Sinhronizacija procesa

1 Uvod

Već smo videli da se procesi mogu izvoditi konkurentno ili paralelno. Takođe, na prethodnim predavanjima smo prikazali raspoređivanje procesa i opisali kako planer CPU-a brzo zamenjuje procese kako bi se osiguralo njihovo istovremeno izvršavanje. To znači da se jedan proces može samo delimično izvršiti pre nego što je raspoređen drugi proces za izvršavanje. Zapravo, proces se može prekinuti u bilo kom trenutku njegovog izvršavanja, a procesorsko jezgro može biti dodeljeno drugom procesu u cilju izvršavanja njegovih instrukcija. Kod paralelnog izvršavanja, dva (ili više) tokova instrukcija (koji su predstavljeni različitim procesima) mogu istovremeno da se izvršavaju na zasebnim procesorskim jezgrama.

U ovom poglavlju objašnjavamo kako konkurentno ili paralelno izvršavanje može doprineti problemima koji uključuju integritet podataka koji se dele u nekoliko procesa.

Razmotrimo najpre primer kako se to može dogoditi. Analiziramo problem proizvođač-potrošač, koji je reprezentativan za operativne sisteme, a koji smo predstavili na prethodnim predavanjima, kada smo opisali kako se *ograničeni bafer* (bounded buffer) može koristiti da bi se mogla deliti memorija među procesima. Kao što smo istakli, naše rešenje istovremeno je dozvolilo najviše BUFFER SIZE - 1 elemenata u baferu. Pretpostavimo da želimo da izmenimo algoritam da otklonimo ovaj nedostatak. Jedna mogućnost je da se doda celobrojni brojač, inicijalizovan na 0. Brojač se povećava svaki put kada dodamo novu stavku u bafer i smanjuje se svaki put kada uklonimo jednu stavku iz bafera. Kod za proces proizvođača može se modifikovati tada na sledeći način:

```
item next_produced;
while (true) {
  /* popuni podatak next_produced sa željenim informacijama */
  while (counter == BUFFER SIZE)
    ; /* ne radi ništa, čekaj da se bafer isprazni */
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER SIZE ;
  counter++;
}
```

Kod potrošača bi se modifikovao na sledeći način:

```
item next_consumed;
while (true) {
  while (counter == 0)
  ; /* bafer prazan, ne radi ništa */
  next_consumed = buffer[out];
  out = (out + 1) % BUFFER SIZE;
  counter--;

  /* iskoristi podatak dobijen kroz next_consumed */
}
```

Iako su obe rutine iznad korektne kada se posmatraju odvojeno, moguće je da one neće raditi očekivano kada se izvršavaju konkurentno. Kao ilustracija problema, pretpostavimo da je vrednost brojača *counter* trenutno 5 i da i proizvođač i potrošač konkurentno izvršavaju liniju „counter++“ i „counter--“, respektivno. Tada, rezultujuća vrednost brojača može biti 4, 5 ili 6! Jedini korektan rezultat je svakako *counter*==5, koji se dobija kada se oba procesa

izvršavaju samostalno. Kako je moguće da se dobije pogrešna vrednost brojača? Naredba *counter++* će se izvršavati na mašinskom jeziku kao:

```
reg1 = counter
reg1 = reg1 + 1
counter = reg1
```

gde je reg1 jedan od lokalnih CPU registara. Slično, naredba *counter-* - se implementira na sledeći način:

```
reg2 = counter
reg2 = reg2 - 1
counter = reg2
```

gde je opet reg2 lokalni CPU registar. Čak iako su reg1 i reg2 zapravo isti registri (npr. akumulator registar), podsećamo da će sadržaj registara biti sačuvan i rekonstruisan prilikom zamene konteksta.

Konkurentno izvršavanje ove dve naredbe je ekvivalentno sekvencijalnom izvršavanju u kojem su instrukcije na niskom nivou „prepletene“ u nekom proizvoljnom redosledu. Jedan od njih bi bio:

```
T0: proizvođač izvršava reg1 = counter {reg1 = 5}
T1: proizvođač izvršava reg1 = reg1 + 1 {reg1 = 6}
T2: potrošač izvršava reg2 = counter {reg2 = 5}
T3: potrošač izvršava reg2 = reg2 - 1 {reg2 = 4}
T4: proizvođač izvršava counter = reg1 {counter = 6}
T5: potrošač izvršava counter = reg2 {counter = 4}
```

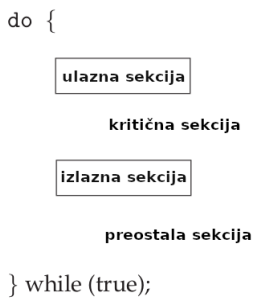
što rezultuje pogrešnim stanjem brojača (*counter* = 4), što indicira da se 4 elementa nalaze u baferu, kada je zapravo 5 elemenata upisano. Ako bismo samo izmenili redosled izvršavanja T4 i T5 dobili bismo opet pogrešno stanje (*counter* = 6).

Došli bismo do ovog pogrešnog stanja jer smo dozvolili da oba procesa konkurentno manipulišu brojačem. Ovakva situacija, u kojoj više procesa pristupa istim podacima i istovremeno manipuliše istim podacima, a ishod izvršenja zavisi od određenog redosleda u kojem se taj pristup dešava, naziva se *stanje utrkivanja* (*race condition*). Da bismo se zaštitili od *stanja utrkivanja*, moramo osigurati da samo jedan proces u datom trenutku može manipulisati promenljivom brojač. Da bismo tako nešto obezbedili , zahtevamo da se procesi na neki način sinhronizuju.

Situacije poput ove koja je upravo opisana često se javljaju u operativnim sistemima jer različiti delovi sistema manipulišu resursima. Dodatno, kao što smo naglasili u prethodnim lekcijama, sve veći značaj više-jezgarnih sistema doprineo je povećanju fokusa na razvoju aplikacija u višestrukim nitima izvršavanja. U takvim se aplikacijama nekoliko niti - koje veoma verovatno dele podatke među sobom - pokreću istovremeno i paralelno na različitim jezgrima. Veliki deo ove lekcije posvećujemo sinhronizaciji procesa i koordinaciji između kooperativnih procesa.

2 Problem kritične sekcije

Razmatranje sinhronizacije procesa započinjemo diskusijom o takozvanom *problemu kritične sekcije*. Razmotrimo sistem koji se sastoji od n procesa $\{P_0,$



Slika 1: Generalna struktura tipičnog procesa Pi

P_1, ..., P_{n-1}}. Svaki proces ima segment koda, koji se naziva kritični sekcija, u kojem proces može menjati zajedničke promenljive, ažurirati tabelu, upisivati u datoteku i tako dalje. Važna karakteristika sistema je da, kada jedan proces izvršava svoju kritičnu sekciju, nije dozvoljeno nijednom drugom procesu da se izvršava u svojoj kritičnoj sekciji. Odnosno, nije dozvoljeno da se dva procesa izvršavaju u svojim kritičnim sekcijama istovremeno. Problem kritične sekcije odnosi se na definisanje protokola koji procesi mogu da koriste da sarađuju. Svaki proces mora zatražiti dozvolu za ulazak u svoju kritičnu sekciju. Deo koda koji sprovodi ovaj zahtev zove se *ulazna sekcija* (eng. *entry section*). Sama kritična sekcija je praćena *izlaznom sekcijom* (eng. *exit section*), a ostatak koda je, takozvana, *preostala sekcija* (eng. *remainder section*). Opšta struktura tipičnog procesa P_i prikazana je na slici 1. Ulazna i izlazna sekcija predstavljeni su pravougaonicima kako bi se istakla važnost ovih segmentata koda.

Rešenje problema sa kritičnim sekcijama mora da ispunjava sledeća tri zahteva:

1. Međusobna (uzajamna) isključivost (*Mutual Exclusion*) : Ako proces P_i izvršava svoju kritičnu sekciju, drugi procesi tada ne mogu izvršavati svoje kritične sekcije

2. Napredak (*Progress*) : Ako se se nijedan proces ne izvršava u svojoj kritičnoj sekciji, a neki procesi žele da uđu u svoje kritične sekcije, tada samo oni mogu konkurisati u odlučivanju koji će sledeći proces ući u kritičnu sekciju, a ovaj izbor mora se obaviti brzo i bez preteranog odlaganja

3. Ograničeno čekanje (*Bounded Waiting*) : Postoji ograničenje na broj puta koliko je ostalim procesima dozvoljeno da uđu u svoje kritične sekcije nakon što je proces podneo zahtev za ulazak u njegovu kritičnu sekciju, a pre nego što mu je zahtev odobren.

U datom trenutku, mnogi procesi mogu biti aktivni u kernel modu u okviru operativnog sistema. Kao rezultat, kod koji implementira operativni sistem (kernel kod) podložan je nekolicini mogućih stanja utrkivanja. Uzmimo za primer strukturu podataka u okviru kernela koja čuva listu svih otvorenih datoteka u sistemu.

Ova lista mora biti izmenjena kada se nova datoteka otvori ili postojeća zatvori (dodavanje datoteke na listu ili uklanjanje sa liste). Ako su dva procesa otvarala datoteke istovremeno, nezavisna ažuriranja ove liste mogu rezultovati stanjem utrkivanja. Ostale strukture podataka kernela koje su sklone mogućim stanjem utrkivanja uključuju: strukture za praćenje raspodele memorije, održavanje lista procesa i za praćenje obrade prekida. Programeri kernela moraju da obezbede da operativni sistem ne bude podložan stanju utrkivanja prilikom rada sa ovim strukturama.

Za obradu kritičnih sekcija u operativnim sistemima koriste se dva pristupa: prisvojivi (preemptive) i neprisvojivi (non-preemptive) kerneli. Prisvojivi kernel omogućava prekidanje izvršenja procesa dok se izvšava u režimu kernela. Neprisvojivi kernel ne dozvoljava da se proces koji se izvršava u režimu kernela prekine; proces u kernel modu će se izvršavati sve dok ne izađe iz kernel moda, blokira ili dobrovoljno ne prepusti kontrolu nad CPU-om. Očigledno je da neprisvojivi kernel u suštini ne omogućava uslove za postojanje stanja utrkivanja u radu sa kernel strukturama podataka jer je u okviru kernela aktivan samo jedan proces u datom trenutku. Ovo ne važi za prisvojive kernele, koji moraju biti pažljivo dizajnirani kako bi se osiguralo da deljeni kernel podaci ne budu podložni stanju utrkivanja. Prisvojivi kerneli se posebno teško dizajniraju na višejezgarnim arhitekturama, jer su u ovim okruženjima moguća dva (ili više) kernel procesa koji se istovremeno izvršavaju na različitim procesorima.

Zašto bi onda bilo ko favorizirao prisvojive kernele umesto neprisvojivih? Prisvojivi kernel može da ima bolju odzivnost, jer postoji manji rizik da će se proces u režimu kernela izvršavati proizvoljno dugo pre nego što će procesor prepustiti procesima koji čekaju. Štaviše, prisvojivi kernel je pogodniji za programiranje u realnom vremenu, jer će omogućiti real-time procesu da prekine proces koji se trenutno izvršava u kernel modu.

3 Patersonov algoritam

U nastavku prikazujemo klasično softversko rešenje problema kritične sekcije koji se naziva i Petersonovo rešenje. Zbog načina na koji savremene računarske arhitekture izvršavaju osnovne instrukcije mašinskog jezika, kao što su učitavanje i skladištenje podataka iz/u memorije, ne postoje garancije da će Petersonovo rešenje ispravno raditi na takvim arhitekturama. Međutim, predstavljamo ovo rešenje jer pruža dobar algoritamski opis rešenja problema sa kritičnim sekcijama i ilustruje određene složenosti dizajniranja softvera koji vodi računa o zahtevima međusobne isključivosti, napretka i ograničenog čekanja. Petersonovo rešenje je ograničeno na dva procesa koji naizmenično izvršavaju kritične sekcije i preostale sekcije. Procesi su označeni sa P_i i P_j. Petersonovo rešenje zahteva da dva procesa dele dve promenljive:

```
int turn;
boolean flag[2];
```

Promenljiva *turn* pokazuje čiji je red da uđe u svoju kritičnu sekciju. To jest, ako je $turn == i$, tada se proces P_i može izvršiti u njegovoj kritičnoj sekciji. Niz

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        kritična sekcija

    flag[i] = false;

        preostala sekcija

} while (true);
```

Slika 2: Patersonov algoritam

flag koristi se za označavanje da li je proces spreman da uđe u svoju kritičnu sekciju. Na primer, ako je *flag*[i] true, ova vrednost ukazuje da je P_i spreman da krene sa izvršavanjem svoje kritične sekcije.

Algoritam je prikazan na slici 2.

Da bi ušao u kritičnu sekciju, proces P_i prvo postavlja *flag*[i] na true, a zatim *turn* na vrednost j, omogućavajući drugom procesu da uđe u kritičnu sekciju ako on to želi. Ako oba procesa pokušaju da uđu istovremeno, *turn* će biti postavljen na i i j otprilike istovremeno. Samo jedna vrednost od te dve će opstati; druga će se pojaviti, ali će biti odmah prepisana. Konačna vrednost *turn* promenljive određuje koji od dva procesa će prvi ući u svoju kritičnu sekciju.

Da bismo dokazali da je ovo rešenje tačno, moramo da pokažemo da je:

1. Očuvana međusobna isključivost
2. Uslov za napredak je ispunjen
3. Uslov ograničenog čekanja je ispunjen

Da bismo dokazali osobinu 1, primetimo da svaki P_i ulazi u svoju kritičnu sekciju samo ako je zadovoljeno *flag*[j] == false ili *turn* == i. Takođe imajte na umu da, ako oba procesa žele istovremeno da izvršavaju svoje kritičnim sekcije, biće *flag*[0] == *flag*[1] == true. Ova dva zapažanja impliciraju da P_i i P_j ne mogu izaći iz svojih while petlji istovremeno, jer vrednost *turn* može biti ili 0 ili 1, ali ne može nikako biti i jedno i drugo. Dakle, jedan od procesa, recimo P_j, može uspešno da izađe iz svoje while petlje, dok P_i ostaje u njoj sve dok je P_j u kritičnoj sekciji. Kao rezultat, uzajamna isključivost je zadovoljena.

Da bismo dokazali osobine 2 i 3, primetimo da proces P_i može biti sprečen da uđe u kritičnu sekciju samo ako je zaglavljen u while petlji kada je ispunjeno *flag*[j] == true i *turn* == j. Ako P_j nije spreman za ulazak u kritičnu sekciju, *flag*[j] == false, te P_i može ući u svoju kritičnu sekciju. Ako je P_j postavio *flag*[j] na true i takođe stigne sa izvršavanjem do svoje while petlje, tada je ili *turn* == i ili *turn* == j. Ako je *turn* == i, P_i će ući u kritičnu sekciju,

a ako je $turn == j$, P_j će ući u kritičnu sekciju. Međutim, kada P_j izađe iz nje, postaviće $flag[j]$ na false, omogućavajući P_i ulaz u njegovu kritičnu sekciju. Ako P_j resetuje $flag[j]$ na true (pre nego što P_i stigne da proveri promenjen uslov čekajući u while petlji), P_j mora tada takođe postaviti i *turn* na *i*. Dakle, pošto P_i ne menja vrednost promenljive *turn* prilikom izvršavanja while petlje, P_i će ući u kritičnu sekciju (*napredak*) nakon najviše jednog ulaska od strane P_j (*ograničeno čekanje*).

4 Hardverska podrška za sinhronizaciju

Upravo smo opisali jedno softversko rešenje problema kritične sekcije. U nastavku ćemo istražiti još nekoliko rešenja problema kritičnih sekcija koristeći tehnike u rasponu od hardvera do softverskog API-ja dostupnog i programerima kernela i programerima aplikacija. Sva ova rešenja su zasnovana na premisi *zaključavanja* - zaštiti kritičnih sekcija korišćenjem *katanca* (eng. lock). Kao što ćemo videti, dizajniranje takvih *katanaca* može biti prilično sofisticirano.

Započinjemo predstavljanjem nekih jednostavnih hardverskih instrukcija koje su dostupne na mnogim sistemima i pokazujemo kako se mogu efikasno koristiti u rešavanju problema sa kritičnim sekcijama. Hardverska podrška može značajno olakšati bilo koji zadatak programiranja i poboljšati efikasnost sistema.

Problem sa kritičnim sekcijama mogao bi da se reši jednostavno u okruženju s jednim procesorom ako bismo sprečili da se prekidi pojave dok se deljena promenljiva menja. Na ovaj način, mogli bismo biti sigurni da će se trenutni niz instrukcija moći izvršiti neometano, bez prisvajanja procesora od strane operativnog sistema i dodeljivanja procesora drugom procesu. Kao rezultat, bilo koje druge instrukcije su onemogućene, tako da se ne mogu desiti neočekivane izmene na deljenim varijablama. To je često pristup koji koriste neprisvojivi kerneli.

Nažalost, ovo rešenje nije izvodljivo u višeprocesorskom (višejezgarnom) okruženju. Onemogućavanje prekida na multiprocesorskom sistemu može biti vremenski zahtevno jer se poruka prosleđuje svim procesorima. Ova poruka odlaže ulazak u svaku kritičnu sekciju, a efikasnost sistema opada drastično kao posledica. Osim toga, ovakav pristup bi imao loš uticaj na sistemski sat, ukoliko se on ažurira iz prekida.

Mnogi savremeni računarski sistemi zato nude posebne hardverske instrukcije koje omogućavaju da ili proverimo i modifikujemo sadržaj memorijske reči ili da na atomički način zamenimo sadržaj dve reči. Ove posebne instrukcije možemo koristiti za rešavanje problema kritičnih sekcija na relativno jednostavan način, a zovu se *test_and_set*() i *compare_and_swap*() instrukcije.

Instrukcija *test_and_set*() može se definisati kao što je prikazano na slici 3.

Važna karakteristika ove instrukcije je da se izvodi atomički. Prema tome, ako se dve instrukcije *test_and_set*() izvršavaju istovremeno (svaka na različitom CPU-u), one će se izvršavati sekvencijalno, jedna posle druge, nekim proizvoljnim redosledom. Ako mašina podržava *test_and_set*() instrukciju, tada možemo implementirati međusobnu isključivost deklaracijom boolean promen-

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Slika 3: Test_and_set instrukcija

```
do {
   while (test_and_set(&lock))
      ; /* ne radi ništa */

      /* kritična sekcija */

   lock = false;

      /* preostala sekcija */
} while (true);
```

Slika 4: Test_and_set instrukcija korišćena za međusobnu isključivost

ljive *lock*, inicijalizovane sa *false*. Struktura procesa P_i koji bi koristio ovakav pristup prikazan je na slici 4.

Instrukcija compare_and_swap(), za razliku od prethodne, radi sa tri operanda i definisana je na slici 5.

Operand *value* je postavljen na *new_value* samo ako je izraz (*value == expected) istinit. Bez obzira na to da li se promena vrednosti desila ili ne, *compare_and_swap*() uvek vraća originalnu vrednost promenljive *value*. Poput instrukcije *test_and_set*(), *compare_and_swap*() se izvšava na atomički način. Međusobna isključivost može se omogućiti na sledeći način: globalno varijabla (*lock*) je deklarisana i inicijalizovana je na 0. Prvi proces koji poziva *compare_and_swap*() postaviće *lock* na 1. Zatim će ući u svoju kritičnu sekciju, jer je prvobitna vrednost *lock* bila jednaka očekivanoj vrednosti 0. Naknadni pozivi *compare_and_swap*() neće uspeti, jer *lock* više nije jednak očekivanoj vrednosti 0. Kada proces izađe iz kritičnog sekcije, on postavlja *lock* na 0, što omogućava

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Slika 5: Compare_and_swap instrukcija

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /*  ne radi ništa  */

      /* kritična sekcija */

   lock = 0;

      /*preostala sekcija */
} while (true);
```

Slika 6: Compare_and_swap instrukcija korišćena za međusobnu isključivost

drugom procesu da uđe u kritičnu sekciju. Struktura procesa P_i koji bi koristio ovakav algoritam prikazana je na slici 6.

Iako ovi algoritmi zadovoljavaju uslov uzajamne isključivosti, oni ne zadovoljavaju uslov ograničenog čekanja. Na slici 7 predstavljamo drugi algoritam koji koristi instrukciju *test_and_set*() i koji zadovoljava sve zahteve u vezi sa problemom kritične sekcije. Zajednički korišćene strukture podataka su:

```
boolean waiting[n];
boolean lock;
```

Ove strukture podataka inicijalizovane su na *false*. Da bismo dokazali da je uslov za uzajamnu isključivost ispunjen, primetimo da proces P_i može ući u svoju kritičnu sekciju samo ako *waiting*[*i*] == *false* ili *key* == *false*. Vrednost *key* može postati *false* samo ako se izvrši *test_and_set*(). Prvi proces koji će izvršiti *test_and_set*() će videti da je key == *false*, dok svi ostali moraju da sačekaju. Promenljiva *waiting*[*i*] može postati false samo ako neki drugi proces napusti kritičnu sekciju - samo je jedno *waiting*[*i*] postavljeno na false, što obezbeđuje održavanje zahteva za međusobnom isključivošću.

Da bismo dokazali da je uslov za *napredak* ispunjen, primetimo da argumenti predstavljeni kod međusobne isključivosti takođe važe i ovde, jer proces koji izlazi iz kritične sekcije ili postavlja *lock* u *false* ili postavlja *waiting*[*j*] na *false*. Oboje omogućavaju da se nastavi proces koji čeka da uđe u svoju kritičnu sekciju.

Da bismo dokazali da je uslov ograničenog čekanja ispunjen, primetimo da, kada proces napusti svoj kritični deo, skenira niz koji čeka u cikličkom redosledu (i + 1, i + 2, ..., n - 1, 0 , ..., i - 1). Nakon toga on označava prvi proces u ovom redosledu koji se nalazi u *ulaznoj sekciji* (*waiting*[*j*] == true) kao sledeći za ulazak u kritičnu sekciju. Svaki proces koji čeka da uđe u kritičnu sekciju će to učiniti u roku od n - 1 ciklusa.

Sama implementacija atomičnih *test_and_set*() i *compare_and_swap*() instrukcija detaljnije je obrađena u knjigama o arhitekturi računara.

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* kritična sekcija */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* preostala sekcija */
} while (true);
```

Slika 7: Uzajamna isključivost sa ograničenim čekanjem korišćenjem test_and_set()

5 Muteksi

Hardverska rešenja problema sa kritičnim sekcijama predstavljena u prethodnom odeljku su komplikovana, i uglavnom nedostupna programerima. Umesto toga, dizajneri operativnih sistema su kreirali softverske alate za rešavanje problema kritičnih sekcija. Najjednostavniji od ovih alata je muteks zaključavanje (izraz muteks kao skraćenica za međusobnu isključivost - MUTual EXclusion).

Muteks koristimo kako bismo zaštitili kritične sekcije i tako sprečili nastanak stanja utrkivanja. Da bi ušao u kritičnu sekciju, proces mora *prisvojiti* (aquire) muteks (kaže se često i *zaključati*), a kada napušta kritičnu sekciju on ga *oslobađa* (release, kaže se često i *otključati*). Operativni sistem obezbeđuje posebne funkcije za ove dve operacije kao što je prikazano ispod:

```
do
{
        Acquire (prisvoji muteks)
        kritična sekcija
        Release (oslobodi muteks)
        preostala sekcija
}
while (true);
```

Muteks se sastoji od logičke promenljive čija vrednost daje indikaciju da li je muteks dostupan ili ne. Ako je moguće prisvojiti muteks, poziv *Aquire*() uspeva, nakon čega je muteks nedostupan. Proces koji pokuša da prisvoji nedostupan muteks biće blokiran sve dok se on ne oslobodi. Definicija *Aquire*() je sledeća:

```
acquire()
{
        while (!available);
```

```
        /* busy wait */
        available = false;
}
```

Definicija *Release*() je sledeća:

```
release()
{
        available = true;
}
```

Izuzetno je važno da se pozivi *Aquire*() ili *Release*() moraju obavljati **na atomički način**.

Glavni nedostatak implementacije koja je ovde data je činjenica da se *čeka u petlji* (*busy wait*) prilikom prisvajanja muteksa. Dok je proces u kritičnoj sekciji, svaki drugi proces koji pokušava da uđe u svoju kritičnu sekciju biće zaglavljen u petlji unutar poziva *Aquire*() funcije. Zbog toga se ova vrsta zaključavanja muteksa naziva i *spinlock* jer se proces „vrti“ dok se čeka da katanac (muteks) postane dostupan (isti problem vidimo sa primerima koda koji ilustruju instrukcije *test_and_set*() i *compare_and_swap*(). Ovo kontinualno izvršavanje u petlji očigledno je problem u stvarnom sistemu multi-programiranja, gde se jedan procesor deli između višestrukih procesa. Čekanje na ovaj način troši CPU cikluse koje bi neki drugi proces mogao produktivnije iskoristiti. Međutim, *spinlock*-ovi imaju prednost koja se ogleda u tome što nije potrebna zamena konteksta kada proces čeka na zaključavanje, a zamena konteksta može trajati dosta vremena. Prema tome, kad se očekuje da se muteksi drže na kratko, spinlock-ovi su korisni. Oni se često koriste u multiprocesorskim sistemima gde jedna nit može da se „zavrti“ na jednom procesoru, dok druga nit izvršava svoju kritičnu sekciju na drugom procesoru.

Kasnije u ovom poglavlju ispitujemo kako se zaključavanje muteksa može koristiti za rešavanje klasičnih problema sa sinhronizacijom.

6 Semafori

Muteksi, kao što smo ranije spomenuli, uglavnom se smatraju najjednostavnijim alatima za sinhronizaciju. U ovom odeljku ispitujemo robusniji alat koji se može ponašati slično kao muteks, ali takođe može pružiti sofisticiranije načine na koje procesi mogu sinhronizovati svoje aktivnosti.

Semafor S je celobrojna promenljiva koja, sa izuzetkom inicijalizacije, omogućava pristup samo kroz dve standardne atomičke operacije: *wait*() i *signal*(). Operacija *wait*() prvobitno je nazvana *P* (od holandskog Proberen, što znači „testirati“), dok se operacija *signal*() prvobitno označavala *V* (od Verhogen, „inkrementirati“).

Definicija operacije *wait*() je:

```
wait(S)
{
        while (S <= 0 )
                ; // busy wait
        S--;
}
```

Definicija operacije *signal*() je:

```
signal(S)
{
        S++;
}
```

Sve modifikacije celobrojne vrednosti semafora u operacijama *wait*() i *signal*() moraju se izvršiti atomički. To jest, kada jedan proces modifikuje vrednost semafora, nijedan drugi proces ne može istovremeno da menja tu istu vrednost semafora. Pored toga, u slučaju *wait*(S), ispitivanje celobrojne vrednosti S (S $\leq$ 0), kao i njegove moguće modifikacije (S--), moraju se takođe izvršiti bez prekida. Najpre ćemo da vidimo kako se semafori mogu koristiti.

6.1 Korišćenje semafora

Operativni sistemi često razlikuju brojačke i binarne semafore. Vrednost brojačkog semafora može biti proizvoljno velika, dok vrednost binarnog semafora može biti u opsegu između 0 i 1. Dakle, binarni semafori se ponašaju slično kao muteksi. Zapravo, na sistemima koji ne omogućavaju zaključavanje putem muteksa, binarni semafori se mogu koristiti umesto njih za implementaciju uzajamne isključivosti. Brojački semafor se može koristiti za kontrolu pristupa datom resursu koji se sastoji od ograničenog broja instanci. Semafor se inicijalizuje na broj raspoloživih resursa (odnosno instanci tog resursa). Svaki proces koji želi koristiti resurs izvodi operaciju *wait*() na semaforu (čime se smanjuje brojač). Kada proces oslobodi resurs, on izvodi operaciju *signal*() (povećavajući vrednost brojača). Kada brojač semafora dođe do 0, to je indikacija da se koriste svi resursi. Nakon toga će se procesi koji žele koristiti resurs blokirati sve dok brojač ne bude veći od 0.

Takođe, možemo da koristimo semafore za rešavanje različitih problema u vezi sa sinhronizacijom. Na primer, razmotrite dva konkurentno izvršavana procesa: P1 sa iskazom S1 i P2 sa iskazom S2. Pretpostavimo da zahtevamo da se S2 izvrši tek nakon završetka S1. Ovu šemu možemo lako implementirati tako što ćemo dozvoliti da P1 i P2 dele semafor *synch*, inicijalizovan na 0. U procesu P1 tada dodajemo iskaze

```
S1;
signal(synch);
```

U procesu P2:

```
wait(synch);
S2;
```

Pošto je *synch* inicijalizovan na 0, P2 će izvršiti S2 tek nakon što P1 pozove *signal*(*synch*), što će se desiti tek nakon izvršenja iskaza S1.

6.2 Implementacija semafora

Podsetimo se da primena muteks zaključavanja koja je prethodno elaborirana ima problem u vezi sa čekanjem u petlji. Prethodno opisane definicije semaforskih operacija *wait*() i *signal*() nisu rešile taj problem. Da bismo isključili

korišćenje čekanja u petlji, izmenićemo definiciju operacija *wait*() i *signal*() na sledeći način: Kada proces izvrši operaciju *wait*() i utvrdi da vrednost semafora nije pozitivna, mora sačekati. Međutim, umesto da čeka u petlji, proces će se blokirati. Operacija blokiranja postavlja proces u red čekanja semafora, a stanje procesa menja se u stanje *čekanja*. Zatim se kontrola prenosi CPU planeru, koji bira drugi proces koji treba da se izvrši. Proces koji je blokiran i čeka na semafor *S*, treba ponovo pokrenuti kada neki drugi proces pozove *signal*() operaciju na semaforu *S*. Proces koji je čekao se ponovo pokreće *wakeup*() operacijom, koja procesu menja stanje iz stanja čekanja u stanje spreman, nakon čega se proces postavlja u red čekanja spremnih procesa. Da bismo implementirali semafore u skladu sa ovom opisom, definisaćemo semafor na sledeći način:

```
typedef struct
{
        int value;
        struct process *list;
} semaphore ;
```

Svaki semafor sadrži celobrojnu (integer) vrednost *value* kao i listu procesa. Kada proces mora da čeka na semafor, on će biti dodat na listu list. Nakon izvršenja *signal*() operacije, jedan od procesa u listi čekanja na semafor će biti uklonjen iz liste i premešten u stanje spreman. Sada, *wait*() operacija se može definisati kao:

```
wait(semaphore *S)
{
        S->value--;
        if (S->value < 0)
        {
                add this process to S->list;
                block();
        }
}
```

dok *signal*() operacija izgleda ovako:

```
signal(semaphore *S)
{
        S->value++;
        if (S->value <= 0)
        {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Operacija *block*() obustavlja proces koji je poziva. Operacija *wakeup*(P) nastavlja izvršavanje blokiranog procesa *P*. Ove dve operacije obezbeđuje operativni sistem kao osnovne sistemske pozive.

Obratite pažnju da u ovoj implementaciji vrednosti semafora mogu biti negativne, što je suprotno prvobitnoj definiciji (sa čekanjem u petlji) po kojoj vrednosti semafora nikada nisu negativne. Ako je vrednost semafora negativna, njena apsolutna vrednost predstavlja broj procesa koji čekaju na tom semaforu. Ova činjenica rezultat je promene redosleda smanjivanja vrednosti semafora i testa u implementaciji operacije *wait*(). Lista procesa koji čekaju može se lako implementirati povezivanjem kontrolnih blokova procesa (PCB).

Svaki semafor sadrži celobrojnu vrednost *value* i pokazivač na listu PCB-ova. Jedan od načina dodavanja procesa na listu i uklanjanja procesa sa liste, koji obezbeđuje ograničeno čekanje je upotreba FIFO reda čekanja, gde semafor sadrži pokazivače na početak i kraj reda. Međutim, generalno, implementacija liste može koristiti bilo koju strategiju čekanja u redu, a pravilna upotreba semafora ne zavisi od određene strategije čekanja u listi procesa semafora.

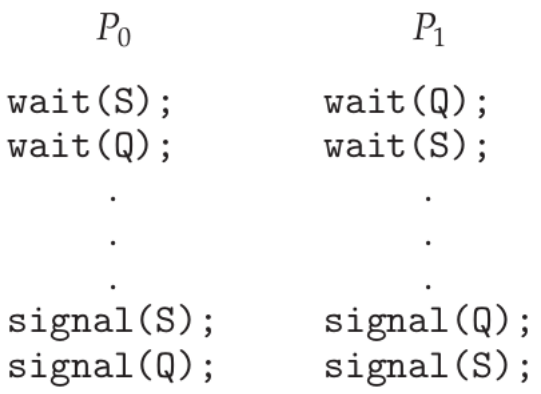
Ključni detalj je da operacije nad semaforom budu izvedene na atomički način. Mora se garantovati da nijedna dva procesa ne mogu istovremeno da izvršavaju operacije *wait*() i *signal*() na istom semaforu. Ovo je problem sa kritičnim sekcijama i u okruženju s jednim procesorom to možemo rešiti jednostavnom zabranom prekida u toku izvršenja ove dve operacije. Ova šema funkcioniše u okruženju s jednim procesorom jer se, kada se zabrane prekidi, instrukcije u sklopu različitih procesa ne mogu preplitati i nema uslova za nastanak stanja utrkivanja. Samo se trenutno pokrenut proces izvršava dok se prekidi ne omoguće ponovo, nakon čega planer ponovo može uspostaviti kontrolu. U multiprocesorskom okruženju, prekidi bi tada morali biti onemogućeni na svakom procesoru. U suprotnom, instrukcije iz različitih procesa (koji rade na različitim procesorima) mogu se „preplitati“ na proizvoljni način. Onemogućavanje prekida na svakom procesoru može biti težak zadatak i dodatno može znatno degradirati performanse sistema. Zbog toga, SMP (*Symmetric MultiProcessing*) sistemi moraju da obezbede alternativne tehnike zaključavanja - kao što su *compare_and_swap*() ili spinlock-ovi da bi se osiguralo atomičko *wait*() i *signal*() izvršavanje.

Usled ove nemogućnosti da se istovremeno izvršavaju *wait*() i *signal*() operacije, očigledno je da ovom modifikovanom definicijom operacija *wait*() i *signal*() nismo u potpunosti eliminisali čekanje u petlji. Umesto toga, odložili smo čekanje u petlji koje se pre dešavalo na ulaznim sekcijama pre kritičnih sekcija aplikativnih programa. Dodatim modifikacijama smo ograničili čekanje na kritične sekcije operacija *wait*() i *signal*(), obzirom da će se sada čekati da se ove operacije izvrše ukoliko više procesa pokuša istovremeno da ih pozove. Ipak, obzirom na to da su ove sekcije kratke (ako su pravilno napišu, trebalo bi da ne sadrže više od desetak instrukcija), ovo je znatno manji problem: kritične sekcije gotovo nikad nisu zauzete, čekanje u petlji se dešava retko, a kada se desi, dešava se samo kratkotrajno. Potpuno drugačija situacija je kod aplikativnih programa čije kritične sekcije mogu biti duge (nekoliko minuta ili čak sati) ili mogu skoro uvek biti zauzete. U takvim slučajevima, čekanje u petlji je krajnje neefikasno.

6.3 Mrtve petlje (*deadlocks*) i izgladnjivanje (*starvation*)

Implementacija semafora sa redom čekanja može rezultovati situacijom kada dva ili više procesa neograničeno čekaju na izvršenje *signal*() operacije od strane samo jednog procesa. Kada se takav scenario desi, kaže se da su ovi procesi u *mrtvoj petlji*.

Da bismo ilustrovali mrtvu petlju, razmatramo sistem koji se sastoji od dva procesa, P_0 i P_1, od kojih svaki ima pristup ka dva semafora, S i Q, postavljenih na vrednost 1:



Slika 8: Nastanak mrtve petlje

Pretpostavimo da P_0 izvrši *wait*(S), a zatim P_1 izvrši *wait*(Q). Kada P_0 izvrši *wait*(Q), on mora sačekati dok P_1 izvrši operaciju *signal*(Q). Slično tome, kada P_1 izvrši *wait*(S), mora sačekati dok P_0 ne izvrši *signal*(S). Pošto se ove operacije *signal*() ne mogu izvršiti, oba procesa P_0 i P_1 se zaustavljaju i biće blokirani u nedogled.

Kažemo da je skup procesa u *mrtvoj petlji* kada svaki proces u skupu čeka događaj koji može prouzrokovati samo drugi proces iz skupa. Događaji od interesa za nas odnose se na prisvajanje i oslobađanje resursa, iako to nisu jedini uzroci nastanka mrtve petlje.

Poseban problem vezan za mrtvu petlju je *neodređeno blokiranje* ili *izgladnjivanje*, situacija u kojoj procesi čekaju neodređeno vreme unutar reda čekanja semafora. Ono nastaje kada procese iz liste čekanja na semafor uklanjamo u LIFO (Last-In, First-Out) maniru.

6.4 Inverzija prioriteta

Izazov za raspoređivanje procesa nastaje kada proces s višim prioritetom mora pročitati ili izmeniti kernel podatke kojima trenutno pristupa proces nižeg prioriteta (ili više procesa sa nižim prioritetom). Pošto su kernel podaci obično zaštićeni *katancem* (bilo koje vrste), proces višeg prioriteta će morati da sačeka da proces nižeg prioriteta završi s resursom. Situacija postaje komplikovanija ako je proces nižeg prioriteta istisnut od strane drugog procesa sa višim prioritetom.

Kao primer, pretpostavimo da imamo tri procesa L, M i H - čiji prioriteti slede redosled L <M <H. Pretpostavimo da za proces H je potreban resurs R, kojem trenutno pristupa proces L. Obično, proces H čeka da L završi sa korišćenjem resursa R. Međutim, pretpostavimo sada da proces M postane izvršan (promeni mu se stanje u stanje izvršavanja), što će blokirati proces L. Indirektno, proces sa nižim prioritetom (proces M), uticaće na to koliko dugo proces H

mora da čeka da L završi sa korišćenjem resursa R.

Ovaj problem je poznat i kao *inverzija prioriteta*. Javlja se samo u sistemima sa više od dva prioriteta, tako da je jedno rešenje ograničavanje na samo dva prioriteta. To je, međutim, nedovoljno za većinu operativnih sistema opšte namene. Tipično, ovi sistemi rešavaju problem inverzije prioriteta primenom protokola o nasleđivanju prioriteta. Prema ovom protokolu, svi procesi koji pristupaju resursima potrebnim procesu višeg prioriteta nasleđuju veći prioritet sve dok ne završe sa dotičnim resursima. Kada završe, njihovi se prioriteti vraćaju na originalne vrednosti. U gornjem primeru, protokol nasleđivanja prioriteta bi omogućio da proces L privremeno nasledi prioritet procesa H, čime sprečava proces M da se izvršava pre njega. Kada je proces L završio sa korišćenjem resursa R, on bi se odrekao nasleđenog prioriteta od H i preuzeo svoj prvobitni prioritet. Pošto bi resurs R sada bio dostupan, proces H (ne M) pokrenuo bi se sledeći.

7 Klasični problemi sa sinhronizacijom

7.1 Problem sa ograničenim baferom

Ovaj problem smo predstavili u jednom od prethodnih predavanja kada smo govorili o proizvođač-potrošač paradigmi. Ovde ćemo prezentovati generalizovanu strukturu ovakve sheme, bez ikakvog ograničavanja na određenu implementaciju. U okviru našeg rešenja, i proizvođač i potrošač dele sledeće strukture podataka:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```

Pretpostavljamo da imamo na raspolaganju n bafera, pri čemu je svaki dovoljno velik da skladišti jednu poruku. *Mutex* semafor obezbeđuje uzajamnu isključivost prilikom pristupa skupu bafera i inicijalizovan je na vrednost 1. *Empty* i *Full* semafori broje koliko ima slobodnih i zauzetih bafera. *Empty* je inicijalizovan na vrednost n, dok je *full* inicijalizovan na vrednost 0.

Kod proizvođač i potrošač procesa dati su ispod. Obratite pažnju na simetriju u ova dva slučaja. Možemo kodove interpretirati tako da proizvođač proizvodi popunjene bafere za potrošača, dok potrošač proizvodi prazne bafere za proizvođača.

```
do
{
        . . .
        /* produce an item in next_produced */
        . . .
        wait(empty);
        wait(mutex);
        . . .
        /* add next_produced to the buffer */
        . . .
        signal(mutex);
        signal(full);
} while (true);
```

```
do
{
        wait(full);
        wait(mutex);
        . . .
        /* remove an item from buffer to next_consumed */
        . . .
        signal(mutex);
        signal(empty);
        . . .
        /* consume the item in next_consumed */
        . . .
} while (true);
```

7.2 Problem sa čitačima i pisačima (*readers - writers problem*)

Pretpostavimo da se podaci dele između nekoliko konkurentnih procesa. Neki od ovih procesa samo čitaju podatke, dok drugi mogu da ažuriraju (tj. da čitaju i pišu) podatke. Razlikujemo ove dve vrste procesa tako što prve zovemo *čitači*, a druge *pisači*. Očigledno je da ako dva čitača istovremeno pristupe zajedničkim podacima, neće doći do negativnih efekata. Međutim, ako pisač i neki drugi proces (bilo čitač ili drugi pisač) istovremeno pristupe deljenim podacima, može doći do neželjenih scenarija.

Da bi se osiguralo da ove poteškoće ne nastanu, potrebno je da pisači imaju ekskluzivno pravo pristupa zajedničkim i deljenim podacima dok upisuju podatke. Ovaj problem sinhronizacije naziva se problem čitača/pisača. Otkako je prvobitno ustanovljen, ovaj problem je korišćen za testiranje gotovo svakog novog primitivnog sinhronizma. Problem čitača/pisača ima nekoliko varijacija, a sve uključuju prioritete. Najjednostavniji, koji se navodi kao prvi čitač/pisač problem, nalaže da nijedan čitač ne može čekati osim ukoliko je pisač već dobio dozvolu za korišćenje deljenog objekta. Drugim rečima, nijedan čitač ne treba da čeka da drugi čitači završe samo zato što pisač čeka. Drugi problem čitača-/pisača zahteva da, kada je pisač spreman, pisač izvrši svoje upisivanje što je pre moguće. Drugim rečima, ako pisač čeka na pristup objektu, novi čitači ne mogu početi da čitaju.

Rešenje bilo kojeg od gornjih problema može rezultovati izgladnjivanjem. U prvom slučaju, pisači mogu biti na neodređenom čekanju, dok u drugom slučaju čitači mogu da budu u istoj situaciji. Iz tog razloga su predložene i druge varijante problema. Ovde predstavljamo rešenja oba problema „čitača/pisača“. Najpre dajemo rešenje prvog problema u kojem čitači imaju prednost (prioritet se daje čitačima). U rešavanju problema procesi čitači dele sledeće strukture podataka:

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

Semafori *mutex* i *rw_ mutex* inicijalizovani su na 1, dok je *read_ count* inicijalizovan na 0. Semafor *rw_ mutex* je zajednički kako za čitače, tako i za pisače. *Mutex* semafor koristi se za osiguravanje uzajamne isključivosti kada se ažurira *read_ count* promenljiva. Ova promenljiva prati koliko procesa trenutno čita

objekat. Semafor *rw_ mutex* funkcioniše kao semafor uzajamne isključivosti za pisače. Koristi ga takođe i prvi ili poslednji čitač koji ulazi ili izlazi iz kritične sekcije. Ne koriste ga čitači koji ulaze ili izlaze dok su drugi čitači u kritičnim sekcijama.

Kod za proces pisača prikazan je ispod:

```
void writer()
{
        wait(rw_mutex);
        . . .
        /* writing is performed */
        . . .
        signal(rw_mutex);
}
```

dok je kod čitača:

```
void reader()
{
        wait(mutex);
        read_count++;
        if (read_count == 1)
                wait(rw_mutex);
        signal(mutex);
        . . .
        /* reading is performed */
        . . .
        wait(mutex);
        read_count--;
        if (read_count == 0)
                signal(rw_mutex);
        signal(mutex);
}
```

Obratite pažnju da, ako je pisač u kritičnoj sekciji, a n čitača čeka, tada jedan čitač čeka na *rw_ mutex*, dok n - 1 čitača čeka na *mutex*. Takođe, primetite da, kada pisač pozove *signal*(*rw_ mutex*), možemo nastaviti s izvršavanjem bilo čitača koji čekaju, bilo jednog pisača koji čeka (ovaj izbor vrši planer).

Druga varijanta problema čitač/pisač je definisana tako da se prednost daje pisačima. Strukture koje se koriste u rešenju ovog problema su:

```
semaphore r_mutex = 1;
semaphore w_mutex = 1;
semaphore resource = 1;
semaphore readTry = 1;
int read_count = 0;
int write_count = 0;
```

Kod pisača je sada:

```
void writer()
{
        //rezerviši ulaznu sekciju za pisače, sprečava stanje utrkivanja
        wait(w_mutex);

        write_count++;

        //proveri da li si ti prvi pisač
        if (write_count == 1){
                //onemogući čitače sve dok poslednji pisač ne završi
                wait(readTry);
        }

        //oslobodi ulaznu sekciju za ostale pisače
```

```
        signal(w_mutex);

        //rezerviši deljenu memoriju samo za sebe
        wait(resource);

        . . .
        /* upisuj željene podatke */
        . . .

        //oslobodi deljenu memoriju za ostale pisače
        signal(resource);

        //rezerviši izlaznu sekciju za pisače, sprečava stanje utrkivanja
        wait(w_mutex);

        write_count--;

        //proveri da li si ti poslednji pisač
        if (write_count == 0){
                //omogući čitače da čitaju ukoliko jesi
                signal(readTry);
        }

        //oslobodi ulaznu sekciju za ostale pisače
        signal(w_mutex);
}
```

Dok je sada kod za pisače:

```
void reader()
{
        //indikacija da čitač želi da čita
        wait(readTry);

        //rezerviši ulaznu sekciju za čitače, sprečava stanje utrkivanja
        wait(r_mutex);
        read_count++;
        //proveri da li si ti prvi čitač
        if (read_count == 1){
                //ukoliko si prvi zaključaj deljeni resurs
                wait(resource);
        }

        //oslobodi ulaznu sekciju za ostale čitače
        signal(r_mutex);

        //indikacija da si gotov sa pokušajem pristupa resursu
        signal(readTry);

        . . .
        /* čitaj podatke bezbedno */
        . . .

        //rezerviši izlaznu sekciju za čitače, sprečava stanje utrkivanja
        wait(r_mutex);
        read_count--;

        //proveri da li si ti poslednji čitač
        if (read_count == 0){
                //ukoliko jesi otključaj deljeni resurs
                signal(resource);
        }

        //oslobodi izlaznu sekciju za ostale čitače
        signal(r_mutex);
}
```

Problem čitača/pisača i njegova rešenja su na nekim sistemima generalizova-

ni kroz implementaciju čitač/pisač zaključavanja (reader/writer lock). Da bi se zaključao čitač/pisač katanac potrebno je navesti način zaključavanja: pristup čitanja ili pisanja. Kada proces želi samo čitanje deljenih podataka, on zahteva da se čitač/pisač zaključa u režimu čitanja. Proces koji želi da izmeni zajedničke podatke mora da zahteva zaključavanje u režimu pisanja. Višestrukim procesima je dozvoljeno da istovremeno zaključaju katanac u režimu čitanja, ali samo jedan proces može zaključati katanac za pisanje, jer je za pisače potreban ekskluzivan pristup. Čitač/pisač zaključavanja su najviše korisna u sledećim situacijama:

- U aplikacijama gde je lako identifikovati koji procesi samo čitaju deljene podatke, a koji procesi samo pišu deljene podatke
- U aplikacijama koje imaju više čitača nego pisača. To je zato što je čitač/pisač zaključavanje skuplje, za njega je uglavnom potrebno više vremena (ili više resursa) nego u slučaju korišćenja semafora ili muteksa. Sa druge strane, povećana konkurentnost čitanja omogućena kroz više konkurentnih čitača nadoknađuje ove dodatne troškove (*overhead*) koji nastaju usled uspostavljanja čitač/pisač zaključavanja.