# Testiranje konkurentnih programa

# Uvod

*Konkurentni program* sadrži dve ili više niti koje se izvršavaju uporedo i rade zajedno da izvrše neki zadatak.

Kada se program izvršava, operativni sistem stvara *proces* koji sadrži programski kod
i podatke i upravlja procesom dok se program ne završi.

*Nit* je jedinica kontrole u okviru procesa:

- kada nit radi, izvršava funkciju u programu - "glavna Nit" izvršava "glavni" program a druge Niti izvršavaju druge funkcije.
- Svaka nit ima svoj: stek, kopiju CPU registara (uključujući stek pointer, program counter)
- Niti u višenitnom procesu dele podatke, programski kod, resurse, adresni prostor, i stanje procesa

Operativni sistem dodeljuje procesor(e) između procesa/niti u sistemu:

- operativni sistemi bira proces i proces bira nit, ili
- niti se direktno raspoređuju od strane operativnog sistema.

U principu, svako spremna nit dobija deo vremena (nazvan *kvantum*) procesora.

- Ako nit čeka nešto, oslobađa procesor.
- Kada istekne kvant vremena dodeljen niti, izvršavanje niti se privremeno prekida da se dozvoli drugoj spremnoj niti da se izvršava.

Prebacivanje procesora od jednog procesa ili niti na drugi je poznat kao izmena konteksta.

više procesora => više niti može izvršiti u isto vreme.

jedan CPU => niti se smenjuju u izvršavanju

*Politika raspoređivanja* može da uzme u obzir prioritet niti. Ako je politika raspoređivanja *fer*, to znači da svaka spremna nit na kraju dobija da se izvrši.

# Komunikacija niti

Da bi niti da rade zajedno, one moraju da komuniciraju.

Jedan od načina za niti da komuniciraju jeste pristup zajedničkoj memoriji. Niti u istom programu mogu referencirati globalne varijable ili pozvati metode na zajedničkom objektu.

Konkurentni programi pokazuju ne-deterministički ponašanje - dva izvršavanja istog programa sa istim ulazom mogu da proizvedu različite rezultate.

## Nedeterminističko ponašanje pri izvršavanju

Primer 1. Pretpostavimo da je ceo broj x inicijalno jednak 0.

| Thread1 | Thread2 | Thread3 |
|---|---|---|
| (1) x = 1; | (2) x = 2; | (3) y = x; |

Konačna vrednost za y je nepredvidiva, ali se očekuje da će biti ili 0 ili 1 ili 2. Ispod su neki od mogućih prepletanja ova tri iskaza.

(3), (1), (2) konačna vrednost y je 0
(2), (1), (3) konačna vrednost y je 1
(1), (2), (3) konačna vrednost y je 2

Memorijski hardver garantuje da se operacije čitanja i pisanja celobrojne promenljive ne preklapaju.

U principu, nedeterminističko ponašanje pri izvršavanju izaziva jednu ili više sledećih stvari:

- nepredvidiva stopa napredovanja izvršavanja niti na jednom procesoru (zbog izmene konteksta između niti)
- nepredvidiva stopa napredovanja izvršavanja niti na različitim procesorima (zbog razlike u brzini procesora)
- upotreba nedeterminističkih programskih konstrukcija, koje čine nepredvidive izbore između dve ili više mogućih akcija.

*Nedeterministički rezultati ne ukazuju nužno na prisustvo greške*. Niti se često koriste za modelovanje objekata u realnom osvetu koji su po prirodi nedeterministički.

*Nedeterminizam obezbeđuje fleksibilnost u dizajnu*. Npr. konačni bafer kompenzuje razlike u brzinama rada proizvođača i potrošača. Redosled depozita i povlačenja novca sa računa je nedeterministički.

*Nedeterminizam i konkurentnost su srodni koncepti*. Konkurentni događaji A i B mogu biti modelovani kao nedeterministički izbor između dva sleda događaja: (A sledi B) ili (B sledi A). Mogući broj prepletanja eksplodira kada se broj konkurentnih događaja povećava.

*Nedeterminizam je inherentno svojstvo konkurentnih programa*.Teret suočavanja sa ne-determinizam pada na programatoru, koji mora da obezbedi da niti su pravilno sinhronizovani, bez nametanja nepotrebnih ograničenja koja samo smanjuju nivo konkurentnost.

Nedeterminističko izvršavanje stvara posebne probleme u toku testiranja i debagovanja.

# Atomske Akcije

Stanje programa sadrži vrednost za svaku promenljivu definisanu u programu i drugih implicitnih varijabli, kao što je programski brojač.

Atomska akcija transformiše stanje programa, na nedeljiv način: {x == 0} x = 1; {x == 1}.

Transformacija stanja obavljena tokom atomske akcije je nedeljiva ako druge niti mogu videti stanje programa, kao što se pojavljuje pre ili posle akcije, ali ne i neko srednje stanje dok se akcija dešava.

Do izmene konteksta može doći dok jedna nit obavlja atomsku akciju sve dok se dozvoljava drugim nitima da vide ili se mešaju sa akcijom, dok je ona u toku.

Pojedinačne mašinske instrukcije kao što su load, add, subtract i store tipično se izvršavaju atomično; ovo je garantovano memorijskim hardverom.

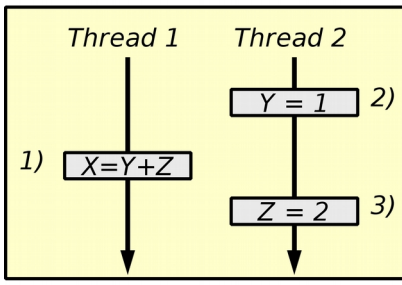
U Javi, dodela vrednosti od 32 bita ili manje garantovano da sprovodi atomično, tako da je iskaz, kao što je x = 1 za celobrojnu promenljivu x atomska akcija. U principu, međutim, izvršenje nekog iskaza dodele ne mora biti atomično.

## Ne-atomski aritmetički izrazi i iskazi dodeljivanja

Prepletanje od mašinskih instrukcija iz dva ili više izraza ili iskaza dodele može da proizvede neočekivane rezultate.

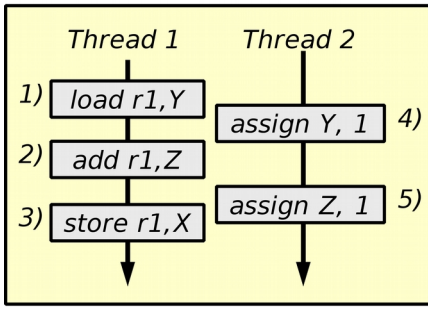
Primer 2. Pretpostavimo da su y i z inicijalno 0.

*Java*




| Execution orders | Final values of X |
|---|---|
| 1) → 2) → 3) | 0 |
| 2) → 1) → 3) | 1 |
| 2) → 3) → 1) | 3 |

*Non-Java*



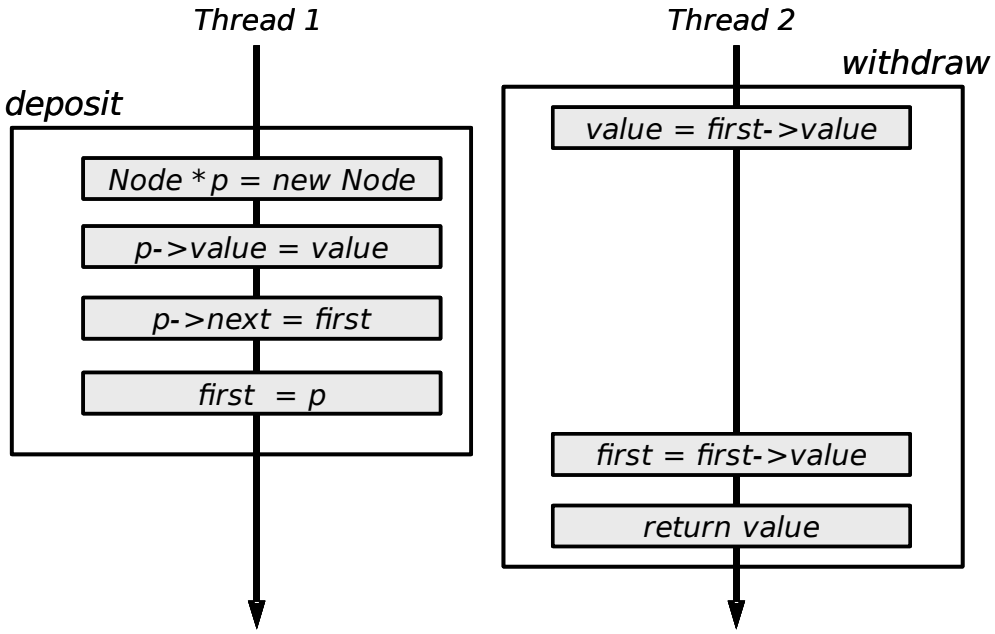

| Execution orders | Final values of X |
|---|---|
| 1) → 2) → 3) → 4) → 5) | 0 |
| 4) → 1) → 2) → 3) → 5) | 1 |
| 1) → 4) → 5) → 2) → 3) | 2 |
| 4) → 5) → 1) → 2) → 3) | 3 |

## Ne-atomske grupe izveštaja.

Druga vrsta neželjenog nedeterminizma u konkurentanom programu je izazvana prepletanjem grupe iskaza, iako svaki iskaz može biti atomski.

Primer 3. Promenljiva first ukazuje na prvi čvor u listi. Pretpostavimo lista je neprazna.

```
class Node {
public:
  valueType value;
  Node* next;
}
Node* first;              //first points to the first Node in the list;
```




Na kraju ovog niza iskaza:

- first i dalje ukazuje na uklonjenu stavku
- deponovana stavka je izgubljena.

Da bi se rešio ovaj problem, metode deposit i withdraw moraju se implementirati kao atomske akcije.

## Sinhronizacija niti

Primeri u ovom poglavlju ilustrovali su bagove koji mogu da nastanu prilikom pristupa deljenim promenljivima koji nisu pravilno sinhronizovani.

Jedna vrsta sinhronizacije se zove uzajamno isključivanje.

**Uzajamno isključivanje** omogućava da grupa atomskih akcija, nazvana kritična sekcija, ne može da se izvrši od strane više od jedne niti u jednom trenutku. To jest, kritična sekcija mora biti izvršena kao atomska akcije.

Neuspeh da se pravilno primeni kritična sekcija je greška pod nazivom **trka podataka** (**data race**).

Drugi tip sinhronizacije se naziva uslovna sinhronizacija.

**Uslovna sinhronizacija** obezbeđuje da stanje programa zadovoljava određeni uslov pre nego što počne neka radnja.

Za povezane liste u Primeru 3, postoji potreba i za uslovnu sinhronizaciju i uzajamnim isključivanjem.

- Lista ne sme da bude u praznom stanju kada se metod withdraw dozvoli da ukloni stavku,
- međusobno isključenje je potrebno da se osigura da se deponovane stvari ne gube i ne bivaju uklonjene više puta.

# Testiranje i otklanjanje grešaka u višenitnim programima

Konvencionalni pristup testiranju i debagovanju sekvencijalnog programa:

(1) Izabere se skup test primera

(2) Pokrene se program po jednom za svaki ulaz i uporede rezultati testa sa očekivanim rezultatima.

(3) Ako test primer nađe grešku, pokrene se program ponovo sa istim ulazom u cilju prikupljanja informacija za otklanjanje grešaka i pronađe se greška koja je izazvala neuspeh testa.

(4) Nakon što je greška ispravljena, izvršiti program ponovo sa svakim od test primera radi provere da li je greška ispravljena i da, usput, nisu uvedeni novi defekti (tzv. "regresiono testiranje") .

Ovaj proces nije u originalnom obliku primenljiv na konkurentne programe.

## Problemi

Neka je CP neki konkurentni program. Višestruko izvršavanje CP sa istim ulazima može da proizvede različite rezultate. Ovo nedeterminističko ponašanje stvara sledeće probleme tokom testiranja i debagovanja CPa:

**Problem 1**. Kada testiramo CP za ulaz X, jedno izvršavanje nije dovoljno da se utvrdi ispravnost CP za X. Čak i ako CP sa ulazom X izvršimo uspešno mnogo puta, moguće je da će buduća izvršenja CZ za X proizvesti netačne rezultate.

**Problem 2**. Kada debagujemo neuspešno izvršavanje CP za ulaz X, ne postoji garancija da će ovo izvršavanje biti ponovljeno izvršavanjem CP za X.

**Problem 3.** Nakon što je CP modifikovan radi ispravke greške otkrivena tokom neuspelog izvršenja CP za ulaz X, jedno ili više uspešnih izvršavanja CP za X tokom regresionog testiranja ne podrazumeva da je otkrivena greška ispravljena ili da nema novo uvedenih grešaka.

Postoje mnoga stvari koja se moraju realizovati u cilju rešavanja ovih problema:

**Ponavljanje programa** (replay): Programeri oslanjaju na debagovanje tehnike koje pretpostavljaju program kvarovi mogu se reprodukovati. Ponavljanje izvršenja konkurentnog programa se zove "program replay".

**Praćenje programa** (Tracing): Pre izvršenja može se ponoviti ono mora pratiti. Ali šta tačno znači to ponove se izvršenje? Trag izvršavanja u opštem slučaju sadrži informacije o redosledu akcija koje izvršava svaka nit. Dodatno kod distribuiranih sistema postoji problem u sagledavanju redosleda akcija na različitim računarima (ako nema globalnog časovnika).

**Izvodljivost sekvence**: za niz akcija koji je dozvoljen programom (moguć) se kaže da je izvodljiva sekvenca. Testiranje podrazumeva utvrđivanje da li je ili nije data sekvenca izvodljiva ili nije. Kako se sekvence biraju?

- Dozvoliti izvršavanje sekvenci bez determinizma (**nedeterminističko testiranje**) u određenom vremenu. Najlakše, ali neefikasno (ne garantuje rešavanje problema).

- Forsirati odabrane sekvence za izvršavanje (**determinističko testiranje**). Izbor sekvenci koje su efikasne u otkrivanju grešaka je teško učiniti. Test pokrivenost kriterijum može da se koristi za usmeravanje izbor testova i da odredi kada da prestane testiranje.

- **Testiranje zasnovano na prefiksu**: početke sekvenci izabrati deterministički. Potom sledi nedeterminističko izvršavanje koje forsira definisani početak i dalje nastavlja nedeterministički.

# Praćenje, testiranje i ponovno izvršavanje za semafore i brave

Sadržaj:

- Tehnika za otkrivanje narušavanja uzajamnog isključivanja.
- Praćenje i ponovna reprodukcija izvršavanja tokom debagovanja.
- Detekcija zastoja
- Testiranje dostižnosti

## Nedeterminističko testiranje Lockset algoritmom

Utrkivanje podataka(data race) predstavlja neuspeh da se pravilno implementiraju kritične sekcije za pristup neatomskih deljenim promenljivim.

Konkurentni program može biti testiran za trke podataka:

- pratiti pristupe zajedničkim promenljivama i uveriti se da je svaka promenljiva pravilno zaključana pre nego što joj se pristupi.

- Izvršavati program više puta sa istim test primerom u cilju povećanja naše šanse za pronalaženje trke podataka.

Ova vrsta testiranja se zove nedeterminističko testiranje.

Nedeterminističko testiranje konkurentnog programa CP podrazumeva sledeće korake:

1. Izabrati skup ulaza za CP.

2. Za svaki izabrani ulaz X, izvršavati CP mnogo puta i ispitati rezultat svakog izvršenja.

Svrha nedeterminističkog testiranja jeste da se ostvari onoliko varijanti ponašanja koliko je to moguće.

- Na žalost, eksperimenti su pokazali da su programi imaju tendenciju da ispoljavaju isto ponašanje od izvršenja do izvršenja.

- Takođe, "efekat sonde"(uticaj test okruženja i test koda) može onemogućiti neke otkaze da se ispolje.

Da bi se povećala verovatnoća ostvarivanja različitih ponašanja:

- promeniti algoritam raspoređivanja koji koristi operativni sistem, npr. promeniti vrednost vremenskog kvanta koji se koristi za round robin raspoređivanje.

- ubaciti sleep(t) iskaze u program sa intervalom spavanja t odabranim nasumično.

  Izvršavanje iskaza sleep forsira izmenu konteksta i time posredno utiče na raspoređivanje niti.

  Najlakše je koristiti posebnu biblioteku sa sinhronizacionim primitivama(npr. binarySemaphore, countingSemaphore, mutexLock) koje su prilagođene testiranju. Kada se želi ova opcija, sleep iskazi izvršavaju se na početku metoda P(), V(), lock() i unlock().

  Za otkrivanje trke podataka, kombinujemo nedeterminističko testiranje sa lockset algoritmom:

- proverava da li sve promenljive slede dosledno disciplinu blokiranja u kojem je svaki deljena promenljiva zaštićena bravom.

- za svaku promenljivu, utvrđuje da li postoji neka brava koja jee uvek zaključana kad god se promenljivoj pristupa.

  Za deljenu promenljivu v, neka skup CandidateLocks(v) sadrži one brave koje su štitile v tokom dosadašnjeg izvršavanja programa. Dakle, brava l je u CandidateLocks(v) ako je, u toku prethodnog izvršenjavanja, svaka nit koja je pristupala v držala l u trenutku pristupa.

  CandidateLocks(v) se izračunava na sledeći način:

- Kada se nova promenljiva v inicijalizuje, njen za kandidat skup smatra se da drži sve moguće brave.

- Kada se v pristupa za čitanje ili pisanje od strane niti T, CandidateLocks(v) se ažurira.

  Nova vrednost CandidateLocks(v) je presek CandidateLocks(v) i skupa brava koje drži nit T.

Na osnovu ovog iterativnog algoritma:

- ako neka brava l dosledno štiti v, ona će ostati u CandidateLocks(v) i kada se CandidateLocks(v) ažurira.
- ako CandidateLocks(v) postane prazan, to znači da ne postoji brava koja štiti v dosledno.

Lockset Algoritam:

```
/ / Neka LocksHeld(T) označava skup brava koje trenutno drži nit T
Za svaku deljenu promenljivu v, inicijalizovati CandidateLocks(v) na skup svih brava.
Na svaki pristup v za čitanje ili upis od strane niti T:
  CandidateLocks(v) = CandidateLocks(v) ∩ LocksHeld(T);
if(CandidateLocks(v) == {})
  izdati upozorenje;
```

Na slici ispod, pristup Thread1 deljenoj promenljivoj s je prvi put zaštićen je prvo sa mutex1 zatim sa mutex2. Ovo kršenje uzajamnog isključivanja može da se detektuje lockset algoritmom:

| Thread1 | LocksHeld(Thread1) | CandidateLocks(s) |
|---|---|---|
| | {} | {mutex1,mutex2} |
| mutex1.lock(); | {mutex1} | {mutex1,mutex2} |
| s = s+1; | {mutex1} | {mutex1} |
| mutex1.unlock(); | {} | {mutex1} |
| | | |
| mutex2.lock(); | {mutex2} | {mutex1} |
| s = s+1; | {mutex2} | {} |
| mutex2.unlock(); | {} | {} |

- CandidateLocks(a) se inicijalizuje na {mutex1, mutex2} i ažurira se kada se pristupa s.
- Kada Thread1 zaključava mutex1, LocksHeld(Thread1) postaje {mutex1}.
- Kada se s pristupa u prvom iskazu dodele, CandidateLocks(a) postaje mutex1, tj. presek skupova CandidateLocks(a) i LocksHeld(Thread1).
- Kada se izvrši drugi iskaz, dodele Thread1 drži bravu mutex2, a jedini kandidat za zaključavanje s je mutex1.
- Posle preseka CandidateLocks(a) i LocksHeld(Thread1) CandidateLocks(a) postaje prazan.

Lockset algoritam je otkrio da nema brave koja dosledno štiti deljenu promenljivu s.

Kao tehnika nedeterminističkog testiranja, lockset algoritam ne može da dokaže da je program slobodan od trke podataka.

## SYN-sekvence za semafore i brave

Izvršavanje uporednog programa možemo se okarakterisati kao niz događaja nad sinhronizacionim objektima.

Redosled sinhronizacije dogadjaja se zove SYN-sekvenca.

Postoji nekoliko načina da se definiše SYN-sekvenca i definicija SYN-sekvence utiče na dizajniranje rešenja za ponovno izvršavanje programa.

Neka CP bude konkurentni program koji koristi deljene promenljive, semafore i brave.

Rezultat izvršavanja CP sa datim ulazom zavisi od(nepredvidljivog) poretka u kojem se pristupa zajedničkim promenljivama, semaforima i bravama u CP.

- Semaforima se pristupa upotrebom operacija P i V
- Bravama se pristupa upotrebom operacija *lock* i *unlock*
- Deljenim promenljivama se pristupa upotrebom operacija *read* i *write*.

Sinhronizacioni objekti u CP su njenove deljene promenljive, semafori i brave. Sinhronizacini događaji u CP su izvršavanje operacija read/write, P/V i lock/unlock nad ovim objektima.

SYN-sekvenca za binarni semafor ili brojački semafor s je sled događaja sledećih tipova:

- završetak operacije P
- završetak operacije V
- početak operacije P koja se nikada nije završila zbog zastoja ili izuzetka
- početak operacije V koja se nikada nije završila zbog zastoja ili izuzetka

Ovakvu sekvencu događaja nazivamo PV-sekvenca od s. Događaj u PV-sekvenci se označava identifikatorom(ID) niti koja izvršava P ili V operaciju.

Redosled kojim niti završe P i V operacije nije obavezno isti kao i redosled u kome oni pozivaju P i V ili čak isti kao i redosled u kome P i V operacije počinju.

Za operacije koje su završile, upravo njihov redosled završetka mora biti ponovo reprodukovan, jer ovaj redosled određuje rezultat izvršenja programa.

Takođe treba ponavljati početke operacija koji ne završavaju, tako da će se isti događaji, izuzeci i zastoji javljajati tokom ponovnog izvršenja programa.

SYN-sekvenca za bravu l je sled događaja sledećih tipova:

- završetak operacije *lock*
- završetak operacije *unlock*
- početak operacije *lock* koja nikada nije završen zbog zastoja ili izuzetka
- početak operacije *unlock* koja se nikada nije završila zbog zastoja ili izuzetka

Ovakvu sekvencu nazivamo LockUnlock-sekvenca za l. Događaj u LockUnlock-sekvenci se označava identifikatorom(ID) niti koja izvršava *lock* ili *unlock*.

Primer: Razmotrimo jednostavan program. Konačna vrednost deljene promenljive x je ili 1 ili 2.

```
 binarySemaphore mutex(1);
Thread1          Thread2
mutex.P();       mutex.P();
x = 1;           x = 2;
mutex.V();       mutex.P(); / / greška: treba da bude mutex.V();
```

(Ilustracija ReadWrite-sekvence i PV-sekvence)

Moguća ReadWrite-sekvenca deljene promenljive x je:

(1, 0, 0),(2, 1, 0). / / Iz Odeljka 2, format je(ID niti, broj verzije, ukupno čitalaca)

Ovo označava da x-u prva pristupila nit Thread1 a potom Thread2.

Pošto Thread1 pristupiti x-u prva, PV-sekvenca za mutex mora biti:

1, 1, 2, 2

ukazuje da Thread1 obavlja svoj P i V operacije pre Thread2. Druga P operacija u Thread2 je greška. Ova P operacija će početi ali neće završiti i trebalo je da bude V operacija.

SYN-sekvence za konkurentni program CP je kolekcija ReadWrite-sekvenci, PV-sekvenci, i LockUnlock-sekvenci. Postoji po jedna sekvenca za svaku zajedničku promenljivu, semafor i bravu u programu.

SYN-sekvenca za program u listingu sadrži ReadWrite-sekvencu za x i PV-sekvence za mutex:

((ReadWrite-sekvenca za x:(1,0,0),(2,1,0), PV-sekvenca za mutex:(1, 1, 2, 2)).

Ovo je ***parcijalni redosled*** događaja, odnosno dešavanja na jednom objektu su(potpuno) uređena, ali redosled događaja između različitih objekata nije naveden.

SYN-sekvenca takođe može biti jedinstven ***totalno uređeni*** sled događaja svih sinhronizacionih objekata. Totalno uređena sekvenca događaja koja je u skladu sa gore delimično uređenom sekvencom je:

1,(1, 0, 0), 1, 2,(2, 1, 0), 2.

U principu, mogu postojati dve ili više potpuno uređene sekvence koje su u skladu sa datim parcijalnim uređenjem pošto se dva istovremena događaja mogu pojaviti u ukupnom totalnom poretku u bilo kom redosledu.

Definicija SYN-sekvence ima za cilj da uhvati značenje ponavljanja jednog izvršavanja programa drugi put. Pretpostavimo da, kada se program iznad izvršava:

- Thread2 izvršava mutex.P() i blokira jer je Thread1 je u kritičnoj sekciji.

- Tokom ponavljanja ovog izvršenja, pretpostavimo da Thread2 izvršava svoju prvu mutex.P() operaciju bez blokiranja, jer je Thread1 već izvršio svoje mutex.P() i mutex.V() operacije.

Ova dva izvršavanja nisu identična, ali su oni dovoljno blizu?

U oba izvršavanja:

- sekvenca *završenih* P() i V() operacija je ista

- konačna vrednost x je 2.

Dakle, smatramo da drugo izvršavanje programa ponavlja prvo.

## Praćenje i reprodukovanje jednostavne PV-sekvence i LockUnlock-sekvence

Menjanje Metoda P() i V().

Praćenje: identifikator niti koja završava poziv metoda s.P() ili s.V() se beleži i čuva u trace datoteci za s.

Reprodukcija: pretpostavimo da svaki semafor ima dozvolu, nazvanu PV-dozvola.

- Nit mora da poseduje semaforsku PV-dozvolu pre nego što izvede P() ili V() operaciju na tom semaforu.
- Redosled kojim nit prima semaforsku PV-dozvolu se zasniva na PV-sekvenci koja se reprodukuje.
- Nit zahteva i oslobađa semaforsku PV-dozvolu pozivom metoda

requestPermit() i releasePermit().

```
void P() {
    if(replayMode)
            control.requestPermit(ID);
    / / Kod za zaključavanje ovog semafora pojavljuje se ovde
    if(replayMode)
      control.releasePermit();
    / * Ostatak tela P() * /
    if(traceMode)
      control.traceCompleteP(ID);
    / / Kod za otključavanje ovog semafora pojavljuje se ovde
}

public void final V() {
  if(replayMode)
    control.requestPermit(ID);
  / / Kod za zaključavanje ovog semafora pojavljuje se ovde
  if(replayMode)
    control.releasePermit();
  / * Ostatak tela za V() * /
  if(traceMode)
    control.traceCompleteV(ID);
  / / Kod za otključavanje ovog semafora pojavljuje se ovde
}
```

## Menjanje Metode lock() i unlock().

Implementacije metoda lock() i unlock() u klasi mutexLock su izmenjene baš kao metode P() i V().

- Klasa mutexLock sadrži pozive requestPermit() i releasePermit() pre i posle operacije blokade u mutexLock.

- Pozivi traceCompleteLock() i traceCompleteUnlock() pojavljuju se na kraju njihovih kritičnih sekcija.

Zastoji (deadlocks) i izuzeci:

- Kada se ponavljaju operacije koje proizvode deadlock, pozivajuća nit neće biti blokirana unutar tela operacije, već će biti blokirana na pozivu requestPermit() pre operacije.

- Događaji koji uključuju izuzetke koji se javljaju u toku izvršenja P, V, lock, ili unlock se ne ponavljaju. Ali trag ukazuje da bi izvršenje ovih operacija bacilo izuzetak, što je verovatno dovoljno da pomogne debagovanje programa.

Klasa kontrole praćenja/reprodukcije.

Svaki semafor i brava povezani su sa kontrolnim objektom.

- Replay režim: kontrolni objekat ubacuje jednostavne SYN-sekvence na semaforu ili bravi i obrađuje pozive requestPermit() i releasePermit().

- Trace režim: kontrolni objekat prikuplja sinhronizacione događaje koji se dešavaju i snima ih u trace fajlu.

Kada nit pozove requestPermit() ili jedan od trace metoda, prosleđuje svoj identifikator (ID).

C + + klase *control* prikazana je u listingu na sledećoj strani.

Kada se kreira kontrolni objekat za semafor s, on čita jednostavnu PV-sekvencu s u vektor *SYNsequence*.

Metod *requestPPermit*(): Niti koje pokušavaju da izvrše operaciju van redosleda stavljaju se na čekanje u metodu *requestPPermit*() na posebnom semaforu u nizu *Threads*.

Nit koristi svoj ID da odredi na kom semaforu u nizu Threads da čeka.

Metod *releasePPermit*(): povećava *index* da bi sledeća operacija u *SYNsequence* mogla da se dogodi. Ako je nit koja treba da izvrši sledeću operaciju blokirana u *requestPPermit*(), biće probuđena.

```
class control {
public:
  control() {
    / * Unos celobrojnih ID-eva u SYNsequence;  inicijalizacija nizova threads i
    hasRequested * /
  }
  void requestPermit(int ID) {
    mutex.lock();
    if(ID != SYNsequence [index]) {// nit ID treba da izvrši sledeći događaj?
      hasRequested [ID] = true;    // Ne; postavi fleg da se zabeleži zahtev
      mutex.unlock();
      threads[ID].P();.            // čekaj dozvolu
      hasRequested [ID] = false;   // Izbriši fleg i izađi iz requestPermit
    }
    else mutex.unlock();           // Da, izlaz iz requestPermit.
  }
  void releasePermit() {
    mutex.lock();
    ++index;
    if(index <SYNsequence.size()) {// Da li postoji još dešavanja za replay?
      // Da li je sledeća nit već tražila dozvolu?
      if(hasRequested [SYNsequence [index]])
        threads[SYNsequence[index]].V(); // Da, probudi je.
    }
    mutex.unlock();
  }
  void traceCompleteP(int ID) {...} // snimanje celobrojnog IDa
  void traceCompleteV(int ID) {...} // snimanje celobrojnog IDa
private:
  // PV-sekvenca ili LockUnlock-sekvenca, niz celobrojnih IDeva
  vector SYNsequence;
  binarySemaphore* threads; //  Svi semafori su inicijalizovani na 0
  // hasRequested[i] je true ako je nit i zaustavljena u requestPermit(); init na false
  bool* hasRequested;
  int index = 0; // SYNsequence[index] je ID naredne niti koja treba da izvrši događaj
  mutexLock mutex; // napomena: ova brava se ne prati niti reprodukuje
}
```

(C++ klasa control za reprodukovanje PV-sekvenci i LockUnlock-sekvenci)

Da bismo ilustrovali rad kontrolora, razmotrimo ispravljenu verziju jednostavnog

programa iz prvog primera.

| Thread1 | Thread2 |
|---|---|
| mutex.P(); | mutex.P(); |
| x = 1; | x = 2; |
| mutex.V(); | mutex.V(); |

Pretpostavimo da je PV-sekvenca zabeležena tokom izvršenja ovog programa: 1, 1, 2, 2.

Pretpostavimo da Thread2 pokuša da izvrši mutex.P() prvo pa poziva requestPermit(2) pre nego što Thread1 pozove requestPermit(1):

- Pošto je vrednost index-a 0 i vrednost SYNsequence[index] je 1 a ne 2, Thread2 blokira sebe u requestPermit() izvršavanjem Threads[2].P()

- Kada nit Thread1 najzad pozove requestPermit(1), biće joj dozvoljeno da izađe iz requestPermit() i izvrši svoju mutex.P() operaciju.

- Thread1 će zatim pozvati releasePermit(). Metod releasePermit() povećava index na 1 i proverava da li nit koja treba da izvrši sledeću P/V operaciju je već pozvala requestPermit().

- Sledeća nit je SYNsequence[1], koji je 1. Thread1 nije zvao requestPermit() za sledeću operaciju, tako da se ništa dalje dešava u releasePermit().

- Konačno Thread1 poziva requestPermit(1) da zahteva dozvolu da izvrši svoju mutex.V() operaciju. Thread1 dobija dozvolu, izvršava mutex.V() i poziva releasePermit().

- Metod releasePermit() povećava index do 2 i zaključuje da je Thread2 nit koja izvršava sledeću P/V operaciju.

- Thread2, pošto je već pozvala requestPermit(), je i dalje blokirana u Threads[2].P(). Na to ukazuje vrednost hasRequested[2], koja je true. Zbog toga releasePermit() poziva Threads[2].V()

- Ovo omogućava Thread2 da izađe iz requestPermit() i obavi svoju mutex.P() operaciju.

- Thread2 će na kraju tražiti i dobiti dozvolu za svoju mutex.V() operaciju, čime se završava reprodukcija.

## Detekcija zastoja

Mrtvi zastoj (deadlock) definišemo kao situaciju u kojoj je jedna ili više niti postala zauvek blokirana.

Neka CP bude konkurentni program koji sadrži niti koje koriste semafore i brave za sinhronizaciju. Pretpostavimo da postoji izvršenje CP koje odgovara SYN-sekvenci S, a na kraju S, postoji nit T koja zadovoljava sledeće uslove:

- T je blokirana zbog izvršenja P(), V(), ili lock() operacije.
- T će ostati zauvek blokirana, bez obzira na to šta drugi će uraditi ostale niti.

Za nit T kaže se da je u mrtvom zastoju na kraju S, i za CP se kaže da ima mrtav zastoj. Zastoj u KP je globalni zastoj ako je svaka nit u CP ili blokirano ili završena, u suprotnom, to je lokalni zastoj.

U operativnim sistemima, procesi traže resurse (npr., štampače i datoteke) i ulaze u stanja čekanja ako te resurse drže drugi procesi. Ako zahtevani resurs nikada ne može postati dostupan, onda procesi ne mogu izaći iz stanja čekanja i nastaje mrtav zastoj.

Informacija o tome koji proces čeka na resurs koji drži drugi proces može se predstaviti na grafu čekanja na resurse (wait-for graph):

- grana iz čvora Pi ka Pj pokazuje da proces Pi čeka da proces Pj oslobodi resurs koji Pi potražuje.
- zastoj postoji u sistemu, ako i samo ako graf čekanja sadrži ciklus.
- Operativni sistem periodično poziva algoritam koji traži cikluse u grafu čekanja.

Detekcija zastoja korišćenjem wait-for grafa nije uvek primenljiva za konkurentne programe. Nit blokirana u P() operaciji, na primer, ne zna koje od ostalih niti mogu da je deblokiraju, pa je time relacija čekanja među nitima nepoznata.

Drugi pristup: Pretpostavimo da do zastoja dolazi ako su sve niti u programu trajno blokirane. Pretpostavimo, takođe, da se od svih niti očekuje da jednom završe rad.

Za otkrivanje zastoja, uvodi se brojač niti koji nisu završile svoj run() metod, i brojač trenutno blokiranih niti, i upoređuju se vrednosti ta dva brojača:

- numThreads brojač se uvećava kada nit počinje svoj run() metod i dekrementira kada nit završi svoj run() metod.

- blockedThreads brojač se uvećava kada se nit blokira u P(), V(), ili lock() opearciji i umanjuje kada se nit odblokira u toku V() ili unlock() operacije. blockedThreads brojač takođe treba da se ažurira u drugim blokirajućim metodima kao što join().

- Ako su numThreads i blockedThreads brojači ikada jednaki(i ne-nula), onda su sve niti su blokirane, a mi pretpostavljamo da je došlo do zastoja.

Ovaj pristup može da se koristi za otkrivanje globalnih zastoja, ali ne lokalne zastoje, jer zahteva da su sve ne-blokirane niti završene.

Kada se otkrije zastoj, događaji koji dovode do zastoja mogu se pratiti i ponoviti koristeći metode opisane ranije.

Neke relacije čekanja među nitima mogu biti opisane wait-for grafom:

- nit blokirana u lock() operaciji, zna da nit koja poseduje bravu može da je odblokira. Tako je odnos čekanja među nitima i bravama poznat.
- kada je jedna nit pokuša da zaključa dve brave u jednom redosledu, a druga nit pokušava da ih zaključa u obrnutom redosledu, do zastoja može doći.
- takav zastoj će biti označen ciklusom u wait-for grafu.

Detekcija zastoja je ugrađena u Java VM:

- Ova alatka se poziva preko prečice Ctrl + (za Linux ili Solaris operativni sistem) ili Ctrl-Pause/Break(za Microsoft Windows) na komandnoj liniji, dok aplikacija radi.
- Ako je aplikacija blokirana jer su dve ili više niti uključene u ciklus da dobiju bravu, prikazuje se spisak niti koje su uključene u zastoj.

Primer

```
import java.util.*;
import java.util.concurrent.locks.*;

public class Deadlock {

  public static void main(String[] args) {
                    final Lock lock1 = new ReentrantLock();
    final Lock lock2 = new ReentrantLock();

    Thread t1 = new Thread() {
      public void run() {
        try{
          lock1.lock();
          sleep(1000);
          lock2.lock();
        } catch(InterruptedException e) {}
      }
    };

    Thread t2 = new Thread() {
      public void run() {
        try{
          lock2.lock();
          sleep(1000);
          lock1.lock();
        } catch(InterruptedException e) {}
      }
    };
```

```
    t1.start();
    t2.start();
  }
}
Found one Java-level deadlock:
=============================
"Thread-1":
  waiting for ownable synchronizer 0x23cd2928, (a java.util.concurrent.locks.ReentrantLock$NonfairSync),
  which is held by "Thread-0"
"Thread-0":
  waiting for ownable synchronizer 0x23cd2950, (a java.util.concurrent.locks.ReentrantLock$NonfairSync),
  which is held by "Thread-1"
```

Ova alatka neće naći zastoj koji uključuje jednu ili više niti koje su trajno blokirane u monitoru, čekajući notify koji nikada ne dolazi, što je slično situaciji u kojoj je nit blokirana u P() operaciji čekajući V() operaciju koja nikada neće doći.

## Testiranje dostižnosti (reachability testing) za semafore i brave

Nedeterminističko testiranje je lako za sprovođenje, ali može biti vrlo neefikasno. Moguće je da će se neka ponašanja izvršavati mnogo puta, dok druga neće uopšte.

Za jedno rešenje sa zastojem problema filozofa koji jedu (dining philosophers) sa pet filozofa:

- zastoj nije otkriven tokom 100 normalnih izvršavanja.
- posle ubacivanja nasumičnih kašnjenja, deadlock je detektovan u osamnaest od sto izvršavanja.
- za deset filozofa, deadlock je detektovan u samo četiri od stotinu izvršavanja.

Kada se broj niti povećava, a time i ukupan broj mogućih ponašanja programa, očigledno postaje sve teže da se otkriju zastoji primenom nedeterminističkog testiranja.

Osim toga, instrumentacija programa da se vrši praćenje i otkrivanje zastoja može stvoriti efekat sonde koji sprečava da se uoče neki propusti.

Testiranje dostižnosti nam omogućava da ispita sva ponašanja jednog programa, ili barem onoliko različitih ponašanja, koliko je praktično, na "sistematski" način.

Pod sistematski, misli se da se svaka SYN-sekvenca izvršava samo jednom i moguće je znati kada su sve SYN-sekvence izvršene.

Testiranje dostižnosti kombinuje nedeterminističko testiranje i reprizno izvršavanje programa.

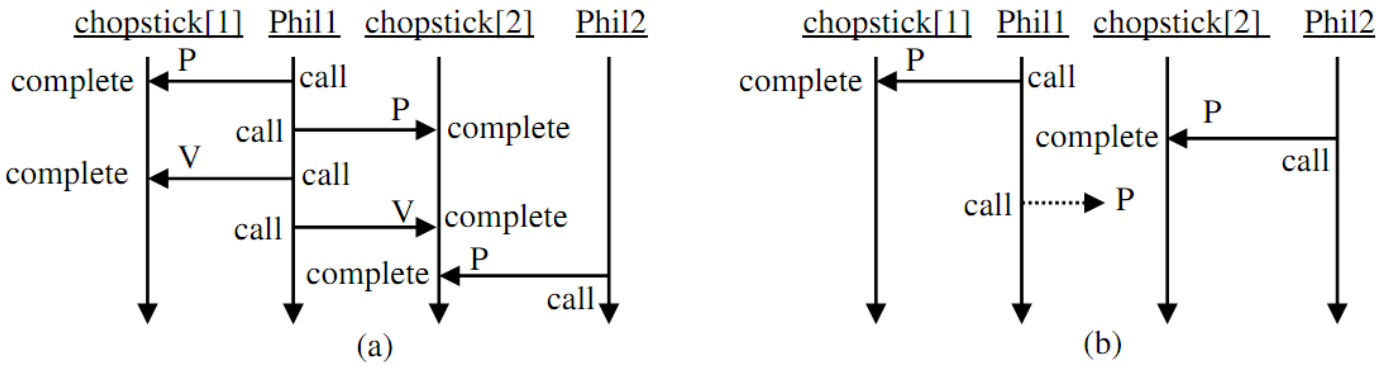
Tokom testiranja dostižnosti, SYN-sekvenca generisana nedeterminističkim izvršavanjem programa prati se kao i obično.

- Trag izvršavanja snima ponašanje koje se zapravo dogodilo.
- Za trag možemo definisati alternativne ponašanja koji su mogle da se dese, ali nisu.

Ova alternativna ponašanja nazivaju se "varijante utrke" (race variants) u tragu. Reprodukovanje varijanti utrke u toku narednih test izvršavanja programa osigurava da se posmatra drugačije ponašanje od ponašavnja osnovnog traga.

Slici je prikazan deo traga izvršavanja programa problem filozofa.

Filozof 1 i filozof 2 trkaju se da pokupe deljeni štapić 2 (ili viljušku 2), pri čemu Filozof 1 završava svoju P() operaciju na štapiću 2 pre nego što filozof 2 može da završi svoju P() operaciju.




U principu, postoji trka između poziva na P() ili V() operacije na istom semaforu ako ovi pozivi mogu biti završeni u drugačijem redosledu tokom drugog izvršenja (sa istim ulazom programa).

Slika b prikazuje varijantu trke izvršenja na slici a.

- U ovoj varijanti trke, Filozof 2 osvaja trku ka štapiću 2 i pokupi ga pre nego što Filozof 1 može da ga zgrabi.
- Isprekidana strelica pokazuje da je filozof 1 pozvao P() operaciju za štapić 2, ali je nije završio u varijanti trke. Ovaj poziv će biti završen u izvršavanju koje reprodukuje (i produžava do završetka ili deadlocka) ovu varijantu.

U principu, varijanta trke predstavlja početni deo jedne SYN-sekvence.

Testiranje dostižnosti koristi reprodukciju do tačke da se ostvare događaji u izabranoj varijanti trke, a zatim omogućava da program nastavi izvršavanje nedeterministički, tako da kompletna sekvenca može da se prati.

Može postojati mnogo kompletnih SYN-sekvenci koje imaju datu varijantu trke na početku.

- Jedna od tih sekvenci će biti uhvaćena u nedeterminističkom delu izvršavanja.

- Kompletirana sekvenca može se analizirati da se izvede još varijanti trke, koje se mogu koristiti za generisanje još tragova, i tako dalje.

Kada se replay primenjuje na varijantu trke na slici b:

- Filozof 1 i Filozof 2 će pokupiti svoje levo štapiće, što je deo scenarija zastoja u kojoj svi filozofi drže svoj levi štapić i čekaju svoj desni.
- Iako kompletni scenario zastoja ne mora odmah da se ostvari, testiranje dostižnosti obezbeđuje da se svaka moguća PV-sekvence programa za filozofe izvrši i tako otkrije zastoj.