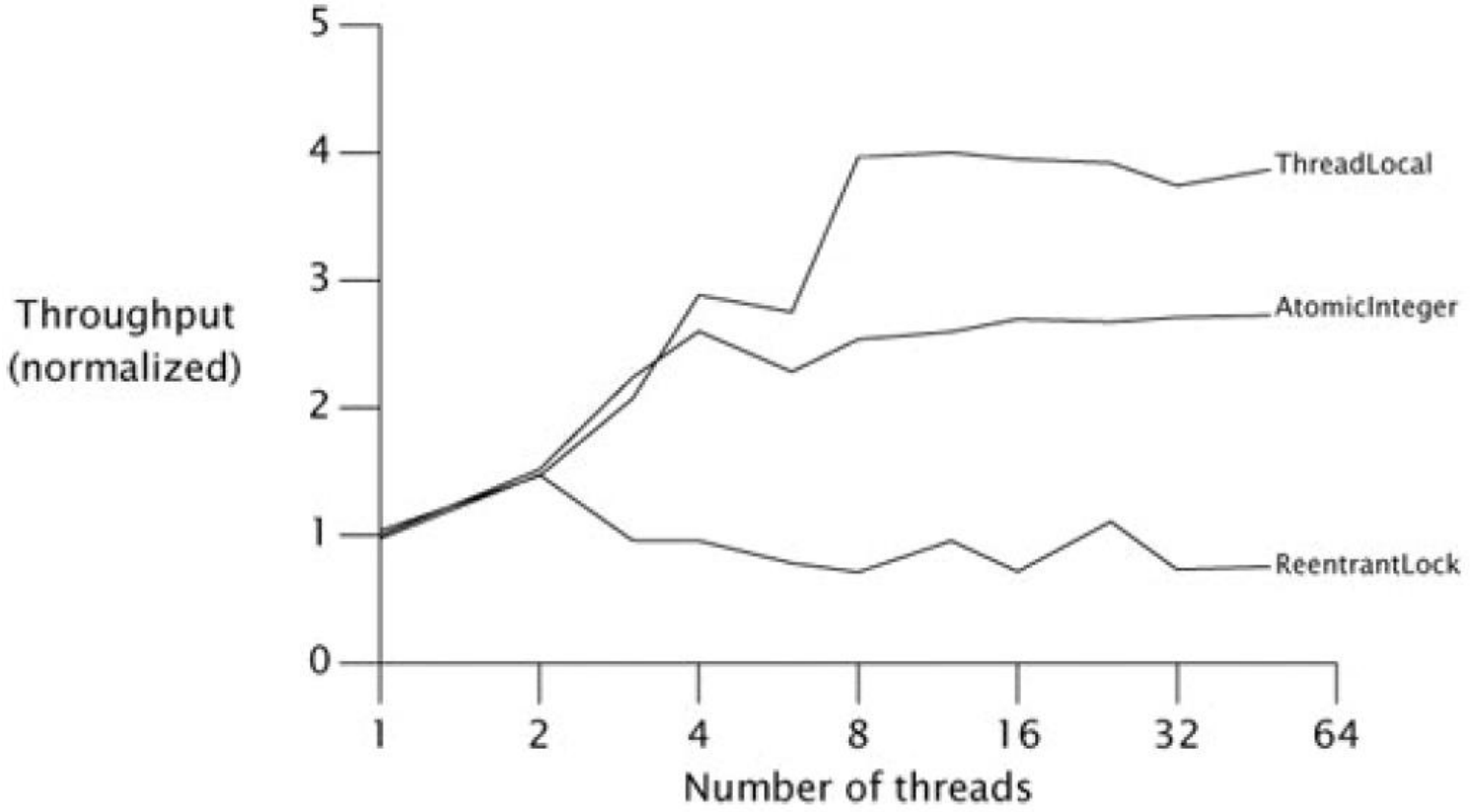
```
public Object take() throws IE {
  lock.lock();
  try {
    while (count == 0) notEmpty.await();
    Object x = items[takeptr];
    if (++takeptr == items.length) takeptr = 0;
    --count;
    notFull.signal();
    return x;
  } finally {lock.unlock();}
}
```

Konkurentno programiranje u Javi verzije 5 – 7

- **Atomarne varijable** su na neki način generalizacija volatilnih varijabli (volatile variables) koje su postojale i prije Jave 5.
- 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, 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**.

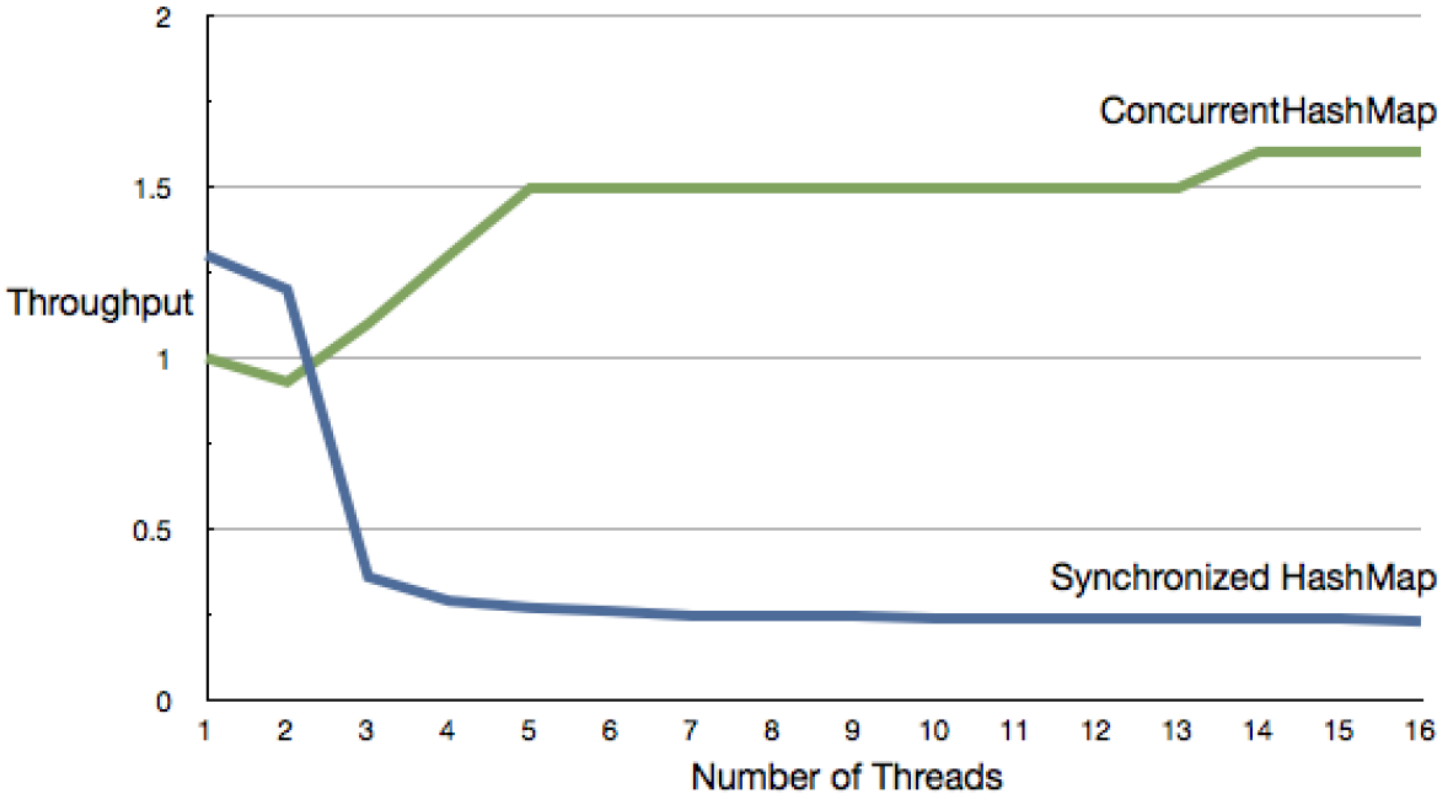
Konkurentno programiranje u Javi verzije 5 – 7

- Propusnost za ReentrantLock, atomarne varijable i tzv. **ThreadLocal varijable** (nešto slično kao kada u BP sesija čita svoju vlastitu verziju podataka u UNDO tablespaceu):



Konkurentno programiranje u Javi verzije 5 – 7

- Usporedba propusnosti ConcurrentHashMap i sinkronizirane kolekcije (8 - jezgreni procesor):



Java 8 - najvažnije nove mogućnosti: Streams, lambda, default metode

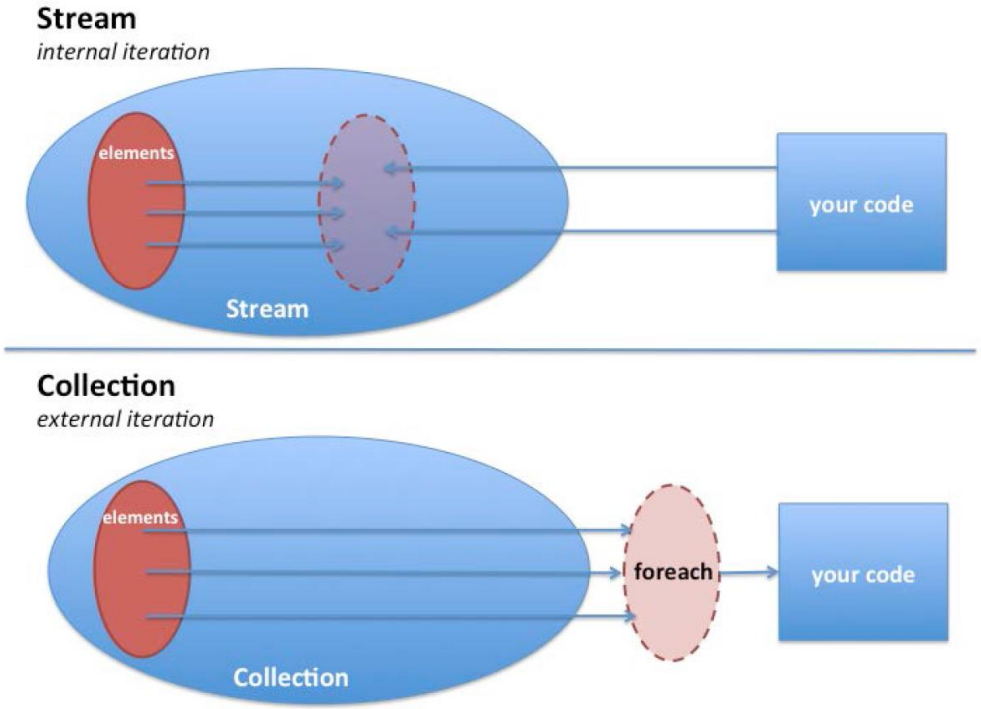
- Na temelju onoga što čitamo i čujemo, mogli bismo zaključiti da je najvažnija nova mogućnost u Javi 8 **lambda izraz** (ili kraće, **lambda**).
- Inače, lambda izraz je (u Javi) naziv za metodu bez imena. U pravilu je ta metoda funkcija, a ne procedura. Zato možemo reći i da **lambda izraz je anonimna funkcija**, koja se može javiti kao parametar (ili povratna vrijednost) druge funkcije (koja je, onda, funkcija višeg reda).
- U Javi 8 pojavile su se i tzv. **default metode** u Java sučeljima (interfaces). One, zapravo, predstavljaju uvođenje **višestrukog nasljeđivanja implementacije** u Javu.
- Međutim, lambda izrazi i default metode su, na neki način, posljedica uvođenja treće važne mogućnosti u Javi 8, a to su **Streamsi**, koji nadograđuju dosadašnje Java kolekcije.

Java 8 Streams (java.util.stream) - zašto su potrebni?

- Pojava **masivno višejezgrenih procesora** traži bolji i lakši način izrade paralelnih programa od dosadašnjih načina. Jedan (dobar) način je da se koristi funkcijsko programiranje, a naročito paralelne kolekcije.
- Java nema paralelne kolekcije (collections), ali su zato uvedeni Streamsi, kako bi se postojeće kolekcije "zaogrnule" u (paralelne) Streamse i mogle (indirektno) paralelizirati.
- **Kako bi se olakšalo korištenje Streamsa, bilo je potrebno uvesti i lambda izraze i default metode**. Međutim, lambda izrazi i default metode mogu biti korisni i za ostale svrhe, ne samo tvorcima API-a, već i "običnim" programerima.
- Za razliku od dosadašnjih kolekcija, koje sadrže sve svoje elemente u memoriji, Streamse možemo shvatiti kao **"vremenske kolekcije"**, čiji se elementi (kojih teoretski može biti i beskonačan broj) stvaraju po potrebi.

Java 8 – razlika između eksterne i interne iteracije

- "Stare" Java kolekcije koriste eksternu (eksplicitnu) iteraciju, koju piše programer. Streamsi imaju internu iteraciju.
- **Streamsima se može poslati naš kod, kao lambda izraz**, za definiranje obrade elemenata:



Java 8 Streams programiranje je deklarativno (što, ne kako), kao SQL

```
// 1. "klasična" obrada "klasične" kolekcije - for petlja
private static void checkBalance(List<Account> accList) {
  for (Account a : accList)
    if (a.balance() < a.threshold) a.alert();
}
// 2. primjena lambda izraza za obradu "klasične" kolekcije
private static void checkBalance(List<Account> accList) {
  accList.forEach(
   (Account a) -> { if (a.balance() < a.threshold) a.alert(); }
  );
}
// 3. primjena Streamsa za paralelizaciju "klasične" kolekcije
private static void checkBalance(List<Account> accList) {
  accList.parallelStream().forEach(
   (Account a) -> { if (a.balance() < a.threshold) a.alert(); }
  );
}
```

Java 8 default metode

- Kako je već rečeno, default metode su u Javi 8 trebale prvenstveno zbog uvođenja Stream API-a, iako su default metode korisne i za ostale svrhe (ne samo tvorcima API-a).
- Kod uvođenja Streamsa, bilo je potrebno nadograđivati brojna postojeća Java sučelja, tj. dodavati im nove metode. Međutim, **kad dodajemo nove metode u sučelje, moramo mijenjati sve klase** koje ga (direktno ili indirektno) nasljeđuju, što može značiti izmjenu milijuna redaka programskog koda.
- Default metode su riješile taj problem. **Sučelje sada može imati i implementaciju metode**, a ne samo deklaraciju.
- Java 8 sučelje sa default metodom **nije tako moćno kao Scala trait**. Scala trait može imati i ne-default metode, može imati stanje (atribute), može se komponirati za vrijeme runtimea, može pristupati instanci klase koja ga nasljeđuje. Ništa od toga Java 8 sučelje (sa default metodama) ne može.

Java 8 default metode

```
// 1. ako bismo postojeće sučelje Iterable
// htjeli proširiti sa novom metodom forEach,
// morali bismo mijenjati svaku klasu
// koja (direktno ili indirektno) nasljeđuje to sučelje
public interface Iterable<T> {
  public Iterator<T> iterator();
  public void forEach(Consumer<? super T> consumer);
}

// 2. default metode rješavaju taj problem
public interface Iterable<T> {
  public r<T> iterator();
  public default void forEach(Consumer<? super T> consumer) {
    for (T t : this) {
      consumer.accept(t);
    }
  }
}
```

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

- Kako je već rečeno, u Javi 5 su kroz paket java.util.concurrent uvedeni **executori** (tj. sučelja Executor, ExecutorService, Callable, Future, klase Executors, ThreadPoolExecutor, FutureTask i dr.)
- Executori, za razliku od direktnog rada s klasom Thread, pomažu da se programeri koncentriraju na kreiranje **zadataka** koji će se poslati executoru na izvršavanje, a optimizaciju izvršavanja rade executori, **koji koriste thread pool**.
- Executori najčešće kreiraju fiksni thread pool:
  ExecutorService ex = Executors.**newFixedThreadPool**(4);
- Može se kreirati i cached thread pool, kod kojeg se automatski kreira onoliko Java dretvi koliko je potrebno:
  … = Executors.**newCachedThreadPool**;
- Moguće je napraviti i thread pool sa samo jednom dretvom:
  … = Executors.**newSingleThreadExecutor**;

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

- Executorima se može predati zadatak koji je instanca klase sučelja **Runnable**.
- Ako je potrebno da zadatak vrati rezultat, onda se zadatak kreira kao instanca klase sučelja **Callable**. Budući da Callable (kao i Runnable) zadatak može izvršavati asinkrono, kod asinkronog izvršavanja potrebna je i instanca klase sučelja **Future** za dobijanje statusa i rezultata rada Callable.
- U našem ćemo primjeru koristi Callable i Future.
- **Zadatak je: naći koliko ima prim (prostih) brojeva među prvih N (npr. 10 000 000) prirodnih brojeva (N zadajemo).**
- Naravno, želimo zadatak riješiti kroz paralelno programiranje, tako da maksimalno koristimo procesorske resurse. Naravno, cilj je postići minimalno vrijeme za izvršenje zadatka.

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

- **Pitanje je kako podijeliti zadatak na podzadatke.**
- Npr., ako je zadatak pretraživati prim brojeve do 10 milijuna, možemo kreirati 10 podzadataka, od kojih će svaki pretraživati milijun brojeva, ili kreirati 10 000 podzadataka, od kojih će svaki pretraživati tisuću brojeva … itd.
- **Zapravo, prvo pitanje je koliko Java dretvi kreirati? Možda onoliko koliko ima podzadataka? To najčešće ne bi bilo dobro!**
- Java VM ne radi baš idealno sa prevelikim brojem Java dretvi (možda je par tisuća već previše). Jedan od razloga je i trošenje memorije za stack (svake) Java dretve.
- Osim toga, budući da su Java dretve najčešće realizirane kroz dretve operacijskog sustava, zamjena konteksta dretvi nije baš bez troškova (iako se kaže da je puno "jeftinija" od zamjene konteksta procesa operacijskog sustava).

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

- **Kako odrediti (relativno) optimalan broj Java dretvi?**
- Ako je problem računski vrlo intenzivan, tj. uglavnom ovisi o procesoru, a ne o I/O (Input/Output) operacijama, onda broj Java dretvi možemo postaviti tako da bude jednak (ili malo veći) broju HW (hardverskih) dretvi.
- Ako je problem IO intenzivan, onda se može kreirati više Java dretvi nego što ima HW dretvi. Za (približno) određivanje broja Java dretvi može se koristiti jednostavna formula:
  broj_Java_dretvi =
  broj_HW_dretvi / (1 – koeficijent_blokiranja)

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

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

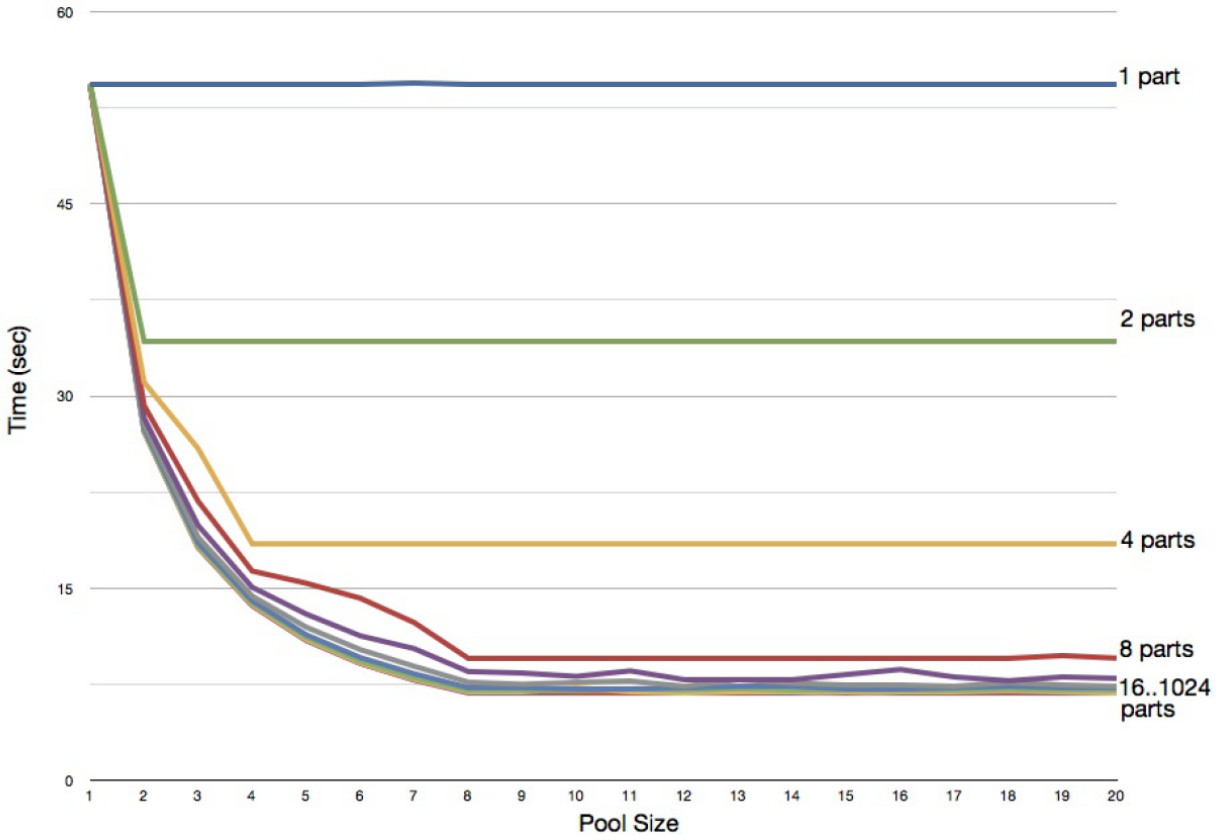
- IO intenzivni problemi mogu imati koeficijent koji se približava jedinici. Ako je koeficijent blokiranja npr. 50%, onda je po ovoj formuli preporučeni broj Java dretvi duplo veći od broja HW dretvi.
- No, sada treba odrediti kako podijeliti problem, tj. koliko ćemo imati podzadataka. Svaki podzadatak radit će konkurentno, pa ih svakako treba biti barem onoliko koliko ima Java dretvi.
- Ako bi broj podzadataka bio potpuno jednak broju Java dretvi, time bi se ignorirala priroda problema koji treba rješavati. Naime, u tom slučaju bi dijelovi programa trebali biti savršeno dobro izbalansirani.
- U našem zadatku, ako se broj dijelova naivno odredi tako da se raspon brojeva podijeli sa brojem dretvi, zanemaruje se važna činjenica da je lakše naći prim brojeve u donjim dijelovima raspona brojeva, nego u gornjim dijelovima.

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

- U općenitom slučaju može se primijeniti relativno jednostavna tehnika: broj podzadataka treba biti dovoljno velik da se iskoriste postojeće Java dretve, tj. **ne smije se desiti da neke Java dretve ostanu neiskorištene.**
- Dakle, broj podzadataka svakako mora biti veći od broja Java dretvi. Naravno, nije lako odrediti (a ponekad niti moguće) koliki točno treba biti taj broj – najčešće treba eksperimentirati. Uglavnom se pokazuje da se kod početnog povećanja broja podzadataka dobije značajno povećanje performansi, a sa daljnjim povećanjem broja, povećanje performansi je sve manje (performanse se mogu i smanjiti).
- Iza slike slijedi program, koji ima ove ulazne parametre:
  - **number**: gornja granica do koje se traže prim brojevi;
  - **poolSize**: broj Java dretvi;
  - **numberOfParts**: broj podzadataka.

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

- Pronalaženje prim brojeva na µP sa 8 HW dretvi (4 jezgre * 2) - prikaz efekta mijenjanja broja Java dretvi (PoolSize, os x) i broja podzadataka (različite krivulje):



Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka


```
import java.util.concurrent.ExecutorService; ...
public class ExecutorPrimeFinder { ...
  public static void main(final String[] args) {
    if (args.length < 3) {
    System.out.println("Usage: number poolSize numberOfParts");
    } else {
      final int number = Integer.parseInt(args[0]);
      final int poolSize = Integer.parseInt(args[1]);
      final int numberOfParts = Integer.parseInt(args[2]);
      ExecutorPrimeFinder task = new ExecutorPrimeFinder();
      final long startTime = System.nanoTime();
      final long numberOfPrimes =
        task.countPrimes(number, poolSize, numberOfParts);
      final long endTime = System.nanoTime();
      System.out.printf("Number of primes under %d is %d\n",
        number, numberOfPrimes, numberOfParts);
      System.out.println("Time (seconds) taken is " +
        (endTime - startTime) / 1.0e9);
    }}}
```

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka


```
protected int countPrimes
(final int number, final int poolSize, final int numberOfParts)
{ int count = 0;
  try {
    final List<Callable<Integer>> partitions =
      new ArrayList<Callable<Integer>>();
    final int chunksPerPartition = number / numberOfParts;

    for(int i = 0; i < numberOfParts; i++) {
      final int lower = (i * chunksPerPartition) + 1;
      final int upper = (i == numberOfParts - 1) ?
        number : lower + chunksPerPartition - 1;
      partitions.add(new Callable<Integer>() {
        public Integer call() {
          return countPrimesInRange(lower, upper);
        }
      });
    } ...
```

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka


```
protected int countPrimes
(final int number, final int poolSize, final int numberOfParts)
{
...
    final ExecutorService executorPool =
     Executors.newFixedThreadPool(poolSize);

    final List<Future<Integer>> resultFromParts =
     executorPool.invokeAll(partitions,10000,TimeUnit.SECONDS);

    executorPool.shutdown();

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

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka


```
public int countPrimesInRange(final int lower, final int upper)
{
  int total = 0;
  for(int i = lower; i <= upper; i++) {
    if (isPrime(i)) total++;
  }
  return total;
}

private boolean isPrime(final int number) {
  if (number <= 1) return false;
  if (number == 2) return true;
  if (number % 2 == 0) return false;
  for(int i = 3; i <= Math.sqrt(number); i = i + 2) {
    if (number % i == 0) return false;
  }
  return true;
}
```

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

- Kako je već rečeno, u Javi 7 je uveden **ForkJoin framework**, koji je, zapravo, specijalna verzija executora.

- Mark Reinhold, Chief Architect, Java Platform Group u "Divide and Conquer Parallelism with the Fork/Join Framework" (07.2011):

The fork/join framework minimizes per-task overhead for compute-intensive tasks

– Not recommended for tasks that mix CPU and I/O activity

A portable way to express many parallel algorithms

– Code is independent of execution topology

– Reasonably efficient for a wide range of core counts

– Library-managed parallelism

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

- Mark Reinhold, Chief Architect, Java Platform Group u "Divide and Conquer Parallelism with the Fork/Join Framework" (07.2011):

No silver bullet - Many point solutions:

- Divide & conquer (fork/join)
- Work queues + thread pools
- Bulk data operations (select / map / reduce)
- Actors
- Software transactional memory (STM)
- GPU-based SIMD-style computation

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

- "Divide and Conquer" ("Divide et impera", "Podijeli pa vladaj" ili "Zavadi pa vladaj" – stara rimska poslovica).
- Implementaciju ExecutorService sučelja u slučaju ForkJoin frameworka radi klasa **ForkJoinPool**.
- Tipično se ForkJoinPool instanci šalje samo jedan zadatak (task), a onda ForkJoinPool instanca i zadatak zajedno primjenjuju tehniku Divide and Conquer.
- Broj Java dretvi u poolu se može zadati eksplicitno ili implicitno:
  -- **broj Java dretvi se zadaje eksplicitno** (u ovom slučaju 8)
  ForkJoinPool fjPool = new ForkJoinPool(8);

  -- **ili implicitno** (na računalu sa 8 HW dretvi, bit će ih 8)
  -- framework koristi metodu **Runtime.availableProcessors()**
  ForkJoinPool fjPool = new ForkJoinPool();

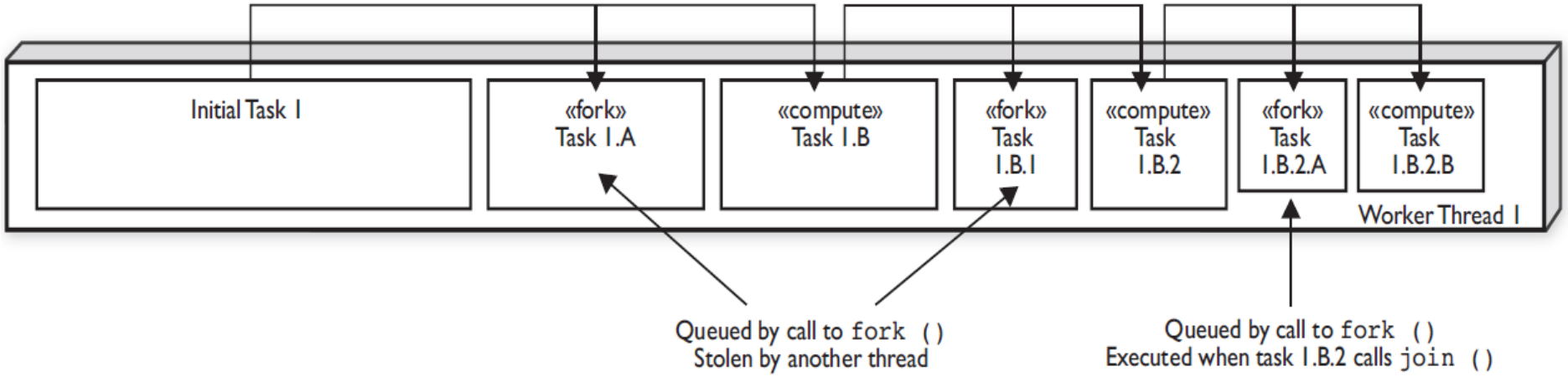
Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

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

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

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

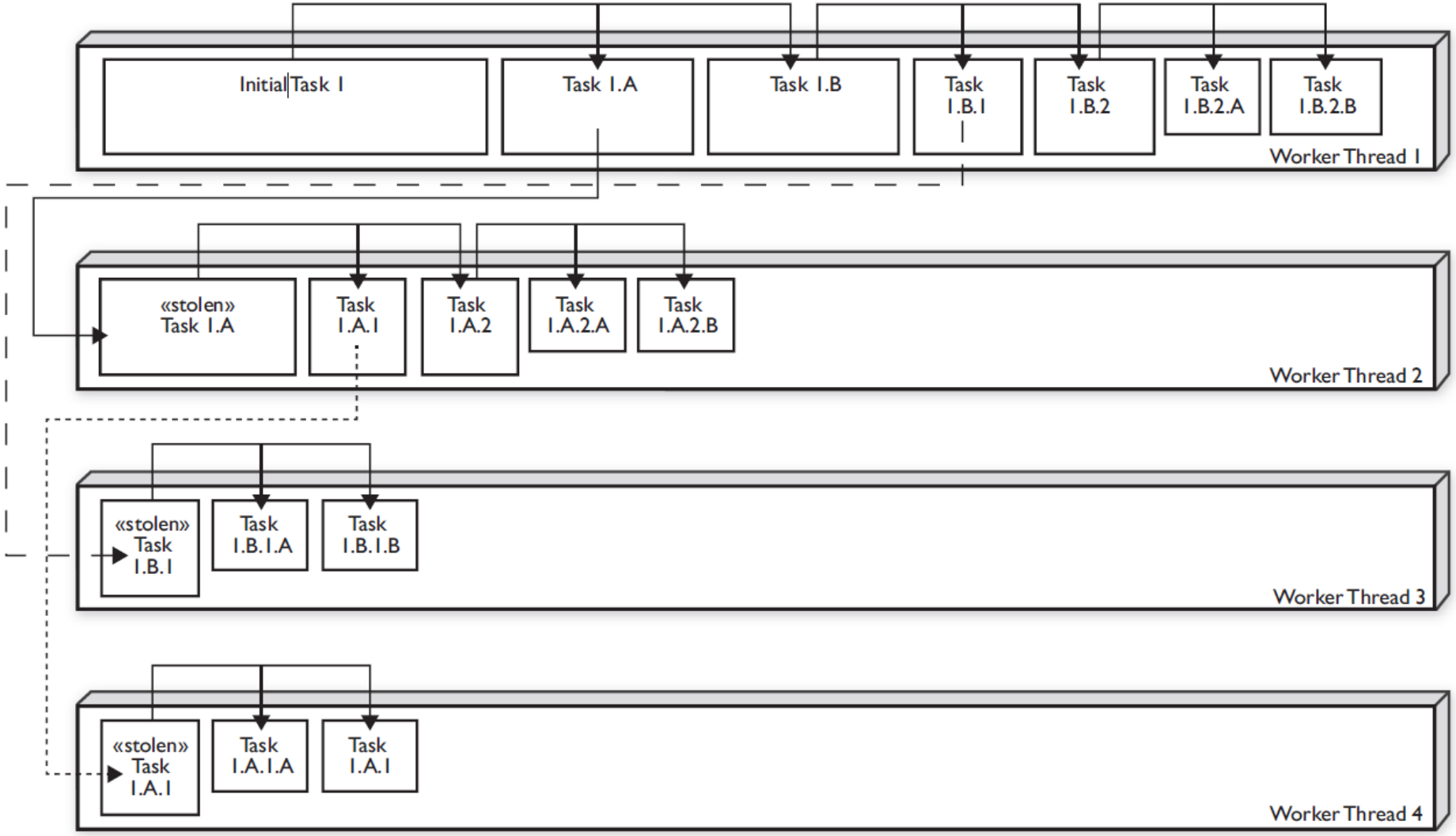
- Za razliku od većine ExecutorService implementacija, kod ForkJoin frameworka **svaka Java dretva u ForkJoinPool-u ima svoj red (queue)** podzadataka na kojima radi.
- Metoda fork() stavlja ForkJoinTask u red tekuće Java dretve.
- Inicijalno je zauzeta samo jedna Java dretva - kada joj pošaljemo (cijeli)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



Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

- Ključna značajka ForkJoin frameworka je **Work Stealing** (krađa posla). **Java dretve kradu posao (podzadatak) drugoj Java dretvi iz njenog reda podzadataka, i stavljaju ga u svoj red** (nešto što većina nas ne bi nikad napravila ☺).
- Vrlo je važan redoslijed stavljanja podzadataka u red. **Podzadaci koji se prvi stavljaju u red trebaju predstavljati veći dio posla.** Npr. na početku imamo jedan zadatak, koji pokriva 100% posla. Njega dijelimo u dva podzadatka, svaki po (otprilike) 50% posla. Prvi stavljamo u red, a drugi obrađujemo, pri čemu drugi opet dijelimo u polovice, itd.
- One Java dretve koje nemaju posla (tj. njihov red je prazan), kradu posao drugim Java dretvama, i to tako da uzmu posao (podzadatak) **koji je najstariji u redu, a to je istovremeno i najveći posao** iz reda druge Java dretve.
- Sljedeća slika pokazuje ta događanja, sa 4 Java dretve.

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka



Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

- Kada bi se zadatak mogao savršeno dijeliti na jednake polovice, ne bi niti bilo potrebe za ForkJoin frameworkom, mogli bismo koristiti i standardne executore.
- No, to je u praksi rijetko tako. Npr., u našem slučaju je lakše pretraživati prim brojeve po donjoj polovici raspona brojeva, nego gornjoj (jer gornja polovica sadrži veće brojeve).
- **Zato je važno podijeliti posao na dovoljno veliki broj podzadataka, tako da (sve do samog kraja) niti jedna Java dretva ne ostane bez posla** (tj. treba biti tako da jedna dretva uvijek može ukrasti posao drugoj dretvi).
- Dijeljenje na podzadatke se, za razliku od primjera sa običnim executorima, radi implicitno – ne zadaje se broj podzadataka, **već se određuje kada je podzadatak dovoljno mali**.
- Nažalost, ne postoji opća metoda kojom se ispravno određuje veličina tog malog podzadatka – **to se radi eksperimentalno**.

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka

- Podzadatak na kojem se poziva metoda join() može tada biti već gotov, jer ga je možda ukrala druga Java dretva, i već napravila. Ili može biti ukraden, ali ga druga dretva upravo radi, pa tada treba čekati da završi. Treći je slučaj da podzadatak nije ukraden i tada ga radi tekuća dretva.
- Poziv join() metode u compute() metodi treba biti jedna od zadnjih radnji, iza poziva fork() metode i rekurzivnog poziva compute() metode. Budući da je to jako važno, **postoji metoda invokeAll(a2, a1); koja zamjenjuje niz a1.fork(); a2.compute(); a1.join();**
- Kako je već rečeno, zadatak (task) treba biti instanca podklase apstraktnih klasa RecursiveAction ili RecursiveTask.
- **RecursiveAction se koristi onda kada nije potrebno vraćati rezultat, a RecursiveTask kada je potrebno vraćati rezultat** (kao u našem primjeru).

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka


```
import java.util.concurrent.ForkJoinPool;
...
public class ForkJoinPrimeFinderTask
  extends RecursiveTask<Integer>
{ private static int threshold;
  private int start;
  private int end;
  public ForkJoinPrimeFinderTask
    (final int theStart, final int theEnd)
  { start = theStart; end = theEnd; }

  public static void main(final String[] args) {
    if (args.length < 1 || args.length > 3) {
      System.out.println("Usage: number poolSize threshold
        OR  number poolSize  OR  number");
      return;
    }
  ...
}
```

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka


```
public static void main(final String[] args) { ...
  final int number = Integer.parseInt(args[0]);
  ForkJoinPrimeFinderTask task =
    new ForkJoinPrimeFinderTask(1, number);
  ForkJoinPool fjPool;
  if (args.length == 2 || args.length == 3) {
    fjPool = new ForkJoinPool(Integer.parseInt(args[1]));
  } else { fjPool = new ForkJoinPool(); }
  if (args.length == 3) {
    threshold = Integer.parseInt(args[2]);
  } else { threshold = 100;
  final long startTime = System.nanoTime();
  final long numberOfPrimes = fjPool.invoke(task);
  final long endTime = System.nanoTime();
  System.out.printf("Number of primes under %d is %d\n",
    number, numberOfPrimes);
  System.out.println("Time (seconds) taken is " +
    (endTime - startTime) / 1.0e9);
}
```

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka


```
protected Integer compute() {
  if (end - start <= threshold) {
    return countPrimesInRange(start, end);
  } else {
    int halfWay = ((end - start) / 2) + start;

    ForkJoinPrimeFinderTask t1 =
      new ForkJoinPrimeFinderTask(start, halfWay);
    ForkJoinPrimeFinderTask t2 =
      new ForkJoinPrimeFinderTask(halfWay + 1, end);

    t1.fork();
    int count2 = t2.compute();
    int count1 = t1.join();
    return count1 + count2;
  }
}
```

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka


```
-- ovo je isto kao u primjeru Executora
public int countPrimesInRange(final int lower, final int upper)
{
  int total = 0;
  for(int i = lower; i <= upper; i++) {
    if (isPrime(i)) total++;
  }
  return total;
}

private boolean isPrime(final int number) {
  if (number <= 1) return false;
  if (number == 2) return true;
  if (number % 2 == 0) return false;
  for(int i = 3; i <= Math.sqrt(number); i = i + 2) {
    if (number % i == 0) return false;
  }
  return true;
}
```

Usporedba Java 5/6 Executora i Java 7 ForkJoin frameworka


```
ExecutorPrimeFinder 10000000  1 10000 Number of primes under 10000000 is 664579
Time (seconds) taken is 12,70
ExecutorPrimeFinder 10000000  2 10000
Time (seconds) taken is 6,60
ExecutorPrimeFinder 10000000  4 10000
Time (seconds) taken is 3,20
ExecutorPrimeFinder 10000000  8 10000
Time (seconds) taken is 3,18
ExecutorPrimeFinder 10000000 16 10000
Time (seconds) taken is 3,17

ForkJoinPrimeFinderTask 10000000  1 1000 Number of primes under 10000000 is 664579
Time (seconds) taken is 14,41
ForkJoinPrimeFinderTask 10000000  2 1000
Time (seconds) taken is 7,36
ForkJoinPrimeFinderTask 10000000  4 1000
Time (seconds) taken is 3,91
ForkJoinPrimeFinderTask 10000000  8 1000
Time (seconds) taken is 3,28
ForkJoinPrimeFinderTask 10000000 16 1000
Time (seconds) taken is 3,25
ForkJoinPrimeFinderTask 10000000 16 100
Time (seconds) taken is 2,93
```

Zaključak

- **Mooreov zakon** vrijedi i dalje. No, radni takt procesora praktički je prestao rasti oko 2005. godine. **Umjesto povećanja brzine**, proizvođači **povećavaju broj CPU-a** (jezgri) na jednom mikroprocesorskom čipu.
- 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, a razlog za to objašnjava **Amdahlov zakon**.
- Nažalost, **konkurentne programe** nije lako pisati. Iako dretve rade paralelno, moramo investirati veliki trud da implementiramo tehnike koje ih sprečavaju da na loš način utječu jedna na drugu.

Zaključak

- 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) **Compare And Swap (CAS)**, ili sličnoj.
- Korištenje lokota (zaključavanje) ima dosta mana, a najveća je 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 na CAS. Njih je posebno teško smisliti i **često su vrlo neintuitivni**.
- Osnovna teškoća sa svim današnjim sinkronizacijskim operacijama je da one **rade na samo jednoj riječi memorije**, što tjera na korištenje kompleksnih i neprirodnih algoritama.

Zaključak

- Zato je izmišljena **transakcijska memorija** (TM), koja može biti **softverska** (STM), **hardverska** (HTM) ili **hibridna**. Postoje implementacije STM-a na razini jezika (npr. jezik Clojure) ili na razini biblioteka (npr. jezik Scala).
- Što se tiče **HTM**-a, danas postoje barem tri mikroprocesora koja ju podržavaju - **Azul Systems Vega, IBM BlueGene/Q**, **Intel Haswell.**
- Postoje i neka softverska rješenja, koja omogućavaju **relativno jednostavno i sigurno konkurentno programiranje u Javi** (preciznije, paralelno programiranje). Između ostalih, to su **executori** i **ForkJoin framework** (koji je specijalna vrsta executora). Nažalost, nisu svi zadaci prikladni za paralelizaciju pomoću tih rješenja.

Literatura

- Brinch Hansen, P. (1998): Java insecure Parallelism, članak, Syracuse University, http://brinch-hansen.net/papers/1999b.pdf (rujan 2012.)
- Cliff C. (2010): Azul's Experiences with Hardware / Software Co-Design (prezentacija), Azul Systems, blogs.azulsystems.com/cliff (kolovoz 2011.)
- 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 (travanj 2015.)
- Da Luo, Z., Nir-Buchbinder, J., Das, R. (2010): Java concurrency bug patterns for multicoresystems - Six lesser known Java concurrency bug patterns, DeveloperWorks, IBM, http://www.ibm.com/developerworks/java/library/j-concurrencybugpatterns/index.html (travanj 2015.)
- Drepper U. (2007): What Every Programmer Should Know About Memory (white paper), Red Hat Inc., http://www.akkadia.org/drepper/cpumemory.pdf (kolovoz 2011.)
- Evans, B.J., Verburg M. (2013): The Well-Grounded Java Developer – Vital techniques of Java 7 and polyglot programming, Manning Publications
- Fernandez Gonzalez J. (2012): Java 7 Concurrency Cookbook, Packt Publishing
- Haring, R. (2011): The Blue Gene/Q Compute Chip, prezentacija, IBM BlueGene Team, ftp://public.dhe.ibm.com/common/ssi/ecm/en/dcw03006usen/DCW03006USEN.PDF (rujan 2012.)
- Harris, T. (2010): Transactional Memory (2.izdanje), Morgan & Claypool

Literatura

- Harris, T. (2011): Multi-Core programming, prezentacija, Microsoft Research, http://www.cl.cam.ac.uk/teaching/1112/R204/slides-tharris.pdf (travanj 2015.)
- Herlihy, M., Shavit, N. (2008): The Art of Multiprocessor Programming, Elsevier / Morgan Kaufmann Publishers
- Horstmann, C.S., Cornell, G. (2008): Core Java: Volume 1 - Fundamentals (8.izdanje), Prentice Hall / Sun Microsystems
- Göetz B. i ostali (2006): Java Concurrency in Practice, Addison-Wesley
- Langer A., Kreft K. (2013): Lambda Expressions in Java (Tutorial), Angelika Langer Training/Consulting, www.AngelikaLanger.com (travanj 2015.)
- Lea D. (1999): Concurrent Programming in Java - Design Principles and Patterns (2.izdanje), Addison-Wesley
- Meyer, B. (1997): Object-Oriented Software Construction, Prentice Hall
- **Prokopec A. (2014): Learning Concurrent Programming in Scala, Packt Publishing**
- 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 (travanj 2015.)
- Sierra K, Bates B. (2105): OCA/OCP Java SE 7 Programmer I & II Study Guide, McGraw-Hill Education
- Subramaniam, V. (2011): Programming Concurrency on the JVM – Mastering Synchronization, STM, and Actors, The Pragmatic Bookshelf, Texas / Raleigh