Predavanje 4

Sinhronizacija i deljenje resursa

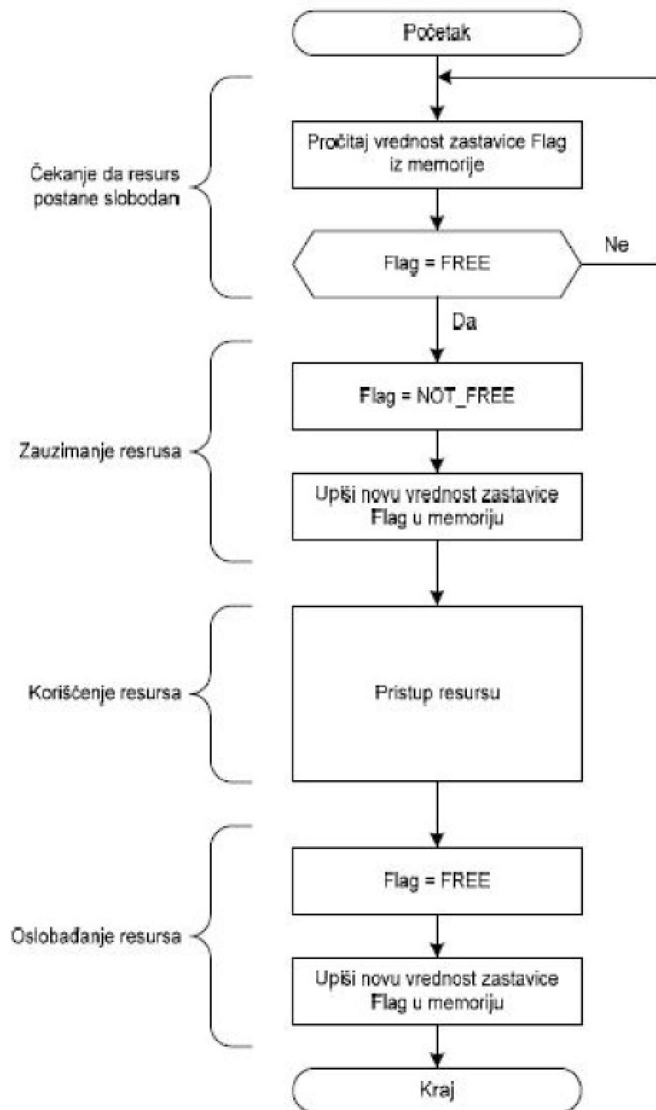
Kod jednoprocesorskog multitasking RTOS sistema, hardverske komponente, kao što su CPU, memorija, U/I uređaji (disk, displej) predstavljaju zajedničke resurse svih zadataka. U slučaju CPU-a, problem deljivosti rešen je uvođenjem dispečera koji obezbeđuje kontrolisano, uzajamno isključivo, korišćenje CPU-a od strane spremnih zadataka (zadataka u stanju Ready). Slično, u slučaju ostalih hardverskih komponenti, moraju biti predviđeni posebni mehanizmi koji će obezbediti korektnu deljivost. U deljive resurse se mogu svrstati i strukture podataka, kao što su promenljive, baferi, nizovi, kojima pristupa više od jednog zadatka.

Rešenje ovog problema se sastoji u implementaciji tzv. uzajamne isključivosti (mutual exclusion), tj. mehanizma koji će obezbediti da deljivom resursu u bilo kom trenutku može da pristupa najviše jedan zadatak. Međutim, iako jednostavan u svoj definiciji, impementacija mehanizma uzajamne isključivosti odnosi se na mnoge, često suptilne, probleme.

4.1 Sinhronizacija i deljenje resursa bez korišćenih RTOS mehanizama

4.1.2 Algoritam signalne zastavice (Varijanta 1)

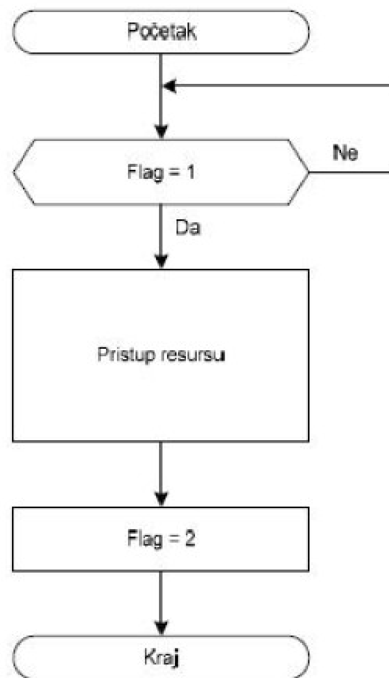
Deljivom resursu pridružena je zastavica (flag). To je binarna promenljiva, smeštena u operativnoj memoriji, čija vrednost (FREE, NOT FREE) ukazuje na tekući status resursa (slobodan, zauzet). Zadatak koji namerava da pristupi resursu, najpre ispituje zastavicu. Ako je stanje zastavice FREE, resurs je slobodan; zadatak najpre postavlja zastavicu u stanje NOT-FREE; pristupa resursu, i nakon završenog obraćanja resursu, postavlja zastavicu u stanje FREE. U suprotnom, ako je stanje zastavice NOT FREE, zadatak se zaustavlja i čeka da stanje zastavice postane FREE. Na slici 1 prikazan je dijagram toka koji opisuje mehanizam signalne zastavice. Da bi bila ispitana, zastavica mora najpre biti prenesena iz operativne memorije u interni registar procesora. Deo programa koji obezbeđuje pristup deljivom resursu se zove „kritična sekcija” (critical section). Opisani mehanizam kontrole pristupa deljivom resursu je jednostavan i lak za implementaciju, međutim nije u potpunosti korektan, jer pod izvesnim uslovima, može se dogoditi da dva ili više zadatka dobiju pravo pristupa deljivom resursu.
Kritičan vremenski interval je onaj između trenutka kada zadatak (recimo Z1) ustanovi da je zastavica u stanju FREE i trenutka kada postavi zastavicu stanje NOT-FREE. Naime, ako u tom vremenskom intervalu, zadatak Z1 bude istisnut nekim drugim zadatkom Z2 koji takođe zahteva pristup istom deljivom resursu i taj drugi zadatak dobija indikaciju da je resurs slobodan i nesmetano ulazi u kritičnu sekciju. Ukoliko za vreme dok je Z2 u kritičnoj sekciji dođe do ponovne promene konteksta i Z1 nastavi sa izvršenjem javlja se konflikt. Kaže se da algoritam za kontrolu pristupa deljivom resursu nije bezbedan.

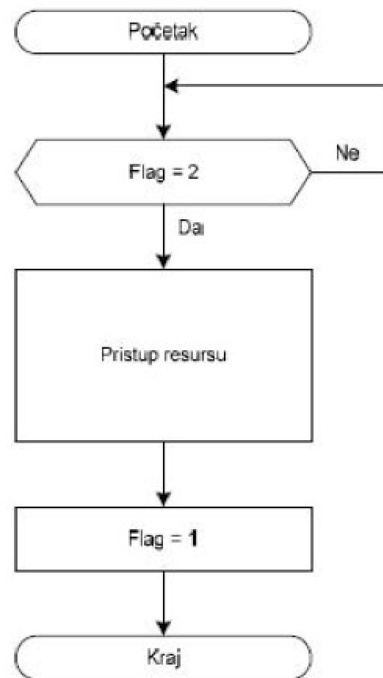


Slika 1: Algoritam signalne zastavice

4.1.2 Bezbedni algoritam signalne zastavice (Varijanta 2)

Kod ove varijante, zastavica ne ukazuje na tekući status deljivog resursa, već na zadatak koji kao sledeći ima pravo korišćenja resursa. Zastavica zadržava tekuću vrednost do trenutka kada zadatak, označen kao sledeći, ne oslobodi resurs. Algoritam je opisan dijagramom toka na slici 2. Pretpostavimo da u sistemu postoje dva zadatka Z1 i Z2 i da je tekuća vrednost zastavice F=1, što znači da pravo pristupa resursu ima zadatak Z1. Neka su oba zadatka započela proceduru pribavljanja prava pristupa deljivom resursu. S obzirom na stanje zastavice, prednost dobija zadatak Z1, koji ulazi u kritičnu sekciju, dok zadatak Z2 ostaje da čeka, neprekidno ispitujući stanje zastavice. Kada zadatak Z1 završi korišćenje resurs, on postavlja zastavicu u stanje F=2, što je signal zadatku Z2 da može da uđe u kritičnu sekciju.
Očigledan nedostatak ovog algoritma je u činjenici da forsira strogo naizemnični pristup deljivom resursu od strane dva zadatka. Naime, zadatak Z1 (ili Z2) ne može dva puta uzastopno da dobije pravo korišćenja resursa. Ovaj nedostatak postaje izraženiji kada u sistemu postoji više od dva zadatka.

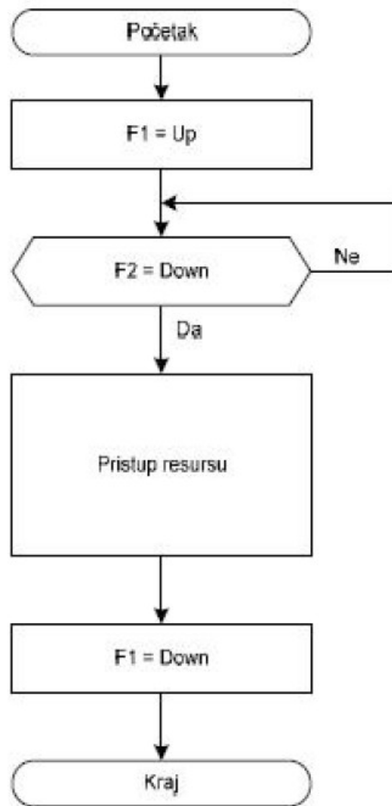


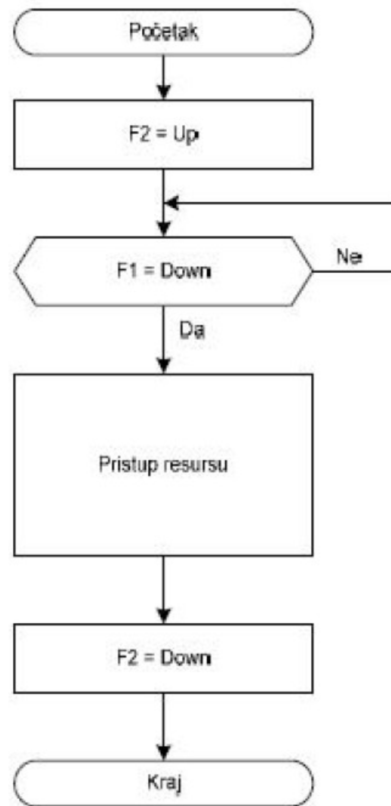


Slika 2: Bezbedni algoritam signalne zastavice

4.1.3 Algoritam sa dve signalne zastavice (Varijanta 3)

Svaki zadatak poseduje svoju zastavicu koju koristi da bi iskazao zahtev (nameru) za korišćenjem deljivog resursa. „Podignuta” zastavica (F=„Up”) ukazuje da takav zahtev postoji, dok spuštena zastavica (F=„Down”) ukazuje da zahtev ne postoji. Algoritam radi na sledeći način (sl 6). Pretpostavimo da je resurs slobodan (obe zastavice su spuštene, F1=F2=„Down”) i da zadatak Z1 „želi” da pristupi resursu. Zadatak Z1 podiže svoju zastavicu (F1=„Up”); ispituje stanje zastavice F2, i pošto je ona spuštena, ulazi u kritičnu sekciju. Neka, u međuvremenu, i zadatak Z2 dobije potrebu korišćenja deljivog resursa. Z2 podiže svoju zastavicu (F2=„Up”); ispituje zastavicu F1 i pošto je ona podignuta, Z2 se zaustavlja i čeka da se zastavica zadatka Z1 spusti. Zadatak Z1, nakon napuštanja kritične sekcije spušta svoju zastavicu (F1=’Down’), što predstavlja dozvolu zadatku Z2 da uđe u kritiču sekciju.





Slika 3: Algoritam sa dve signalne zastavice

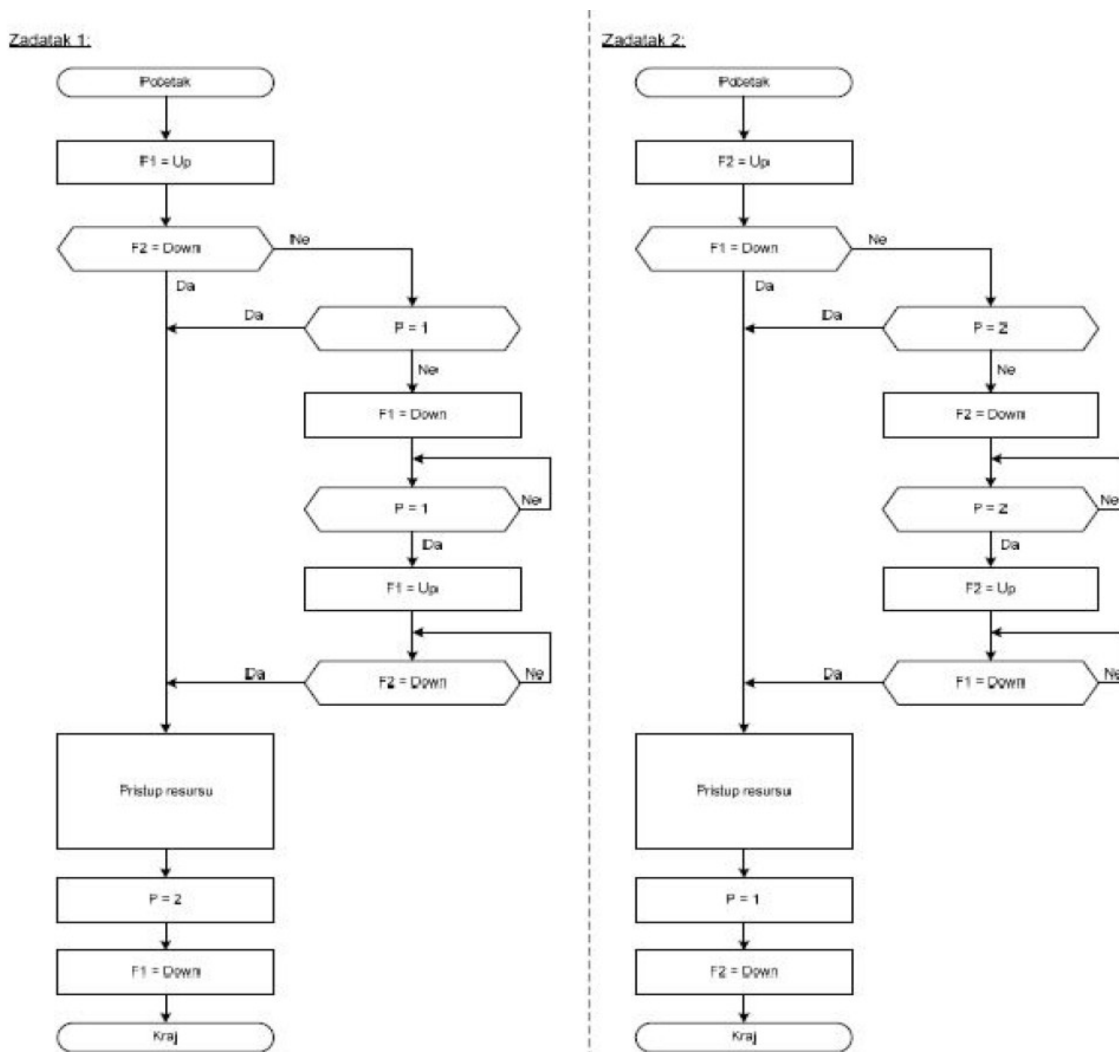
Međutim, i ovo rešenje poseduje jedan ozbiljan problem, koji je poznat pod nazivom deadlock (mrtva petlja). Pretpostavimo da je u prethodno opisanom scenariju do promene konteksta (istiskivanja zadatka Z1 zadatkom Z2) došlo u trenutku kada je zadatak Z1 podigao svoju zastavicu, a pre nego sto je stigao da ispita zastavicu zadatka Z2. Zadatak Z2, koji se sada izvršava, podiže svoju zastavicu i pošto je zastavica zadatka Z1 podignuta, zaustavlja se i ostaje u stanju čekanja. Počev od ovog trenutka, obe zastavice su podignute i oba zadatka čekaju da se spusti zastavica onog drugog zadatka kako bi nastavili sa radom. Drugim rečima, resurs je trajno zauzet, a sistem „ukočen”, tj. došlo je do deadlock-a. U opštem slučaju, deadlock se javlja u sitacijama kada je u sistemu od k zadataka uspostavljena ključna zavisnost, tako što Z1, da bi nastavio, čeka Z2, Z2 čeka Z3 i tako redom do Zk koji da bi nastavio sa radom čeka Z1.

Jedan od načina da se razreši gornja situacija sastoji se u tome da svaki od zadataka, periodično, dok čeka na oslobađanje resursa, za neko kratko vreme, spusti, a zatim ponovo podigne svoju zastavicu, dajući tako prvenstvo i šansu onom drugom zadatku da uđe u kritičnu sekciju. Međutim, rešavanje problema deadlock-a na ovaj način dovodi do jednog drugog problema koji se zove „gladovanje” (starvation). Može se desiti da oba zadatke rade „istom brzinom” i da stalno u sinhronizmu, obavljaju iste aktivnosti: „spusti zastavicu” -> „čekaj neko vreme” -> „podigni zastavicu” -> „ispituj zastavicu drugog zadatka”. Pod ovakvim uslovima, zadaci će uvek u isto vreme da daju pravo prvenstva drugom zadatku, tako da će u vremenu dok ispituju zastavicu drugog zadatka ona uvek biti podignuta. Znači, postoje vremenski intervali kada je resurs dostupan (dok su obe zastavice spuštene), ali ni jedan od zadataka ne uspeva da dobije pravo korišćenja resursa. Treba napomenuti, da je u realnim uslovima, verovatnoća da dva zadatka beskonačno dugo rade u

strogom sinhronizmu, veoma mala. Ipak, nepredvidljivo dugo vreme čekanja na razrešenje deadlock-a nije poželjno kod hard RTOS.

4.1.4 Dekerov algoritam (Varijanta 4)

Korektno rešenje problema kontrole pristupa deljivom resursu postiže se uvođenjem još jedne, treće, zastavice, tzv. zastavice prioriteta, koja se koristi samo u slučajevima kada postoje istovremeni zahtevi za korišćenjem deljivog resursa (slika 4). Stanje zastavice prioriteta, slično kao kod Varijante 2, ukazuje na zadatak kome se u slučaju istovremenog zahteva daje prioritet za pristup resursu. U ostalim slučajevima, algoirtam radi na isti načina kao Varijanta 3.



Slika 4: Dekerov algoritam

Kao što je to naglašeno kod opisa Varijante 3, kada oba zadatka istovremeno ispostave zahtev za korišćenjem deljivog resursa, obe zastavice, pridružene zadacima Z1 i Z2, prelaze u stanje „Up”, i zbog toga oba zadatka ostaju stanju u kome u nedogled ispituju stanje zastavice onog drugog zadatka. Modifikacija, u odnosu na varijantu 3, se sastoji u tome što pod ovakvim uslovima svaki zadatak dodatno ispituje zastavicu prioriteta. Pretpostavimo da je stanje zastavice prioriteta P=2. To znači da se prioritet daje zadatku Z2. Shodno tome, zadatak Z2 ulazi u kritičnu sekciju, a zadatak

Z1 spušta svoju zastavicu, zaustavlja se i čeka da se stanje zastavice prioriteta promeni na P=1. Kada zadatak 2 završi sa korišćenjem resursa, on najpre menja stanje zastavice prioriteta na P=1. Zadatak1, prilikom prve naredne provere zastavice prioriteta, zaključuje da je ona postavljena na vrednost na koju čeka. Kao posledica toga, zadatak Z1 najpre podiže svoju zastavicu, a zatim ispituje zastavicu zadatka Z2 i kako je ona još uvek podignuta, Z1 ulazi u stanje čekanja, stalno ispitujući sada zastavicu Z2. Konačno, zadatak Z2 spušta svoju zastavicu, što je znak zadatku Z1 da može da uđe u kritičnu sekciju.

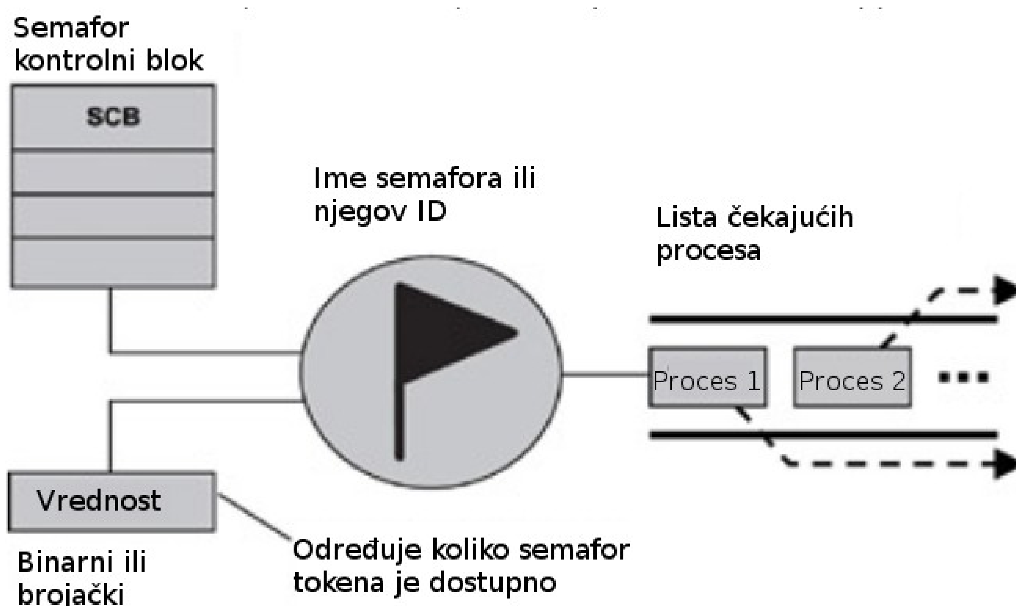
4.2 Semafori

Kao što je naglašeno ranije, za više konkurentnih niti izvršavanja u okviru aplikacije ključno je da imaju način da se sinhronizuju i koordiniraju ekskluzivnim pravom pristupanja deljenim resursima. Kako bi omogućio ovakav mehanizam RTOS kernel implementira semafore i prateće servise i infrastrukturu za rad sa semaforima. Ovde će biti više reči o:

1. definisanju semafora
2. tipičnim operacijama sa semaforima
3. tipičnom upotrebom semafora

4.2.1 Definisanje semafora

Semafor (često se naziva i semafor-token) je kernel objekat koji jedan ili više niti izvršavanja mogu zauzeti ili osloboditi u cilju sinhronizacije.
Kada se semafor kreira, kernel mu dodeljuje odgovarajući Semafor Kontrolni Blok (Semaphore Control Block - SCB), jedinstveni ID, vrednost (binarni semafor ili brojač) i listu sa čekajućim procesima (process-waiting list) kao što je prikazano na slici 5.



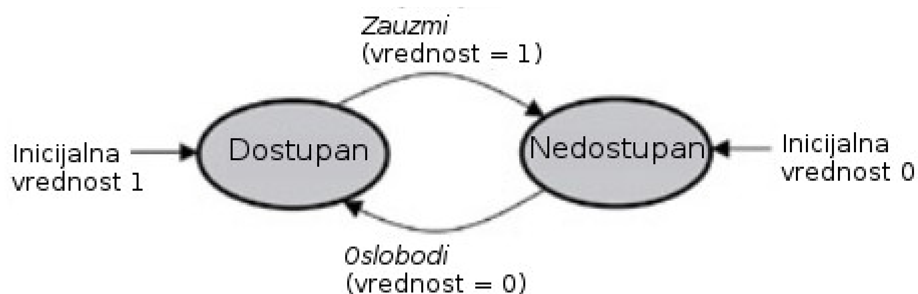
Slika 5: Semafor i njegova struktura

Semafor je kao ključ koji dozvoljava odgovarajućim procesima da vrše određene operacije ili jednostavno pristupaju određenim resursima. Ukoliko neki proces može da zauzme semafor, on može i da izvrši željenu operaciju ili pristupi resursu. Jedan semafor može biti zauzet konačan broj puta (slično kao duplikat ključa koji može da se iskoristi u istu svrhu,a li ne više od onoliko koliko duplikata uopšte postoji kod stanodavca). Slično, kada je dati semafor zauzet maksimalan broj puta, nije više moguće zauzeti semafor, sve dok neko ne oslobodi semafor. Kernel prati broj zauzeća semafora, kao i broj oslobađanja semafora, ažuriranjem vrednosti brojača koji je prilikom kreiranja semafora inicijalizovan na vrednost semafora. Kada neki proces zauzme semafor, brojač je umanjen za jedan, dok se u slučaju oslobađanja semafora on uvećava za jedan. Ukoliko brojač dođe do 0, semafor nema više preostalih mogućnost korišćenja. Kao rezultat, sledeći pokušaj zauzimanja semafora od strane nekog procesa će biti neuspešan i rezultovaće odlaskom procesa u blokirajuće stanje u kome će čekati na semafor da bude oslobođen i postane dostupan.

Lista sa čekajućim procesima prati sve procese koji su blokirani i čekaju na semafor. Ovi blokirani procesi se čuvaju u listi koja je ili First In First Out (FIFO) ili je lista u kome je prvi proces sa najvišim prioritetom. Kada semafor postane dostupan, kernel dopušta prvom procesu iz reda čekanja da zauzme semafor. Kernel tada premešta ovaj odblokirani proces ili direktno u aktivno stanje, ukoliko je to proces sa nejvećim prioritetom, ili u ready stanje, dok ne postane proces sa najvećim prioritetom i ne krene sa izvršavanjem.Naravno, sama implementacija ove liste čekajućih procesa se razlikuje od kernela do kernela, ali je uvek u pitanju isti princip, opisan iznad. Kernel najčešće podržava više tipova semafora među kojima su binarni, brojački i muteksi (mutual-exclusion).

A. Binarni semafori

Binaran semafor može imati samo dve vrednosti, 0 i 1. Kada je vrednost binarnog semafora 0, semafor se smatra nedostupnim, a kada je vrednost semafora 1 smatra se da je dostupan. Prilikom kreiranja binarnog semafora on može biti inicijalizovan kao dostupan ili nedostupan. Dijagram stanja ovakvog semafora je prikazan na slici 6.

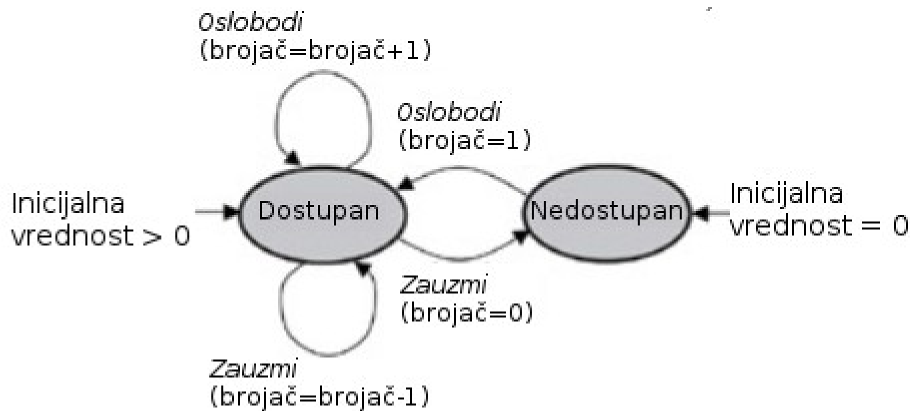


Slika 6: Binarni semafor i njegova mašina stanja

Binarni semafori se tretiraju kao globalni resursi, što znai da se dele od strane svih procesa koji ih koriste. Pošto je semafor globalan resurs, svakom procesu je dozvoljeno da ga oslobodi, čak i ukoliko se radi o procesu koji ga nije zauzeo.

B. Brojački semafori

Brojački semafori koriste brojač kako bi omogućili zauzimanje i oslobađanje semafora više puta. Prilikom kreiranja brojačkih semafora, vrednosti semafora se dodeljuje brojač koji označava koliko se puta u samom startu može istovremeno zauzeti semafor (kaže se i koliko semafor-tokena je na raspolaganju). Ukoliko je ovaj brojač na početku inicijalizovan sa 0, znači da se ne može zauzeti od strane procesa. Ukoliko je brojač veći od 0, to znači da ga je moguće zauzimati u paraleli onoliko puta kolika je vrednost brojača (bez oslobađanja), kao što je prikazano na slici 7.



Slika 7: Brojački semafor i mašina stanja

Jedan ili više procesa može da zauzima semafor-tokene brojačkog semafora sve dok njih više ne bude na raspolaganju. U trenutku kada se to desi (brojač=0) semafor prelazi is stanja raspoloživ u stanje neraspoloživ. Kako bi se vratio nazad, barem jedan semafor-token mora da bude oslobođen.
Isto kao i binarni semafori, brojački semafori su takođe globalni resursi, koji se dele između svih procesa koji ih trebaju. Ova osobina omogućava svakom procesu da oslobodi semafor i svaka ovakva operacija povećava vrednost brojača za jedan, čak i ako je ovakvo oslobađanje došlo od strane procesa koji nije inicijalno ni zauzeo semafor.
U nekim implementacijama brojačkih semafora, brojač je ograničen. Ograničen brojač je brojač u kome je inicijalna vrednost brojača, postavljena prilikom kreiranja semafora, zapravo i maksimalna vrednost brojača (gornja granica). Neograničeni brojački semafori dozvoljavaju da vrednost brojača raste preko prvobitno inicijalizovane vrednosti, i u njima je maksimalna vrednost najčešće određena samim tipom brojača (unsigned int ili unsigned long).

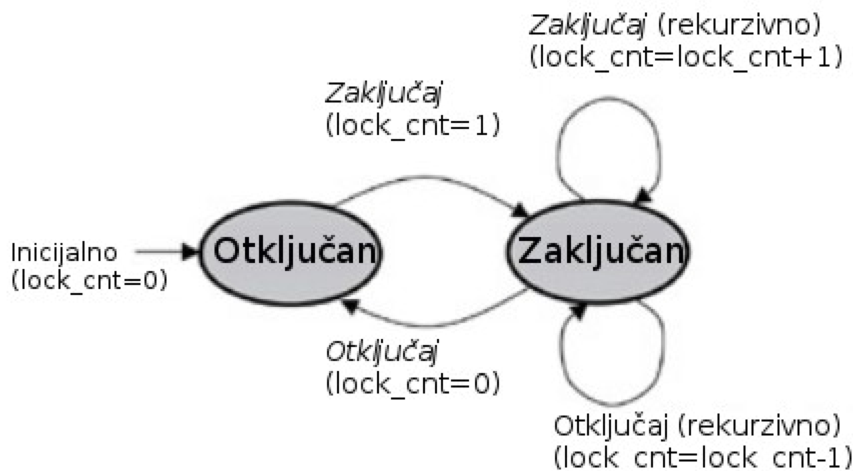
C. Muteksi

Muteks (mutual-exclusion) semafor je specijalan tip binarnog semafora, koji omogućava posedovanje, rekurzivni pristup, bezbednost prilikom brisanja procesa i eventualno dodatne protokole koji su neophodni za rešavanje problema nastalih usled uzajamnog ekskluzivnog deljenja resursa (npr. priority inversion avoidance protocol).
Nasuprot stanjima dostupan i nedostupan koje smo imali kod binarnih i brojačkih semafora, stanja muteksa su zaključan i otključan (1 i 0). Muteks je inicijalno kreiran u otključanom stanju, u kome može da bude zauzet od stane nekog procesa. Nakon što se zauzme, muteks prelazi u zaključano stanje i suprotno, kada proces oslobodi muteks, on prelazi u otključano stanje. Neki kerneli čak koriste termine zaključavanje i otključavanje (lock and unlock) umesto termina zauzmi i oslobodi (acquire i release).

U zavisnosti od implementacije, muteks može da podržava i dodatne funkcije koje se ne mogu pronaći kod binarnih ili brojačkih semafora.

Posedovanje Muteksa

Posedovanje muteksa je ostvareno kada proces prvi put zaključa muteks njegovim zauzimanjem. Suprotno, proces gubi posed muteksa nakon što ga otključa njegovim oslobađanjem. Kada proces poseduje muteks, nijedan drugi proces ne može da ga otključa ni zaključa. Podsećamo, kod binarnih semafora koji su svakako slični muteksima, ovo ne važi jer bilo koji proces u bilo kojem trenutku može da oslobodi semafor, čak i ukoliko ga nije prethodno zauzeo.



Slika 8: Muteks i mašina stanja

Rekurzivno zaključavanje

Većina implementacija muteksa takođe omogućava rekurzivno zaključavanje, koje dozvoljava procesu koji poseduje muteks da ga više puta zauzme dok je u zaključanom stanju. U zavisnosti od implementacije ovaj mehanizam je ili automatski ugrađen u muteks, ili je neophodno dozvoliti ga eksplicitno prilikom kreiranja muteksa. Muteksi koji omogućavaju ovakav mehanizam se nazivaju rekurzivni muteksi.
Ovakav muteks je najkorisniji u situacijama kada proces, koji zahteva ekskluzivno pravo korišćenja deljenog resursa, poziva jednu ili više rutina koje takođe zahtevaju pristup istom resursu. Rekurzivni muteks u takvom scenariju omogućava ugnježdene pokušaje da se zaključa muteks i to će biti urađeno uspešno, dok bi u suprotnom ovakav pokušaj doveo do mrtve petlje (deadlock), koja nastaje kada dva ili više procesa pređu u blokirano stanje čekajući na resurse koje su uzajamno zaključali. O mrtvim petljama i algoritmima za izbegavanje istih će biti više reči kasnije.
Kao što se vidi na slici 8, kada rekurzivni muteks se prvi put zaključa, kernel registruje proces koji ga je zaključao kao vlasnika muteksa. U narednim pokušajima, kernel koristi interno ažuriranje broja zaključavanja za svaki muteks, kako bi pratio i evidentirao broj puta koliko je proces (trenutni vlasnik muteksa) rekurzivno zauzeo proces. Da bi se korektno otključao muteks, muteks mora biti oslobođen isti broj puta. U primeru sa slike 4, lock count prati ne samo dva stanja muteksa (0 za otključan i 1 za zaključan) već i broj rekurzivnih zaključavanja (lock count veće od 1). U drugim implementacijama, moguće je ovo realizovati korišćenjem dva različita brojača: binarni koji prati samo stanje muteksa, i poseban brojač koji prati koliko je puta zauzet muteks dok je u zaključanom stanju od strane vlasnika muteksa.
Svakako ovo ne treba mešati sa brojačkim muteksima. Osim različite prirode, brojač kod muteksa je uvek neograničen što omogućava proizvoljan broj rekurzivnih zauzimanja muteksa.

Bezbedno brisanje procesa

Neke muteks implementacije takođe imaju ugrađen mehanizam zaštite procesa prilikom brisanja. Prevremeno brisanje procesa je izbegnuto korišćenjem zabrane/dozvole brisanja procesa prilikom zaključavanja/otključavanja muteksa. Dozvoljavanjem ove mogućnosti, obezbeđeno je da se proces ne može obrisati sve dok poseduje muteks. Tipično, ovaj način protekcije je aktiviran podešavanjem prikladnih inicijalizacionih opcija prilikom kreiranja muteksa.

Izbegavanje inverzije prioriteta

Inverzija prioriteta se pojavljuju uglavnom u loše dizajniranim real-time embedded aplikacijama. Javlja se u slučajevima kada se proces višeg prioriteta blokira čekajući na resurs koji je zauzet od strane procesa niskog prioriteta, koji je prekinut u izvršavanju od strane procesa srednjeg prioriteta. Efekat ovakve situacije je da se prioritet procesa sa visokim prioritetom invertovao u nizak prioritet procesa koji ga drži blokiranim.

Dozvoljavanjem specifičnih protokola koji su tipično ugrađeni u mutekse, omogućava se izbegavanje ovakve inverzije prioriteta. Dva tipična protokola koji se koriste u ovu svrhu su:

- *Protokol nasleđivanja prioriteta*
  obezbeđuje da se prioritet procesa nižeg prioriteta povećava do nivoa procesa višeg prioriteta koji zahteva pristup tom istom resursu u trenutku kada se dešava inverzija prioriteta. Izmenjeni prioritet procesa se tada vraća na svoju originalnu vrednost nakon što proces oslobodi kritičan muteks.
- *Maksimalno povećavanje prioriteta* (ceiling priority protocol)
  obezbeđuje da se prioritet procesa koji je zauzeo muteks automatski povećava to najvišeg prioriteta svih mogućih procesa koji mogu zahtevati pristup tom istom muteksu, i to u trenutku kada je muteks zauzet, sve do trenutka kada se muteks oslobodi. Kada se muteks oslobodi, prioritet procesa koji je do tog trenutka bio vlasnik muteksa, se vraća na originalnu vrednost.

4.2.2 Tipične operacije sa semaforima

Tipične operacije koje programere koriste u radu sa semaforima su:
-kreiranje i brisanje semafora
-zauzimanje i oslobađanje semafora
-brisanje liste procesa koji čekaju na semafor
-dobijanje informacija o semaforima

A. Kreiranje i brisanje semafora

Operacija	Opis
Create	Kreira semafor
Delete	Briše semafor

Ipak, nekoliko stvari mora biti razmatrano prilikom kreiranja i brisanja semafora. Ukoliko kernel podržava različite tipove semafora, različiti pozivi mogu da se koriste za kreiranje binarnih, brojačkih semafora i muteksa:
- za binarne je potrebno definisati inicijalno stanje semafora, kao i listu procesa koji čekaju na njega
- za brojačke takođe inicijalno stanje semafora (brojač) i listu procesa koji ga koriste

- za mutekse listu procesa koji ga koriste i dozvoljavanje dodatnih funkcionalnosti vezanih za mutekse, koje su eventualno podržane.
Semafori se mogu obrisati od strane procesa korišćenjem njegovog ID-ja i pozivom sistemskom poziva za brisanje semafora. Naravno, treba imati u vidu da brisanje semafora nije ista operacija kao i oslobađanje semafora. Kada se semafor obriše, svi procesi koji su blokirani i čekaju na oslobađanje semafora se odblokiraju od strane kernel-a, i premeštaju se ili u ready stanje ili odmah u aktivno stanje (running), u zavisnosti od njihovog prioriteta.
Bilo koji proces, koji pokuša da zauzme obrisani semafor, dobiće grešku prilikom sistemskog poziva, jer semafor više ne postoji. Dodatno, nikako ne treba brisati semafor dok se koristi (zauzet je). Ovakva akcija može izazvati postojanje ne-validnih podataka, ili ozbiljnih problema ukoliko taj semafor štiti deljeni resurs ili neku kritičnu sekciju koda.

B. Zauzimanje i oslobađanje semafora

Operacija	Opis
Acquire	Zauzimanje semafora
Release	Oslobađanje semafora

Same akcije zauzimanja i oslobađanja semafora mogu imati različite nazive od kernela do kernela (npr. take-give, sm_p i sm_v, pend i post, lock i unlock,..). Nezavisno od imena, te aktivnosti se odnose na iste akcije. Tipično, procesi zahtevaru zauzimanje semafora na sledeće načine:
- čekanje zauvek (wait forever)
    Proces ostaje u blokiranom stanju dok se ne oslobodi semafor kako bi ga on mogao zauzeti;
- čekanje sa time-out-om (wait with time-out)
    Proces ostaje u blokiranom stanju dok ne zauzme semafor ili dok ne istekne predefinisani period čekanja na semafor. Ukoliko istekne time-out, proces se izbacuje iz liste procesa koji čekaju na semafor i prebacuje se ili u ready stanje ili u aktivno stanje.
- bez čekanja (do not wait)
    Proces zahteva semafor i ukoliko ne može da ga zauzme, proces ne odlazi u blokirajuće stanje.

Važno je spomenuti da ISR (prekidna rutina) takođe može osloboditi binarne i brojačke semafore. Većina kernela ne dozvoljava prekidnim rutinama da zaključavaju i otključavaju mutekse, obzirom da takva akcija uglavnom nema previše smisla. Slično, prilično besmislena bi bila i akcija zauzimanja binarnih i brojačkih semafora iz prekidne rutine.

C. Čišćenje liste procesa koji čekaju na semafor

Operacija	Opis
Flush	Odblokiranje svih procesa koji čekaju na semafor

Ova operacija je korisna kada je potrebno poslati broad-cast poruku grupi signala. Na primer, programer koristi više procesa koji treba da završe određene aktivnosti nakon kojih treba da sačekaju na poslednjeg među njima. Ovo će uraditi tako što nakon okončanja svog posla pokušaju da zauzmu semafor koji im nije dostupan, nakon čega odlaze u blokirajuće stanje. Nakon što poslednji proces završi sa svojim aktivnostima, on poziva operaciju brisanja liste procesa koji čekaju na zajednički semafor. Ova operacija oslobađa sve procese koji čekaju na semafor. Ovakav

mehanizam se često naziva i “*thread rendezvous*”, kada je tokom izvršavanja više niti potrebno da se sve one “sastanu” u određenom stanju kako bi se izvršila sinhronizacija njihovih aktivnosti.

D. Dobijanje informacija o semaforu

Operacija	Opis
Show info	Prikaži informacije o semaforu (generalno)
Show blocked processes	Zatraži listu ID-jeva svih procesa koji su u datom trenutku blokirani čekajući na semafor

4.2.3 Tipična upotreba semafora

Semafori su korisni kako za sinhronizaciju izvršavanja većeg broja procesa, tako i za manipulisanje pristupom deljenim resursima. Primeri koji će biti navedeni u nastavku prikazuju razne načine korišćenja semafora. Naravno, u cilju pojednostavljivanja, ovde nisu prikazane sve mogućnosti korišćenja semafora.

A. Čekaj i signaliziraj (wait-and-signal) sinhronizacija

Dva procesa mogu da komuniciraju u cilju sinhronizacije bez ikakve razmene podataka. Na primer binarni semafor može da se koristi od strane dva procesa kako bi se koordinirao transfer kontrole izvršavanja kao što je prikazano na slici ispod.

Slika 9: Binarni semafor u čekaj i signaliziraj scenariju

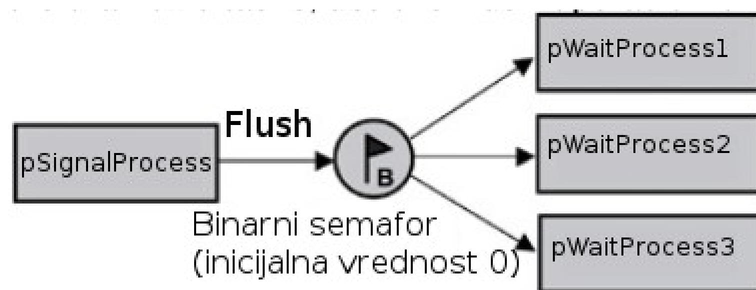
U ovakvoj situaciji, binarni semafor je inicijalno nedostupan (vrednost mu je 0). pWaitProcess ima viši prioritet i prvi se izvršava. Tokom izvršavanja procesa on zahteva semafor i biva blokiran obzirom na to da je semafor nedostupan. Ovo daje priliku procesu pSignalProcess nižeg prioriteta da krene sa izvršavanjem. U nekom trenutku, pSignalProcess oslobađa binarni semafor i na taj način odblokira pWaitProcess. Pseudo kod koji opisuje ovakav scenario je prikazan ispod.

```
pWaitProcess ( )
{
	:
	Acquire binary semaphore token
	:
}

pSignalProcess ( )
{
	:
	Release binary semaphore token
	:
}
```

Obzirom na činjenicu da je pWaitProcess višeg prioriteta u odnosu na pSignalProcess, čim postane oslobađen pWaitProcess prekida izvršavanje pSignalProcess-a i počinje da se izvršava.

B. Čekaj i signaliziraj za više procesa (multiple-process wait-and-signal) sinhronizacija

U situacijama kada je potrebno sinhronizovati više od dva procesa, koristi se flush operacija na listi procesa koji čekaju binarni semafor, kao što je prikazano na slici 10. Kao i u prethodnom primeru, binarni semafor je na početku nedostupan. procesi višeg prioriteta pWaitProcess1, pWaitProcess2 i pWaitProcess3 se izvršavaju i vrše neko procesiranje podataka, kada svaki od njih završi, pokušava da zauzme nedostupan semafor i biva blokiran. Nakon što su sva tri procesa visokog prioriteta blokirana, kontrola se prepušta pSignalProcess-u, koji će kada za to dođe vreme pozvati Flush komandu na semaforu, i istovremeno odblokirati sva tri procesa visokog prioriteta.



Slika 10: Čekaj i signaliziraj za više procesa scenario

```
pWaitProcess ()
{
	:
	Do some processing specific to process
	Acquire binary semaphore token
	:
}
pSignalProcess ()
{
	:
	Do some processing
	Flush binary semaphore's process-waiting list
	:
}
```

Obzirom na činjenicu da su pWaitProcess-i višeg prioriteta, odmah po oslobađanju semafora pWaitProcess sa najvišim prioritetom će nastaviti da se izvršava. Treba obratiti pažnju da je vrednost binarnog semafora, nakon što se na njemu izvrši flush operacija, nedefinisan i da zavisi od implementacije.

C. Sinhronizacija praćenjem kredita (credit-tracking)

U nekim slučajevima frekvencija kojom se izvršava signalni proces je veća u poređenju sa procesom koji čeka na signale. U takvom scenariju je neophodno prebrojavati pojavljivanje signala.

Brojački semafor omogućava upravo ovakvu funkcionalnost. Korišćenjem brojačkog semafora, signalni proces može da nastavi sa izvršavanjem i inkrementira stanje brojača u skladu sa tim. Sa druge strane, proces koji čeka, kada se odblokira, nastavlja izvršavanje nezavisno, kao što je prikazano na slici 11.

Još jednom, inicijalno stanje brojačkog semafora je 0, što ga čini nedostupnim na početku. Proces, nižeg prioriteta pWaitProcess pokušava da zauzme semafor ali je blokiran jer je on nedostupan, sve dok pSignalProcess ne oslobodi semafor. Čak i tada, pWaitProcess ne može da nastavi sa izvršavanjem, sve dok pSignalProcess višeg prioriteta ne oslobodi CPU nakon blokirajućeg poziva (ili jednostavno odlažući svoje izvršavanje na određeno vreme).

Slika 11: Praćenje kredita pomoću semafora

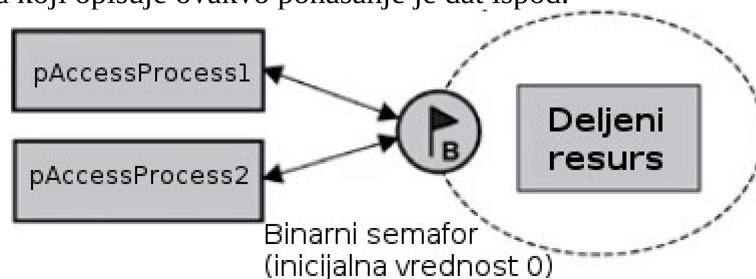
```
pWaitProcess ()
{
       :
       Acquire counting semaphore token
       :
}

pSignalProcess ()
{
       :
       Release counting semaphore token
       :
}
```

Pošto je pSignalProcess višeg prioriteta, i izvršava se učestanošću nezavisnom od procesa nižeg prioriteta, on će povećavati brojač semafora više puta pre nego što pWaitProcess nastavi sa izvršavanjem nakon svog prvog zauzimanja semafora. Kao rezultat, brojački semafor dozvoljava određeni "kredit" koji zapravo predstavlja broj puta koliko pWaitProcess može da se izvrši pre nego što semafor opet postane nedostupan. Ukoliko se vremenom frekvencija izvršavanja pSignalProcess-a smanji, pWaitProcess ima mogućnost da ga "dostigne" i, vremenom, smanji vrednost brojačkog semafora na 0, nakon čega će opet biti blokiran.
Ovakav pristup je koristan kada pSignalProcess oslobađa semafore učestalo, omogućavajući pWaitProcess da ga dostigne nakon određenog vremena.
Korišćenje ovakvog mehanizma zajedno sa prekidnom rutinom (ISR) koja igra ulogu signalnog procesa, svakako može biti korisno, takođe. Prekidi sami po sebi imaju viši prioritet u odnosu na procese. Dakle, prekidna rutina sa visokim prioritetom se poziva u skladu sa hardverskim događajima i tipično ostavlja popriličan "posao" procesu nižeg prioriteta da ga odradi tokom vremena.

D. Sinhronizacija pristupa jednostruko deljenom resursu

Jedan od najčešće korišćenih primena semafora je u slučaju kontrole uzajamno ekskluzivnog prava korišćenja deljenih resursa. Deljeni resurs može biti memorijska lokacija, struktura podataka ili jednostavno I/O uređaj-praktično bilo šta što se deli između dva ili više konkurentnih niti izvršavanja. Semafor se koristi u ovakvom scenariju kako bi se serijalizovao pristup deljenom resursu, kao što je to prikazano na slici 12.

U ovakvom scenariju, binarni semafor je kreiran inicijalno u dozvoljenom stanju (vrednost mu je 1) i koristi se kako bi zaštitio deljeni resurs. U cilju pristupanja deljenom resursu, proces 1 ili proces 2 mora najpre da zauzme binarni semafor, pre nego što nastavi sa upisom ili čitanjem iz deljenog resursa. Pseudo kod koji opisuje ovakvo ponašanje je dat ispod.



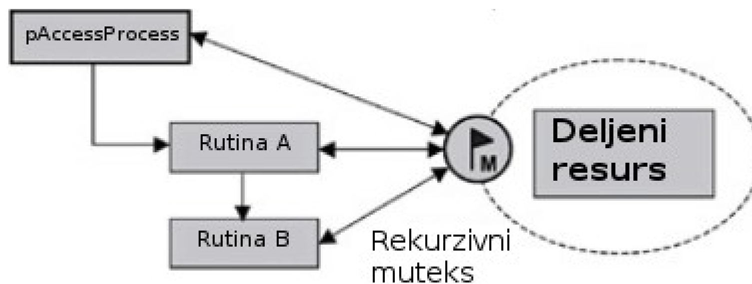
Slika 12: Kontrola pristupa deljenim resursima korišćenjem semafora

```
pAccessProcess ()
{
	:
	Acquire binary semaphore token
	Read or write to shared resource
	Release binary semaphore token
	:
}
```

Ovako organizovan kod serijalizuje pristup deljenom resursu. Ukoliko se pAccessProcess 1 izvršava prvi, on pravi zahtev za zauzimanjem semafora koji će biti uspešan jer je semafor dostupan. Nakon što zauzme semafor, ovaj proces dobija pravo pristupa deljenom resursu i ima mogućnost čitanja ili pisanja u njega. U međuvremenu, proces pAccessProcess 2 višeg prioriteta se aktivira kao rezultat nekog eksternog događaja. Pokušava da zauzme isti semafor, ali odlazi u blokirajuće stanje jer pAccessProcess 1 još uvek poseduje semafor. Nakon što pAccessProcess 1 oslobodi semafor, pAccessProcess 2 postaje odblokiran i nastavlja sa izvršavanjem.
Jedna od opasnosti sa ovakvim pristupom jeste ta da bilo koji proces može slučajno osloboditi binarni semafor, čak i proces koji nije nikada ni zauzeo semafor. Ukoliko se ovako nešto desi, moguće je da se desi da oba procesa zauzmu semafor i nakon toga nastavljaju sa čitanjem/pisanjem koristeći deljeni resurs zaštićen semaforom. Kako bi se obezbedilo da se ovako nešto ne desi, u ovakvim slučajevima je bolje koristiti muteks umesto binarnog semafora. Obzirom da muteks podržava osobinu posedovanja, on će obezbediti da samo proces koji je uspešno zauzeo (zaključao) semafor, može da ga oslobodi (otključa ga).

E. Rekurzivna sinhronizacija pristupa deljenim resursima

U određenim situacijama embedded programer ima potrebu da omogući procesu pristup deljenom resursu u rekurzivnom maniru. Ovakva situacija može postojati ukoliko, na primer, pAccessProcess poziva rutinu A, koja opet poziva rutinu B, pri čemu sve tri rutine zahtevaju pristup istom deljenom resursu, kao što je to prikazano na slici 13.



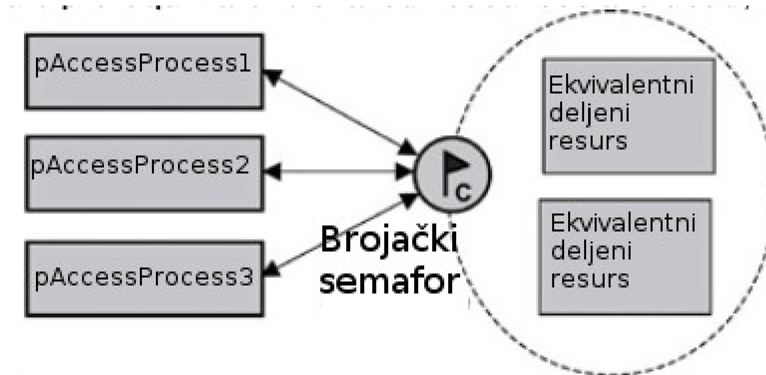
Slika 13: Rekurzivno pristupanje deljenim resursima korišćenjem muteksa

Ukoliko se semafor koristi u ovakvom scenariju, proces će se blokirati, pri čemu dolazi do mrtve petlje (deadlock). Ukoliko se rutina poziva iz konteksta procesa, rutina inherentno postaje deo procesa sama po sebi. Ukoliko se rutina A izvršava, dakle, ona se svakako izvršava kao deo pAccessProcess procesa. Ukoliko ova rutina pokuša, efekat je isti kao da pAccessProcess pokušava ponovo da zauzme semafor koji je već prethodno zauzeo, nakon čega odlazi u blokirajuće stanje.
Jedno moguće rešenje ovog problema je korišćenje rekurzivnog muteksa. Nakon što pAccessProcess zaključa muteks, on postaje "vlasnik" tog muteksa. Svaki naredni pokušaj samog procesa, ili rutine koja je pozvana iz konteksta tog procesa, da se zaključa već zaključan muteks će biti uspešna. Kao rezultat, ukoliko rutina A i B pokušaju da zaključaju muteks, ovo će uspeti bez blokiranja. Pseudo kod koji opisuje ovakvo ponašanje je

```
pAccessProcess (){
       :
       Acquire mutex
       Access shared resource
       Call Routine A
       Release mutex
       :
}
Routine A ()
{
       :
       Acquire mutex
       Access shared resource
       Call Routine B
       Release mutex
       :
}
Routine B ()
{
       :
       Acquire mutex
       Access shared resource
       Release mutex
       :
}
```

F. Sinhronizacija pristupa višestrukim deljenim resursima

U slučajevima kada je više ekvivalentnih deljenih resursa na raspolaganju, brojački semafor se pokazuje kao vrlo koristan, kao što je prikazano na slici 14.



Slika 14: Pristup višestrukim deljenim resursima

Treba obratiti pažnju na činjenicu da ovakva struktura nije moguća ukoliko deljeni resursi nisu ekvivalentni. Stanje brojača brojačkog semafora je inicijalno postavljeno na broj deljenih ekvivalentnih resursa, u ovom primeru 2. Kao rezultat, prva dva procesa koji zahtevaju semafor će biti usluženi. Ipak, treći proces koji pokuša da zauzme semafor će biti blokiran sve dok jedan od procesa koji su prethodno zauzeli semafor ne oslobodi svoj semafor. Pseudo kod koji opisuje ponašanje procesa u ovakvom slučaju je (pri čemu se sličan pseudo kod koristi za pAccessProcess 1, 2 i 3):

```
pAccessProcess ()
{
	:
	Acquire a counting semaphore token
	Read or Write to shared resource
	Release a counting semaphore token
	:
}
```

Slično kao u slučaju binarnog semafora, ovakav dizajn može izazvati problem ukoliko se semafor oslobodi od strane nekog od procesa koji nije originalno ni zauzeo semafor. Ukoliko je kod jednostavan, ovo možda i neće biti problem. Ukoliko, pak, ovo nije slučaj, korišćenje muteksa umesto semafora rešava ovaj problem.

Kao što je prikazano na slici 13 posebni muteksi mogu biti dodeljeni svakom od deljenih resursa. Kada pokušava da zaključa muteks, svaki proces pokušava da zauzme prvi muteks u “neblokirajućem” maniru. Ukoliko ovo ne urodi plodom, proces će pokušati da zauzme drugi muteks u blokirajućem maniru.

```
pAccessProcess ()
{
      :
      Acquire first mutex in non-blocking way
            If not successful then acquire 2nd mutex in a blocking way
      Read or Write to shared resource
      Release the acquired mutex
      :
}
```

U ovakvom scenariju, procesi 1 i 2 uspešno zaključaju svoje mutekse i dobijaju pravo korišćenja deljenih resursa. Kada proces 3 pokuša da zaključa prvi muteks u “neblokirajućem” maniru, ukoliko je muteks slobodan, proces 3 ga zaključa i garantovan mu je pristup prvom deljenom resursu. U slučaju da nije tako, proces 3 pokušava da zaključa drugi muteks, ali ovaj put blokirajući. Ukoliko je drugi muteks još zaključan, proces 3 će blokirati i čekati na drugi muteks dok ne postane otključan.