Programske paradigme

— Konkurentno programiranje —

Milena Vujošević Janičić

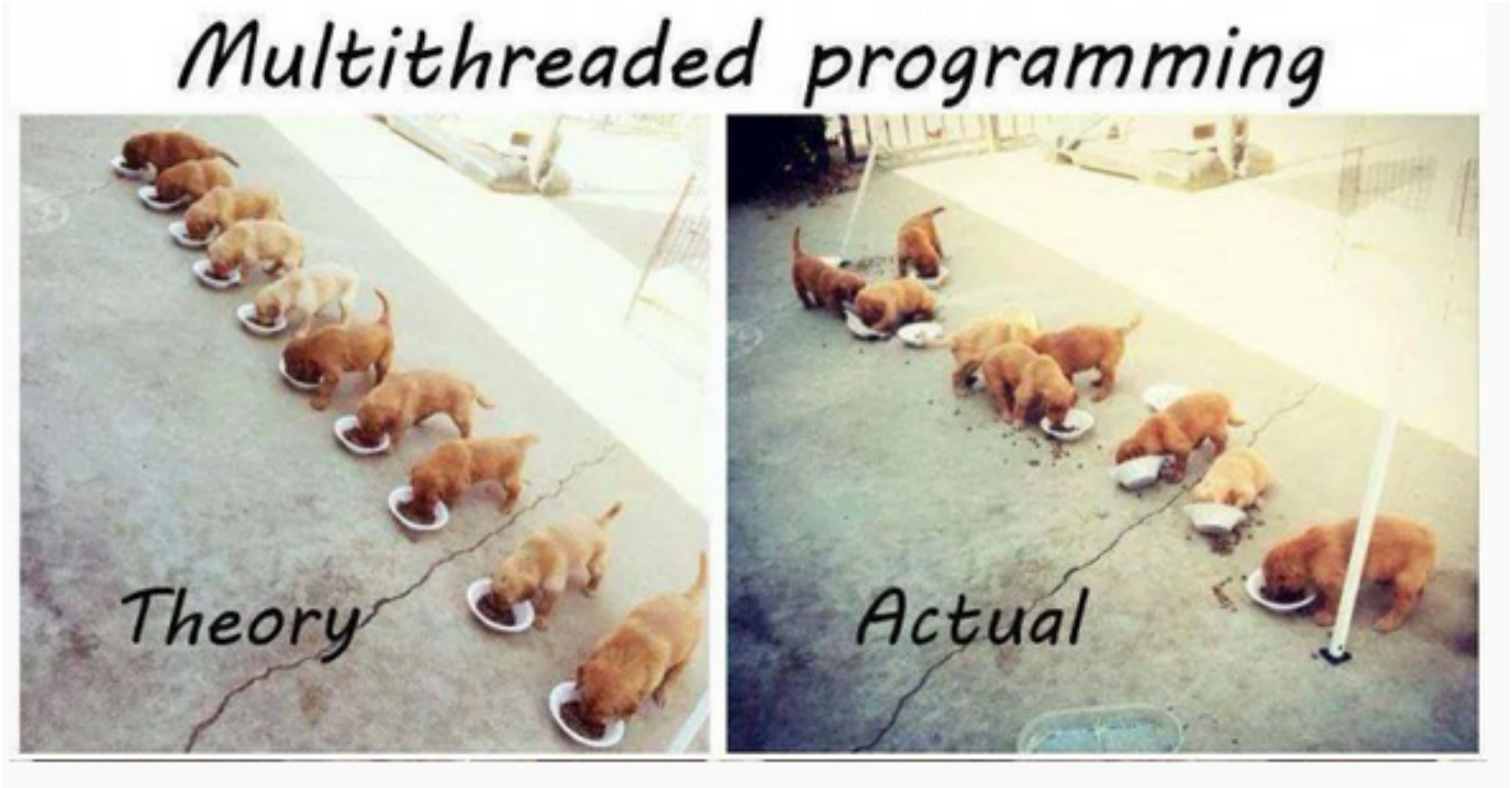
www.matf.bg.ac.rs/~milena

Matematički fakultet, Univerzitet u Beogradu

Pregled

Pregled

Teorija i praksa konkurentnog programiranja



Konkurentna paradigma

Konkurentna paradigma

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

- Koji su razlozi (motivacija) za korišćenje konkurentnog programiranja?
  - Podrška logičkoj strukturi problema
  - Dobijanje na brzini
  - Rad sa fizički nezavisnim uređajima

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

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 je u velikoj meri nezavisan, 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.

Dobijanje na brzini

- Ukoliko imamo više procersora, 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 pitanji) ili slanjem poruka.
- 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.

Zajednički rad fizički nezavisnih uređaja

- Aplikacije koje rade distribuirano korišćenjem različitih mašina, bilo da su u pitanju lokalno povezane mašine ili Internet — distribuirano programiranje
- Procesi međusobno šalju poruke da bi razmenili informacije.
- Može se shvatiti kao vrsta paralelnog izračunavanja ali sa drugačijom međusobnom komunikacijom koja nameće nove izazove.
- Postoje jezici dizajnirani za distrbuirano programiranje, ali oni nisu u širokoj upotrebi

Veza sa programskim jezicima

- Na prvi pogled, konkurentnost može da izgleda kao jednostavan koncept.
- Pisanje konkurentnih programa je značajno teže od pisanja sekvencijalnih programa.
- Za osnovne koncepte konkurentnog programiranja potrebno je obezbediti odgovarajuću podršku u programskom jeziku.

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.
- Zadatak je implementacije jezika da korišćenjem operativnog sistema preslika logičku konkurentnost u odgovarajući hardver.

Veza sa programskim jezicima

- Osnovni problemi se odnosi na pitanja komunikacije i sihronizacije procesa, kao i pitanje pristupa zajedničkim podacima.
- Veliki izazov za programere, dizajnere programskih jezika i dizajnere operativnih sistema (dobar deo podrške za konkurentnost obezbeđuje operativni sistem)

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.
  - Distribuirana paradigma specijalizacija paralelene paradigme u kojoj su procesori i fizički razdvojeni.
- Međutim, sa stanovišta semantike ove podele nemaju značajnu ulogu, one su bitne po pitanju implementacije i performansi.

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.

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 i portabilnost

- Amdahlov zakon — mali delovi programa koji se ne mogu paralelizovati će ograničiti mogućnost ukupnog ubrzavanja paralelizacijom (npr ako je α =10% koda koji ne može paralelizovati, onda je maksimalno ubrzanje paralelizacijom $1/\alpha$, tj u ovom slučaju 10 puta, bez obzira na broj procesora koji se dodaju).
- Dodatno usporavanje — komunikacija.
- Portabilnost se odnosi na nezavisnost od konkretne arhitekture — životni vek hardvera je kratak, stalno izlaze nova hardverska rešenja i dobar algoritam mora da ne zavisi od hardvera.

Pregled

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.

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 se razlikuje od potprograma na tri načina
  - Zadaci mogu da počinju implicitno, ne moraju eksplicitno da budu pozvani.
  - Kada program pokrene neki zadatak, ne mora uvek da čeka na njegovo izvršavanje pre nego što nastavi sa svojim.
  - Kada se izvršavanje zadatka završi, kontrola ne mora da se vrati na mesto odakle je počeli izvršavanje.
- Svaki zadatak u programu može da bude podržan od strane jedne kontrolne niti ili procesa.

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.

Teški zadaci

- Teškima 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 (stanje izvršavanja koje može biti spreman, radi, čeka...), 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.

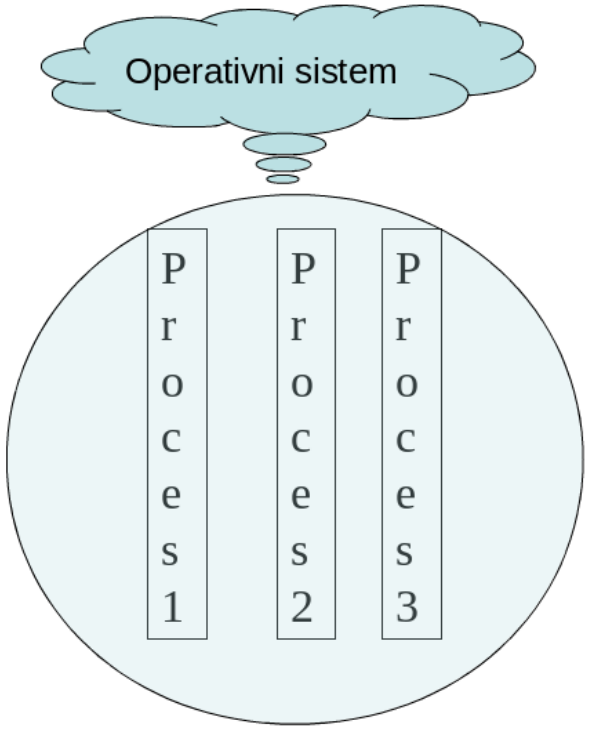
Upravljanje lakim zadacima

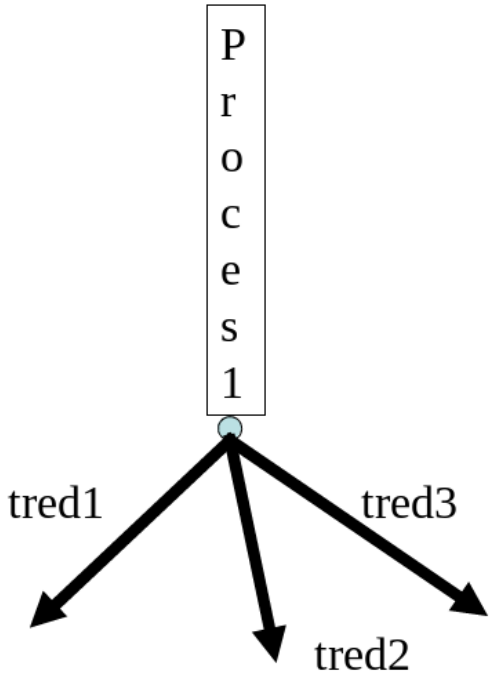
- Upravljanje nitima ostvaruje se preko korisničkih biblioteka. Tri najpoznatije biblioteke su:
  - POSIX Pthreads
  - Java threads
  - Win32 threads
- Kada je reč o opertivnim sistemima, niti su podržane u:
  - Windows XP/2000 , Vista, 7, ...
  - Linux
  - Solaris, Tru64 UNIX, Mac OS X

Primer

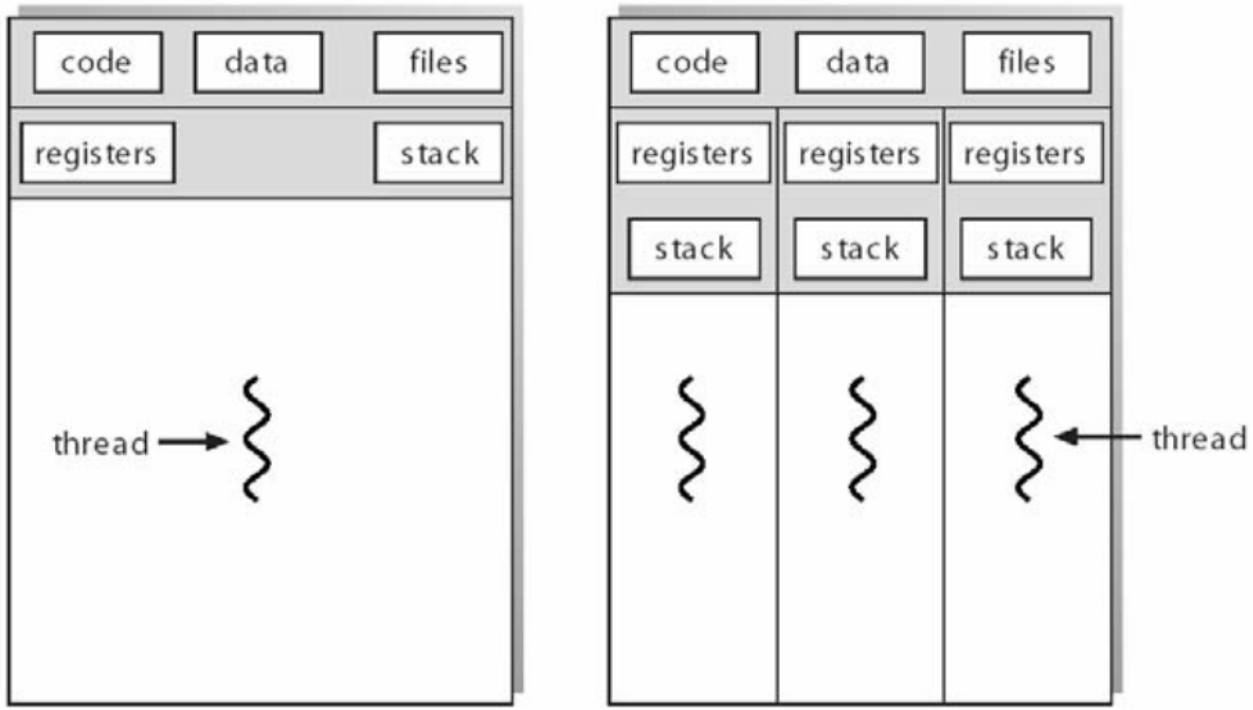
- Terminologija je nekonzistentna
- Npr, u C-u teški zadaci su procesi (koji imaju svoj adresni prostor), laki zadaci su niti (koje imaju zajednički adresni prostor).
- Kreiranje lakih zadataka je efikasnije od kreiranja teških.
- Sistemski poziv `fork` za kreiranje novih procesa
- Biblioteka `pthread`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

Odnos teških i lakih zadataka (procesa i niti)





Odnos teških i lakih zadataka (procesa i niti)



Paralelizacija zadataka ili podataka

- Osnovna odluka koju programer mora da donese kada piše paralelni program je kako da podeli posao.
- Šta paralelizovati?
- 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.

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 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. Ili, igrica može da koristi posebnu nit za svaki objekat koji se pokreće.

Primer

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

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

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.

Primer

- Proizvod komponenti vektora
- Skalarni proizvod dva vektora
- Množenje matrica
- Pronalaženje prostih brojeva u nizu
- Pronalaženje brojeva koje imaju određenu osobinu (prosti, savršeni, blizanci, Armstrongovi...) i pripadaju intervalu [n,m]

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) ili slanjem 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

Slanje poruka

- Slanje 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
- Slanje poruka može se ostvariti na razne načine
- Mogu se koristiti tokovi, signali, cevi, soketi, kanali (mehanizmi međuprocesne komunikacije)...

Pouzdanost slanja poruka

- Ukoliko je slanje poruka u okviru iste mašine, onda se ono smatra pouzdanim.
- Slanje poruka u distribuiranim sistemima nije pouzdano jer podaci putuju kroz mrežu gde mogu da se zagube i tu su potrebni dodatni mehanizmi i protokoli komunikacije
- 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

Sinhronizacija kod deljene memorije — saradnja

- Postoje dve vrste sinhronizacije 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

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

Sinhronizacija kod deljene memorije — takmičenje

- Situacija koja vodi do ovih problema naziva se uslov takmičenja (eng. race condition) jer se dva ili više zadataka takmiče 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

- Iako se široko koriste, semafori se takođe 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)

Implementacija sinhronizacije

- I kod deljene memorije i kod slanja poruka, sinhronizacija može da se implementira na dva načina: zauzetim čekanjem ili blokiranjem
- 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

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*)

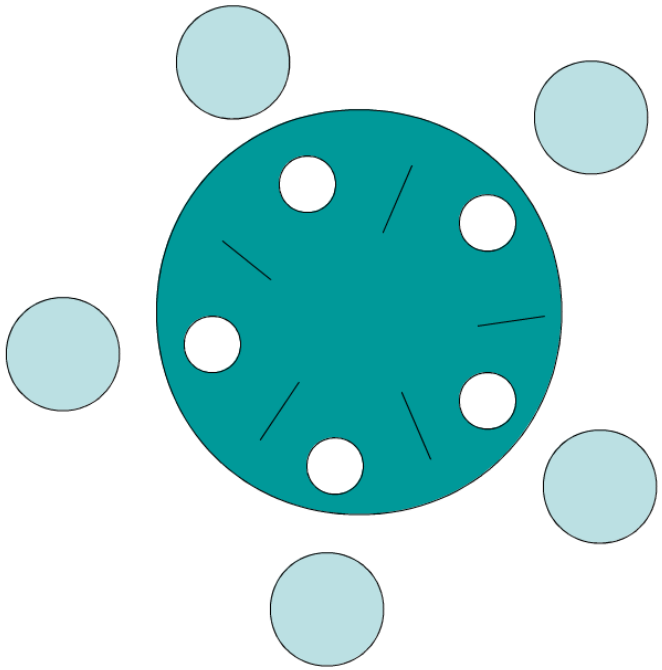
- Na primer, pretpostavimo da oba zadatka, A i B, zahtevaju resurse X i Y da bi mogli da završe svoj posao. Ukoliko se desi da zadatak A dobije resurs X, a zadatak B resurs Y, tada, da bi nastavili sa radom, zadatku A treba resus Y a zadatku B treba resurs X, i oba zadatka čekaju onaj drugi da bi mogli 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 zadatak sprečava izvršavanje drugog.
- 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

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

Pregled

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.

Literatura

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