Operativni sistemi: interna struktura i principi dizajna

Poglavlje 5
Konkurentnost: međusobno isključivanje i sinhronizacija

Sedmo izdanje

William Stallings

Više procesa

- Projektovanje operativnog sistema bavi se upravljanjem procesima i nitima:
  - Multiprogramiranje
  - Multiprocesiranje
  - Distribuirana obrada



Konkurentnost i deljeni podaci

- Konkurentni procesi mogu deliti podatke radi komunikacije, razmene informacija,...
- Niti istog procesa mogu deliti globalni adresni prostor
- Konkurentno deljenje može izazvati probleme
- Na primer: izgubljena ažuriranja

Konkurentnost

Ključni pojmovi

atomična operacija	Funkcija ili akcija implementirana kao niz od jedne ili više instrukcija koji deluje nedeljivo, tj. nijedan drugi proces ne može da vidi međustanje ni da prekine operaciju. Garantuje se da će se niz instrukcija izvršiti kao celina ili se uopšte neće izvršiti, bez vidljivog uticaja na stanje sistema. Atomičnost garantuje izolaciju od konkurentnih procesa.
kritična sekcija	Deo koda unutar procesa koji zahteva pristup deljenim resursima i koji se ne sme izvršavati dok je neki drugi proces u odgovarajućem delu koda.
mrtva petlja	Situacija u kojoj dva ili više procesa ne mogu da nastave jer svaki od njih čeka da neki od ostalih nešto uradi.
živa petlja	Situacija u kojoj dva ili više procesa neprestano menjaju svoja stanja kao odgovor na promene u drugim procesima, a da pritom ne obavljaju nikakav koristan posao.
međusobno isključivanje	Zahtev da, dok je jedan proces u kritičnoj sekciji koja pristupa deljenim resursima, nijedan drugi proces ne može biti u kritičnoj sekciji koja pristupa bilo kom od tih deljenih resursa.
stanje trke	Situacija u kojoj više niti ili procesa čita i upisuje deljeni podatak, a konačni rezultat zavisi od relativnog vremenskog redosleda njihovog izvršavanja.
izgladnjivanje	Situacija u kojoj raspoređivač neograničeno dugo zaobilazi proces koji je spreman za izvršavanje; iako proces može da nastavi, nikada nije izabran.

Tabela 5.1 Neki ključni pojmovi vezani za konkurentnost

Teškoće konkurentnosti

- Deljenje globalnih resursa
- OS-u je teško da optimalno upravlja dodelom resursa
- Teško je pronaći greške jer rezultati nisu deterministički ni ponovljivi



Stanje trke

- Nastaje kada više procesa ili niti čita i upisuje deljene podatke
- Konačni rezultat zavisi od redosleda izvršavanja
  - „gubitnik“ trke je proces koji poslednji ažurira podatke i on određuje konačnu vrednost promenljive



Potreba za međusobnim isključivanjem

- Ako pristup deljenim podacima nije kontrolisan, procesi ili niti mogu videti te podatke u nekonzistentnom stanju
- Rezultat konkurentnog izvršavanja zavisiće od redosleda u kome se instrukcije prepliću.
- Greške zavise od vremenskih odnosa i obično nisu ponovljive.

Jednostavan primer

- Pretpostavimo da P1 i P2 izvršavaju ovaj kod i dele promenljivu **a**
- Procesi mogu biti prekinuti u bilo kom trenutku.
- Pretpostavimo da je P1 prekinut posle naredbe za unos, a da se zatim P2 izvrši u celosti
- Znak koji P1 ispiše biće onaj koji je pročitao P2 !!

```
static char a;

void echo()
{
    cin >> a;
    cout << a;
}
```

U čemu je problem?

- Ovo je primer *stanja trke*
- Pojedinačni procesi (niti) izvršavaju se sekvencijalno i izolovano, ali ih konkurentnost dovodi u interakciju.
- Moramo sprečiti konkurentno izvršavanje procesa dok menjaju iste podatke. Moramo obezbediti *međusobno isključivanje*.

Problem kritične sekcije

- Kada proces izvršava kod koji radi sa deljenim podacima (ili resursima), kažemo da je proces u svojoj kritičnoj sekciji (CS) za te deljene podatke
- Moramo obezbediti međusobno isključivanje pri izvršavanju kritičnih sekcija.
- Samo jedan proces u jednom trenutku može biti u svojoj CS (za te deljene podatke ili resurs).

Problem kritične sekcije

- Obezbeđivanje međusobnog isključivanja garantuje da će se povezane CS izvršavati *serijski* umesto *konkurentno*.
- Problem kritične sekcije svodi se na to kako obezbediti mehanizme za međusobno isključivanje, tako da akcije konkurentnih procesa ne zavise od redosleda preplitanja njihovih instrukcija

Problem kritične sekcije

- Procesi/niti moraju da zatraže dozvolu za ulazak u CS i da signaliziraju kada iz nje izađu.
- Struktura programa:
  - ulazna sekcija: zahteva ulazak u CS
  - izlazna sekcija: javlja da je CS završena
  - preostala sekcija (RS): kod koji ne uključuje deljene podatke i resurse.
- Problem CS postoji i na višeprocesorskim i na jednoprocesorskim sistemima.

Mrtva petlja i izgladnjivanje

- Mrtva petlja: dva ili više procesa trajno su blokirani jer svaki čeka na resurs koji ekskluzivno drži neki od ostalih.
- Izgladnjivanje: procesu se uvek iznova uskraćuje pristup nekom resursu zaštićenom međusobnim isključivanjem, iako resurs povremeno postaje dostupan.

Zahtevi za međusobno isključivanje

- Međusobno isključivanje: mora biti obezbeđeno
- Neometanje: proces koji se zaustavi ne sme ometati druge procese
- Bez mrtve petlje i izgladnjivanja
- Napredak: procesu se ne sme uskratiti pristup kritičnoj sekciji kada je nijedan drugi proces ne koristi
- Ne prave se pretpostavke o relativnim brzinama procesa ni o broju procesa
- Proces ostaje u svojoj kritičnoj sekciji samo konačno vreme



Međusobno isključivanje: hardverska podrška

- **Zabrana prekida**
  - jednoprocesorski sistem
  - zabrana prekida garantuje međusobno isključivanje



- **Nedostaci:**
  - efikasnost izvršavanja može primetno da opadne
  - ovaj pristup ne funkcioniše u višeprocesorskoj arhitekturi



Međusobno isključivanje: hardverska podrška



- Specijalne mašinske instrukcije
  - Instrukcija „uporedi i zameni“ (Compare&Swap)
    - naziva se i instrukcija „uporedi i razmeni“
    - vrednost u memoriji se **poredi** sa testnom vrednošću
    - ako je stara vrednost u memoriji = testna vrednost, u tu memorijsku lokaciju upisuje se nova vrednost
    - uvek se vraća stara vrednost iz memorije
    - izvršava se atomično, u hardveru.

Međusobno isključivanje: hardverska podrška

- Instrukcija „uporedi i zameni“
  - Definicija hardverske instrukcije u pseudokodu:



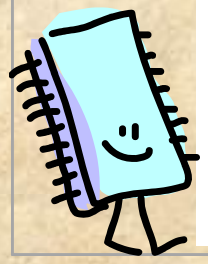
```
compare_and_swap (word, test_val, new_val)
old_val = word;
if (old_val == test_val)
    word = new_val;
return old_val;
```

Instrukcija „uporedi i zameni“

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true) {
      while (compare_and_swap(bolt, 0, 1) == 1)
           /* do nothing */;
       /* critical section */;
       bolt = 0;
       /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ... ,P(n));

}
```

(a) Compare and swap instruction



word = bolt
test_val = 0
new_val = 1

Ako je bolt 0 kada se izvrši C&S, uslov je netačan i P ulazi u svoju kritičnu sekciju. (ostavlja bolt = 1)
Ako je bolt = 1 kada se izvrši C&S, P nastavlja da izvršava while petlju. To je aktivno čekanje (ili vrtenje u petlji)

Slika 5.2 Hardverska podrška za međusobno isključivanje

Specijalne mašinske instrukcije: prednosti

- Primenljive na proizvoljan broj procesa, na jednom procesoru ili na više procesora koji dele glavnu memoriju
- Jednostavne i lako proverljive
- Podržavaju više kritičnih sekcija, od kojih svaku definiše posebna promenljiva



Specijalne mašinske instrukcije: nedostaci

- Koristi se aktivno čekanje, pa proces, dok čeka na pristup kritičnoj sekciji, i dalje troši procesorsko vreme
- Moguće je izgladnjivanje kada proces napusti kritičnu sekciju, a čeka više procesa
- Moguća je mrtva petlja ako se koristi raspoređivanje po prioritetima



Mehanizmi konkurentnosti

Semafor	Celobrojna vrednost koja se koristi za signalizaciju između procesa. Nad semaforom se mogu izvršavati samo tri operacije, sve atomične: inicijalizacija, umanjenje i uvećanje. Operacija umanjenja može dovesti do blokiranja procesa, a operacija uvećanja do deblokiranja procesa. Naziva se i **brojački semafor** ili **opšti semafor**
Binarni semafor	Semafor koji uzima samo vrednosti 0 i 1.
Muteks	Sličan binarnom semaforu. Ključna razlika je u tome što proces koji zaključa muteks (postavi vrednost na nulu) mora biti onaj koji ga otključava (postavlja vrednost na 1).



Semafor

Promenljiva sa celobrojnom vrednošću nad kojom su definisane samo tri operacije:

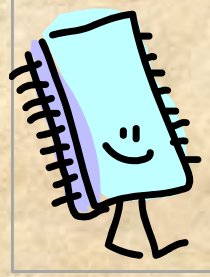
Osim ove tri operacije, ne postoji drugi način da se semafor ispita ili promeni

1) **Inicijalizuje se nenegativnim celim brojem**
2) **Operacija semWait umanjuje vrednost**
3) **Operacija semSignal uvećava vrednost**

Primitive semafora

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

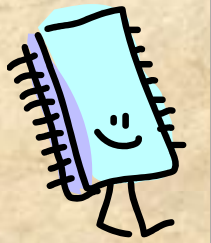
Figure 5.3 A Definition of Semaphore Primitives



Primitive binarnog semafora

```
struct binary_semaphore {
      enum {zero, one} value;
      queueType queue;
};
void semWaitB(binary_semaphore s)
{
      if (s.value == one)
            s.value = zero;
      else {
               /* place this process in s.queue */;
               /* block this process */;
      }
}
void semSignalB(semaphore s)
{
      if (s.queue is empty())
            s.value = one;
      else {
               /* remove a process P from s.queue */;
               /* place process P on ready list */;
      }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives



Međusobno isključivanje

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.6 Mutual Exclusion Using Semaphores

Muteks

- Muteks (mutex) je „brava“ za kritičnu sekciju
- **`lock(m)`:** ako je muteks otključan, proces ga zaključava i ulazi u kritičnu sekciju; inače se blokira
- **`unlock(m)`:** proces otključava muteks; ako drugi procesi čekaju, jedan od njih nastavlja
- Za razliku od binarnog semafora, muteks otključava samo proces koji ga je zaključao

```
lock(m);
/* kritična sekcija */
unlock(m);
```

Problem proizvođača i potrošača

Opšta situacija:

- jedan ili više proizvođača generišu podatke i smeštaju ih u bafer
- jedan potrošač uzima stavke iz bafera, jednu po jednu
- u svakom trenutku samo jedan proizvođač ili potrošač može pristupati baferu

Problem:

- obezbediti da proizvođač ne dodaje podatke u pun bafer, a potrošač ne uzima podatke iz praznog bafera

Rešenje pomoću semafora



```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```



Slika 5.13 Rešenje problema proizvođača i potrošača sa ograničenim baferom pomoću semafora

Pregled

- Konkurentni procesi, niti
- Pristup deljenim podacima/resursima
- Potreba za obezbeđivanjem međusobnog isključivanja
- Hardverski mehanizmi su ograničeno korisni
- Semafori: mehanizam OS-a za međusobno isključivanje i druge probleme sinhronizacije
- Standardni/brojački semafor
- Binarni semafor i muteks
- Problem proizvođača i potrošača

Rezime



Teme

- Multiprogramiranje, multiprocesiranje, distribuirana obrada

Međusobno isključivanje

- U svakom trenutku samo jedan proces sme da pristupa resursu
- Jedan pristup koristi posebne mašinske instrukcije
  - npr. instrukciju „uporedi i zameni“ (compare&swap)

Semafori

- Služe za signalizaciju između procesa i za međusobno isključivanje
- Brojački i binarni semafor; problem proizvođača i potrošača

Muteks

- Brava koju može da otključa samo proces koji ju je zaključao