Paralelno izvršavanje

Podsetnik

- Paralelno izvršavanje = više zaista nezavisnih lanaca izvršavanja instrukcija
  - pretpostavka: svaki od ovih lanaca vidi istu (do izvesnih granica) sliku memorije
  - u suprotnom, sistem je distribuiran
- Konkurentno izvršavanje = sukcesivno izvršavanje odlomaka lanaca instrukcija vezanih za pojedinačne procese tako da se dobije iluzija istovremenog rada
- Konkurentno i paralelno izvršavanje se mogu kombinovati

Terminologija

- Proces = kontekst izvršavanja (od ranije)
- Sužena definicija: kontekst koji ima:
  - nezavisnu mapu radne memorije
  - nezavisne pokazivače na izvesne sistemske resurse (npr. deskriptori za pristup fajlovima i mrežnim konekcijama)
- Lanac = "Thread":
  - konkurentno izvršavanje u kontekstu jednog procesa
  - deljena slika memorije
  - deljeni deskriptori

Implementacija

- Klasični Unix:
  - proces se dobija sistemskim pozivom **fork**()
  - za "thread" se pokreće LWP (*lightweight process*)
- Linux:
  - primitivna operacija je **clone**(), koja može/ne mora da deli adresni prostor
  - **fork**() se izvodi iz **clone**()
  - korisnička konkurentnost je i dalje moguća

Osnovni problem paralelnog rada

- Da bi bili korisni, procesi moraju na neki način međusobno da komuniciraju
- Najjednostavnija komunikacija je preko zajednički vidljivog bloka memorije
- Problem je očuvanje sekvencijalne semantike izvršavanja pri mogućnosti paralelnih izmena memorije
- Sve manifestacije paralelnog rada se na kraju posmatraju u skladu s uticajem na ovaj problem

Trivijalna ilustracija

- Šta se sve može desiti prilikom paralelnog izvršavanja ovog koda?
  - potpitanje: a šta sa konkurentnim izvršavanjem

```
flag = False;
if not flag:
    flag = True
    # kritični deo
    print('uradi samo jednom')
```

Garancije pristupa

- Generalno, želimo da postignemo da se kritičnom delu može pristupiti samo iz jednog procesa
- Postoje različite strukture podataka i protokoli njihovog korišćenja koje ovo omogućavaju
- Hardver mora da pruža neku vrstu garancije, inače je čisto softversko rezonovanje nemoguće

Generička struktura: semafor

- U opštem slučaju, brojač koji upravlja pristupom nekom resursu
- Dokle god je vrednost brojača veća od nule, pristup je moguć
- Dve operacije:
  - umanjivanje (operacija P): ako vrednost za jedan. Ako je posle ovoga vrednost negativna, smesti proces u red i čekaj
  - uvećavanje (operacija V): uvećaj vrednost za jedan. Ako je vrednost bila negativna, uzmi prvi proces iz reda čekanja
- Semafor ima početnu vrednost veću od nule koja kaže koliko je instanci resursa na raspolaganju

Muteks

- Binarni semafor = muteks (mutex: *mutual exclusion*)
- Samo jedna instanca resursa je na raspolaganju
- Veoma često korišćena struktura, u implementaciji i praktičnom rezonovanju se tretira posebno u odnosu na generički semafor

Praktična implementacija

- **Atomičke** operacije: procesor(i) garantuje/u da tokom njihovog izvršavanja samo jedan proces na čitavom sistemu ima pristup nekom resursu, obično memorijskoj lokaciji
- Nekoliko mogućih operacija:
  - proveri i postavi (**test-and-set**): postavi vrednost lokacije na 1, vrati staru vrednost
  - preuzmi i uvećaj (**fetch-and-add**)
  - uporedi i zameni (**compare-and-swap**)

Proveri i postavi

- Muteks implementiran pomoću ove instrukcije (pseudokod)

```
while test_and_set(mutex) == 1:
    pass
# kritični deo
print('uradi samo jednom')
# oslobodi muteks
mutex = 0
```

Sinhronizacija memorijskog pristupa

- Procesori zbog povećanja performansi smeju da preuređuju redosled izvršavanja instrukcija
  - ako zaključe da naredna instrukcija u nizu ne utiče na rezultat cele sekvence
  - problem: ovo zaključivanje je lokalno
- U prethodnoj implementaciji, mora se postići da se **mutex** postavi na nulu tek kada su svi prethodni memorijski pristupi završeni
- Način da se ovo postigne je uvođenje *memorijskih barijera*
  - garancije da će promene i/ili pristupi memoriji biti lokalno ili globalno vidljivi u određenom redosledu
  - granularnost i efektivnost zavise od procesora
  - npr. x86 ima jače osnovne garancije u odnosu na ARM procesore

Dodatne komplikacije

- Višeprocesorski sistemi: sinhronizacija keševa
  - sve promene u jednom kešu moraju se propagirati ostalima
  - redosled promena vrednosti mora ostati očuvan
  - različiti protokoli za koherentnost keševa
- Takođe: sinhronizacija MMU mapiranja za virtuelnu memoriju
  - ako se mapiranje promeni na jednom sistemu, TLB zapisi se moraju očistiti svugde (*TLB shootdown*)

Opšti tretman paralelizacije

- **Amdalov zakon** daje teorijsku granicu ubrzanja izvršavanja nekog zadatka pri poboljšanju raspoloživih resursa
- Često korišćen pri proceni dobitaka od paralelizacije
- Ključno: nisu svi delovi nekog problema podložni paralelizaciji
- Parafrazirano: ako je dobitak u paralelizaciji manji od troška paralelizovanja, ne isplati se

Drugačiji pristupi konkurentnosti (1)

- CSP model (*Communicating Sequential Processes*)
- Formalizam za predstavljanje načina interakcije u konkuretnim sistemima
- Osnova je da procesi nemaju zajednički vidljivo promenljivo stanje
- Komunikacija se odvija pomoću kanala, procesi su anonimni
- Primer implementacije: programski jezik Go

Drugačiji pristupi konkurentnosti (2)

- Aktorski model
- Takođe predstavlja model za konkurentnu interakciju i programiranje
- Ima sličnosti sa CSP: komunikacija je takođe putem poruka, samo je lokalno stanje aktora promenljivo (nema globalnih promena)
- Razlike: aktori su imenovani, procesi ne; prenos poruka je potpuno asinhron; kanali za prenos su implicitni
- Primer implementacije: programski jezik Erlang

Drugačiji pristupi konkurentnosti (3)

- Transakciona memorija
  - grupe memorijskih pristupa se tretiraju slično transakcijama kod baza podataka
  - moguće su i hardverska i softverksa implementacija
- Funkcionalno programiranje
  - veoma širok pojam
  - ključan je oslonac na nepromenljive strukture i njihovu transformaciju
  - programski jezici kao što su Haskell, (donekle) Scala i Rust