Programske paradigme

— Konkurentno programiranje —

Milena Vujošević Janičić

Matematički fakultet, Univerzitet u Beogradu

Sadržaj

1 Uvod u konkurentno programiranje

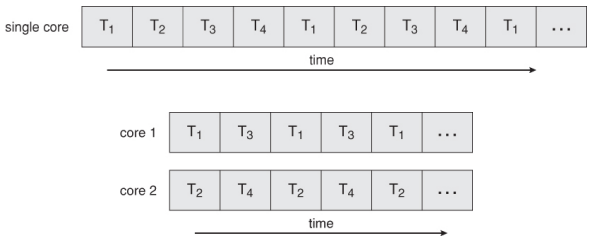
Teorija i praksa konkurentnog programiranja

1.1 Definicija i motivacija

Konkurentna paradigma

Konkurentna paradigma

Konkurentnu paradigmu karakteriše više procesa koji se izvršavaju u istom vremenskom periodu, a koji imaju zajednički cilj.



Konkurentna paradigma

- Konkurenost nije nova ideja
- Veliki deo teorijskih osnova datira još iz 1960tih godina, a već Algol 68 sadrži podršku za konkurentno programiranje
- Međutim, široko rasprostranjen interes za konkurentnost poslednjih dvadesetak godina. Uzroci:
  - **Podrška logičkoj strukturi problema** — Porast broja grafičkih, multimedijalnih i veb-zasnovanih aplikacija koje se sve prirodno predstavljaju konkurentnim nitima
  - **Dobijanje na brzini** — Dostupnost jeftinih višeprocesorskih mašina
  - **Rad sa fizički nezavisnim uređajima** — Potreba za umrežavanjem računara (farme računara) i različitih uređaja

1.2 — Podrška logičkoj strukturi problema

Podrška logičkoj strukturi problema

- Konkurentni mehanizmi su originalno izmišljeni za rešavanje određenih problema u okviru operativnih sistema, ali se sada koriste u raznim aplikacijama.
- Mnogi programi, moraju da vode računa istovremeno o više nego jednom zadatku koji su u velikoj meri nezavisni, pa je u takvim situacijama logično da se zadaci podele u različite kontrolne niti.

Primer

- Primer aplikacje čiji se dizajn oslanja na konkurentnost je veb razgledač (eng. *web browser*)
- Brauzeri moraju da izvršavaju više različitih funkcionalnosti istovremeno (konkurentnost na nivou jedinica), npr primanje i slanje podataka serveru, prikaz teksta i slika na ekranu, reagovanje na korisničke akcije mišom i tastaturom...
- U ovom slučaju konkurentnost može da se odnosi i na **konkurentnost u užem** smislu koja obuhvata jedan procesor, a konkurentnost se ostvaruje isprepletanim izvršavanjem različitih niti ili procesa.

1.3 — Dobijanje na brzini

Dobijanje na brzini

- Ukoliko imamo više procesora, treba ih iskoristiti da bi se dobilo na brzini.
- Više procesora odgovara **paralelnom programiranju**.
- Termin paralelno programiranje može da se odnosi i na više procesora na različitim mašinama, ali se najčešće misli na višeprocesorsku mašinu.
- Procesi međusobno komuniciraju preko zajedničke memorije (ukoliko je višeprocesorska mašina u pitanju) ili slanjem poruka (to može uvek da se koristi, bez obzira na vrstu hardvera).
- Ako je posao dobro podeljen na različite procesore, jasno je da se time dobija na brzini — slično kao što će četiri radnika brže okrečiti stan nego što bi to uradio jedan radnik

Dobijanje na brzini

- Dobijanje na brzini može da se ostvari i kod jednoprocesorske mašine konkurentnim izvršavanjem. Program, po prirodi, ne mora da bude u potpunosti sekvencijalan i to se može iskoristiti.
- Na primer, imamo problem u okviru kojeg treba da se učita velika količina podataka i da se na osnovu tih podataka izvrše redom nekakva izračunavanja

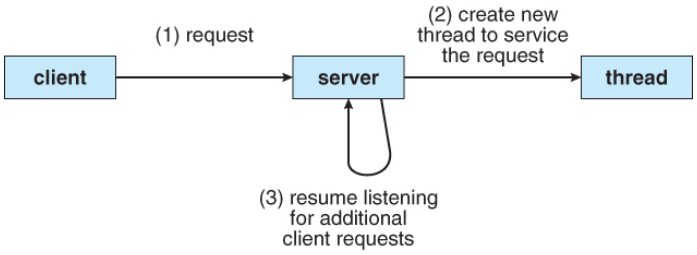
- Sekvencijalno rešenje: učitati sve podatke, izvršiti izračunavanja, ispisati rezultat
- Konkurentno rešenje: učitavati podatke i koristiti vreme između dva unosa podataka za izvršavanje izračunavanja.
- Pošto je učitavanje podataka spor proces, u konkurentnom slučaju rezultati izračunavanja će biti dostupni ubrzo nakon završetka učitavanja (jer je vreme između dva učitavanja, koje je u prvom rešenju neiskorišćeno, u konkurentnom pristupu iskorišćeno za izračunavanje), dok u sekvencijalnom pristupu to nije slučaj, s obzirom da izračunavanje počinje tek nakon kompletiranog učitavanja

1.4 — Zajednički rad fizički nezavisnih uređaja

Zajednički rad fizički nezavisnih uređaja

- Aplikacije koje rade distribuirano korišćenjem različitih mašina, bilo da su u pitanju lokalno povezane mašine ili Internet — **distribuirano programiranje**
- Većina distribuiranih sistema pokreće nezavisne programe na svakom umreženom procesoru, i koristi slanje poruka za komunikaciju
- Može se shvatiti kao vrsta paralelnog izračunavanja ali sa drugačijom međusobnom komunikacijom koja nameće nove izazove.
- Komunikacija je najskuplji deo distribuiranog programiranja i efikasni distribuirani algoritmi teže da smanje potrebu za komunikacijom
- Postoje jezici dizajnirani za distribuirano programiranje, ali oni nisu u širokoj upotrebi već se kod postojećih programa koriste dodatne biblioteke koje omogućavaju ovu vrstu programiranja.

Primer: servisno orijentisani sistemi (klijent-server arhitektura aplikacije)



Klijent-server model je primer distribuirane strukture aplikacije koja deli posao na server, koji obezbeđuje neku vrstu servisa, i na klijenta koji zahteva servis. Obično klijenti i serveri komuniciraju putem mreže. Klijenti obično iniciraju komunikaciju sa serverom, koji čeka na zahteve i po prijemu ih obrađuje.

Primeri distribuiranih sistema

- www — informacije i deljenje podataka
- *Online* igre sa velikim brojem igrača
- Lokalizovani sistemi različitih komponenti: automobil, avion
- Telekomunikacione mreže koje uključuju radne stanice i mobilne telefone
- *Peer-to-peer* aplikacije koje se pokreću na mreži računara *ravnomernom* podelom zadatka na delove (strukturirane i nestrukturirane mreže)

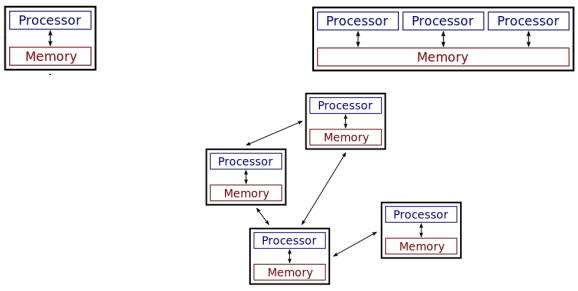

- ...

1.5 Hijerarhijska podela konkurentnosti

Hijerarhijska podela konkurentnosti

- Hijerarhijski:
  - Konkurentna paradigma je najširi pojam, tj konkurentnu paradigmu karakteriše više procesa koji se izvršavaju u istom vremenskom periodu.
  - Paralelna paradigma je specijalizacija koja obuhvata postojanje više procesora i omogućava istovremeno izvršavanje.
  - Distribuirana paradigma specijalizacija paralelene paradigme u kojoj su procesori i fizički razdvojeni.
- Ove razlike su bitne po pitanju implementacije jezika i performansi koje se postižu.

Hijerarhijska podela konkurentnosti



Konkurentnost u užem smislu

Paralelnost u užem smislu

Distribuirani sistem

Hijerarhijska podela konkurentnosti

- Često se termin konkurentno programiranje koristi za konkurentno programiranje u užem smislu (jedan procesor), a paralelno programiranje se koristi za paralelno programiranje u užem smislu (višeprocesorska mašina, ne i distribuirano programiranje)
- Razlika između konkurentne paradigme u užem smislu i paralelnog programiranja u užem smislu semantički nije izražena

2 Podrška, skalabilnost i portabilnost

2.1 Vrste konkurentnosti

Podrška konkurentnom programiranju

- Osnovna ideja konkurentnog programiranja je u podeli poslova tako da se mogu obavljati u izvesnoj količini nezavisno i istovremeno
- Na prvi pogled, konkurentnost može da izgleda kao jednostavan koncept, međutim pisanje konkurentnih programa je značajno teže od pisanja sekvencijalnih programa.
- Za osnovne koncepte konkurentnog programiranja potrebno je obezbediti odgovarajuću podršku u programskom jeziku ili kroz sam jezik, ili kroz odgovarajuće biblioteke
  - Ukoliko se podrška obezbeđuje kroz biblioteku, treba napomenuti da biblioteka mora da bude na odgovarajući način povezana sa jezikom i njegovim kompajlerom (kako bi se mogla obezbediti atomičnost izvršavanja operacija).

Vrste konkurentnosti

- Vrste konkurentnosti
  - Fizička konkurentnost (podrazumeva postojanje više procesora)
  - Logička konkurentnost
- Sa stanovišta programera i dizajna programskog jezika, logička konkurentnost je ista kao i fizička u kontekstu konkurentnog i paralelnog programiranja (u užem smislu). Zadatak je implementacije jezika da korišćenjem operativnog sistema preslika logičku konkurentnost u odgovarajući hardver.
- Kada je u pitanju distribuirano programiranje, stvari su malo složenije, i pisanje distribuiranih programa najčešće podrazumeva svesnost programera da će se stvari izvršavati distribuirano.

2.2 Nivoi konkurentnosti

Nivoi konkurentnosti

- Postoje četiri osnovna nivoa konkurentnosti
  1. Nivo instrukcije — izvršavanje dve ili više instrukcija istovremeno
  2. Nivo naredbe — izvršavanje dve ili više naredbi jezika višeg nivoa istovremeno
  3. Nivo jedinica — izvršavanje dve ili više jedinica (potprograma) istovremeno
  4. Nivo programa — izvršavanje dva ili više programa istovremeno
- Prvi i četvrti nivo konkurentnosti ne utiču na dizajn programskog jezika.
  - Prvi se odnosi na karakteristike hardvera i mogućnost kompajlera da te karakteristike iskoristi
  - Četvrti se odnosu na mogućnosti operativnog sistema da pokrene više nezavisnih programa istovremeno

2.3 Skalabilnost

Osnovni cilj: skalabilnost i portabilnost

- Konkurentnim programiranjem treba da se proizvedu skalabilni i portabilni algoritmi.
- Skalabilnost se odnosi na ubrzavanje izvršavanja porastom broja procesora. Ovo je važno jer se broj dostupnih procesora stalno povećava.
- Naravno, u razmatranju skalabilnosti mora se uzeti u obzir i priroda problema i njegova prirodna ograničenja (porast broja procesora nakon neke granice ne mora da ima pozitivan efekat).
- Idealna bi bila linearna skalabilnsot, ali ona je retka.

Skalabilnost

Faktor ubrzanja paralelizacijom je broj koji pokazuje koliko puta se program izvršavan na više procesora izvršava brže nego kada se taj isti program izvršava na jednom. Formula ubrzanja je sledeća:

$$S = \frac{T_s}{T_p}$$

gde T_s predstavlja vreme izvršavanja programa na jednom procesoru a T_p vreme izvršavanja programa na više procesora.

Skalabilnost

- Amdalov zakon — mali delovi programa koji se ne mogu paralelizovati će ograničiti mogućnost ukupnog ubrzavanja paralelizacijom: ako je α procenat koda koji se ne može paralelizovati, onda je maksimalno ubrzanje paralelizacijom $1/\alpha$
- Na primer,
  - Ako je α =10% onda je maksimalno ubrzanje paralelizacijom 10 puta, bez obzira na broj procesora koji se dodaju.
  - Ako je α =20% onda je maksimalno ubrzanje paralelizacijom 5 puta, bez obzira na broj procesora koji se dodaju.

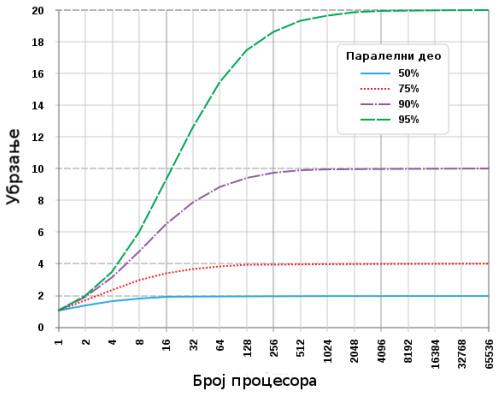
Skalabilnost

Objašnjenje Amdalovog zakona: izvršavanje paralelnog programa na paralelnom računaru uglavnom obuhvata i deo operacija koje se ne mogu izvršavati paralelno. Označimo sa α deo programa koji se mora izvršavati sekvencijalno na jednom procesoru, a ostatak $(1 - \alpha)$ se može izvršiti paralelno. Ako je N broj procesorskih jedinica, formula ubrzanja je:

$$S = \frac{T_s}{T_p} = \frac{T_s}{T_s \cdot (\alpha + \frac{1-\alpha}{N})}$$

Ova formula nam pokazuje da ubrzanje nikada ne može preći $1/\alpha$, tj. broj procesorskih jedinica ne utiče na deo programa koji se mora izvršavati sekvencijalno.

Amdalov zakon



Amdalov zakon

- Amdalov zakon nam govori da ne treba da očekujemo previše od paralelizacije i da treba dobro da razumemo prirodu algoritma koji paralelizujemo da bi znali da procenimo šta paralelizacijom možemo da dobijemo
- Dodatno treba uzeti u obzir i usporavanje koje nameće komunikacija između različitih niti izvršavanja.

2.4 Portabilnost

Portabilnost

- Portabilnost se odnosi na nezavisnost od konkretne arhitekture — životni vek hardvera je kratak, stalno izlaze nova hardverska rešenja i dobar algoritam ne zavisi od hardvera.
- Međutim, poznavanje ciljanog hardvera omogućava implementaciju efikasnijeg rešenja i često se uzima u obzir. Zbog toga je potrebno poznavati ciljnu arhitekturu hardvera.
- Pod arhitekturom hardvera najveću razliku čine distribuirani sistemi u odnosu na sisteme sa deljenom memorijom, mada i različite arhitekture sistema sa deljenom memorijom mogu imati značajnu ulogu

Portabilnost

- U okviru sitema sa deljenom memorijom, to mogu biti simetrični multiprocesori i višejezgarni procesori, ali i grafičke kartice (engl. GPU — graphics processing unit)
- U okviru distribuiranih sistema, to mogu biti različite vrste mreža, npr mreže radnih stanica (*network of workstations*) ili mreže u opštem smislu (heterogeni dinamički resursi koji su geografski distribuirani na više administrativnih domena i u vlasništvu različitih organizacija).

3 Osnovni koncepti

Osnovni koncepti

- Osnovni koncepti koje ćemo diskutovati se uglavnom donose na konkurentno i paralelno programiranje u užem smislu
- Osnovna pitanja se odnose na realizaciju **komunikacije** i **sihronizacije** procesa, kao i **pristupa zajedničkim podacima**
- Veliki izazov za programere, dizajnere programskih jezika i dizajnere operativnih sistema (dobar deo podrške za konkurentnost obezbeđuje operativni sistem)

Osnovni koncepti

- Za konkurentno programiranje potrebna je podrška u okviru programskog jezika, ili u okviru biblioteka
- Da bi se razmatrala podrška koju je potrebno da programski jezik pruži, najpre je potrebno razumeti osnovne koncepte konkurentnosti.
- Nažalost, terminologija u okviru različitih autora, programskih jezika i operativnih sistema nije konzistentna — zato je jako bitno suštinski razumeti problematiku

3.1 Zadatak, nit, proces

Zadatak, posao (eng. *task*)

- Zadaci (eng. *task*), niti (eng. *thread*), procesi (eng. *process*)
- Zadatak ili posao je jedinica programa, slična potprogramu, koja može da se izvrši konkurento sa drugim jedinicama istog programa.

Zadatak, posao (eng. *task*)

- Zadatak se razlikuje od potprograma na tri načina
  - Zadaci mogu da počinju implicitno, ne moraju eksplicitno da budu pozvani.

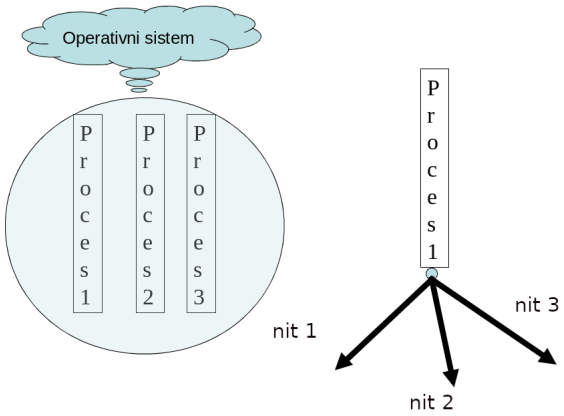
    ```
    #pragma omp parallel for
    for (int i = 0; i < element.size(); ++i)
        element[i] =  2 * i + 1;
    ```

  - Kada program pokrene neki zadatak, ne mora uvek da čeka na njegovo izvršavanje pre nego što nastavi sa svojim. Na primer, server kada napravi nit koja opslužuje klijenta, nastavlja dalje sa svojim radom.
  - Kada se izvršavanje zadatka završi, kontrola ne mora da se vrati na mesto odakle je počela izvršavanje.

Teški i laki zadaci

- Zadaci se dele na dve opšte kategorije: *heavyweight* i *lightweight*.
- Teški imaju svoj sopstveni adresni prostor dok laki dele isti adresni prostor.

Odnos teških i lakih zadataka (u C terminologiji: odnos procesa i niti)



Teški zadaci

- Teškim zadacima upravlja operativni sistem obezbeđujući deljenje procesorskog vremena, pristup datotekama, adresni prostor.
- Prelaz sa jednog teškog zadatka na drugi vrši se posredstvom operativnog sistema uz pamćenje stanja prekinutog procesa (skupa operacija)

Teški zadaci

- Stanje teškog zadatka obuhvata:
  - Podatke o izvršavanju, vrednosti registara, brojač instrukcija
  - Informacije o upravljanju resursima (informacije o memoriji, datoteke, ulazno-izlazni zahtevi i ostali resursi)
- **Promena konteksta** je promena stanja procesora koja je neophodna kada se sa izvršavanja jednog teškog zadatka prelazi na izvršavanje drugog teškog zadatka: potrebno je zapamtiti u memoriji stanje zadatka koji se prekida i na osnovu informacija u memoriji rekonstruisati stanje zadatka koji treba da nastavi izvršavanje.

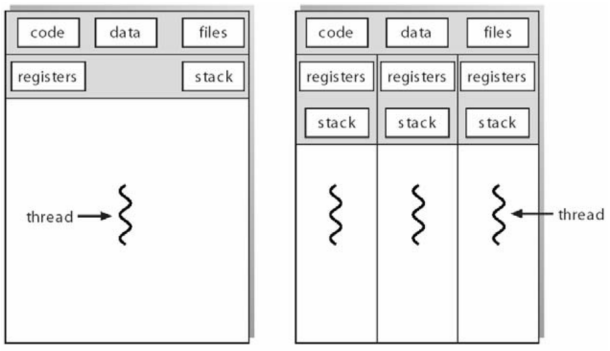
Teški zadaci

- Promena konteksta je skupa, a to nije zanemarljivo: promena konteksta se dešava veoma često, od nekoliko puta do par stotina ili hiljada puta u jednoj sekundi
- Podaci o stanju procesa zauzimaju memoriju koja nije zanemarljiva, a bitna je i u kontekstu promašivanja u kešu koje značajno utiče na performanse sistema

Laki zadaci

- Koncept lakih zadataka se uvodi kako bi se omogućio efikasniji prelaz sa jednog na drugi zadatak
- Za razliku od teških zadataka, laki ne zahtevaju posebne računarske resurse, već postoje untar jednog teškog zadatka.
- Podaci koji se vode na nivou teškog zadatka su zajednički za sve lake zadatke koje on obuhvata
- Podaci o lakom zadatku obuhvataju samo njegovo stanje izvršavanja, brojač instrukcija i vrednosti radnih registara
- Prelaz sa jednog lakog zadatka na drugi je stvar izmene sadržaja radnih registar i pamćenja brojača instrukcija i stanja izvršavanja i vrši se brzo.

Odnos teških i lakih zadataka (u C terminologiji: odnos procesa i niti)



Upravljanje lakim zadacima

- Upravljanje nitima najčešće se ostvaruje preko korisničkih biblioteka (npr pthreads, OpenMP, Java threads, Win32 API ...)
- Kada je reč o opertivnim sistemima, niti su podržane u svim modernim operativnim sistemima:
  - Windows, počevši od verzije XP/2000
  - Linux
  - Solaris, Tru64 UNIX, Mac OS X, Android

Primer

- Terminologija je nekonzistentna
- Npr, u C-u teški zadaci su procesi (koji imaju svoj adresni prostor), laki zadaci su niti (koje imaju zajednički adresni prostor).
- Kreiranje lakih zadataka je efikasnije od kreiranja teških.
- Sistemski poziv `fork` za kreiranje novih procesa
- Biblioteka `pthread` za kreiranje niti

3.2 Izbor nivoa konkurentnosti

Paralelizacija zadataka ili podataka

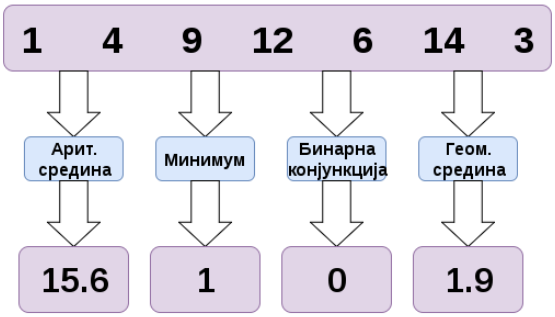
- Osnovna odluka koju programer mora da donese kada piše paralelni program je kako da podeli posao.
- Šta paralelizovati?
- Ne postoji jedinstven odgovor na ovo pitanje, već je za svaki problem neophodno sagledati mogućnosti

Paralelizacija zadataka ili podataka

- Pretpostavimo da imamo niz brojeva i da za svaki element niza treba da ispitamo njegovih svojstava. Konkretno, potrebno je za svaki broj iz niza brojeva odrediti koja od narednih svojstva ispunjava:
  - broj je prost,
  - broj je savršen,
  - broj je deljiv sumom svojih cifara,
  - broj ima paran broj delilaca,
  - broj je jednak zbiru kubova svojih cifara.

  Ili, na primer, za niz brojeva izračunati
  - aritmetičku sredinu
  - minimum
  - geometrijsku sredinu
  - binarnu konjunkciju
- Šta paralelizovati?

Paralelizacija zadataka



Пример паралелизације задатака: примена различитих функција над истим подацима

Paralelizacija zadataka

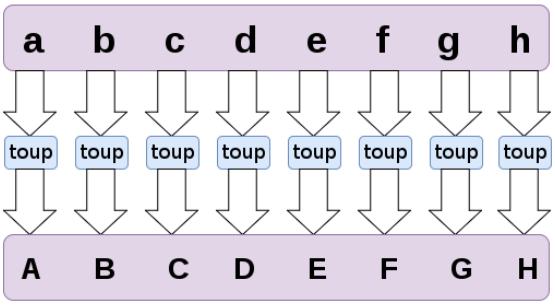
- Najčešća strategija, koja dobro radi na malim mašinama, je da se koriste različite niti za svaki od glavnih programerskih zadataka ili funkcija.
- Na primer, kod procesora reči, jedan zadatak bi mogao da bude zadužen za prelamanje paragrafa u linije, drugi za određivanje strana i raspored slika, treći za pravopisnu proveru i proveru gramatičkih grešaka, četvrti za renderovanje slika na ekranu...
- Ova strategije se obično naziva **paralelizam zadataka**.
- Nedostatak ove strategije je da ne skalira prirodno dobro ukoliko imamo veliki broj procesora.

Paralelizacija zadataka ili podataka

- Ako ponovo razmotrimo ovaj primer, tj za svaki broj iz niza brojeva odrediti koja od narednih svojstva ispunjava:
  - broj je prost,
  - broj je savršen,
  - broj je deljiv sumom svojih cifara,
  - broj ima paran broj pravih delilaca,
  - broj je jednak zbiru kubova svojih cifara.

  vidimo da ova svojstva imaju medjusobne zavisnosti, kao i da je ispitivanje svojstva za jedan broj nezavisno od ispitivanja svojstva za drugi broj
- To znači da je u ovom slučaju moguće primeniti paralelizaciju podataka (moguće je i u prvom primeru takođe primeniti paralelizaicju podataka, ali na nešto kompleksniji način)

Paralelizacija podataka



Пример паралелизације података: примена функције *toupper* над сваким словом појединачно

Paralelizacija podataka

- Za dobro skaliranje na velikom broju procesora, potreban je paralelizam podataka.
- Kod paralelizma podataka, iste operacije se primenjuju konkurentno na elemente nekog velikog skupa podataka.
- Na primer, program za manipulaciju slikama, može da podeli ekran na n manjih delova i da koristi različite niti da bi procesirao svaki taj pojedinačni deo.
- Igrica može da koristi posebnu nit za svaki objekat koji se pokreće.
- U primeru računanja svojstava brojeva, svaka nit računa sva predložena svojstva nad svojim delom niza

Primer

- Program koji je dizajniran da koristi paralelizaciju podataka najčešće se zasniva na paralelizaciji petlji (paralelizam na nivou naredbi): svaka nit izvršava isti kod ali korišćenjem različitih podataka.
- Na primer, u C#, ukoliko se koristi *Parallel FX Library*

```
Parallel.For(0, 100, i => { A[i] = foo(A[i]); } );
```

ili u C++-u ako se koristi biblioteka OpenMP

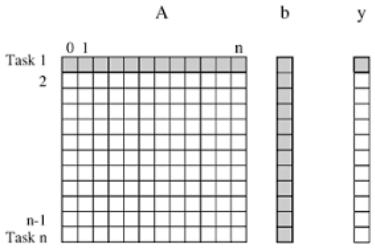
```
#pragma omp parallel for
for (int i = 0; i < element.size(); ++i)
    element[i] =  2 * i + 1;
```

Primer

- Neophodno je da programer zna da su pozivi funkcije `foo` međusobno nezavisni, ukoliko nisu, neophodna je *sinhronizacija*.
- Idealno bi bilo kada bi kompajler mogao sam da zaključi da je nešto nezavisno i da sam obavi paralelizaciju umesto nas, ali, nažalost, u opštem slučaju to nije moguće.
- Paralelizacija podataka prirodna je za neke vrste problema, ali nije za sve.

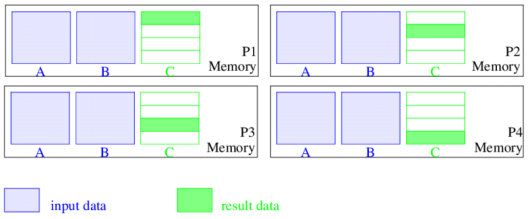
Primeri

- Množenje matrice i vektora



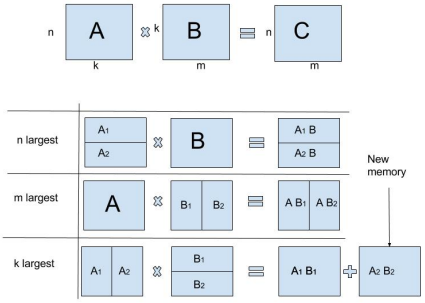
Primeri

- Množenje matrica



Primeri

- Množenje matrica



Primeri

- Proizvod po komponentama vektora $A = (a_1, ...a_n)$ i $B = (b_1, ...b_n)$,

$$A \cdot B = (a_1 \cdot b_1, ..., a_n \cdot b_n)$$

- Podela na n jednakih delova
- Pretpostavka o uniformnosti podataka?
- Podela tako da svaka nit uzima po jedan element

Primeri

- Pronalaženje prostih brojeva u nizu
- Pronalaženje brojeva koje imaju određenu osobinu (prosti, savršeni, blizanci, Armstrongovi...) i pripadaju intervalu [n,m]

3.3 Komunikacija i sinhronizacija

Komunikacija

- Zadaci međusobno moraju da komuniciraju
- Komunikacija se odnosi na svaki mehanizam koji omogućava jednom zadatku da dobije informacije od drugog
- Komunikacija se može ostvariti preko
  - zajedničke memorije (ukoliko zadaci dele memoriju)
  - razmenom poruka

Zajednička memorija

- Ukoliko imamo zajedničku memoriju, izabranim promenljivama se može pristupiti iz različitih zadataka (ovo se, pre svega, odnosi na komunikaciju između lakih zadataka)
- Da bi dva zadatka komunicirala, jedan upiše vrednost promenljive, a drugi je jednostavno pročita
- U ovom slučaju, važan je redosled čitanja i pisanja promenljivih i potrebno je starati se o ispravnom korišćenju resursa i eventualnim sukobima

Razmena poruka

- Razmena poruka se može upotrebljavati uvek, i kada zadaci imaju i kada nemaju zajedničku memoriju.
- U tom slučaju, da bi se ostvarila komunikacija, jedan zadatak mora eksplicitno da pošalje podatak drugom
- Razmena poruka može se ostvariti na razne načine
  - Ukoliko se koriste mehanizmi međuprocesne komunikacije na istoj mašini, postoje tokovi, signali, cevi, soketi, kanali
  - Za komunikaciju može se koristiti i mreža i u distribuiranim sistemima poruke moraju da putuju preko mreže

Pouzdanost slanja poruka

- Ukoliko je slanje poruka u okviru iste mašine, onda se ono smatra pouzdanim.
- Razmena poruka u distribuiranim sistemima nije pouzdano jer podaci putuju kroz mrežu gde mogu da se zagube
- Za slanje poruka u distribuiranim sistemima koriste se različiti protokoli

Sinhronizacija

- Sinhronizacija se odnosi na mehanizam koji dozvoljava programeru da kontroliše redosled u kojem se operacije dešavaju u okviru različitih zadataka.
- Sinhronizacija je obično implicitna u okviru modela slanja poruka: poruka mora prvo da se pošalje da bi mogla da se primi, tj ako zadatak pokuša da primi poruku koja još nije poslata, mora da sačeka pošiljaoca da je najpre pošalje
- Sinhronizacija nije implicitna u okviru modela deljene memorije, na primer, ukoliko se eksplicitno ne pobrinemo, zadatak može da pročita staru vrednost promenljive pre nego što je ta vrednost zamenjena novom od strane nekog drugog zadatka

Implementacija sinhronizacije

- I kod deljene memorije i kod slanja poruka, sinhronizacija može da se implementira na dva načina: zauzetim čekanjem ili blokiranjem
- Zauzeto čekanje (busy-waiting synchronization) — zadatak u petlji stalno proverava da li je neki uslov ispunjen (da li je poruka stigla ili da li deljena promenljiva ima neku određenu vrednost)
- Blokirajuća sinhronizacija — zadatak svojevoljno oslobađa procesor, a pre toga ostavlja poruku u nekoj strukturi podataka zaduženoj za sinhronizaciju. Zadatak koji ispunjava uslov u nekom trenutku naknadno, pronalazi poruku i sprovodi akciju da se prvi zadatak odblokira, tj da nastavi sa radom.
- Zauzeto čekanje nema smisla na jednom procesoru

3.4 — Sinhronizacija kod deljene memorije

Sinhronizacija kod deljene memorije — saradnja

- Postoje dve vrste potrebe za sinhronizacijom kada zadaci dele podatke: saradnja i takmičenje
- Sinhronizacija saradnje je neophodna između zadatka A i zadatka B kada A mora da čeka da B završi neku aktivnost pre zadatka A da bi zadatak A mogao da počne ili da nastavi svoje izvršavanje
- Primer: proizvođač-potrošač

Sinhronizacija kod deljene memorije — takmičenje

- Sinhronizacija takmičenja je neophodna između dva zadatka kada oba zahtevaju nekakav resurs koji ne mogu istovremeno da koriste, npr ako zadatak A treba da pristupa deljenom podatku x dok B pristupa x, tada zadatak A mora da čeka da zadatak B završi procesiranje podatka x
- Primer: korišćenje štampača

Sinhronizacija kod deljene memorije — takmičenje

- Na primer, ukoliko zadatak A treba da vrednost deljene promenljive uveća za jedan, dok zadatak B treba da vrednost deljene promenljive uveća dva puta, onda, ukoliko nema sinhronizacije, kao rezlutat rada ovih zadataka mogu da se dese različite situacije.

```
x = 3;
A:                  B:
   x = x + 1;           x = 2*x;
```

Sinhronizacija kod deljene memorije — takmičenje

- Na mašinskom nivuo, imamo sledeća tri koraka: uzimanje vrednosti, izmena, upisivanje nove vrednosti
- Bez sinhronizacije, ukoliko je početna vrednost 3, mogući ishodi su:
  - 8 — ako se A prvo izvrši, pa zatim B
  - 7 — ukoliko se B prvo izvrši, pa zatim A
  - 6 — ukoliko A i B uzmu istovremeno vrednost, ali je A prvi upiše nazad, pa je zatim B prepiše
  - 4 — ukoliko A i B uzmu istovremeno vrednost, ali je B prvi upiše nazad, pa je zatim A prepiše
- Kako bi to izgledalo na primeru štampača?

Sinhronizacija kod deljene memorije — takmičenje

- Situacija koja vodi do ovih problema naziva se **uslov takmičenja** ili **nadmetanje za resurse** (eng. *race condition*) jer se dva ili više zadataka takmiče/nadmeću da koriste deljene resurse i ponašanje programa zavisi od toga ko pobedi na takmičenju, tj ko stigne prvi
- Osnovna uloga sinhronizacije je da za sekvencu instrukcija koju nazivamo kritična sekcija obezbedi da se izvrši atomično, tj da se sve instrukcije kritične sekcije izvrše bez prekidanja.

Semafori i monitori

- Za kontrolisanje pristupu deljenim resursima, koristi se zaključavanje ili uzajamno isključivanje
- Za uzajamno isključivanje mogu se koristiti npr semafori ili monitori
- Semafori su prvi oblik sinhronizacije, implementiran već u Algol 68, i dalje prisutni, npr u Javi
- Semafori imaju dve moguće operacije, P i V (ili wait i release)
- Nit koji poziva P atomično umanjuje brojač i čeka da n postane nenegativan. Nit koji poziva V atomično uvećava brojač i budi čekajuću nit, ukoliko postoji.

Semafori i monitori

- Semafori se smatraju kontrolom niskog nivoa, nepogodnom za dobro struktuiran i lako održiv kod.
- Korišćenje semafora lako dovodi do grešaka i do uzajamnog blokiranja.
- Drugi koncept su monitori koji enkapsuliraju deljene strukture podataka sa njihovim operacijama, tj čine deljene podatke apstraktnim tipovima podataka sa specijalnim ograničenjima.

- Monitori su prisutni npr u Javi (modifikator synchronized), C# (klasa Monitor)...
- I monitori se takođe smatraju kontrolom niskog nivoa.

Muteksi i katanci

- Mutex — mutual exclusion (postoje u C++-u, ne postoje u Javi)
- Semantika katanca — osnovni način bezbednog deljenja podataka u konkurentnom okruženju
- Atomično zaključavanje, samo jedna nit može zaključati podatak, ako je muteks već zaključan, nit koja želi da ga zaključa mora da sačeka na njegovo otključavanje
- Katanci — različiti pristupi pod različitim uslovima (npr više njih može da čita, samo jedan može da piše, tada niko ne sme da čita)

Zaključavanje podataka

- Zaključavanje podataka je veoma skupa operacija
- Ako postoji veliki broj katanaca, i ako se zaključava/otključava često, to može da uspori rad aplikacije
- Ako postoji mali broj katanaca, i ako se zaključava velika količina podataka istovremeno, to može da dovede do dužeg čekanja i opet do sporijeg rada

3.5 —— Koncept napredovanja

Koncept napredovanja (eng. *liveness*)

- Kod sekvencionalnog izvršavanja programa, program ima karakteristiku napredovanja ukoliko nastavlja sa izvršavanjem, dovodeći do završetka rada programa u nekom trenutku (ukoliko program ima osobinu da se završava, npr nema beskonačnih petlji)
- Opštije, koncept napredovanja govori da ukoliko neki događaj treba da se desi (npr završetak rada programa) da će se on i desiti u nekom trenutku, odnosno da se stalno pravi nekakav progres tj napredak.
- U konkurentnoj sredini sa deljenim objektima, napredak zadatka može da prestane, odnosno može da se desi da program ne može da nastavi sa radom i da zbog toga nikada ne završi svoj rad.

Koncept napredovanja (eng. *liveness*)

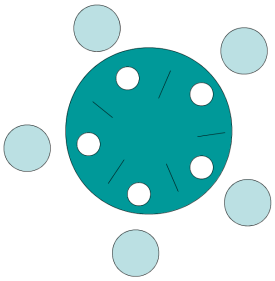
- Na primer, pretpostavimo da oba zadatka, A i B, zahtevaju resurse X i Y da bi mogli da završe svoj posao. Ukoliko se desi da zadatak A dobije resurs X, a zadatak B resurs Y, tada, da bi nastavili sa radom, zadatku A treba resus Y a zadatku B treba resurs X, i oba zadatka čekaju onaj drugi da bi mogli da nastave sa radom. Na taj način oba gube napredak i program ne može da završi sa radom normalno.
- Prethodno opisan način gubitka napretka naziva se uzajamno blokiranje, smrtonosno blokiranje (eng. *deadlock)*
- *Deadlock* je ozbiljna pretnja pouzdanosti programa i zahteva ozbiljna razmatranja i u jeziku i u dizajnu programa.

Koncept napredovanja (eng. *liveness*)

- *Livelock* (živo blokiranje)- kad svi zadaci nešto rade, ali nema progresa
- *Lockout* - *Individual starvation* (individualno izgladnjivanje) - mogućnost da jedan ili više zadataka sprečavaju izvršavanje nekog zadatka.
- Treba obezbediti:
  - uzajamno isključivanje kritičnih sekcija,
  - pristipačnost resursima,
  - poštenost u izvršavanju

Filozofi za večerom

Problem filozofa koji večeraju (obeduju) Hoare, 1985.



http://www.usingcsp.com/cspbook.pdf

3.6 —— Odnos konkurentnosti, potprograma i klasa

Odnos konkurentnosti i potprograma

- Jedinstveno pozivanje — ne sme se istovremeno upotrebljavati u različitim nitima (npr koristi globalne promenljive ili globalne resurse na način koji nije bezbedan)

- Ulazne (engl. reentrant) — može se upotrebljavati na različitim nitima ali uz dodatne pretpostavke (da se ne koriste nad istim podacima)
- Bezbedne po niti (engl. thread-safe) — sme se bezbedno upotrebljavati bez ograničenja
- Slično važi i za klase

Debagovanje konkurentnih programa

- Zašto je teško naći grešku u konkurentnim programima?
- Preporuke bezbednog konkurentnog programiranja

3.7 — Model razmene poruka Actor

Primer modela razmene poruka — *Actor*

- Razmena poruka rešava neke od pomenutih problema
- Primer modela razmene poruka — *Actor*
- Model razmene poruka Aktor podrazumeva arhitekturu SN i može se upotrebljaviti i na distribuiranim sistemima
- Arhitektura SN (*Share Nothing*) podrazumeva da se podaci između čvorova sistema međusobno ne dele, odnosno, podaci se međusobno rasporede i svaki čvor sistema ima odgovornost za svoje podatke ali i potpunu slobodu nad svojim podacima
- Svaka komponenta ovog modela ima svoje stanje koje jeste promenljivo, ali ga nikada ne deli sa drugima.

Primer modela razmene poruka — *Actor*

- Komponente modela Aktor predstavljaju objekti koji se nazivaju Aktori. Oni formiraju hijerarhijsku strukturu u kojoj svaki objekat ima poštansko sanduče i njegov zadatak je da obradi svaku poruku koju dobije
- Poruke koje stižu u sanduče se čuvaju u redu, tj obrađuju se redom koje su stigle
- Aktor reaguje na primljene poruke određenom akcijom, koja zavisi od same poruke
- Kao odgovor na poruku, Aktor može promeniti svoje stanje, može napraviti još komponenata Aktor (svoju decu), slati poruke drugim komponenatama ili zaustaviti rad svoje dece ili sebe
- Aktor ima svoje lokalno stanje koje je promenljivo, ali sa drugima ne deli ništa osim što komunicira porukama
- Aktor kada pošalje poruku nastavlja dalje sa radom (tj, ne blokira se)

Primer modela razmene poruka — *Actor*

- Model Aktor je pogodan u aplikacijama kojima je moguće rasporediti posao na veliki broj manjih, nezavisnih poslova. Tada svaki manji posao predstavlja zadatak jedne komponente tipa Aktor
- Roditelji prikupljaju rezultate izvršavanja svoje dece
- Dizajn aplikacije odgovara načinu organizacije poslovne kompanije gde se poslovi dele po odeljenjima, a odeljenja dele poslove dalje sve dok posao ne postane dovoljno jednostavan da ga može odraditi pojedinačni radnik

Primer modela razmene poruka — *Actor*

- Model Aktor nije pogodan ukoliko je potrebno intenzivno deljenje podataka. Na primer, rad sa bazama podataka i transakcijama je primer kada se programeri obično odlučuju za neko drugo rešenje
- Model Aktor enkapsulira rad sa nitima čime nalazi na višem nivou apstrakcije koja omogućava udobniji rad. Međutim, viši nivo apstrakcije povlači slabije performanse, tj bolje performanse se mogu postići direktnim radom sa nitima umesto sa modelom Aktor

Primer modela razmene poruka — *Actor*

- Biblioteka Akka predstavlja implementaciju modela Aktor na Java virtuelnoj mašini (može se koristiti za Javu, Scalu, Kotlin...)
- Proto.actor — Cross-platform actors for .NET, Go, Java and Kotlin
- CAF — The C++ Actor Framework
- Actix — A Rust library built on the Actor Model
- Erlang koristi Actor model za svoju konkurentnost
- ...

4 Distribuirani sistemi

4.1 Definicija, pristupi, karakteristike, prednosti, izazovi, vrste

Distribuirani sistemi — definicija

- Distribuirani sistem je sistem koji se sastoji od više komponenti koje su locirane na različitim mašinama i koje komuniciraju i koordiniraju sa ciljem da izgledaju kao jedan koherentni sistem krajnjem korisniku
- Komponente koje mogu biti sastavni deo distribuiranog sistema su sve mašine koje imaju mogućnost da se zakače na mrežu, imaju svoju lokalnu memoriju i mogu da komuniciraju putem slanja poruka

Distribuirani sistemi — pristupi i karakteristike

- Osnovni pristupi funkcionisanja distribuiranih sistema
  - Mašine rade zajedno oko zajedničkog cilja i korisnik vidi rezultat kao jedinstven
  - Svaka mašina ima svog korisnika i distribuirani sistem omogućava deljenje resursa ili komunikaciju mašine i korisnika
- Osnovne karakterisitke:
  - sve komponente rade konkurentno (paralelno),
  - ne postoji globalni časovnik (tj ne postoji mogućnost globalne sinhronizacije putem časovnika, već to mora da se softverski obezbedi) i
  - sve komponente mogu da prestanu sa radom neočekivano i nezavisno jedna od druge

Prednosti — horizontalno skaliranje

- Kada sistem raste i kada se zahtevi sistemu povećavaju, potrebno je unaprediti hardver
  - Vertikalno skaliranje
  - Horizontalno skaliranje
- Vertikalno skaliranje — poboljšavanje karakteristika hardvera jedne mašine
- Vertikalno skaliranje je dobro dok je moguće, ali nakon određene tačke lako se uočava da i najbolji hardver nije dovoljan za srednju/veliku količinu saobraćaja/komunikacije/izračunavanja
- Vertikalno skaliranje može da poboljša performanse sve do najnovijih mogućnosti hardvera. Ove mogućnosti se pokazuju kao nedovoljne za kompanije sa srednjim ili većim obimom posla

Prednosti — horizontalno skaliranje

- Horizontalno skaliranje — dodavanje više računara umesto poboljšavanja hardvera pojedinačnog računara
- Važna osobina horizontalnog skaliranja je što ne postoji gornji limit skaliranja: kada se performanse degradiraju potrebno je dodati novu mašinu, i uvek može da se doda nova mašina po potrebi
- Prednost distribuiranog programiranja — horizontalna skalabilnost:
  - S obzirom da se izračunavanje vrši nezavisno na svakom čvoru, jednostavno je i u opštem slučaju nije skupo dodavanje novih čvorova i funkcionalnosti kada je to potrebno

Prednosti — pouzdanost

- Pozdanost — Distriburani sistem u opštem slučaju ne bi trebalo da ima nikakve posledice ukoliko jedna mašina iz bilo kog razloga otkaže. (Većina distribuiranih sistema se razvijaju da budu tolerantni na padove, što je neophodno jer sistemi mogu biti sazdani od stotina čvorova koji rade zajedno.)
- S druge strane, ukoliko postoji samo jedna mašina koja u potpunosti obavlja zadatak, onda se sa njenim padom gubi funkcionalnost u potpunosti.

Prednosti — performanse

- Performanse — Distribuirani sistemi mogu da budu ekstremno efikasni jer posao može da se podeli i pošalje različitim mašinama. Dodatno, distribuirani sistemi mogu da reše neke zadatke i izazove koje nije moguće rešiti na jednoj mašini.
- Komunikacija između čvorova sistema može da predstavlja izazov za performanse jer je vreme za paket koji putuje mrežom fizički ograničeno brzinom svetlosti. Na primer, najkraće moguće vreme za zahtev koji treba da otputuje i da se vrati kroz optički kabl između Njujorka i Sidneja je 160ms. Međutim, dobro dizajnirani distribuirani sistemi dozvoljavaju korišćenje čvorova u oba grada koji omogućavaju da poruka ne putuje do udaljenog čvora već samo do najbližeg čvora.

Izazovi — Raspoređivanje poslova

- Kompleksan dizajn aplikacije, konstrukcija i debagovanje koji su potrebni da bi se izgradio efikasan distribuirani sistem mogu da budu veoma zahtevni i opterećujući
- Raspoređivanje poslova — Distribuirani sistem treba da odluči koji posao treba da se izvršava, kada treba da se izvršava i gde treba da se izvršava. Raspoređivanje svakako ima ograničenja koja u nekim situacijama mogu da vode do nedovoljne iskorišćenosti hardvera ili do nepredvidljivog vremena izvršavanja.

Izazovi — Kašnjenje i posmatranje

- Latentnost, kašnjenje (eng. latency) — Što je šire sistem distrbiuiran, to je veće kašnjenje koje se oseća zbog komunikacije. Ovo obično vodi ka tome da je neophodno birati između dostupnosti, konzistentnosti i latentnosti
- Posmatranje — Prikupljanje, obrada, prikazivanje i praćenje metrika korišćenja hardvera je izazovan zadatak. Padovi su neizbežni i neophodno je ispratiti padove kako bi se bolje razumela njihova priroda i kako bi se napravile odgovarajuće pravovremene rekacije i prevencija. Kada je distribuiran sistem kompleksan, posmatranje i razumevanje padova postaje veliki izazov.

Različite arhitekture distribuiranih sistema

- Distribuirani sistemi u opštem slučaju mogu da imaju različite arhitekture:
  - Klijent-server — Klijenti kontaktiraju server za podatke i onda dobijene podatke formatiraju i prikazuju krajnjem korisniku. Korisnik takođe može da napravi izmene i da ih prenese na server tako da postanu trajne.
  - Troslojna arhitektura (eng. *three-tier*) — Informacije o klijentu su sačuvane u srednjem sloju umesto na klijentu, kako bi se pojednostavila razvoj i stavljanje u rad aplikacije (eng. *deployment*). Ova arhitektura se često koristi kod veb aplikacija.
  - Višeslojna arhitektura (eng. *n-tier*) — Koristi se kada server ima potrebu da prosledi zahtev dodatnim servisima na mreži
  - Ravnopravni učesnici (eng. *peer-to-peer*) — Ne postoje dodatne mašine koje se koriste da obezbede servise ili da upravljaju resursima. Odgovornost je uniformno distribuirana između mašina na sistemu.

Različite arhitekture distribuiranih sistema

- Pored monolitnih aplikacija sa različitim nivoima slojevitosti, za distribuirane sisteme može se koristiti i mikroservisna arhitektura
- Mikroservisna arhitektura uvodi kompleksnost u razne delove razvoja ali ima veliki broj benefita, pre svega u lakšoj podeli poslova, većoj skalabilnosti, povećanoj pouzdanosti i otpornosti na padove, daje mogućnost korišćenja različitih tehnologija...
- Posebno predavanje na temu mikroservisa

4.2 Distribuirani delovi sistema

Šta distribuiramo?

- Distribuirana skladišta podataka
- Distriburane transakcije
- Distriburano izračunavanje
- Distribuirano slanje poruka

4.3 — Distribuirana skladišta podataka

Distribuirana skladišta podataka

- Teorema **CAP** je dokazana 2002. godine. Ona tvrdi da u okviru distribuiranog sistema ne mogu biti istovremeno prisutne osobine konzistentnosti, dostupnosti i tolerantnosti na razdvajanje
  - Konzistentnost (eng. **C***onsistency*) — Ono što se upiše i pročita redom je ono što se očekuje (šta se dešava ukoliko se podaci upišu u jedan čvor a čitaju sa drugog?)

- Dostupnost (eng. **A***vailability*) — Ceo sistem nikada ne umire, tj svaki čvor sistema koji nije otkazao vraća odgovor
- Tolerantnost na razdvajanje (eng. **P***artition tolerant*) — Sistem nastavlja da radi i zadržava zadate garancije konzistentnosti/dostupnosti uprkos mogućim razdvajanjima odnosno prekidima u komunikaciji u okviru mreže

Partition Tolerant — neophodnost

- Ukoliko imamo dva čvora koji prihvataju informacije i njihova veza se prekine — kako oba mogu da budu dostupna i da obezbede konzistentnost? Ukoliko je njihova veza prekinuta, ne postoji način da jedan čvor zna šta drugi čvor radi. To znači da u tom slučaju čvor može ili da prekine sa radom, i da postane nedostupan, ili da nastavi sa radom ali u tom slučaju ne može se garantovati konzistentnost, odnosno podaci postaju nekonzistentni.
- Dakle, na osnovu ovoga, pošto je razdvojenost nešto što se ne može sprečiti, ostaje da se izabere između jake konzistentnosti i dostupnosti u uslovima prekida u komunikaciji u okviru mreže

Availability

- Praksa pokazuje da većina aplikacija više vrednuje dostupnost od konzistentnosti. U zavisnosti od aplikacije, stroga konzistentnost najčešće uopšte nije ni neophodna.
- Takođe, ova vrsta izbora nije uvek uzrokovana potrebom za 100% dostupnošću, već pre zbog toga što zbog latentnosti (kašnjenja) mreže opšta sinhronizacija mašina usled zahteva za strogom konzistentnošću može da bude problematična i može loše da utiče na performanse i druge karakteristike sistema. U skladu sa time, aplikacije obično biraju rešenja koja favorizuju dostupnost

4.4 — Distribuirane transakcije

ACID osobine — Zahtevi za tranaskcije za nedistribuirane baze

- **A***tomicity* — Atomičnost garantuje da će svaka transakcija biti tretirana kao jedinstvena jedinica koja ili uspeva kompletno ili kompletno ne uspeva
- **C***onsistency* — Konzistentnost obezbeđuje da svaka transakcija može da prevede bazu podataka iz jednog ispravnog u drugo ispravno stanje
- **I***solation* — Izolovanost obezbeđuje da će svako konkurentno izvršavanje transakcija ostaviti bazu podataka u istom stanju kao što bi to bilo da su se transakcije izvršavale sekvencijalno
- **D***urability* — Trajnost garantuje da će jednom urađena transakcija biti trajna, tj da će njeni rezultati biti prisutni čak i ukoliko se desi pad sistema

BASE osobine — Zahtevi za tranaskcije za distribuirane baze

- **B***asically* **A***vailable* — Sistem uvek vraća odgovor
- **S***oft state* — Sistem može da se menja vremenom, čak i kada nema novih ulaza (zbog postizanja konzistentnosti)
- **E***ventual consistency* — U odsustvu novih ulaza, podaci će se raširiti po svim čvorovima pre ili kasnije i distriburani sistem će postati konzistentan

4.5 — Distribuirano izračunavanje

Distribuirano izračunavanje

- Distribuirano izračunavanje (eng. *distributed computing*) je tehnika podele velikog zadatka (npr. agregacija 100 milijardi podataka) koji ni jedan pojedinačni računar ne može praktično da izvrši, u puno manjih zadataka, tako da svaki manji može da se izvrši na pojedinačnoj mašini
- Rešavanje početnog problema se onda svodi na podelu velikog zadatka na manje zadatke, izvršavanje tih zadataka paralelno i agregiranje izračunatih podataka na odgovarajući način
- Ovakav pristup omogućava horizontalnu skalabilnost — za veći zadatak potrebno je uključiti veći broj čvorova u izračunavanje

Distribuirano izračunavanje

- Jedan od najranijih inovatora u ovoj oblasti je bio Google koji je zbog potrebe za obradom ogromne količine podataka morao da izmisli novu paradigmu distriburanog izračunavanja — *MapReduce*. Oni su objavili rad na tu temu 2004. godine i zajednica otvorenog koda je kasnije kreirala *Apache Hadoop* koji se bazira na tome
- Slično, Spark

Distribuirano izračunavanje

- Distribuirano izračunavanje može da koristi podelu podataka (kao što je to već opisano) ali i podelu zadataka
- Podela zadataka odgovara distribuiranim aplikacijama kod kojih različiti čvorovi obavljaju različite zadatke (ili različite grupe čvorova obavljaju različite zadatke)
- Primer: Erlang virtuelna mašina omogućava distribuiranje Erlang aplikacija (korišćenjem modela Aktor)
- Primer: Korišćenje mikroservisa za različite servise koje obezbeđuje aplikacija

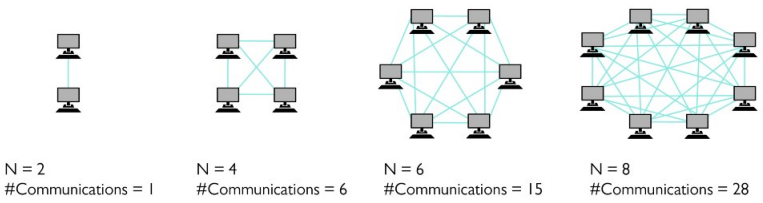
Distribuirano izračunavanje

- Primer, platforma za elektronsko poslovanje
  - Slanje i primanje elektronske pošte o specijalnim ponudama trenutnim korisnicima
  - Sastavljanje liste korisnika i njihovih kupovina kako bi se bolje razumeli proizvodi koji ih iteresuju i kako bi se u skladu sa time uradile odgovarajuće akcije
  - Dopuna liste korisnika sa novim korisnicima koji su se onlajn registrovali
  - Prihvatanje recenzije/pregleda proizvoda iz različitih izvora kako bi se olakšalo buduće donošenje odluka
  - Prihvatanje različitih metoda plaćanja
  - Odgovaranje na onlajn pitanja korisnika, bilo putem zaposlenih ili putem *chatbot*-a.

4.6 — Distribuirano slanje/primanje poruka

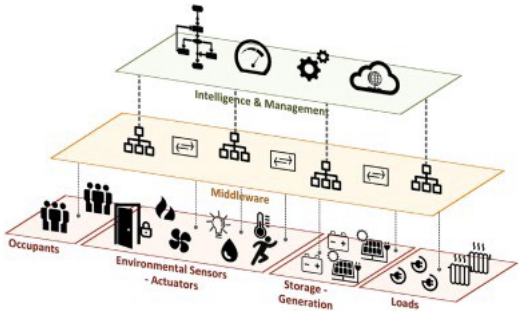
Distribuirano slanje/primanje poruka

- Problem komunikacije slanjem poruka — ko kome šalje poruke, ko prima čije poruke?



Distribuirano slanje/primanje poruka

- Sloj koji je odgovoran za slanje/primanje poruka — *Message Oriented Middleware* (MOM)

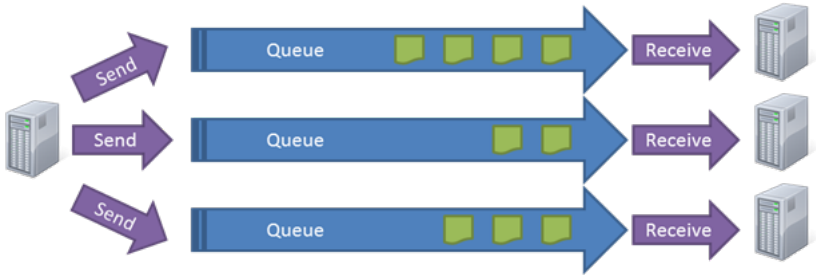


Distribuirano slanje/primanje poruka

- Distribuirano slanje/primanje poruka (engl. *distributed messaging*) se zasniva na konceptu pouzdanog reda poruka (eng. *reliable message queuing*). Poruke se smeštaju u red asinhrono između aplikacija klijenta i sistema za poruke. Distriburani sistem za poruke obezbeđuje pouzdanost (engl. *reliability*), skalabilnost (engl. *scalability*) i trajnost (eng. *persistence*) isporuke/primanja poruka
- Dve vrste sistema
  - *PTP — Point To Point MOM*
  - *Pub/Sub MOMs — Publisher Subscriber MOMs*

Distribuirano slanje poruka

- *PTP — Point To Point MOM* PTP paradigma je paradgima gde se komunikacija vrši *jedan-na-jedan*, odnosno jedan pošiljaoc proizvodi poruku za tačno jednog primaoca

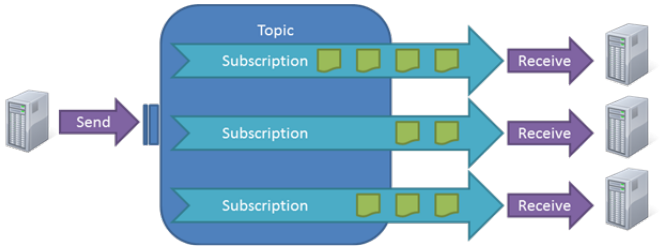


Distribuirano slanje/primanje poruka

- Većina interesantnih obrazaca prate model objavi-pretplati se (engl. *publish-subscribe*), gde pošiljaoc (objavljivač, izdavač, engl. *publisher*) šalje poruku svima koji su se za to pretplatili (engl. *subscriber*)

Distribuirano slanje poruka

- *Pub/Sub MOMs — Publisher Subscriber MOMs* Ovde je glavno obeležje **tema** (engl. *topic*). Izdavač šalje poruke sa temom pri čemu jedan ili više primalaca može da se pretplati na primanje tih poruka. Ovo je paradigma *jedan-prema-puno*



Distribuirano slanje/primanje poruka

- Jednom kada pošiljalac objavi poruku, pretplatioci mogu da prime odabrane poruke uz pomoć opcije fitriranja. Obično postoje dve vrste filtriranja:
  - filtriranje zasnovano na temi (engl. *topic-based filtering*) i
  - filtriranje zasnovano na sadržaju (engl. *content-based filtering*).
- Alati: Apache Kafka, RabbitMQ, JMS (Java Message Service), ActiveMQ, ZeroMQ, Kestrel

Tipovi poruka kod klijent-server arhitekture — zahtev/odgovor

- Obrasci za zahtev/odgovor (engl. *request/response patterns*)
- Na primer, kod klijent-server arhitekture
  - Pošalji i zaboravi (engl. *Fire and Forget*) — Klijent pošalje poruku serveru i ne očekuje da server pošalje nazad odgovor. Klijent šalje poruku kada njemu to odgovara
  - Zahtev za odgovorom (engl. *Request Response*) — Klijent šalje poruku i očekuje da mu server odgovori sa tačno jednim odgovorom. Klijent ne šalje poruku dok nije spreman da prihvati odgovor.

Tipovi poruka kod klijent-server arhitekture — zahtev/odgovor

- Zahtev za tokom (engl. *Request Stream*) — Klijent šalje serveru zahtev i očekuje od servera odgovor u obliku N poruka. Klijent ne šalje zahtev sve dok nije spreman da obradi odgovore, a server ne šalje odgovore pre nego što klijent potvrdi da očekuje N odgovora
- Zahtev za kanalom (engl. *Request Channel*) — Klijent i server se pripreme da emituju i primaju N poruka jedan od drugog. Ni klijent ni server ne počinju slanje poruka dok ne dobiju signal od ovog drugog da su spremni da prime i obrade N poruka.

5 Veza sa programskim jezicima

5.1 Podrška u okviru tradicionalnih programskih jezika

Podrška konkurentnom programiranju

- Gotovo svi kompilirani programski jezici imaju neku vrstu podrške za konkurentno programiranje
  - Standardni jezici: Ada, C/C++, C#, Objective-C/Swift, Java/Scala/Kotlin, Haskel, Lisp, Erlang (na primer, Parallel and Distributed Programming Using C++)
  - Moderni programski jezici: Rust, Go, Elixir, Elm, Clojure
- Iako postoje jezici dizajnirani za distribuirano programiranje, najčešće se kod postojećih jezika koriste dodatne biblioteke koje omogućavaju ovu vrstu programiranja.

5.2 Java, Scala, Kotlin i Clojure

Java, Scala, Kotlin i Clojure

Java, Scala, Kotlin i Clojure

- Jezici Scala, Kotlin i Clojure se oslanjaju prevenstveno na JVM
- Scala i Kotlin su mešavine OO i funkcionalnog programiranja dok je Clojure funkcionalni jezik
- U Scali i Kotlinu postoji podrška za konkurentno programiranje i model aktor
- Clojure je razvijan sa ciljem rešavanja konkurentnog programiranja

Programski jezik Java

Programski jezik Java

- Java je objektno orijentisani jezik, nastao 1995. godine, Sun Microsystems
- Razvoj i održavanje Oracle Corporation https://www.oracle.com/java/
- Uvodi ideju o prenosivosti: kôd se kompilira do bajtkoda koji se zatim interpretira na JVMu: *Write Once, Run Anywhere (WORA)*

- Na razvoj Jave uticali su jezici Objective-C, C++, Smalltalk, i Eiffel
- Java je uticala na razne programske jezike, pre svega na Scala-u, Kotlin i Clojure koje koriste JVM
- Java je jezik opšte namene, sa velikim primenama u industriji, jedan od najznačajnijih i najzastupljenijih jezika

Programski jezik Java

- Java je multiparadigmatski jezik, daje podršku i za višenitno i za distribuirano programiranje
- Distribuirano programiranje u Javi:
  - Distribuirano *map-reduce* programiranje *Hadoop* i *Spark* razvojni okviri
  - Klijent-server programiranje *Java Socket* i *Remote Method Invocation* interfejsi
  - Programiranje slanjem poruka korišćenjem *Message Passing Interface*
- Akka za Javu i Scalu https://akka.io/ (implementacija modela aktor)
- ...

Programski jezik Java

Hello.java

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

```
$ javac Hello.java
$ java Hello
```

Programski jezik Scala

Programski jezik Scala

- Scala https://www.scala-lang.org/
- **SCA**lable **LA**nguage, jezik koji raste sa potrebama korisnika i sistema
- Scala je jezik opšte namene nastao 2003. godine na univerzitetu EPFL

- Cilj: prevazići ograničenja jezika Java **kombinovanjem OO i funkcionalne paradigme**
- Pogodna za rešavanje različitih problema, počevši od malih nezahtevnih programa, sve do velikih i kompleksnih sistema
- Na razvoj Scale su uticali Lisp, Erlang, Haskell, Java, OCaml

Programski jezik Scala

- Osnovne osobine:
  - Kompatibilnost sa Javom (korišćenje biblioteka, klasa, tipova, interfejsa)
  - Konciznost — kod napisan u Scali je obično kraći od odgovarajućeg koda u Javi
  - Statički tipiziran, jako zaključivanje tipova (type inference)
- Podrška za konkurentno i distribuirano programiranje
  - Biblioteka Akka (model aktor)
  - Distribuirano izračunavanje kroz Spark https://spark.apache.org/docs/0.9.1/scala-programming-guide.html

Programski jezik Scala

- Osnovno korišćenje Skale je na Java VM
- Scala.js je Scala kompajler koji kompilira Scala kod u JavaScript, i time omogućava da se pišu Scala programi koji mogu da se izvršavaju u veb brauzerima
- Scala Native je Scala kompajler koji omogućva korišćenje LLVM kompajlerske infrastrukture kako bi se kreirala izvršna verzija koja koristi prednosti nativnog izvršavanja
- Kompajler sa ciljem spuštanja Scala jezika na .NET razvojno okruženje i njegov *Common Language Runtime* je razvijan od 2004. godine do 2012. godine kada je taj razvoj napušten.

Programski jezik Scala

ekstenzije .scala, .sc Hello.scala

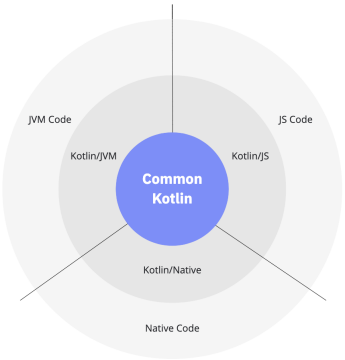
```
object Hello {
    def main(args: Array[String]) = {
        println("Hello, world")
    }
}

$ scalac Hello.scala
$ scala Hello
```

Programski jezik Kotlin

Programski jezik Kotlin

- Kotlin https://kotlinlang.org/
- Pojavio se 2011. godine, razvija ga JetBrains
- Podrška za **multiplatformsko programiranje** je jedan od ključnih prednosti koje Kotlin nudi. Ona smanjuje vreme pisanja i održavanja koda za različite platforme, pri čemu Kotlin zadržava fleksibilnost i benefite prirodnog programiranja
- Na razvoj su uticali Java, Scala, Swift, ML, C#, JavaScript
- Multiparadigmatski: OO programiranje, funkcionalno programiranje, imperativno programiranje, konkurentno programiranje, generičko programiranje

Programski jezik Kotlin



- Zajednički (engl. *common*) Kotlin uključuje jezik, osnovne biblioteke i osnovne alate. Kod napisan na zajedničkom Kotlinu radi na svim platformama
- Sa Kotlinovim multiplatformskim bibliotekama, može se koristiti multiplatformska logika i u zajedničkom i u platformski specifičnom kodu
- Platformski specifične verzije Kotlina (Kotlin/JVM, Kotlin/JS, Kotlin/Native) uključuju proširenja jezika Kotlin i platformski specifične biblioteke i alate.

Programski jezik Kotlin

hello.kt (ekstenzije .kt, .kts, .ktm)

```
fun main(args : Array<String>) {
    println("Hello, World!")
}

prevodjenje u nativni izvršni fajl
(llvm zasnovan kompajler)
kotlinc-native hello.kt -o hello
```

Za razliku od Jave, nije obavezno kreirati klasu za svaki Kotlin program, ali će kompajler za Kotlin da kreira odgovarajuću klasu umesto programera.

Funkcija `println()` poziva interno `System.out.println()`.

Programski jezik Kotlin

- Može se koristiti svugde gde se koristi Java, kao što je razvoj serverske strane aplikacija ili Android aplikacija.
- Odlično radi sa svim postojećim Java bibliotekama i radnim okvirima, a ima iste performanse kao Java.

- https://kotlinlang.org/docs/coroutines-guide.html
- U Kotlinu, korutine su lake niti (slično kao i u programskom jeziku Go)
- kotlinx.coroutines je bogata biblioteka za rad sa korutinama
- Mogu se koristiti kanali za komunikaciju
- Podrška za model actor

Programski jezik Clojure

Programski jezik Clojure

- Clojure https://clojure.org/
- Clojureje dinamički programski jezik opšte namene nastao 2007. godine. Clojure kombinuje pristupačnost i interaktivni razvoj skriptnog jezika sa efikasnom i robusnom infrastrukturom za višenitno programiranje.
- Clojure je kompajlirani jezik, ali je i dalje potpuno dinamičan — svaka funkcionalnost i osobina koju podržava Clojure se razrešeava u toku izvršavanja.

Programski jezik Clojure

- Clojure obezbeđuje lak pristup Javinim okvirima za razvoj (engl. *Java frameworks*)
- Clojure je dijalekat Lisp-a koji nije opterećen sa *backwards compatibility*
- Clojure je pretežno funkcionalni programski jezik i sadrži bogat skup nepromenljivih i postojanih struktura podataka.
- Kada su potrebna promenljiva stanja, Clojure nudi posebno softversko rešenje, koje se naziva sistem transakcione memorije (engl. *transactional memory system*) i sistem reaktivnih agenata (engl. *reactive agent system*) sa cilje da obezbedi čist i korektan višenitni dizajn koda
- Robustan, praktičan i brz

Programski jezik Clojure

Clojure ukratko:

- Vrsta Lisp-a
- Za funkcionalno programiranje
- Simbioza sa etabliranom platformom (JVM)
- Dizajniran za konkurentnost

Programski jezik Clojure

- Pojavio se 2007. godine, poslednja stabilna verzija iz marta 2021.
- Uticaji: Java, Lisp, C++, Erlang, Haskell, ML
- Tekućim razvojem upravlja zajednica, a komercijalnu podršku obezbeđuje kompanija Cognitect
- Pored osnovne podrške za JVM, postoji i podrška za JavaScript i .NET framework

Programski jezik Clojure

ekstenzije .clj .cljs .cljc

```
;; A typical entry point of a Clojure program:
;;   `-main` function
(defn -main ; name
  [& args] ; (variable) parameters
  (println "Hello, World!")) ; body
```

5.3 Erlang, Elixir i Elm

Erlang, Elixir i Elm

Erlang, Elixir i Elm

- Erlang, Elixir i Elm su funkcionalni programski jezici
- Erlang daje osnovu za građenje skalabilnih distribuiranih aplikacija otpornih na padove sa stalnom dostupnošću
- Elixir se oslanja na Erlangovu virtuelnu mašinu
- Elm se često koristi u kombinaciji sa Elixirom

Programski jezik Erlang

Programski jezik Erlang

- https://www.erlang.org/
- Ime Erlang dolazi od danskog matematičara koji se zvao Agner Krarup Erlang, kao i od skraćenice za "**Er**icsson **Lang**uage".
- Osnovne paradigme: funkcionalna i konkurentna

Programski jezik Erlang

- Erlang je jezik koji se koristi za građenje skalabilnih sistema za rad u realnom vremenu (engl. *real-time*) koji imaju potrebu za stalnom dostupnošću (engl. *availability*) i koje neprestano pružaju usluge (engl. *non-stop applications*)
- Primeri: telekomunikacija, bankarstvo, elektronska trgovina (engl. *e-commerce*), kompjuterska telefonija (engl. *computer telephony*) i slanje poruka sa trenutnim odzivom (engl. *instant messaging*)
- Takođe, važna osobina je mogućnost zamene delova koda bez zaustavljanja sistema (engl. *hot swapping*)

Programski jezik Erlang

- Erlangov *runtime* sistem (podrška koja je neophodna da se obezbedi i koja omogućava razvoj aplikacija sa prethodno opisanim osobinma) se naziva OTP
- Erlang/OTP se razvijaju i održavaju u okviru dela firme Ericsson koji se naziva *Open Telecom Platform* (OTP).
- OTP ima ugrađenu podršku za konkurentnost, distribuiranost i za toleranciju grešaka/padova
- Dodatno, OTP uključuje podršku za distribuirane baze podataka, interfejse prema drugim jezicima, debagovanje i slično

Programski jezik Erlang

- Erlang je nastao 1986. godine u okviru firme Ericsson, a 1998. godine je objavljen i postao je otvorenog koda
- Iako je prvobitno izgrađen za telekomunikacione sisteme, Erlang ni na koji način nije specijalizovan za ovaj domen, ne sadrži specijalzovanu podršku za programiranje telefona i drugih telekomunikacionih uređaja.
- U vreme njegovog nastajanja, programi su se uglavnom koristili bez komunikacije sa nekim veb servisom (eng. *desktop-based sotfware*), pa je upotreba Erlang-a bila ograničena na telekomunikacione sisteme.
- Međutim, zahtevi modernih sistema i aplikacija poklapaju se sa nefunkcionalnim zahtevima za koje Erlang pruža podršku, tako da je u poslednje vreme privukao veliku pažnju.

Programski jezik Erlang

- Početna verzija Erlang-a je bila implementirana u Prologu i na nju je uticao jezik Plex, koji se razvialo i koristio u okviru Ericsson-a
- 1992. godine počela je izgradnja BEAM virtuelne mašine koja je kompilirala Erlang u C i koja je omogućila potrebnu efikasnost na osnovu koje je krenula i industrijska upotreba Erlang-a
- Joe Armstrong: "If Java is 'write once, run anywhere', then Erlang is 'write once, run forever'."
- Primeri sistema koji se izvršavaju na Erlangovoj virtuelnoj mašini WhatsApp, WeChat, Riak distribuirana baza, Heroku Cloud.

Programski jezik Erlang

Normalna Erlang aplikacija je izgrađena od stotine malih Erlang procesa:

- Sve je proces
- Procesi su jako izolovani
- Kreiranje i uništavanje procesa je laka/efikasna operacija
- Jedini način interakcije između procesa je slanjem poruka
- Procesi i maju jedinstvena imena
- Ako znaš ime procesa, možeš da mu pošalješ poruku
- Procesi međusobno ništa ne dele
- Rukovanje greškama nije lokalno
- Procesi mogu da urade šta treba ili da padnu (engl. *fail*)

Programski jezik Erlang

- Erlang sadrži nepromenljive podatke, poklapanje obrazaca, funkcionalno programiranje
- Sekvencijalni (imperativni) podskup Erlanga podržava vrednu evaluaciju (engl. *eager evaluation*), jedinstvenu dodelu i dinamičko određivanje tipova
- Ekstenzije .erl, .hrl

```
-module(hello).
-export([hello_world/0]).

hello_world() -> io:fwrite("hello, world\n").
```

Programski jezik Erlang

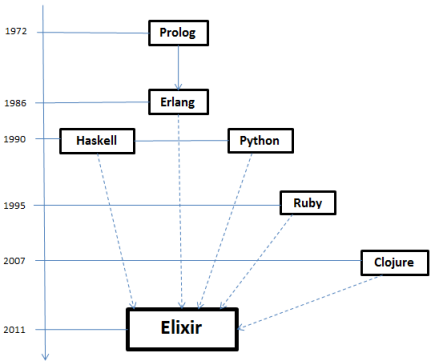
```
-module(fact). % This is the file 'fact.erl', the module and the filename must match
-export([fac/1]). % This exports the function 'fac' of arity 1 (1 parameter, no type, no name)

fac(0) -> 1; % If 0, then return 1, otherwise (note the semicolon ; meaning 'else')
fac(N) when N > 0, is_integer(N) -> N * fac(N-1).
% Recursively determine, then return the result
% (note the period . meaning 'endif' or 'function end')
%% This function will crash if anything other than a nonnegative integer is given.
%% It illustrates the "Let it crash" philosophy of Erlang.
```

Programski jezik Elixir

Programski jezik Elixir

- Elixir https://elixir-lang.org/
- Elixir je nastao godine 2011. godine. Njegovim tvorcem se smatra José Valim.
- Poslednja stabilna verzija mart 2021.
- Elixir je dizajniran za izgradnju skalabilnih i lako održivih aplikacija.
- Poseduje jednostavnu i modernu sintaksu.
- Elixir je funkcionalni jezik, i s obzirom da korsiti Erlangovu virtuelnu mašinu ima izuzetno dobru podršku za rad u distrubiranim sistemima i toleranciju na greške
- U Elixir-u je rađeno mnogo zanimljivih projekata iz oblasti robotike. Takođe se uspešno koristi u razvoju veba i u domenu softvera za uređaje sa ugrađenim računarom

Programski jezik Elixir



Programski jezik Elixir

- Procesi koji ne dele međusobno ništa i komunikacija slanjem poruka (Actor model)
- Erlangove funkcije mogu da budu pozivane direktno iz Elixira i obrnuto, bez uticaja ne vreme izvršavanja, s obzirom da se oba jezika kompiliraju do Erlangovog bajtkoda
- Na razvoj su uticali Clojure, Erlang, Ruby
- Po ugledu na Python koristi sistem za dokumentaciju

Programski jezik Elixir

ekstenzije .ex, .exs hello.exs

```
IO.puts "Hello, World!"
```

```
$ elixir hello.exs
```

Programski jezik Elm

Programski jezik Elm

- Elm https://elm-lang.org/
- Elm je nastao 2012. godine, Evan Czaplicki: "Elm: Konkurento FRP (funkcionalno reaktivno programiranje) za funkcionalne GUI-je"
- Elm je domenski specifičan jezik namenjen isključivo za kreiranje korisničkog interfjesa veb aplikacija, koji ima za cilj da GUI programiranje učini prijatnijim

Programski jezik Elm

- Elm omogućava lako korišćenje, performanse i robusnost
- Elm je statički tipiziran, čisto funkcionalni programski jezik koji se kompilira, tačnije transpilira u JavaScript
- Osnovna filozofija: "No runtime exceptions in practice"
- Koristi se često u kombinaciji sa Elixirom
- Na razvoj uticali Haskell, Standard ML, OCaml, F#

Programski jezik Elm
https://elm-lang.org/examples ekstenzija .elm

```
module HelloWorld exposing (..)

import Html exposing (text)

main =
  text "Hello!"
```

5.4 Rust i Go

Rust i Go


Rust i Go

- Rust i Go po svojoj sintaksi pripadaju familiji C jezika
- Oba jezika su nastala sa ciljem rešavanja problema konkurentnosti
- Rust je dominantno funkcionalan, dok je Go dominantno imperativan jezik

Programski jezik Rust

Programski jezik Rust

- Rust www.rust-lang.org
- Rust se pojavio 2010. godine
- Inicjalno nastala u Mozzila Research
- Februar 2021, osnovana Rust Foundation: AWS, Huawei, Google, Microsoft, i Mozilla
- Poslednja verzija je iz marta 2021.

Programski jezik Rust

- https://doc.rust-lang.org/stable/book/
- Rust je multiparadigmatski jezik dizajniran sa ciljem visokih performansi i sigurnosti, sa posebnim akcentom na bezbednu upotrebu memorije i konkurentnosti.
- Rust pripada C familiji programskih jezika, sintaksa je slična jeziku C++
- Uticaji jezika C++, Erlang, Haskell, ML, Ruby, Swift
- Funkcionalna, imperativna, konkurentna, generička paradigma
- Rust je, počevši od 2016. godine, svake godine izglasan da je najvoljeniji programski jezik na istraživanjima *Stack Overflow Developer Survey*

Programski jezik Rust

ekstenzija .rs

```
fn main() {
    println!("Hello, World!");
}

fn factorial(i: u64) -> u64 {
    match i {
        0 => 1,
        n => n * factorial(n-1)
    }
}
```

```
$ rustc hello.rs
$ ./hello
Hello World!
```

Programski jezik Rust

- Drugi osnovni cilj Rust-a je da omogući pisanje bezbednih i efikasnih konkurentnih programa
- Inicijalno, tim koji je razvijao Rust je mislio da su obezbeđivanje bezbedne upotrebe memorije i sprečavanje problema sa konkurentnošću dva različita problema koji treba da se rešavaju na različite načine. Međutim, vremenom, tim je otkrio da su pristupi koji se zansnivaju na vlasništvu podataka i na sistemu tipova moćan skup alata da se upravlja i bezbednom upotrebom memorije i problemima sa konkurentnošću.

Programski jezik Rust

- Rust obezbeđuje da su mnoge greške sa konkurentnošću zapravo greške u vreme kompilacije umesto greške u fazi izvršavanja
- Prema tome, umesto da se provede ogromna količina vremena za reprodukciju odgovarajuće situacije u kojoj dolazi do konkurentne greške u fazi izvršavanja, nekorektan kôd će biti odbijen u fazi kompilacije i pritom će biti prikazana i poruka o grešci koja i objašnjava problem

- Kao rezultat ovakvog pristupa, greške se ispravljaju u fazi kodiranja umesto potencijalno nakon što se kôd isporuči
- Ovo osobina Rust-a se naziva *nezastrašujuća* konkurentnost (engl. *fearless concurrency*)

Programski jezik Rust

- Nezastrašujuća konkurentnost omogućava pisanje koda koji ne sadrži skrivene greške i koji je lak za refaktorisanje bez uvođenja novih grešaka
- U okviru Rust-a, podržan je i model slanja poruka i model deljene memorije
  - U modelu slanja poruka, koriste se kanali da se šalju poruke između niti
  - U modelu deljene memorije, više niti ima pristup istom podatku

Programski jezik Go

Programski jezik Go

- Programski jezik Go je projekat koji razvija kompanija Google od 2007. godine.
- Postao je javno dostupan kao projekat otvorenog koda 2009. godine i u stalnom je razvoju.
- **Cilj projekta:** kreirati novi statički tipiziran jezik koji se kompilira, a koji omogućava jednostavno programiranje kao kod interpretiranih, dinamički tipizarnih programskih jezika.
- Smatra se da Go pripada C familiji programskih jezika, ali pozajmljuje i adaptira ideje raznih drugih jezika, izbegavajući karakteristike koje dovode do komplikovanog i nepouzdanog koda.

Programski jezik Go

- Go nije objektno-orijentisan, ali podržava određene koncepte kao što su metodi i interfejsi koji pružaju fleksibilnu apstrakciju podataka.
- Go omogućava efikasnu konkurentnost koja je ugrađena u sam jezik, kao i automatsko upravljanje memorijom, odnosno sakupljanje smeća.

- Zbog svojih osobina kao što je konkurentnost, Go je posebno dobar za izradu različitih vrsta serverskih aplikacija, ali je pre svega jezik opšte namene pa se može koristiti za rešavanje svih vrsta problema.
- Go je jezik opšte namene i ima primenu u raznim oblastima
- Osim kompanije Google, koristi se u velikom broju drugih kompanija kao alternativa jezicima poput Python-a i Javascript-a, jer pruža znatno viši stepen efikasnosti i bezbednosti.

Programski jezik Go

ekstenzija .go hello.go

```
package main
import "fmt"

func main() {
    fmt.Println("hello world")
}

$ go run hello-world.go
```

Programski jezik Go

- Konkurentnost u programskom jeziku Go ostvaruje se pomoću gorutina
- Gorutine su laki procesi
  - Gorutine se kreiraju sa stekom male veličine od nekoliko kilobajta, koji u zavisnosti od potreba može da se proširuje. Ova karakteristika omogućava kreiranje i izvršavanje velikog broja gorutina. Go poseduje sopstveni M:N planer izvršavanja gorutina koji mapira proizvoljan broj gorutina M u proizvoljan broj niti operativnog sistema N, što omogućava da se više gorutina može mapirati u jednu nit operativnog sistema.
- Komunikacija može da se ostvari putem deljene memorije ili slanjem poruka korišćenjem kanala

Programski jezik Go

Primer 1: Upotreba kanala

```
func sum(s []int, c chan int) {
    res := 0
    for _, v := range s {
        res += v
    }
    c <- res
}
func main() {
    s := []int{1, 2, 3, 4, 5, 6}
    c := make(chan int)
    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)
    x, y := <-c, <-c
    fmt.Println(x, y, x+y)  // 15 6 21
}             // redosled ispisa brojeva 15 i 6 varira
```

6 Pitanja i literatura

6.1 Pitanja

Pitanja

- Šta je konkurentna paradigma?
- Da li su ideje o konkurentnosti nove? Zbog čega je konkurentnost važna?
- Koji su osnovni nivoi konkurentnosti?
- Koje su vrste konkurentnosti u odnosu na hardver? Kako se to odnosi na programere i dizajn programskog jezika?
- Šta je osnovni cilj koji želimo da ostvarimo razvijanjem konkurentnih algoritama?

Pitanja

- Koji su osnovni razlozi za korišćenje konkurentnog programiranja?
- Navesti primer upotrebe konkurentnog programiranja za podršku logičkoj strukturi programa.
- Da li je dobijanje na brzini moguće ostvariti i na jednoprocesorskoj mašini?
- Koji je hijerarhijski odnos u okviru konkurentne paradigme?
- Šta je zadatak? Na koji način se zadatak razlikuje od potprograma?

Pitanja

- Koje su osnovne kategorije zadataka i koje su karakteristike ovih kategorija?
- Šta je paralelizacija zadataka?
- Šta je paralelizacija podataka?
- Navesti primere paralelizacije zadataka i paralelizacije podataka.
- Koji je odnos ovih paralelizacija?

Pitanja

- Šta je komunikacija?
- Koji su osnovni mehanizmi komunikacije?
- Šta karakteriše slanje poruka u okviru iste mašine, a šta ukoliko je slanje poruka preko mreže?
- Šta je sinhronizacija?
- Kakva je sinhronizacija u okviru modela slanja poruka?
- Kakva je sinhronizacija u okviru modela deljene memorije?
- Koje su dve osnovne vrste sinhronizacije u okviru modela deljene memorije?

Pitanja

- Objasniti sinhronizaciju saradnje.
- Šta je uslov takmičenja?
- Koji su načini implementiranja sinhronizacije?
- Šta je koncept napredovanja?
- Navesti primer uzajamnog blokiranja.
- Navesti primer živog blokiranja.
- Navesti primer individualnog izgladnjivanja.

Pitanja

- Koje vrste uzajamnog isključivanja postoje?
- Koji je odnos konkurentnosti i potprograma/klasa.
- Opisati problem filozofa za večerom.
- Semantika muteksa i katanaca.
- Nedostaju pitanja za distriburane sisteme i vezu sa jezicima

6.2 Literatura

Literatura

- Programming Language Pragmatics, Third Edition, 2009 by Michael L. Scott
- Concepts of Pprogramming Languages, Tenth Edition, 2012 Robert W. Sebasta