Predavanje 5

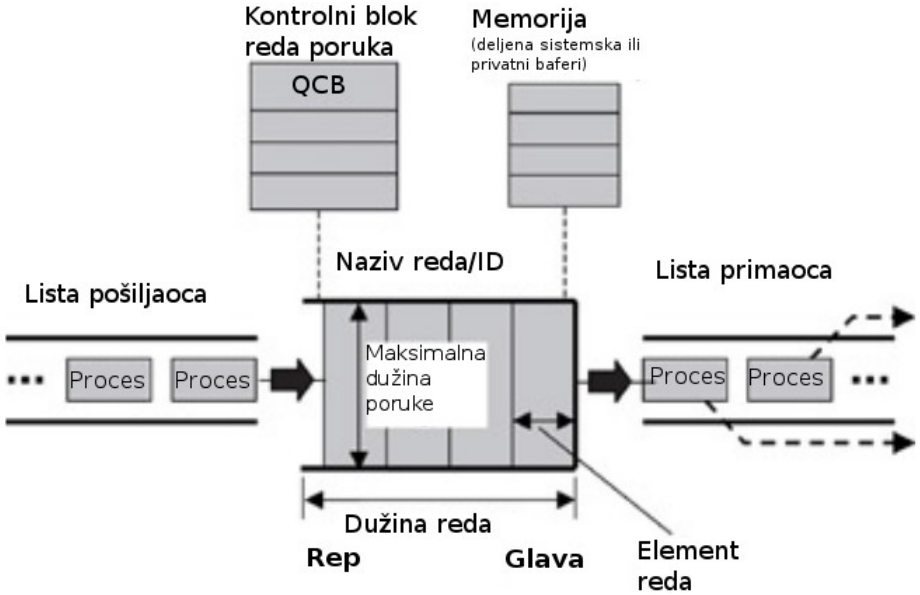
Napredni kernel objekti

5.1 Redovi Poruka

Na prethodnom predavanju je diskutovano kako se dve ili više niti izvršavanja sinhronizuju tokom izvršavanja. U mnogim slučajevima, ipak, sama sinhronizacija nije dovoljna, jer procesi moraju biti u mogućnosti da razmenjuju poruke. Redovi poruka (eng. *Message Queues*) kao i servisi za upravljanje redovima poruka omogućavaju komunikaciju između procesa (eng. *Inter-Process Communication*). Na ovom predavanju će biti više reči o:

- definisanju redova poruka
- stanjima redova poruka
- sadržaju redova poruka
- smeštanju poruka memoriju
- tipičnim operacijama na redovima poruka i
- tipičnim primenama redova poruka

5.1.1 Definicija reda poruka



Slika 1: Red poruka

Red poruka je baferovana struktura preko koje procesi i prekidne rutine šalju i primaju poruke u

cilju komunikacije i sinhronizacije podataka. Red poruka privremeno smešta poruke od procesa koji ih šalje, sve dok ih proces kome su namenjene ne preuzme. Ovakva infrastruktura omogućava razdvajanje slanja od prijema poruka, tj. omogućava procesima nesinhronizovano slanje i prijem poruka.

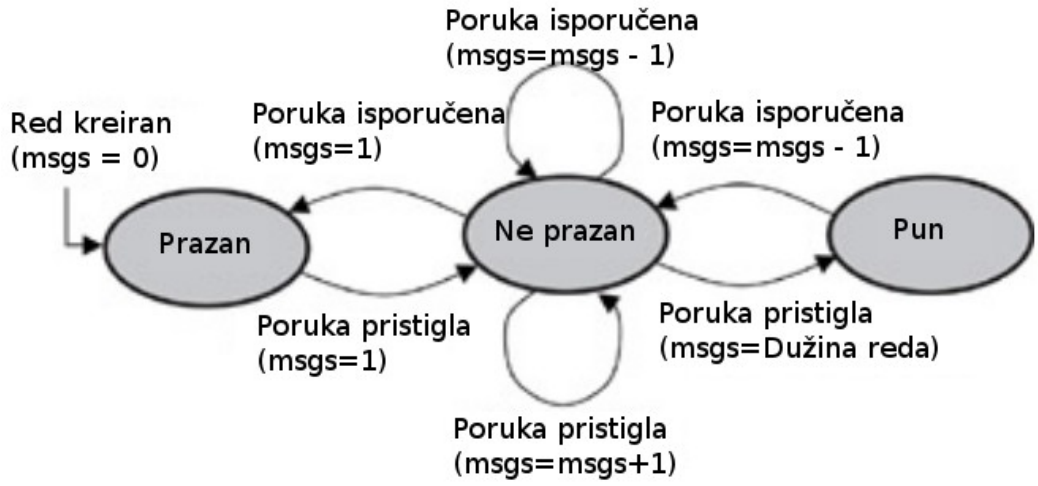
Slično kao i u slučaju semafora, redovi poruka imaju nekolicinu pridruženih komponenti od strane kernela, pomoću kojih kernel kontroliše i upravlja redom poruka. Kada se red poruka prvobitno kreira, dodeljuje mu se *kontrolni blok reda poruka* - QCB (*Queue Control Block*), ime reda poruka, jedinstveni ID, memorijski bafer, odgovarajuća dužina, maksimalna dužina poruke i jedan ili više redova čekajućih procesa.

Nakon što programer odredi koliko je memorije potrebno za realizaciju reda poruka, kernel alocira memoriju za red poruka ili iz deljene memorije rezervisane za sve redove poruka u sistemu ili korišćenjem privatnog memorijskog prostora za svaki pojedinačni red poruka.

Sam red poruka se sastoji od određenog broja elemenata, od kojih svaki može da skladišti po jednu poruku. Elementi koji skladište prvu i poslednju poruku nazivaju se *glava* i *rep* (eng. *head* i *tail*) respektivno. Neki elementi reda mogu biti prazni (ne sadrže nikakve poruke), a ukupan broj elemenata (praznih i onih koji nisu) predstavlja ukupnu veličinu reda, koja se definiše prilikom kreiranja reda poruka.

Kao što je prikazano na slici 1, red poruka ima dve dodeljene liste čekajućih procesa. Lista primaoca se sastoji od procesa koji čekaju na poruke kada je red prazan. Lista pošiljaoca, sa druge strane, sadrži procese koji čekaju kada je red poruka popunjen.

5.1.2 Stanja reda poruka



Slika 2: Red poruka implementiran kao konačni automat

Kao i u slučaju drugih kernel objekata, redovi poruka su implementirani kao mašine stanja (FSM), odnosno konačni automati, kao što je prikazano na slici 2. Kada se red poruka kreira, FSM se nalazi u stanju *prazan*. Proces koji pokuša da primi poruku iz praznog reda poruka, biće blokiran i sačuvan

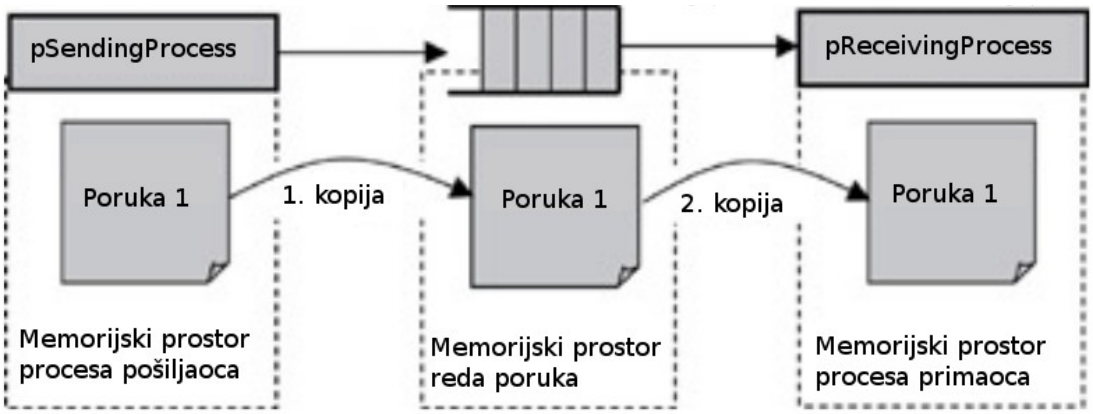
u listi čekajućih procesa primaoca, koja je realizovana kao FIFO ili lista bazirana na prioritetima.

U ovakvom scenariju, ukoliko drugi proces pošalje poruku u red poruka, ona je direktno dodeljena blokiranom procesu. Blokirani proces se tada uklanja iz liste čekajućih procesa, i premešta se ili u stanje *spreman*, ili u *aktivno* stanje. Red poruka u ovom slučaju ostaje prazan jer je poruka uspešno prosleđena. Ukoliko se druga poruka pošalje u isti red poruka i nema procesa koji čekaju na poruku, sam red poruka prelazi u stanje *ne-prazan*. Kako naredne poruke pristižu u red poruka, doći će do situacije u kojoj je broj primljenih poruka u redu jednak ukupnom kapacitetu reda poruka, te on prelazi u stanje *pun*.

Dok je red poruka u ovom stanju, nije moguć prijem novih poruka sve dok neki od primaoca poruka ne preuzme poruku iz reda, čime se oslobađa jedno prazno mesto (isprazni se jedan element reda).

U nekim implementacijama kernela, ukoliko proces pokuša da pošalje poruku u pun red poruka, funkcija slanja poruke vraća vrednost koda greške. U drugim realizacijama, proces koji je slao poruku prelazi u blokirano (ili suspendovano) stanje i premeštaju ga u red čekajućih pošiljaoca poruka.

5.1.3 Struktura redova poruka



Slika 3: Kopiranje prilikom stavljanja poruke u red poruka

Redovi poruka mogu da sadrže najraznovrsnije podatke. Primeri bi bili:

- temperatura očitana od strane senzora
- bitmap slika koja treba da se prikaže na displeju
- tekstualna poruka za prikaz na LCD-u
- taster pritisnut na tastaturi
- paket podataka koji se šalje preko mreže
- ..

Neke od ovih poruka mogu biti veoma dugačke što dovodi do prekoračenja maksimalne dužine poruke, određene prilikom kreiranja reda poruka. Jedan način da se prevaziđe ovaj problem jeste da se u red poruka prosleđuje pokazivač na poruku, umesto same poruke. Čak i u slučaju dugačkih poruka koje mogu da stanu u red poruka, često je bolje koristiti ovu metodu u cilju poboljšanja performansi i korišćenja memorije. Kada proces šalje poruku drugom procesu, poruka se kopira dva puta, kao što je prikazano na slici 3. Prvi put prilikom upisa u red poruka od stane procesa pošiljaoca poruke, pri čemu se poruka kopira iz memorijskog prostora pošiljaoca u memorijski prostor reda. Drugo kopiranje se vrši prilikom kopiranja poruke iz reda poruka u memorijski prostor procesa primaoca poruke. U zavisnosti od implementacije kernela, moguće je da se poruka samo jednom kopira direktno iz memorijskog prostora procesa pošiljaoca u memorijski prostor procesa primaoca poruke. Obzirom na to da je kopiranje bloka podataka veoma često “skupa” operacija, u pogledu performansi i zauzeća memorijskih resursa, dobra je praksa minimizovati broj kopiranja u embedded sistemima, bilo da se to odnosi na kreiranje malih poruka, ili, kada to nije moguće, korišćenjem pokazivača.

5.1.4 Smeštanje poruka u memoriju

Različite implementacije kernela smeštaju poruke u različite memorijske lokacije. U prvoj varijanti koristi se deljena sistemska memorija, u kojoj su svi redovi poruka smešteni u jedinstvenom velikom prostoru memorije. U drugoj varijanti, koriste se zasebni memorijski blokovi, tzv privatni baferi, za svaki od redova poruka.

Prednost korišćenja deljene memorije (eng. memory pools) u ove svrhe je u tome što je na ovaj način garantovano da nikada neće svi redovi poruka biti popunjeni u potpunosti i na ovaj način se svakako čuva memorija kao značajan sistemski resurs. Loša strana ovakvog pristupa leži u činjenici da red “velikih” poruka vrlo lako može da zauzme veći deo deljene memorije, ne ostavljajući dovoljno prostora ostalim redovima. Indikacija da se ovakvo nešto desilo jeste kada red koji nije popunjen počinje da odbacuje pristigle poruke, ili, kada popunjen red poruka nastavlja da prihvata nove poruke.

Privatni baferi zahtevaju znatno više rezervisane memorije obezbeđujući pun kapacitet svih korišćenih redova poruka, čak i u slučaju kada nisu oni svi popunjeni u potpunosti. Sa druge strane, ovakav pristup omogućava se poruke neće kopirati jedne preko drugih, i da će biti dovoljno prostora za sve poruke, što rezultuje povećanjem pouzdanosti.

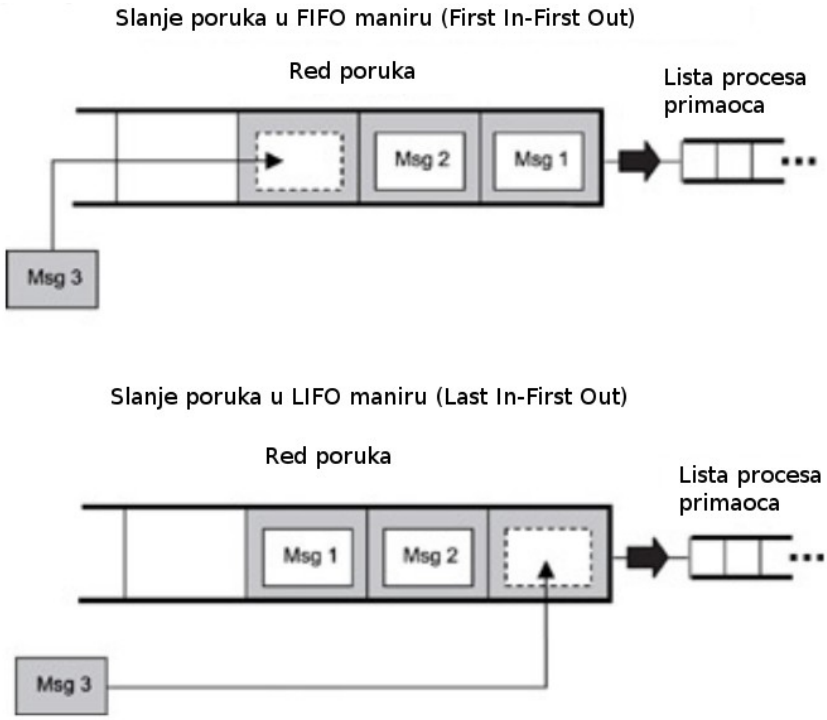
5.1.5 Tipične operacije sa porukama

Operacija	Opis
Create	Kreira red poruka
Delete	Briše kreirani red poruka

Send	Pošalji poruku
Receive	Primi poruku
Broadcast	Pošalji poruku svim primaocima
Show Queue Info	Prikaži informacije o redu poruka
Show queue's process-waiting list	Prikaži listu čekajućih procesa

Kada se kreiraju, redovi poruka se tretiraju kao globalni objekti i nisu posedovani od strane ni jednog pojedinačnog procesa. Tipično, red poruka se koristi od strane grupe procesa ili prekidnih rutina.

Prilikom kreiranja reda poruka, programer mora da donese određene inicijalne odluke u vezi sa kapacitetom reda poruka, maksimalnom veličinom poruke u redu poruka, i način stavljanja procesa u red čekanja na poruke. Brisanje reda poruka automatski odblokirava sve čekajuće procese, dok se poruke koje su se nalazile u redu poruka gube u tom slučaju.



Slika 4: Upis u red poruka u FIFO i LIFO maniru

Prilikom slanja poruka, kernel tipično popunjava red poruka u FIFO maniru pri čemu se svaka nova poruka smešta na kraj (rep) reda.

U nekim implementacijama urgentne poruke se smeštaju na početak (glavu) reda. Na ovaj način je implementiran rad poruka koji je LIFO tipa. Takođe, većina implementacija omogućava prekidnim rutinama da šalju poruke u red poruka. Bez obzira na način popunjavanja reda poruka, to se vrši uvek na jedan od tri načina:

- ne blokirajući (prekidne rutine i procesi)

- blokirajući sa tajm-autom (samo procesi)
- blokirajući (samo procesi)

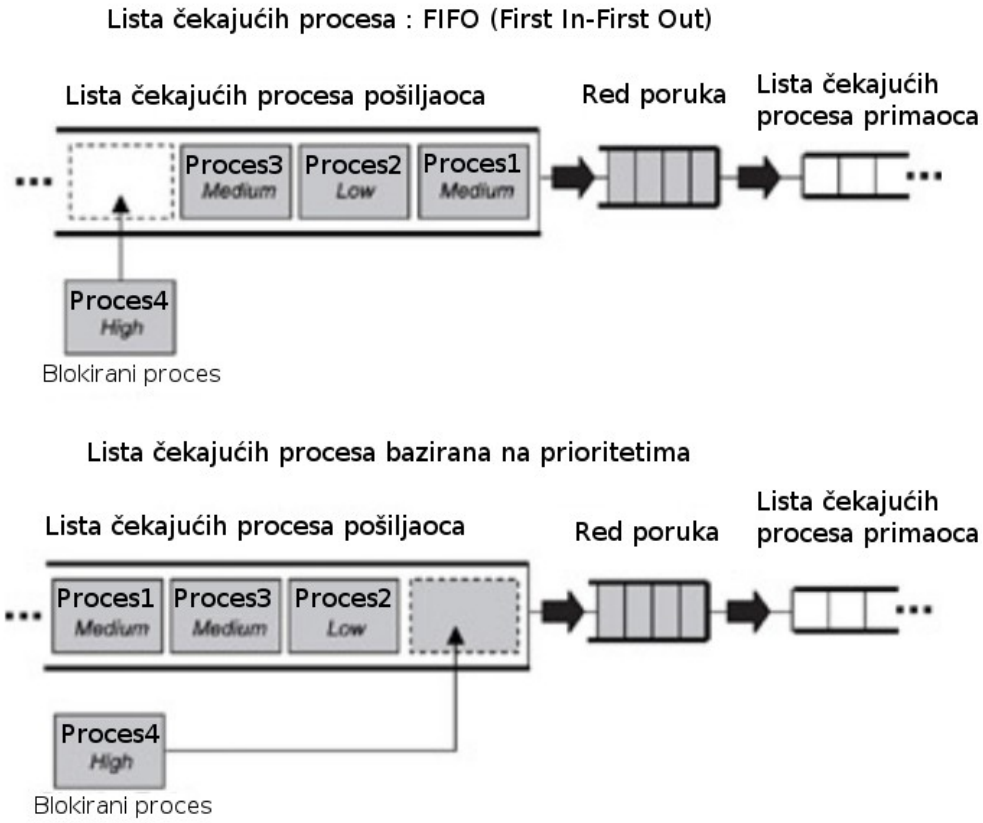
U određenim situacijama poruke moraju biti poslate bez blokiranja pošiljaoca. Ukoliko je red poruka već popunjen, zahtev za slanjem poruke će rezultovati vraćanje koda greške, pri čemu proces ili prekidna rutina mogu da nastave sa izvršavanjem. Jedino na ovaj način poruke mogu da se šalju iz prekidnih rutina, obzirom da prekidne rutine ne mogu biti blokirane.

Ukoliko neki proces blokira prilikom slanja poruke sa tajm-autom, on će biti odblokiran ili kada se oslobodi element reda, ili kada istekne vreme tajm-auta (pri čemu će biti vraćen kod greške na osnovu kojeg proces može da zaključi da nije dočekao svoj red u redu poruka).

Kod prijema poruka, kao i kod slanja, postoje tri načina na koji se ono vrši: blokirajući, blokirajući sa tajm-autom i ne-blokirajući. Jedina razlika u ovom slučaju je u tome što se blokiranje dešava kao posledica reda poruka koji je prazan, a ne popunjen. Čekajući procesi smeštaju se u red čekanja ili u FIFO maniru, ili po prioritetima (Slika 5).

Da bi se red poruka popunio, potrebno je da lista čekajućih procesa primaoca poruka bude prazna, ili da je učestanost slanja poruka veća od učestanosti prijema.

Poruke se iz reda poruka mogu čitati na dva načina: destruktivno i ne-destruktivno. Prilikom destruktivnog čitanja, nakon što proces primi poruku, on će je permanentno izbrisati iz reda i bafera.

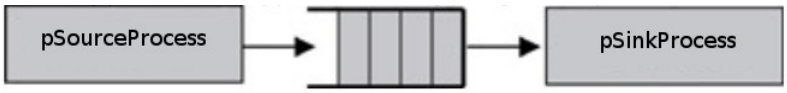


Slika 5: Postavljanje procesa u listu čekajućih procesa

Sa druge strane, ne-destruktivno čitanje omogućava procesu da pogleda poruku bez uklanjanja iste iz reda (poruke koja se nalazi na glavi reda). Iako oba načina prijema poruka mogu biti primenljiva u različitim aplikacijama, činjenica je da ne implementiraju svi kerneli ne-destruktivno čitanje poruka.

5.1.6 Tipična upotreba redova poruka

1. **Jednosmerna komunikacija bez sinhronizacije (non-interlocked)**



Slika 6: Jednosmerna komunikacije bez sinhronizacije

Najjednostavnija primena redova poruka podrazumeva jedan proces koji šalje poruke, red poruka, i proces koji prima poruke.

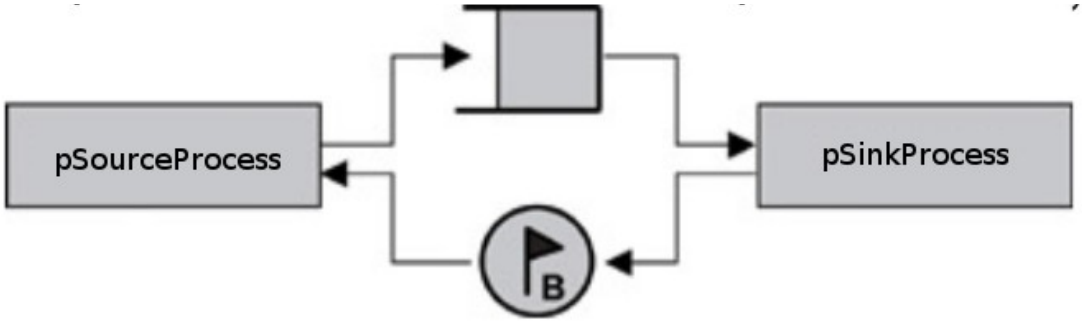
Aktivnosti pSourceProcess-a i pSinkProcess-a nisu sinhronizovane. pSourceProcess jednostavno šalje poruku i ne čeka na potvrdu od strane pSinkProcess-a. Ukoliko je pSinkProcess-u dodeljen viši prioritet, on se prvi izvršava dok se ne blokira prilikom pokušaja čitanja praznog reda poruka. Čim pSourceProcess pošalje poruku u red poruka, pSinkProcess prihvata tu poruku i nastavlja da se izvršava ponovo. Ukoliko je, pak, pSinkProcess-u dodeljen niži prioritet, pSourceProcess će da popuni ceo red poruka, nakon čega će se blokirati prilikom pokušaja upisa poruke u popunjen red. Ovo će dovesti do toga da pSinkProcess biva odblokiran i da nastavi sa preuzimanjem poruka iz reda.

Prekidne rutine tipično koriste ovakav način jednosmerne komunikacije. Proces sličan pSinkProcess-u se izvršava i čeka na pristigle poruke. Kada se desi hardverski triger, prekidna rutina se izvršava i postavlja jednu ili više poruka u red poruka. Nakon što se prekidna rutina završi, pSinkProcess dobija priliku da se ponovo izvršava i da preuzme poruku (poruke) iz reda poruka. Naravno, kada prekidna rutina šalje poruku u red poruka, ona to mora da radi na ne-blokirajući način. Ukoliko je red poruka popunjen kada prekidna rutina pokušava da šalje poruku, ta poruka će biti odbačena.

2. **Jednosmerna komunikacija sa sinhronizacijom (interlocked)**

U određenim situacijama, proces koji šalje poruke može da zahteva sinhronizaciju (handshaking), u vidu potvrde da je primaoc poruke uspešno primio poruku.

U slučaju da poruka iz bilo kog razloga nije primljena u potpunosti, pošiljaoc može ponovo da je pošalje, u ovakvom scenariju. Jednostavan primer je prikazan na slici 7.

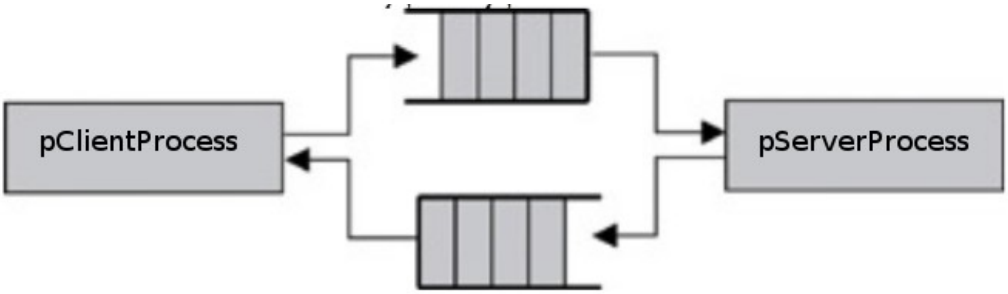


Slika 7: Jednosmerna komunikacija sa sinhronizacijom

Na ovom primeru pSourceProcess i pSinkProcess koriste binarni semafor inicijalno postavljen na 0, i red poruka dužine 1 (često se naziva i mailbox). pSourceProcess šalje poruku u red poruka i blokira se pokušavajući da zauzme semafor. pSinkProcess prima poruku i inkrementira vrednost binarnog semafora. Kao rezultat, aktivira se ponovo pSourceProcess i postavlja sledeću poruku, nakon čega opet biva blokiran.

Semafor u ovakvoj strukturi samo služi da obezbedi sinhronizaciju dva procesa koji komuniciraju.

3. **Dvosmerna komunikacija sa sinhronizacijom**



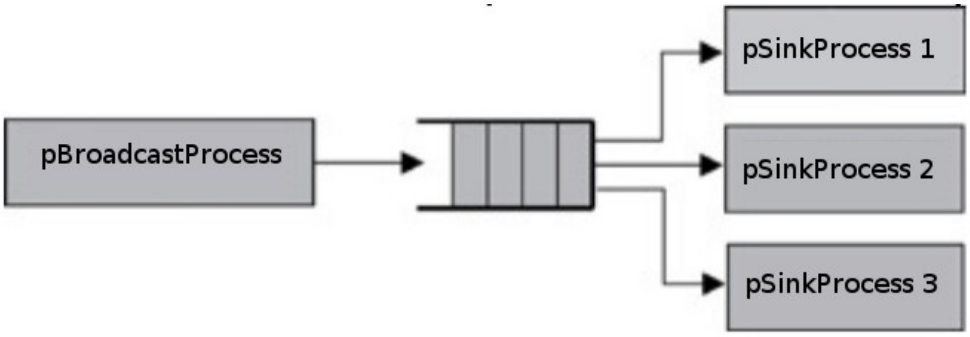
Slika 8: Dvosmerna komunikacija

U slučaju potrebe za dvosmernom komunikacijom, full-duplex (npr. klijent-server komunikacija), implementiraju se dva odvojena reda poruka, kao na slici 8.

U ovakvoj upotrebi, šalje zahteve ka pServerProcess-u korišćenjem reda poruka. pServerProcess ispunjava pristigli zahtev slanjem poruke nazad -u. U ovakvoj topologiji je neophodno imati dva odvojena reda poruka, u slučaju da je neophodno slati podatke. Ukoliko je samo sinhronizacija potrebna, u tu svrhu se može koristiti i semafor.

U primeru sa slike 8, pServerProcess-u je obično dodeljen viši prioritet, što omogućava da brzo odgovara na pristigle zahteve. Ukoliko ima više klijenata u sistemu, svi oni mogu da dele isti red poruka, pri čemu onda pServerProcess koristi posebne redove poruka za komunikaciju sa svakim pojedinačnim klijentom.

4. **Broadcast komunikacija**



Slika 9: Broadcast komunikacija

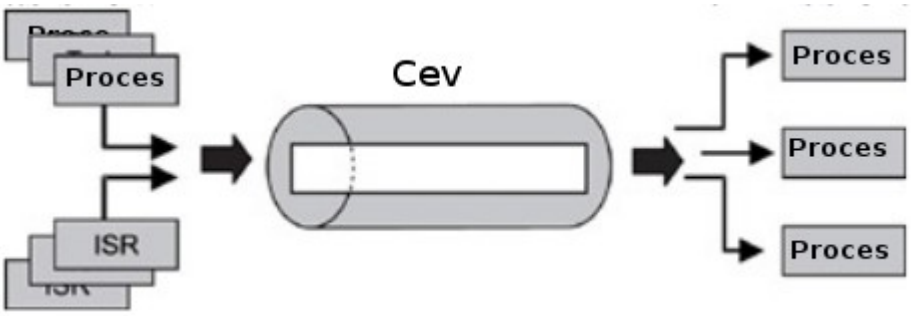
Ukoliko kernel dozvoljava, redovi poruka omogućavaju programeru slanje iste poruke ka više različitih procesa, kao što je prikazano na slici 9.

U ovakvom scenariju, pSinkProcess1, 2 i 3 su blokirani čekajući na poruke. Kada pBroadcastProcess počne da se izvršava, on šalje poruku koja rezultuje odblokiranjem sva tri procesa koji čekaju.

5.2 Cevi (pipes)

Slika 10: Cev sa elementarnim operacijama upisa i čitanja

Cevi su kernel objekti koji omogućavaju nestruktuiranu razmenu podataka i obezbeđuju sinhronizaciju između procesa.. U tradicionalnoj implementaciji, cevi su jednosmeran interfejs za razmenu podataka, kao što je prikazano na slici 10. Postoje dva desktriptora, po jedan za svaki kraj cevi (jedan za upis, drugi za čitanje), i oni su kreirani od strane operativnog sistema u trenutku kreiranja cevi. Podaci se upisuju korišćenjem jednog desktriptora i čitaju korišćenjem drugog. Podaci se čuvaju unutar cevi kao nestruktuiran tok bajtova, i čitaju se u FIFO maniru.



Slika 11: Primena cevi

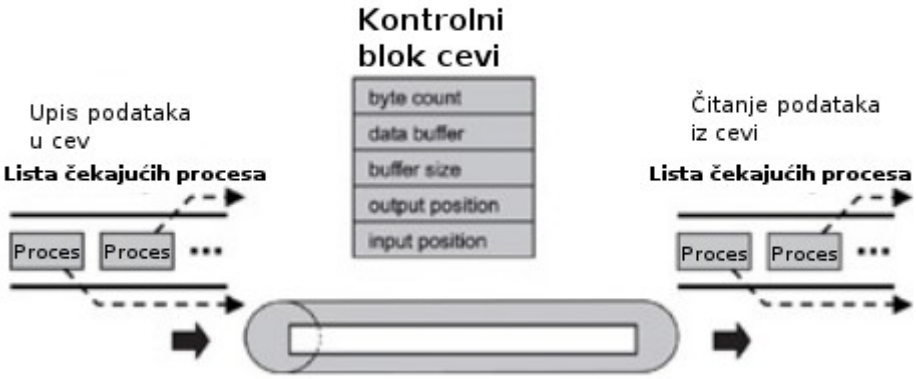
Cevi omogućavaju jednostavan tok podataka u smislu da “čitač” biva blokiran ukoliko je cev prazna (nema podataka), a pisač (eng. writer) se blokira kada je cev puna. Tipično, ova infrastruktura se koristi prilikom razmene podataka između procesa koji “proizvodi” podatke i procesa koji konzumira podatke, kao što je prikazano na slici 11. Takođe je dozvoljeno da postoji više “pisača” i više “čitača” pridruženih nekoj cevi.

Treba primetiti da je cev konceptualno veoma slična redu poruka, ali ipak sa značajnim razlikama:

- za razliku od reda poruka, cev ne može da skladišti veći broj poruka
- umesto toga, podaci koji su skladišteni u cevi su nestruktuirani i predstavljaju niz bajtova
- podaci u okviru cevi ne mogu biti prioritizovani-uvek je tok podataka u FIFO maniru
- cevi poseduju moćan mehanizam selektovanja, kao što će biti opisano u nastavku

Cevi mogu biti dinamički kreirane i uništene. Kernel kreira i održava podatke vezane za kreirena cevi u okviru interne strukture koja se naziva PCB (Pipe Control Block). Struktura PCB-a varira od jedne do druge implementacije, ali u generalnoj formi, sadrži bafer alociran od strane kernela. Veličina bafera je održavana od strane kontrolnog bloka i fiksna je kada je jednom cev kreirana- ne

može se menjati dinamički. Trenutni brojač broja upisanih bajtova, kao i pokazivači na trenutne ulazne i izlazne podatke su takođe deo bloka. Brojač upisanih bajtova daje informacije o tome koliko je podataka dostupno u okviru cevi, pokazivač ulaznih podataka specificira na kome mestu će se izvršiti sledeća operacija upisa. Slično, pokazivač izlaznih podataka specificira na kojoj poziciji će se izvršiti naredna operacija čitanja. Kernel kreira dva desktriptora koji su jedinstveni u okviru I/O prostora sistema i vraća ih procesu koji kreira cev. Dve liste čekajućih procesa su asocirane sa svakom cevi, kao što je prikazano na slici 12. Jedna služi za praćenje procesa koji čekaju na upis u cev, kada je cev popunjena, dok druga lista služi za praćenje procesa koji čekaju prilikom čitanja podataka iz cevi koja je prazna.



Slika 12: Struktura cevi u okviru kernela

Obzirom na činjenicu da je i cev implementirana kao konačan automat (slično kao red poruka), stanja u kojima se može naći cev su prikazana na slici 13.

Slika 13: Cev implementirana kao konačni automat

Kernel tipično podržava dva tipa cevi: imenovane i neimenovane cevi (eng. named pipes i unnamed pipes). Imenovana cev, takođe poznata kao FIFO, ima naziv sličan nazivu datoteka i može se pronaći u okviru fajl-sistema, isto kao da je u pitanju neka tekstualna datoteka ili fajl koji predstavlja uređaj. Bilo koji proces, ili prekidna rutina mogu da koriste ovakvu imenovanu cev i da je referenciraju na osnovu njenog imena. Neimenovane cevi nemaju svoje ime i ne pojavljuju se u

okviru fajl-sistema. One se moraju referencirati korišćenjem deskriptora koji se kreiraju prilikom kreiranja cevi od strane kernela.

5.2.1 Tipične operacije sa cevima

Operacija	Opis
Pipe	Kreira cev
Open	Otvara cev
Close	Briše ili zatvara cev
Read	Čitanje
Write	Pisanje
Fcntl	Omogućava kontrolu nad desktriptorom
Select	Čeka na događaj koji će se desiti sa cevi

Operacijom *pipe* se kreira neimenovana cev. Ova operacija vraća dva deskriptora procesu koji je pozvao ovu metodu, i sva naredna referenciranja se odnose na ove deskriptore. Jedan deskriptor se koristi samo za upis, dok se drugi koristi za čitanje.

Kreiranje imenovane cevi je slično kreiranju datoteke u okviru fajl-sistema. Specifičan sistemski poziv zavisi naravno od same implementacije operativnog sistema. Neki standrardni nazivi za ove metode su *mknod* i *mkfifo*. Obzirom na činjenicu da je imanovana cev prepoznatljiva u okviru fajl-sistema nakon što je kreirana, ona se može otvoriti korišćenjem standardne open operacije. Proces koji otvara cev mora eksplicitno da naglasi da li otvara cev za upis ili za čitanje, i nikako nisu dozvoljeno oba pristupa. Zatvaranje je suprotna operacija od otvaranja u slučaju rada sa cevima. Slično kao i kod otvaranja, operacija zatvaranja može jedino biti izvršena na imenovanoj cevi. U nekim implementacijama, čim se zatvori, cev biva permanentno uklonjena iz sistema.

Operacija čitanja vraća podatak iz cevi procesu koji ju je pozvao. Proces sam specificira koliko podataka želi da čita iz cevi. Proces može odabrati da bude blokiran čekajući na ostatak podataka koje čeka, u slučaju kada je broj podataka koje čeka veći od kapaciteta cevi. Važno je spomenuti da je operacija čitanja iz cevi uvek destruktivna jer se pročitani podatak uklanja iz cevi nakon završetka ove operacije. Dakle, za razliku od redova poruka, cevi se ne mogu koristiti u cilju slanja Broadcast poruka. Ipak, proces može da “konzumira” blok podataka koji pristižu od strane većeg broja “pisača” tokom jedne operacije čitanja.

Operacija upisa dodaje nove podatke već postojećima u okviru cevi. Pozivajući proces specificira količinu podataka koje upisuje u cev, i može odabrati da bude blokiran u slučaju da je kapacitet cevi manji od veličine podataka koje treba upisati.

Ne postoje granice za slanje poruka korišćenjem cevi, obzirom da se podaci i skladište kao nestruktuirane poruke (nizovi bajtova). Ovaj detalj zapravo predstavlja osnovnu razliku između cevi i redova poruka. Obzirom na to da ne postoje zaglavlja poruka (eng. message headers), nemoguće je odlučiti ko je originalno “postavio” podatke u cev. Obzirom da se podaci pristigli u cev ne mogu

prioritizovati, cevi ne treba koristiti u situacijama kada urgentne poruke treba prvo pročitati.

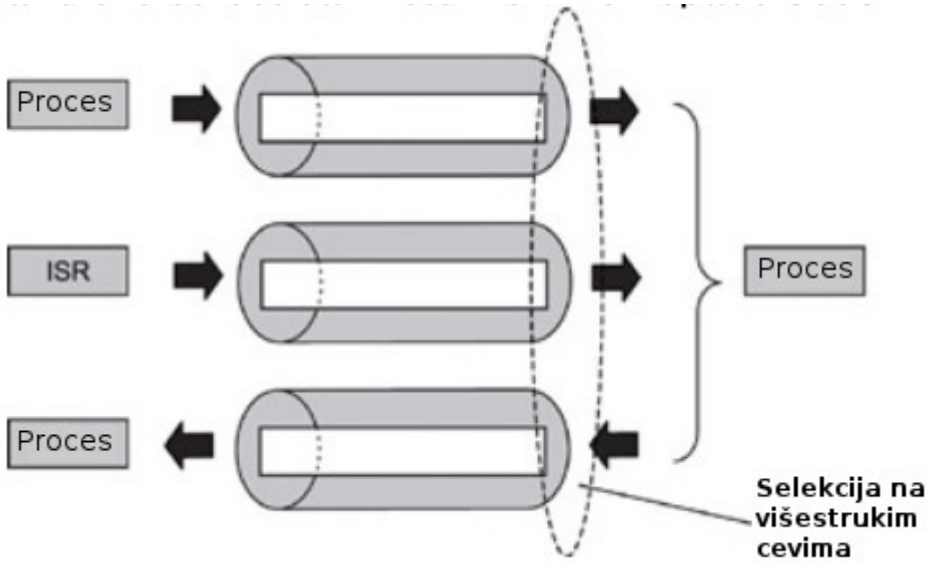
Kontrola cevi se vrši *Fcntl* operacijom koja obezbeđuje generičku kontrolu deskriptora cevi korišćenjem raznoraznih komandi koje kontrolišu ponašanje cevi.

Na primer, tipična implementacija komande je ne-blokirajuća komanda. Ova komanda kontroliše da li je pozivajući proces blokiran prilikom čitanja iz prazne cevi, ili upisa u popunjenu cev. Još jedan tipičan primer komande je *flush* komanda. Ova komanda briše sve podatke iz cevi i reinicijalizuje cev u početno stanje, kao kada je cev kreirana.

Operacija selekcije omogućava procesu da blokira i čeka ne specifičan događaj koji treba da se desi na jednoj ili više cevi. Čekanje se može odnositi na čekanje na podatke koji će postati dostupni, ili čekanje na podatke koji će se isprazniti iz cevi.

Na slici 14 je prikazan primer kada jedan proces čeka prilikom čitanja dve cevi i upisuje u treću. U ovom slučaju, poziv selekcije se vraća kada podaci postanu dostupni u bilo kojem od dve cevi. Ista operacija selekcije se može koristiti za čekanje na prostor koji će se osloboditi za upis novih podataka u donju cev.

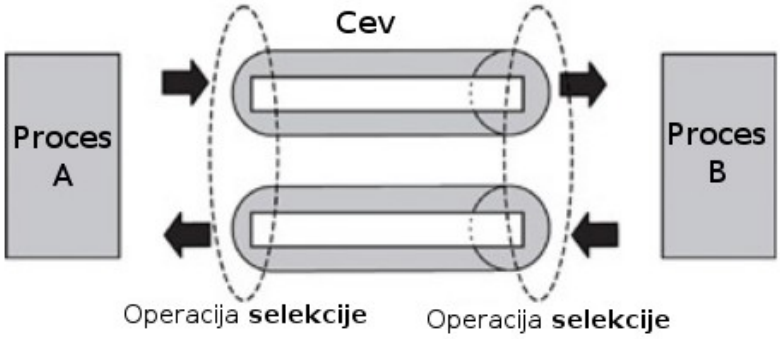
Za razliku od cevi, redovi poruka ne podržavaju operacije selekcije. Kao posledica, dok proces može da ima pristup ka više redova poruka, on ne može da blokira čekajući na dostupnost podataka u okviru grupe praznih redova poruka. Isto ograničenje važi i za upis u redove poruka: proces može da upisuje u više redova poruka, ali ne može da blokira čekajući na slobodno mesto u okviru bilo kojeg reda poruka. Očigledno je da se u ovome zapravo ogleda i osnovna prednost upotrebe cevi u odnosu na redove poruka: prednost korišćenja *selekcije* .



Slika 14: Operacija selekcije

Pošto je cev jednostavan kanal za prenos podataka, uglavnom je namenjen za proces-proces ili prekidna rutina-proces komunikaciju, kao što je prikazano ranije. Osim toga, cevi se koriste i za među-proces sinhronizaciju, kao što je prikazano na slici 15. Ovo je omogućeno na osnovu operacije selekcije.

Proces A i proces B otvaraju dve cevi za inter-proces komunikaciju. Prva cev je otvorena za slanje podataka od procesa A ka procesu B, dok je druga namenjena za potvrdu prijema podataka, od strane procesa B ka procesu A. Oba procesa koriste selekciju na cevima. Proces A može da čeka asinhrono na mogućnost upisa novih podataka, nakon što je proces B pročitao prethodno upisane podatke. Ovo znači da proces A može da inicira ne-blokirajući upis u cev i nakon toga izvršava ostale operacije sve dok cev ne bude dostupna za upis. Istovremeno, proces A može da čeka asinhrono na prijem potvrde od strane procesa B na drugoj cevi. Slično, proces B može da čeka asinhrono na pristizanje poruka od strane procesa A, kao i na dostupnost prostora u drugoj cevi nakon što proces A pročita informaciju o potvrdi prijema poruke.

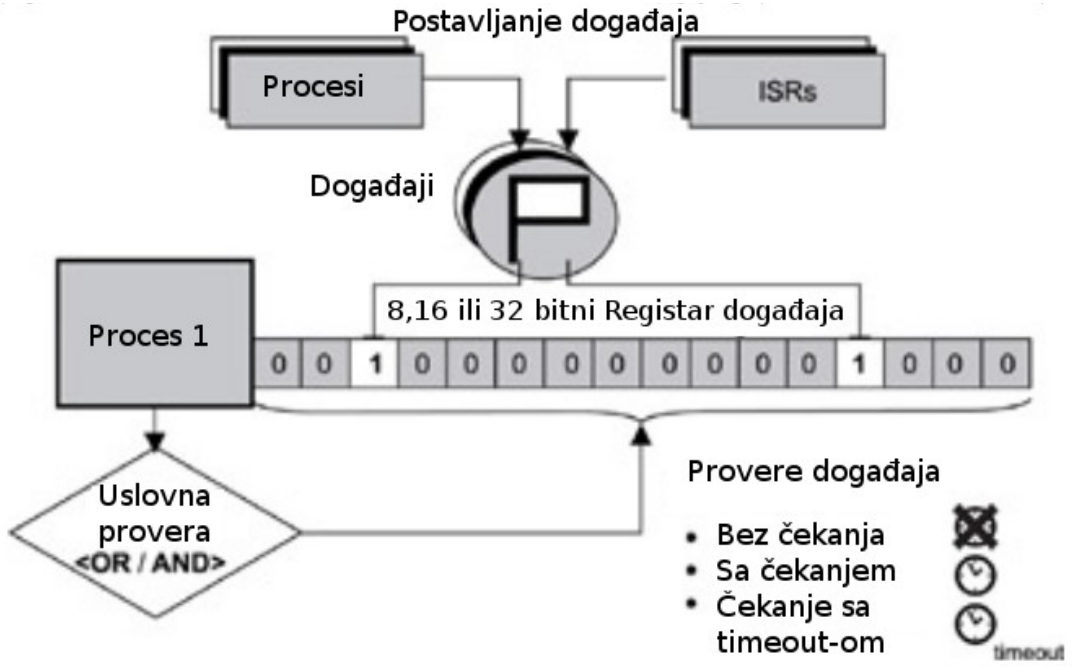


Slika 15: Sinhronizacija dva procesa korišćenjem selekcije

5.3 Registri događaja

Neki kerneli omogućavaju specijalne registre kao deo kontrolnog bloka svakog pojedinačnog procesa (PCB-a), kao što je prikazano na slici 16. Ovaj registar, koji se naziva registar događaja (*event register*) je objekat koji pripada procesu i sadrži grupu binarnih indikatora koji se koriste da prate pojave određenih događaja. U zavisnosti od implementacije kernela, ovi registri mogu biti 8, 16 ili 32 bita širine, ili čak više. Svaki bit u registru se tretira kao binarni indikator (*event flag*) i može se ili setovati ili poništiti.

Korišćenjem ovog registra, proces može da proveri prisustvo određenih događaja koji mogu da kontrolišu samo izvršavanje procesa. Eksterni izvor, npr. prekidna rutina, može da postavlja (setuje) bite u okviru ovog registra, kako bi obavestila proces da se određeni događaj desio. Sama aplikacija definiše događaje koji su asocirani sa svakim pojedinačnim indikatorom, i ovo mora biti usaglašeno između modula koji obaveštava o događaju i modula koji čeka na događaj.



Slika 16: Registar događaja

Proces specificira skup događaja o kojima želi da bude obavešten. Ovaj skup događaja se skladišti u datom registru događaja. Proces takođe definiše tajm-aut koji želi da koristi prilikom čekanja na događaje. Ukoliko se koristi tajm-aut, kernel odblokira proces ukoliko je tajm-aut istekao, pre nego što se desio događaj na koji proces čeka. Proces čak ima mogućnost definisanja kompleksnijih uslova u vezi sa događajima na koje čeka. Na primer, proces može specificirati da želi biti obavešten kada se dese istovremeno događaj 1 i događaj 3, ili kada se desi događaj 2. Ovakva mogućnost obezbeđuje fleksibilnost u definisanju složenih šablona obaveštavanja.

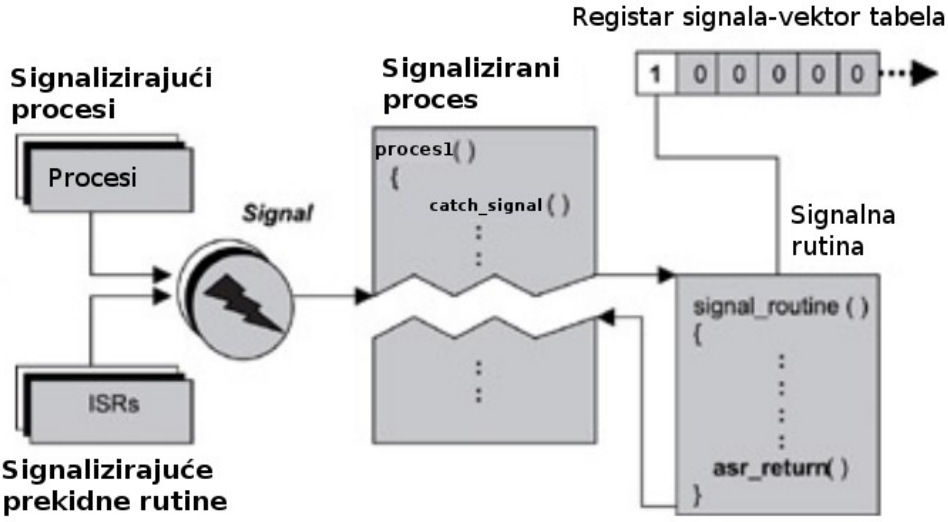
5.4 Signali

Signali su softverski prekidi koji se generišu kada se desi događaj. Prilikom primanja signala, proces koji ga je primio prekida svoje izvršavanje time što se inicira predodređeno asinhrono procesiranje pristiglog signala.

U suštini, signali imaju ulogu da obaveste proces o događaju koji se desio tokom izvršavanja ostalih procesa ili prekidnih rutina. Kao u slučaju prekida, ovi događaji su asinhroni sa proces om koji je obavešavan i ne dešavaju se u nekom predodređenom trenutku tokom izvršavanja proces a. Razlika signala i prekida je u tome što su signali takozvani “softverski prekidi”, koji se generišu izvršavanjem softvera u okviru sistema. Nasuprot njima, redovni prekidi se obično generišu nakon pristizanja signala prekida na jedan od predodređenih eksternih pinova centralne procesorske jedinice (CPU). Prekidi nisu generisani softverskim putem već dejstvom eksternih uređaja.

Broj i tip signal definisanih u okviru datog sistema ne zavisi samo od sistema, već i od korišćenog RTOS-a. Najlakše je razumeti signal kao nešto što je asocirano sa određenim događajem. Događaj

može biti nenameran, kao na primer ilegalna instrukcija (*illegal instruction*) tokom izvršavanja programa, ali i nameran, kao na primer notifikacija od strane jednog proces a drugom procesu da je vreme da se terminira. Osim što proces jasno definiše akcije koje će se preduzeti kada signal pristigne, on nema nikakvu kontrolu trenutka kada će on pristići. Kao posledica, signali pristižu često sporadično kao što je prikazano na slici 17.



Slika 17: Izvršavanje asinhrone signalne rutine

Kada signal pristigne, proces privremeno prekida svoje izvršavanje, i predodređena asinhrona signalna rutina (ASR) se poziva. Svaki signal je identifikovan pomoću celobrojnog podatka, koji se naziva broj signala (*signal number*) ili broj vektora (*vector number*).

Kernel kreira signalni kontrolni blok kao deo PCB-a u kojem postoje četiri skupa signala: prihvaćeni signali, ignorisani signali, čekajući signali i blokirani signali.

Proces može da obezbedi signalnu rutinu za svaki signal koji se nalazi u grupi prihvaćenih signala, ali može i da poziva podrazumevajuću rutinu obezbeđenu od strane kernela. Takođe, moguće je imati jednu rutinu koja se poziva za svaki od pristiglih signala iz grupe prihvaćenih signala.

Ignorisani signali se čuvaju u okviru skupa ignorisanih signala i svaki signal iz ovog skupa ne može da prekine proces .

Signali koji pristižu dok proces izvršava rutinu obrađujući prethodni signal, postavljeni su u skup čekajućih signala (*pending signals*). Ovi signali će biti prosleđeni procesu čim proces završi sa procesiranjem prethodnog signala. Skup čekajućih signala je podskup skupa prihvaćenih signala.

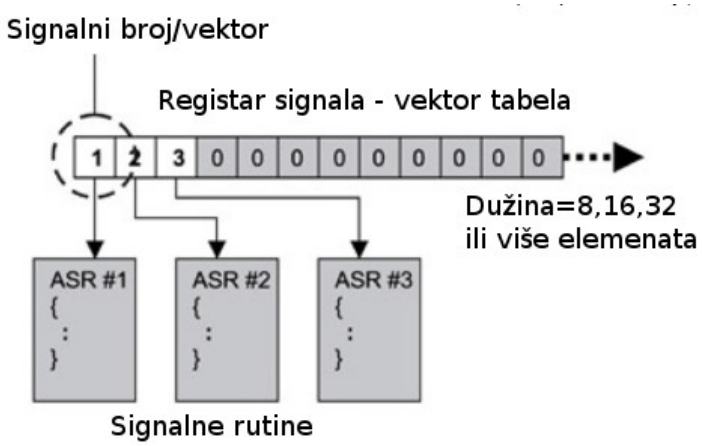
Blokiranjem signala, signal nije ignorisan, već je samo blokirano prihvatanje i reagovanje na signal tokom određenih faza izvršavanja procesa, kada je od suštinske važnosti da proces ne bude prekinut. Proces obaveštava kernel o blokiranim signalima tako što ih postavlja u skup blokiranih signala. Kernel u takvom scenariju neće isporučivati blokirane signale sve dok se oni ne uklone iz skupa.

5.4.1 Tipične operacije sa signalima

Operacija	Opis
Catch	Intalira signalnu rutinu
Release	Uklanja prethodno instaliranu signalnu rutinu
Send	Šalje signal drugom procesu
Ignore	Postavlja signal u skup ignorisanih signala
Block	Postavlja signal u skup blokiranih signala
Unblock	Uklanja signal iz skupa blokiranih signala

Proces može da prihvati signal tek nakon što je instalirao ASR za signal, što se vrši Catch operacijom. Proces može da instalira podrazumevanu rutinu obezbeđenu kernel-om. Čak i u slučaju sopstvene instalirane rutine, rutina može da nakon procesiranja, prosledi kontrolu podrazumevanoj rutini u cilju dodatnog procesiranja.

Nakon što je rutina instalirana za određeni signal, ona se poziva kada se desi taj signal čak i ukoliko je on prihvaćen od strane nekog drugog procesa, ne samo onog koji je instalirao signalnu rutinu. Osim toga, svaki proces može da instalira signalnu rutinu za neki signal. U skladu sa tim, dobra je praksa da proces sačuva prethodno instaliranu rutinu pre nego što instalira sopstvenu i da je restaurira u sistemu nakon što ukloni sopstvenu.



Slika 18: Instaliranje ASR

Na slici 18 je prikazana vektor tabela signala, koju održava kernel. Svaki element vektor tabele je pokazivač ili ofset na ASR. Za signale koji nemaju dodeljene ASR, na odgovarajućem mestu u tabeli se nalazi NULL. Primer sa slike 18 pokazuje tabelu nakon tri *Catch* operacije. Svaka *Catch* operacija instalira jednu ASR, upisivanjem pokazivača ili ofset-a rutine u odgovarajući element vektor tabele.

Release operacija deinstalira signalnu rutinu.

Send operacija omogućava slanje signala od jednog procesa ka drugom.

Ignore operacija obaveštava kernel da određeni skup signala ne treba nikada isporučivati procesu.

Ipak, određeni signali ne mogu biti ignorisani i kada se oni dese, kernel poziva podrazumevenu rutinu.

Block operacija privremeno sprečava signale da budu isporučeni proces u koji poziva ovu operaciju. Na ovaj način se čuva kritična sekcija koda od prekida usled pristiglog signala. Osim toga, blokiranje je često zgodno u cilju sprečavanja konflikta koji može da nastane kada se već izvršava signalna rutina za isti signal. Kada je blokiran, signal ostaje u stanju čekanja i nalazi se u listi čekajućih signala.

Unblock operacija omogućava prethodno blokiranom signalu da se odblokira. Nakon toga, signal se isporučuje trenutno ukoliko se nalazi u čekajućem skupu (*pending signal*).

5.4.2 Tipične upotrebe signala

Neki signali su asocirani sa hardverskim događajima i samim tim su pozvani od strane hardverske prekidne rutine. Prekidna rutina je zadužena da trenutno odreaguje na taj događaj, ali često je potrebno da se pošalje i signal kako bi procesi na koje utiče ovaj događaj imali priliku da izvrše dodatno procesiranje specifično za njih.

Ipak signale treba koristiti pažljivo iz nekoliko razloga:

1. Usled asinhrone prirode pojave signala, kada se koriste za sinhronizaciju između procesa, proces koji prima signal biće prekinut u nedeterminističkom stanju, što svakako nije poželjno.
2. Većina implementacija ne omogućava brojanje ili signala niti stavljanje signala u red čekanja. U takvim implementacijama, višestruki signali će biti ignorisani.
3. Većina implementacija ne podržava prosleđivanje dodatnih informacija sa signalom.
4. Većina implementacija takođe ne obezbeđuje informacije o redosledu pristizanja signala, a signali različitog tipa se tretiraju kao signali istog prioriteta, što često može biti problem.
5. Većina implementacija ne garantuje kada će odblokiran čekajući signal biti isporučen odredišnom procesu.

Ipak neke implementacije kernela uključuju real-time ekstenzije tradicionalnoj infrastrukturi manipulisanja signalima, kao na primer:

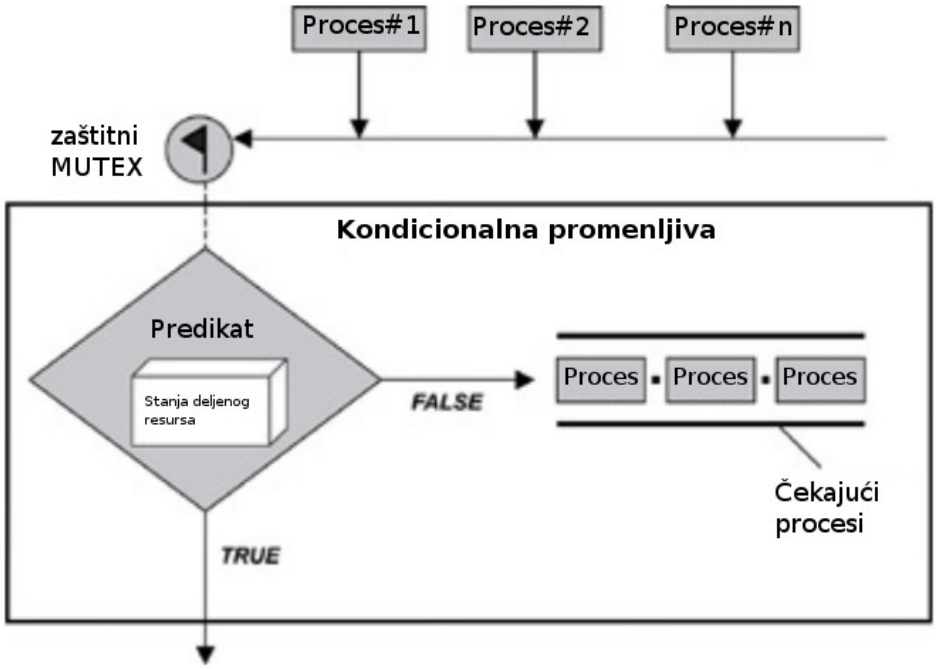
1. Prioritizovano isporučivanje signala bazirano na njihovim prioritetima.
2. Dodatne informacije koje mogu biti asocirane svakom pojedinačnom signalu.
3. Postavljanje višestrukih pristiglih signala istog tipa u red čekanja.

5.5 Kondicionalne promenljive

Procesi često koriste deljene resurse, kao što su datoteke ili komunikacioni kanali. Kada proces treba takav resurs, vrlo često je potrebno da sačeka dok resurs ne stigne u posebno stanje. Najćešće resurs dolazi u traženo stanje pod uticajem nekog drugog procesa. U takvom scenariju, procesu je

potreban način da odredi stanje resursa i jedan način da se to uradi jeste korišćenjem *kondicionalne promenljive*.

Kondicionalna promenljiva je kernel objekat koji je pridružen deljenom resursu, i omogućava procesu da čeka na ostale procese da prevedu deljeni resurs u željeno stanje. Kondicionalna promenljiva može biti pridružena i višestrukim stanjima.



Slika 19: Kondicionalna promenljiva

Kao što je prikazano na slici 19, kondicionalna promenljiva implementira ***predikat***. Predikat je skup logičkih izraza koji se odnose na stanje deljenog resursa i njegova vrednost može biti *true* ili *false*. Proces ima zadatak da evaluira predikat. Ako je vrednost istinita, proces nastavlja sa izvršavanjem. U suprotnom, proces mora da čeka na druge procese koji će promeniti vrednost predikata.

Kada proces proverava kondicionalnu promenljivu, on mora imati ekskluzivno pravo pristupa. U nedostatku istog, drugi proces može promeniti stanje kondicionalne promenljive istovremeno, što će rezultovati time da prvi proces dobije pogrešnu informaciju o stanju promenljive. Dakle, zaštitni muteks se koristi uvek istovremeno sa kondicionalnom promenljivom i on obezbeđuje da će proces imati ekskluzivno pravo pristupa kondicionalnoj promenljivoj sve dok proces ne završi sa njom.

Proces mora najpre da zauzme muteks pre nego što evaluira predikat. Nakon provere predikata, proces oslobađa muteks i, ako je predikat netačan, čeka na uspostaljanje traženog stanja. Uz pomoć kondicionalne promenljive, kernel obezbeđuje da će proces osloboditi muteks i preći u red čekanja na uslov u sklopu jedne atomičke operacije (operacije koja ne može biti prekinuta).

Važno je naglasiti da kondicionalne promenljive nisu mehanizam za sinhronizaciju pristupa deljenom resursu, već ih programeri koriste da omoguće procesima čekanje na događaj koji će prevesti deljeni resurs u željeno stanje.

Kao što je naglašeno ranije, u slučaju kada je vrednost predikata *false*, proces mora da se premesti u listu čekajućih procesa. Nakon što se promeni stanje kondicionalne promenljive, jedan od procesa koji čekaju se aktivira i nastavlja sa izvršavanjem. Kriterijum za odabir procesa koji će nastaviti sa izvršavanjem je ili po prioritetima, ili baziran na FIFO pristupu, ali je uvek definisan od strane datog kernela. Kernel garantuje da će se odabrani proces izmestiti iz liste čekanja, zauzeti muteks i nastaviti svoju operaciju u atomičkom kontekstu. Osnovna karakteristika koncepta kondicionalne promenljive jeste atomičnost otključaj-i-čekaj (unlock-and-wait) i nastavi-i-zaključaj (resume-and-lock) operacija.

Kooperativni procesi definišu stanja deljenog resursa i ova informacija nije deo kondicionalne promenljive, pošto svaki proces definiše različiti predikat ili stanje na koje čeka. Kao rezultat toga, stanje je specifično za proces .

5.5.1 Tipične operacije kondicionalne promenljive

Operacija	Opis
Create	Kreira i inicijalizuje kondicionalnu promenljivu
Wait	Čeka na kondicionalnu promenljivu
Signal	Signalizira kondicionalnoj promenljivoj o prisustvu stanja
Broadcast	Signaliziranje svim čekajućim procesima o prisustvu stanja

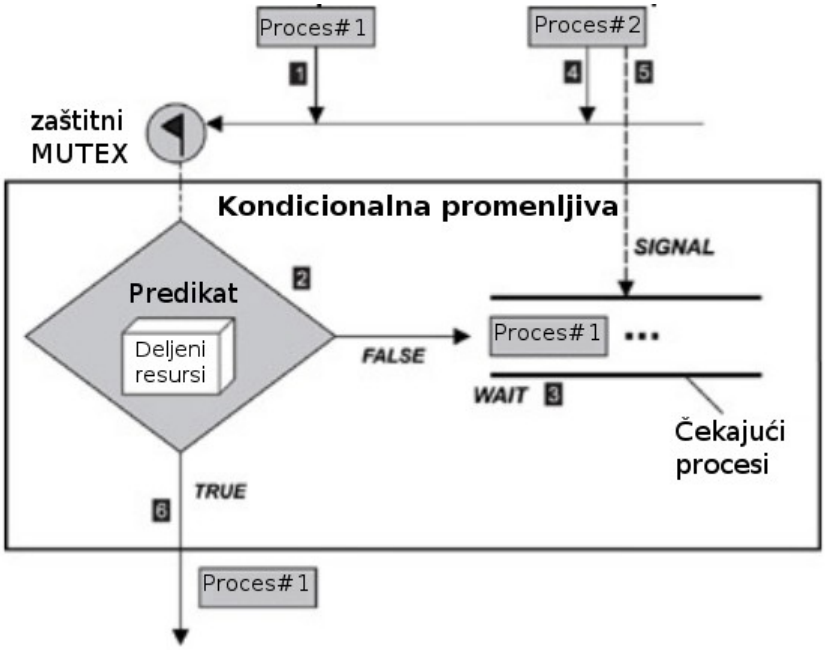
Create operacija kreira kondicionalnu promenljivu i inicijalizuje njen interni kontrolni blok.

Wait operacija omogućava procesu da blokira i čeka na zahtevano stanje deljenog resursa. Da bi pozvao ovu operaciju, proces mora najpre zauzeti uspešno zaštitni muteks. Wait operacija stavlja pozivajući proces u listu čekajućih procesa i oslobađa asocirani muteks u jednoj atomičkoj operaciji.

Signal operacija omogućava procesu da modifikuje kondicionalnu promenljivu u cilju indikacije da je određen uslov dostignut i kao i određeno stanje deljene promenljive. Kako bi pozvao ovu operaciju, signalizirajući proces mora najpre uspešno da zauzme muteks. Signal operacija odblokira jedan od procesa koji čekaju na kondicionalnu promeljivu. Nakon što se završi izvršavanje operacije signaliziranja, kernel ponovo zauzima muteks asociran sa kondicionalnom promenljivom od strane selektovanog procesa i odblokirava ga u jednoj atomičkoj operaciji.

Broadcast operacija budi sve procese koji se nalaze u listi čekajućih procesa kondicionalne promenljive. Jedan od njih je odabran od strane kernela i dodeljen mu je zaštitni muteks. Svi ostali procesi su uklonjeni iz liste čekajućih procesa kondicionalne promenljive i premešteni u listu čekajućih procesa na muteks.

5.5.2 Tipične upotrebe kondicionalne promenljive



Slika 20: Primer upotrebe kondicionalne promenljive

```
Proces 1
Lock mutex
     Examine shared resource
     While (shared resource is Busy)
          WAIT (condition variable)
     Mark shared resource as Busy
Unlock mutex
Proces 2
Lock mutex
     Mark shared resource as Free
     SIGNAL (condition variable)
Unlock mutex
```

U pseudo kodu iznad:

1. proces 1 zaključava zaštitni muteks u prvom koraku
2. Proverava stanje deljenog resursa i pronalazi da je on u stanju BUSY
3. Zahteva Wait operaciju kako bi čekao na resurs da postane dostupan (*Free*)

4. Prevođenje resursa u stanje *Free* se vrši od strane procesa 2, nakon što završi sa korišćenjem resursa. Kako bi promenio stanje, proces 2 prvo zaključa muteks, označava resurs kao free i signalizira procesu 1 da je stanje free dostignuto

Signal o stanju kondicionalne promenljive je izgubljen ako niko ne čeka na njega. U skladu sa tim, proces uvek treba da proveri stanje pre nego što krene da čeka na njega. Takođe, ista provera treba da se izvrši i nakon buđenja procesa, kao preventiva nevalidnih signala za kondicionalnu promeljivu. Iz ovog razloga pseudo kod iznad uključuje while petlju kako bi proverio prisustvno stanja.