Sistemi za rad u realnom vremenu

UVOD

Sistemi za rad u realnom vremenu su računarski sistemi koji upravljaju i nadgledaju fizičke procese. RTS su obično sastavni deo nekog većeg sistema ili uređaja i iz tog razloga se nazivaju ugradjeni (ili *embedded*) mikroprocesorski sistemi. RTS, za razliku od računarskih sistema opšte namene, projektuju se za unapred odredjenu primenu, koja diktira njihovu kako hardversku, tako i softversku strukturu. Ključni deo specifikacije RTS se odnosi na vreme odziva, koje je određeno prirodom fizičkog procesa. Korektnost rada RTS zavisi ne samo od logičke ispravnosti rezultata izračunavanja, već i od vremena za koje se ti rezultati generišu. Drugim rečima, da bi korektno obavio svoj zadatak, RTS mora svoju funkciju da izvrši u unapred zadatom vremenu. Ukoliko se zadato vreme odziva prekorači može doći do otkaza sistema što može imati katastrofalne posledice. Takođe, od RTS se obično zahteva visoko-pouzdani rad i predvidljivo ponašanje u svim okolnostima. U slučajevima kada se RTS ugrađuje u uređaje široke potrošnje, iz razloga ekonomičnosti, zahteva se minimalno korišćenje hardverskih resursa (8/16 bitni CPU, mala koli~ina memorije). Zbog svega navedenog, projektovanje RTS sistema predstavlja složen zadatak.

RTS se sastoji od *upravljačkog* i *upravljanog* sistema. Upravljački sistem (ili *embedded system*) predstavlja elektronski podsistem fizičkog sistema. Upravljani sistem se može posmatrati kao *okruženje* sa kojim upravljački sistem interaguje. Na primer, u slučaju automatizovane fabrike, upravljani sistem je fabrički pogon koji se sastoji od: robota, manipulatora, pokretnih traka, itd, dok upravljački sistem čini računar zajedano sa interfejsom prema operateru, koji upravlja i koordiniše sve aktivnosi u fabričkom pogonu.

Osnovne funkcije RTS sistema su sledeće:

1. Prikuplja informacije o tekućem stanju fizičkog sistema, npr. nadgleda fizičke veličine kao što su: temperatura, pritisak, hemijski sastav.
2. Obrađuje prikupljene informacije na bazi matematičkog modela fizičkog sistema.
3. Generiše izlazne signale koji utiču na ponašenje fizičkog sistema, a u cilju upravljanja ili ostvarivanje neke zadate performansne mere.

Upravljački sistem interaguje sa svojim okruženjem na bazi informacija o okruženju prikupljenih posredstvom *senzora*. Od ključne važnosti za ispravan rad RTS je da stanje okruženja, kako ga upravljački sistem "vidi", bude usklađeno (konzistentno) sa stvarnim stanjem okurženja. U suprotnom, efekti akcija, koje upravljački sistem peduzima na bazi pogrešne predstave o tekućem stanju okruženja, mogu imati katastrofalne posledice. Iz tog razloga, neophodno je obezbediti periodično prikupljanje i pravovremenu obradu informacija. Drugi aspekt vremenske korektnosti RTS odnosi se na fizičke posledice koje mogu u okruženju prouzrokovati nepravovremene akcije upravljačkog sistema. Na primer, ako računar koji upravlja robotom ne izda na vreme komandu za zaustavljanje ili promenu smera kretanja, robot se može sudariti sa nekim durgim objekom koji se nalazi u fabričkom pogonu.

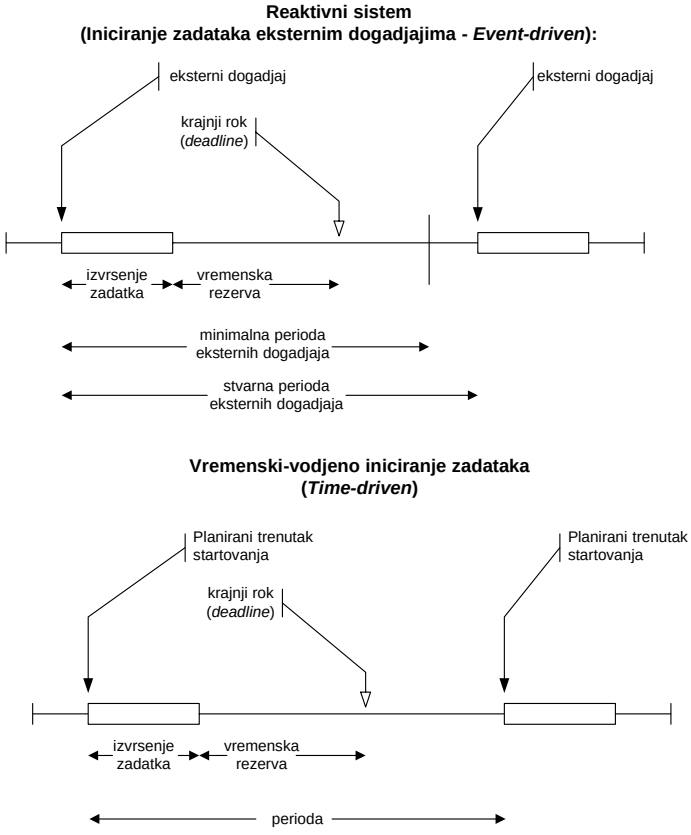
Kod većine RTS, aktivnosti sa pridruženim vremenska ograničenja, koegzistiraju sa aktivnostima za koja takva ograničenja ne postoje ili nisu kritična. Za oba tipa aktivnosti koristićemo termin *zadatak* (ili *real-time task*). Zadatak je softverski modul koji se može pozvati u cilju izvršenja određene funkcije. Zadaci sa vremenskim ograničenjima su *vremenski-kritični* ili *hard-real-time* zadaci, dok su ostali zadaci *soft* zadaci. Idealno, mikroprocesorski sistem bi trebalo da izvršava vremenski-kritične zadatke tako da *svaki*

takav zadatak zadovolji svoja vremenska ograničenja, dok vremenski-nekritične zadatke izvršava tako da se minimizuje njihovo srednje vreme odziva. Upravo neophodnost zadovoljenja vremenskih ograničenja **pojedinačnih** vremenski-kritičnih zadataka predstavlja glavnu teškoću prilikom projektovanja RTS.

Vremenski parametri RTS sistema

Tipično, RTS obavlja veći broj zadataka (aktivnosti) od kojih se svaki inicira kao posledica ispunjenja nekog unapred definisanog uslova. Sa stanovišta načina iniciranja razlikuju se sledeća dva tipa zadataka:

- *reaktivni* (*event-driven* ili *asinhroni*). Zadatak je iniciran pojavom određenog događaja (eksternog ili internog). Pri tome, tačan vremenski trenutak pojave događaja kao i sled događaja nije unapred poznat, već se eventualno poznaje minimalno vreme između dva događaja, prosečno vreme između dva događaja i sl. Zato se ovi zadaci nazivaju i *aperiodični* ili *asinhroni* zadaci.
- *periodični* (*time-driven ili sinhroni*). Zadatak se ponavljaja u regularnim vremenskim intervalima, a perioda iniciranja zadatka se zadaje unapred, u fazi projektovanja sistema. Zato se ovi zadaci nazivaju *sinhrnim* zadacima. Tipično, zadaci koji prikupljaju informacije od senzora su po svojoj prirodi periodični.



Sl. 1 Reaktivni i periodični zadaci

Na Sl. 1 su prikazani vremenski parametri reaktivnih i periodičnih zadataka. Za svaki zadatak, definiše se krajnji rok (*deadline*) kao trenutak do koga izvršenje zadatka mora biti završeno. Vreme izvršenja zadatka mora biti kraće od vremenskog intervala između iniciranja zadatka i njegovog krajnjeg roka. Razlika ova dva vremena predstavlja *vremensku rezervu*.

Kod reaktivnih RTS, vremenski trenutak ponovnog iniciranja zadatak nije unapred poznat, osim što se zna da vremenski interval između dva iniciranja ne može biti kraći od nekog minimalnog, unapred poznatog vremena. Kod vremenski vođenih RTS, zadatak se startuje u pravilnim vremenskim intervalima, pri čemu se, najčešće, krajnji rok poklapa sa trenutkom sledećeg iniciranja.

Kod reaktivnih RTS, startovanje zadataka je inicirano događajima iz okruženja (aperiodični ili asinhroni zadaci), dok se kod vremenski-vodjenih RTS, zadaci startuju periodično. Kod *multitasking* RTS, u sistemu postoji veći broj zadataka koji se izvršavaju nezavisno jedan od drugoga. Ukolko RTS poseduje jedan CPU (jednoprocesorski sistem), u jednom trenutku može se izvršavati najviše jedan zadatak. To znači da ako u isto vreme postoje zahtevi za izvršenjem više od jednog zadatka, svi inicirani zadaci, osim jednog, moraju da čekaju na oslobađanje CPU-a. Odlaganje startovanja zadatka, zbog zauzetosti CPU-a izvršenjem drugih zadataka, može da dovede do prekoračenja krajnjeg roka. Kod većine RTS sistema, redosled izvršenja iniciranih zadataka kontroliše se uz pomoć *prioriteta*. Naime, svakom zadatku se pridružuje nenegativan ceo broj (prioritet) koji odražava stepen hitnosti (kritičnosti) zadatka. Uvek kada postoji više od jednog iniciranog zadatka, za izvršenje se bira zadatak najvišeg prioriteta.

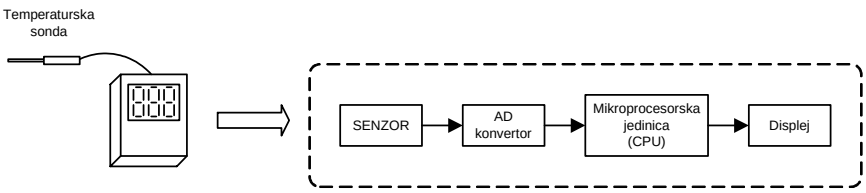
Hard- i soft-RTS

Za zadatak koji nakon iniciranja mora biti završen u nekom unapred zadatom vremenu, kaže se da je *hard*, ili da ima fiksni krajnji rok (*hard deadline*). Prekoračenje krajnjeg roka dovodi do delimičnog ili potpunog otkaza sistema. Rezultati izračunati nakon krajnjeg roka smatraju se pogrešnim ili neupotrebljivim. Ukoliko u RTS sistemu postoji barem jedan zadatak sa fiksnim krajnjim rokom, takav sistem se naziva *hard-RTS*. To ne isključuje postojanje zadatak (aktivnosti) kojima nisu pridruženi fiksni krajnji rokovo, medjutim, sa stanovista korektnog funkcionisanja sistema u smislu pravovremenosti odziva, od ključnog značaja su upravo *hard* zadaci.

Ukoliko u RTS sistemu ne postoje zadaci sa fiksnim krajnjim rokovima, takav sistem se naziva *soft*-RTS. Zadacima u *soft* RTS, pridružen su krajnji rokovi (tzv. *soft-deadline*), ali prekoračenje krajnjeg roka ne dovodi do otkaza sistema, već samo do degradacije nekih performansi sistema. Drugim rečima, prekoračenje krajnjeg roka nije poželjno, ali se može, do izvesne mere, tolerisati.

PRIMER 1.

Razmotrimo primer jednosatavnog RTS čija je struktura prikazana na Sl. 2. Radi se o uređaju za merenje temperature i prikazivanje rezultata merenja na cifarskom displeju. Sistem se sastoji od: temperaturske sonde, A/D konvertora, mikroprocesorsa i cifarskog displeja. Rad sistema (program) je organizovan tako da se u "beskonačnoj" petlji, jedna za drugom, neprekidno obavljaju tri aktivnosti: očitavanje digitalizovana vrednost izmerene temperature, konverzija očitane brojne vrednosti u oblik pogodan za prikaz na displeju i prikazivanje rezultat na cifarskom displeju.



Sl. 2 Jednostavan RTS sistem.

DO FOREVER
{

1. Očitati digitalizovanu vrednost izmerene temperature (npr. 12-bitna neoznačena vrednost);
2. Konvertovati pročitanu vrednost u format pogodan za prikaz na displeju (npr. 3 BCD cifre, XY.Z ^{0}C)
3. Postaviti novi prikaz na displej.

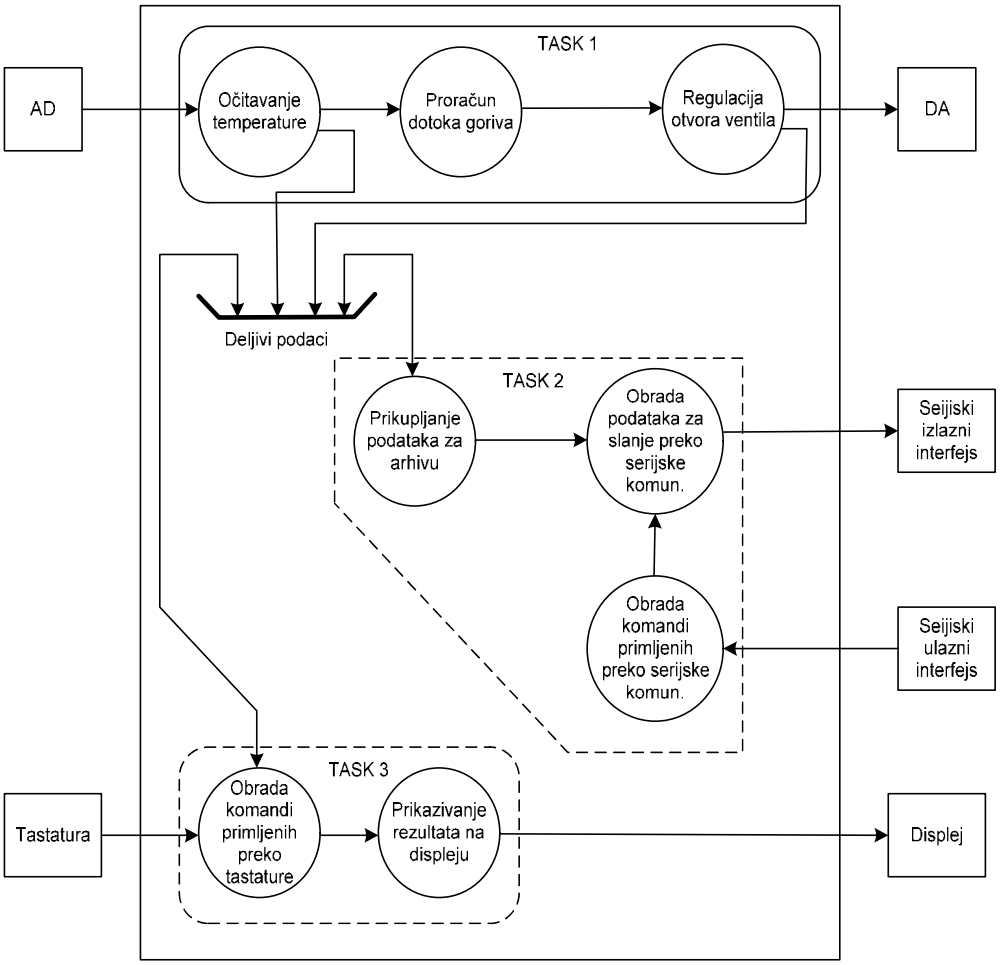
}

U ovom primeru se radi o sekvencijalnom (*single-thread*) programu, tj. program se sastoji od jedinsvene, neprekidne sekvence instrukcija koje se izvršavaju u "beskonačnoj" (ili kontinualnoj) petlji. Naravno, sekvencijalnost programa ne isključuje korišćenje potprograma i procedura. (U konkretnom primeru, programer bi mogao da koristi posebne potprograme za očitavnje AD konvertora i postavljanje novog prikaza na displej.) Kod ovog sistema ne postoje stroga vremenska ograničenja, s obzirom da se rezultat dobija brzo, što dodatno pojednostavljuje projektovanje softvera.

Međutim, iako se radi o jednostavnom sistemu (hadverski i softverski), čak i pod pretpostavkom da se za razvoj softvera koristi neki viši programski jezik (npr. C), programer mora dobro da poznaje strukturu i način funkcionisanja hardvera. tj. projektant mora biti ekspert kako za hardver tako i za softver.

PRIMER 2.

Razmotrimo, sada, jedan nešto složeniji RTS (Sl. 3), koji je po strukturi sličan najvećem broju RTS niske i srednje složenosti. Radi se o sistemu za upravljanje kotlom na tečna goriva. Temperatura se meri temperaturskim senzorom, a rezultujući analogni signal se digitalizuje uz pomoć A/D konverora. Mikorprocesor izračunava signal greške (razliku između zadate i izmerene temperature) i na bazi toga izračunava upravljački signala koji se preko D/A konvertora vodi do ventila, a čija vrednost određuje otvor ventila, tj. protok goriva (funkcija koja odgovara zadatku TASK 1). Pored ovog glavnog zadatka, od sistema se zahteva da obavi i veći broj sekundarnih zadataka. Prvi se odnosi na interakciju sa operaterom, koji posredstvom tastature i displeja treba da ima mogućnost zadavanja temperture i uvida u sve druge informacije o radu sistema (zadatak TASK 3). Drugo, sistem treba da arhivira podatke koje se odnose na rad sistema, kao što su aktivnosti operatera, izmerene vrednosti temperature. Treće, sistem treba da ima mogućnost serijske komunikacije sa nadređenim računarom, u cilju prenosa arhiviranih podataka i upravljanja sa udaljenog mesta) - odgovara zadatku TASK 2.



Sl. 3 Sistem za upravljanje kotlom na tečna goriva.

Mada je po hardverskoj strukturi sistem tek nešto složeniji od sistema iz prethodnog primera, softver razmatranog mikroprocesorskog sistema je značajno komplikovaniji, kako u pogledu funkcije tako i u pogledu unutrašnje organizacije. Organizacija softvera u vidu sekvencijalnog programa više nije najbolje rešenje. Problem predstavljaju zahtevi za obavljanjem većeg broja različitih zadataka, od kojih se neki iniciraju "*asinhrono*" (tačan trenutak kada se javlja potreba za izvršenjem zadatka nije poznat) pod uticajem dogadjaja iz okurženja (kao što je npr. zadatak obrade komande zadate preko tastature ili zahtev za serijskom komunikacijom), dok se drugi startuju periodično u fiksnim, unapred poznatim vremenskim intervalima (*sinhrono*), kao što je npr. zadatak očitavanja temperature u kotlu i izračunavanje vrednosti upravljačkog signala za ventil. Pod ovim uslovima, mogu se javiti situacije kada je potrebno istovremeno (tj. *konkurentno* ili *paralelno*) izvršavati dva ili više zadatka, npr. ako se u toku izračunavanja vrednosti upravljačkog signala, javi zahtev sa tastature. S obzrom da se radi o jednoprocesorskom sistemu, u jednom trenutku može biti izvršavan samo jedan zadatak, dok svi ostali zadaci koji su spremni za izvršenje moraju da čekaju da CPU postane slobodan (tzv. *kvazi-konkurentnost*). Sistemi koji podržavaju kvazi-konkurentnost su *multitasking* sistemi. Problem multitaskinga se uobičajeno rešava pomoću *prekida* (*interrupt*-a), tako što se zadaci realizuju kao rutine za obradu prekida iniciranih odgovarajućim događajima (npr. pritisak na dirku tastature, prekid od tajmera, prekid od komunikacionog interfejsa).

Odlaganje izvršenja pojedinih zadatka zbog zauzetosti CPU-a može da dovede do otkaza sistema ukoliko je odziv koji se određuje zadatkom vremenski kritičan (npr. upravljački signal za ventil mora biti izračunat unutar vremena koje je specificirano algoritmom automatskog upravljanja - npr. ne sme biti duže od perode odmeravanja ulaznog signala). Međutim, nisu svi zadaci vremenski kritični, u smislu da se kod njih može tolerisati i duže kašnjenje (npr. prikaz na displej). S tim u vezi javlja se problem *planiranja izvršenja zadataka*, na način koji će obezbediti da svi zadaci zadovolje zahteve u pogledu brzine odziva. Šta više, RTS treba da **garantuje** da će krajnji rokovi vremenski-kritičnih zadataka biti ispoštovani u svim okolnostima. Uobičajno, ovaj problem se rešava uvođenjem *prioriteta* zadataka. Naime, svakom zadatku, u skladu sa stepenom njegove kritičnosti, dodeljuje se prioritet, a uvek kada postoji više od jednog iniciranog zadatka, za izvršenje se bira zadatak najvišeg prioritetom.

Takođe, zadaci ne moraju biti nezavisni, već može postojati potreba da u toku rada sistema različiti zadaci na neki način interaguju, komuniciraju i sinhronišu. Npr. vrednost upravljačkog signala izračunata od strane TASK_1, zajedno sa izmerenom temperaturom, se arhivira od strane TASK_2. To nameće potrebu da TASK_1, na neki način dostavi podatke TASK_2 i da ga obavesti o tome. U bliskoj vezi sa interakcijom zadataka je problem *deljivih resursa*. Za komunikaciju između zadataka uobičajeno se koriste tehnike zasnovane na *prenosu poruka* ili na *deljivim promenljivim*. Za sinhronizaciju zadataka koriste se mehanizmi kao što su *signali, semafori*, *kritične sekcije.*

Operativni sistemi za rad u realnom vremenu (RTOS)

Sve napred navedeno ukazuje na složenost razvoja RTS aplikacija, kao i na suštinske razlike između aplikacija koje koriste sisteme opšte namene. Takođe, uočava se potreba za podrškom, na sistemskom nivou, za impementaciju napred navedenih mehanizama i tehnika. Takva podrška je operativni sistem, ili tačnije operativni sistem za rad u realnom vremenu (*Real-Time Operating System - RTOS)*. RTOS treba da u što većoj meri "sakrije" hardversku složenost sistema od programera (detaljno poznavanje prekida, A/D konvertora, tajmera), tj. da za pristup hardveru obezbedi standardizovane softverske procedure. Takođe, RTOS treba da obezbedi podršku za multitasking, komunikaciju i sinhronizaciju zadataka, kontrolu pristupa deljivim resurasima… Na taj način, RTOS stvara privid "virtuelne" mašine, nezavisne od konkretne hardverske platoforme, što mogućava projektantu da se usredsredi na rešavanje konkretnog problema na višem nivou (identifikacija zadataka, komunikacija/sinhronizacija zadataka, obezbeđivanje zahtevanih performansi…) i da se u što većoj meri oslobodi implementacionih detalja niskog nivoa.

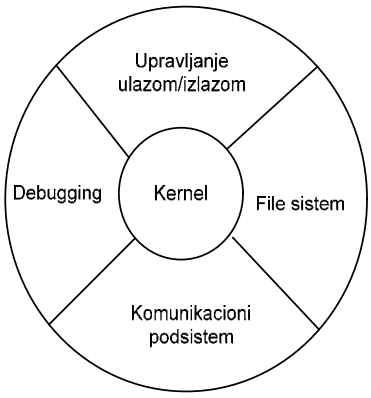
Prednosti korišćenja RTOS-a su:

1) smanjuje cena razvoja (skraćuje se vreme realizacije korisničke aplikacije);
2) povećava pouzdanost sistema, jer se smanjuje verovatnoća programerske greške prilikom programiranja mehanizama niskog nivoa;
3) olakšana portabilnost (prenosivost) - prenos aplikacije na drugu hardversku platformu (npr. dugi CPU).

Struktura RTOS

RTOS pruža brojne servise koji u velikoj meri olakšavaju razvoj RTS aplikacija, tako što obezbeđuju podršku za standardizovan pristup hardveru, konkurentno izvršavanje zadataka, upravljanje resursima. Struktura RTOS sistema prikazana je na Sl. 4. RTOS se projektuje tako da ima modularnu i slojevitu strukturu (arhitekturu). "Srce" RTOS-a čini tzv. jezgro

(*kernel*) koje obezbeđuje podršku za: planiranje izvršenja zadataka, upravljanje memorijom, zaštitu deljivih resursa, obradu prekida, komunikaciju i sinhronizaciju zadataka. Deo "upravljanje ulazom-izlazom" obezbeđuje drajvere standardnih hardversikih uređaja i omogućava hardversku nezavisnost. *File* sistem pruža podršku za kreiranje i brisanje fajlova i direktorijuma, upis/čitanje fajlova kod sistema koji poseduju *hard*-disk. Mrežni nivo implementira komunikacione protokole niskog nivoa i drajvere za mrežnu komunikaciju. *Debugging* nivo obezbeđuje nadgledanje izvršenja zadataka, startovanje/zaustavljanje zadataka što se koristi u toku razvoja i testiranja korisničke aplikacije. Većina RTOS je tako organizovana da se moduli, ili čak celi nivoi, mogu izostaviti, ako nisu neophodni, prilikom instalacije OS na ciljnoj hardverskoj platformi. Ovo je bitna karakteristika, s obzirom da RTS u najvećem broju slučajeva poseduju ograničene hardverske resurse (npr. količina memorija), pa je njihovo ekonomično korišćenje od suštinskog značaja. Ovakva organizacija RTOS se naziva *microkernel architectura*. Kod microkernel RTOS, kernel je obavezni deo OS; on je optimizovan za konkretan CPU, tipično je razvijen u asembleru kako bi bio male veličine i velike brzine rada. Ostali delovi RTOS su obično dostupni projektantu korisničke aplikacije u vidu izvornog koda (npr. u C-u) kao biblioteke funkcija i procedura, koje projektant može po potrebi da uključi u svoj projekat.



Sl. 4 Struktura RTOS-a.

Hardverska nezavisnost

Jedna od primarnih prednosti korišćenja RTOS je dostupnost tzv. drajvera uređaja (*device drivers*). To su softverske rutine koje obezbedjuju pristup i korišćenje različitih tipova istih hardverskih uređaja na standardizovan način. Drajveri uređaja sakrivaju detalje vezane za specifičnosti konkretnog tipa hardverskog uređaja, pružajući tako utisak programeru da radi sa standardnim uređajem. Tipično, RTOS obezbeđuju *drajvere* za sledeće hardverske uređaje: štampač, dispej, tastatura, miš, tajmer, komunikacioni kontroleri, mrežni kontroleri, magistrale. Npr. drajver dispeja obezbeđuje rutine kao što su: ispisivanje karaktera na zadatu poziciju, brisanje displeja, brisanje jedne linije dispeja, ispisivanje jedne linije na displeju, rutine za upravljanje kurzorom, itd. Korišćenje device driver-a obezbeđuje da je aplikacioni deo softvera nezavistan od konkretnog tipa hardverskog uređaja. Na primer, ukoliko se pojavi potreba za ugradnjom drugog tipa displeja (npr. sa većom rezolocijom), biće neophodno instalirati novi *disply-driver*, ali neće biti potrebe vršiti bilo kakve izmene u korisničkoj aplikaciji.

Upravljanje konkurentnim izvršenjem zadataka

Konkurentnost predstavlja mogućnost istovremenog izvršenja više zadataka. "Prava" konkurentnost može se ostvariti samo kod multiprocesorskih sistema, kod kojih se svaki zadatak izvršava na posebnom procesoru. Međutim, ovakvo rešenje je izrazito

neekonomično. Broj zadataka u jednom tipičnom RTS može biti veoma veliki. Takođe, dodela pojedinačnih zadataka zasebnim procesorima, uslovila bi veoma nizak nivo iskorišćenosti procesorskih resursa, s obzriom da po pravilu važi da je u toku rada sistema većina zadatak najveći deo vremena neaktivna, a da se aktiviraju samo po potrebi. (Npr. kod primera 2, zadatak koji obrađuje unos sa tastature aktivan je samo dok operater menja vrednost zadate temperature, a sve ostalo vreme je neaktivan.) Iz ovih razloga, većina savremenih RTS su jednoprocesorski sistemi, a konkurentnost (tačnije, privid konkurentnosti) se postiže alternativnim izvršavanjem aktivnih zadataka (tzv. kvazi-konkurentnost ili *multitasking*). Multitasking se obezbeđuje na nivou kernela. RTOS autonomno planira redosled izvršenje zadatak i obezbeđuje mehanizme za kreiranje, startnovanje i suspenziju zadataka. Sve ove aktivnosti, neophodne za obezbeđivanje multitaskinga, troše dodatno CPU vreme. Takođe, činjenica da više nezavisnih zadataka dele iste hardverske resurse dovodi do pojave problema koji ne postoje kod "prave" konkurentnosti, kao što su otežano ostvarivanje zahteva koji se odnose na brzinu odziva, razrešavaljna konflikta oko korišćenja deljivih resursa, upravljanje dodelom memorije pojedinačnim zadacima i sl.

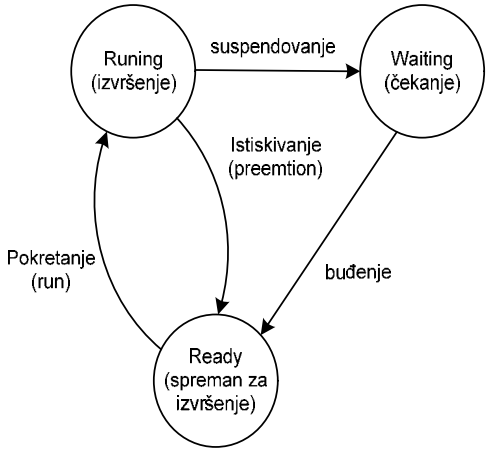
Upravljanje multitaskingom, zasnovano je na konceptu *stanja* zadataka. Naime, u toku rada sistema, svaki zadatak u svakom trenutku se nalazi u jednom od sledeća tri stanja:

1. *Running* - zadatak se trenutno izvršava od strane CPU-a. U jednoprocesorskom, multitasking sistemu, u svakom trenutku, najviše jedan zadataka može biti u ovom stanju.
2. *Ready* - zadatak je aktiviran i spreman za izvršenje; svi resursi neophodni za izvršenje zadatka, osim CPU, su obezbeđeni (memorija). U multitasking sistemu, proizvoljan broj zadataka može biti u ovom stanju i svi oni su uređeni u tzv. *red čekanja* (*queue*). Zadatak koji se nalazi na početku (vrhu) reda čekanja je prvi sledeći zadatak koji će biti izvršen kada se završi izvršenje tekućeg *Running* zadatak.
3. *Waiting* - zadatak je neaktivan i čeka na događaj koji će ga aktivirati. (Kaže se da u ovom stanju zadaci "spavaju" i čekaju na događaj (signal) koji će ih "probuditi"). Događaj može biti: isteklo zadato vreme, signal iz okruženja (npr. od senzora), interni signal generisan od strane nekog aktivnog zadatka. Broj zadataka koji se nalazi u stanju *Waiting* nije ograničen, a među njim ne postoji nikakvo uređenje.

Na Sl. 5 prikazan je dijagram prelaza koji opisuje način upravljanja izvršenjem zadataka kod multitasking RTOS-a. Izvršenje zadatka koji se trenutno izvršava (*Running* zadatak) može biti prekinuto, a zadatak suspendovan i postavljen u stanje *Waiting,* kada je zadatak završio sve predviđene aktivnosti (*termination*), ili ako je zadatak uputio *sistemski poziv* (*system call*)[1]. Takođe, zadatak može "samoinicijativno" da pređe u stanje *Waiting* kada je njegovo izvršenje došlo do tačke kada se čeka na završetak neke dugotrajne U/I aktivnosti. Nakon što je U/I aktivnost obavljena, zadatak se "budi" i postavlja u stanje *Ready*. Izvršenje zadatka može biti privremno prekinuto i pre njegovog završetka ukoliko je prekoračeno maksimalno vreme zauzeća CPU-a ili se u redu čekanja *Ready* zadataka pojavio zadatak sa višim prioritetom od *Running* zadatka. U takvim situacijama, kaže se da je došlo do *istiskivanja* (*preemption*) zadatka. Prebacivanje zadatka iz stanja *Ready* u *Running* (aktivnost *Run*) i iz stanja *Running* u stanje *Ready* (*preempt*), obavlja proces kernela koji se naziva *dispečer* (*dispetcher*). Obe ove aktivnosti uključuju promenu konteksta (*task switching*).

[1] Pojam sistemski poziv se odnosi na sitaciju kada korisnički zadatak poziva neki servis OS, koji je sam za sebe realizovan kao konkurentni zadatak.

Uređenjem reda čekanja *Ready* zadataka upravlja proces kernela koji se naziva *planer* (*scheduler*). Planer se aktivira uvek kada neki zadatak ulazi u stanje *Ready* (bilo da je probuđen i dolazi iz stanja *Waiting*, bilo da je istisnut i dolazi iz stanja *Running*). Uloga planera je da novi *Ready* zadatak, a uskladu sa njegovim prioritetom, ubaci na odgovarajuću poziciju u redu čekanja (rasporedi zadatak). Način na koji se određuje prioritet zadatka zavisi od *politike (ili strategije*) *planiranja* koju planer sprovodi. Od implementirane politike palaniranja u mnogome zavisi da li će zahevi u pogledu brzine odziva biti zadovoljeni.



Sl. 5 Stanja zadatka u RTOS

Planiranje izvršenja zadataka

Planiranje izvršenja zadataka kod većine RTOS zasnovano je na prioritetima. Naime, svakom zadatku se pridružuje prioritet (nenegativan ceo broj); uvek kada postoji više od jednog spremnog zadatka, za izvršenje se bira zadatak sa najvišim prioritetom. Pri tome, generalno, postoje dve strategije planiranje:

1. *Planiranje bez istiskivanja*. Kada izvršenje zadatka počne ono se nastavlja sve dok se zadatak u potpunosti ne izvrši, ili iz nekog drugog razloga pređe u stanje *Waiting*. Drugim rečima, dispečer nema mogućnost da prekine izvršenje započetog zadatka.
2. *Planiranje sa istiskivanjem*. Izvršenje zadatka može biti prekinuto ukoliko se pojavi zadatak višeg prioriteta. Na primer, neka se izvršava zadatak A sa prioritetom 6 i neka su zadaci B i C spremni za izvršenje i imaju prioritete 8 i 10, respektivno. U ovakvoj situaciji, zadatak A biće istisnut i započeće izvršenje zadatka C. Ukoliko se u međuvremenu ne pojavi zadatak za prioritetom većim od 10, zadatak C se izvršava do kraja. U tom trenutku, RTOS kao sledećeg za izvršenje bira zadatak B i tek nakon njegovog završetka, restartuje zadatak A, koji konačno ima šansu da kompletira svoje izvršenje.

Planiranje bez istiskivanja je jednostavnije za realizaciju, međutim kod *hard*-RTS može da dovede do prekoračenja fiksnih krajnjih rokova zadataka visokog prioriteta koji su aktivirani u vremenu dok se izvršava zadatak niskog prioriteta. Na primer, neka je u trenutku t=0, startovano izvršenje zadatka A, čiji je prioritet 5 i vreme izvršenja 100ms. U trenutku t=10ms, kao reakcija na neki urgentni ekserni događaj iniciran je zadatak B, prioriteta 20, vremena izvršenja 10ms, sa fiksnim krajnjim rokom t=50ms (zadatak B mora biti završen do trenutka t=50ms, u suprotnom dolazi do otkaza sistema). Očigledno, kako ne postoji mogućnost da se jednom započeti zadatak prekine, zadatak B prekoračuje svoj krajnji rok. Zbog toga, kog većine RTOS sistema koristi se planiranje sa istiskivanjem koje obezbeđuje

brzu reakciju na događaje visokog prioriteta. Cena koja se plaća je povećano "ekstra" vreme potrebno za čestu promenu konteksta.

Međutim, planiranje sa istiskivanjem, u opštem slučaju, ne rešava problem prekoračenja fiknih krajnjih rokova. Pretpostavimo da se u prethodno opisanom primeru, u trentku t=20ms pojavio zadatak C višeg prioriteta od zadatka B, recimo 30, čije je vreme izvršenja 80ms i krajnji rok u trenutku t=120ms. Pod pretposatvkom da se koristi istiskivanje, u trenutku t=20ms zadatak B biva istisnut zadatkom C, koji zauzima CPU sve do svog okončanja, tj. do trenutka t=100ms. Očigledno u trenutku t=50ms dolazi do otkaza sistema (zadatak B je prekoračio svoj krajnji rok).

Razmatrani primer pokazuje da rešenje problema ostvarivanje zahteva u pogledu vremena odziva vremenski-kritičnih zadataka, treba tražiti u kombinaciji istiskivanja i manipulacije prioritetima zadataka. Ovaj problem je težak, s obzirom da se kod *hard*-RTS od projektanta zahteva *garancija* da će fiksni krajnji rokovi biti *uvek* ispoštovani, bez obzira na redosled i vremena iniciranja zadataka. Pri tome, tačni vremenski trenuci iniciranja zadataka obično nisu unapred poznati (jer su posledica asinhronih događaja koji potiču iz okruženja i nad kojima RTS nema direktnu kontrolu).

Generalno, postoje dva načina za dodelu prioriteta zadacima:

1. *Statički (ili fiksni) proriteti*. Prioriteti se dodeljuju zadacima u fazi projektovanja sistema i ne mogu se menjati u toku rada sistema. Prioriteti se dodeljuju na bazi procenjene *kritičnosti* (ili *urgentnosti*) zadatka. Kriterijumi za procenu kritičnosti mogu biti: učestanost iniciranja zadatka (veća učestalonost, viši proritet), urgentnost (kraće zahtevano vreme odziva, viši prioritet), vreme izvršenja (duže vreme izvršenja, viši prioritet). Tipično, prioritet se određuje na bazi jednog ili kombinacijom više navedenih kriterijuma (npr. količik vremena izvršnja i vremena odziva).
2. *Dinamiči (ili promenljivi) prioriteti*. Prioriteti zadataka nisu fiksni, unapred dodeljeni zadacima, već se određuju i menjaju u toku rada sistema, dinamički, od strane RTOS-a (tj. dispečera), na bazi trenutnog stanja sistema. Dispečer neprekidno prati "razvoj situacije" i pokušava da organizuje izvršenje spremih zadataka na način koji će obezbediti da svi oni ispoštuju zadate krajnje rokove. Informacije koje su pri tome dostupne dispečeru, a na bazi kojih on donosi svoje odluke (koji zadatak izvršavati, kada istisnuti zadatak koji se trenutno izvršava) su: očekivana vremena izvršenja zadatak i njihovi krajnji rokovi.

Način na koji se određuju statički prioriteti, u prvom slučaju, odnosno način na koji dispečer donosi odluke, u drugom slučaju, naziva se *algoritam planiranja.* Tako, razlikujemo *statičke* i *dinamičke* algoritme za planiranje.

Kao što je već napomenuto, kod *hard*-RTS od ključne važnosti je mogućnost garantovanja korektnog rada sistema, u smislu poštovanja fiksnih krajnjih rokova. Idealno, ova garancija bi trebalo da bude 100 procentna, tj. analitički (matematički) dokazna. Nažalost, garanciju ovog tipa moguće je postići samo u slučajevima nekih jednostavnih algoritama planiranja i restriktivnih tipova RTS (kao sto su vremenski-vodjeni RTS). U nastavku će biti navedena dva takva algoritma, pri čemu je prvi statički, a drugi dinamički.

Rate Monotonic Scheduling (RMS). Primenjuje se kod RTS sistema koji se sastoji od n periodičnih (vremenski-vođenih) zadataka, $Z_1, Z_2, \ldots, Z_n$. Za svaki zadatak Z_k poznato je vreme izvršenja C_k i perioda T_k. Polazna pretpostavka je da se krajnji rok zadatka poklapa sa njegovom periodom, tj. svaki zadatak mora biti završen pre isteka tekuće periode. Kod RMS algoritma prioriteti (statički) određuju se na bazi periode zadataka, tako što zadatak sa

najkraćom perodom dobija najviši prioritet. Može se dokazati da je RTS sistem koji koristi RMS algoritam planiranja garantovano korektan ukoliko je ispunjena sledeća nejednakost:

$$\sum_{j=1}^{n}\left(\frac{C_j}{T_j}\right) \le n\left(2^{\frac{1}{n}} - 1\right) \quad (1)$$

U svojoj bazičnoj formi RMS algoritam je primenljiv samo na periodične zadatke. Međutim, većina RTS predstavlja kombinaciju periodičnih i reaktivnih (asinhronih) zadataka. Iako asinhroni zadaci nemaju fiksu periodu iniciranja, obično je poznato minimalno vreme između dva iniciranja (vidi Sl. 1). Tako, RMS algoritam se može proširiti i na slučaj kombinovanih RTS sistema, ako se prilikom analize asinhroni zadaci trentiraju kao periodični sa periodom jednakom minimalnom vremenu između dva iniciranja.

Postavlja se sledeće logično pitanje: RMS algoritam je jednostavan za implementaciju, može se proširiti i na asinhrone zadatke, i obezbeđuje jednostavan način za proveru korektnosti *hard*- RTS sistema. Imajući to u vidu, zašto RTS algoritam ne bi mogao biti univerzalno rešenje?

Odgovor na ovo pitanje je taj što RMS algoritam obezbeđuje relativno nizak nivo iskorišćenosti sistema. Razmotrimo, ponovo, nejednakost (1). Suma sa leve strane nejednakosti predstavlja *nivo iskorišćenosti* CPU-a, tj. deo ukupnog vremena u kome je CPU zauzet izvršenjem zadataka. Na primer, neka za zadatak Z_1 važi: C_1=2ms i T_1=10ms (tj. vreme izvršenja zadatak Z_1 je 2ms, a perioda 10ms). Kada bi u sistemu postojao samo zadatak Z_1, tada bi CPU bio zauzet samo 20% vremena, tj. nivo iskorišćenosti CPU-a bi bio 2/10= 0.2. Neka se sistem proširi zadatkom Z_2, sa C_2=3ms i T_2=12ms. Sada CPU izvršava dva zadatka. Na Z_1 troši 2ms, na svakih 10ms, a na Z_2 3ms na svakih 12ms. Znači nivo iskorišćenosti je sada veći i iznosi 2/10 + 3/12 = 0.45, ili 45%. Očigledno, nivo iskorišćenosti ne može biti veći od 1, tj. od 100%. Sa stanovišta efikasnosti (cene) poželjno je da nivo iskorišćenosti bude što je moguće bliži 1. (Tako se potpunije koriste raspoloživi resursi). Ukoliko je nivo iskorišćenosti nizak, projektant može da prvobitno predviđen mikroprocesor zameni nekim sporijim (jefitinijim). Ugradnjom sprijeg mikroprocesora vremena izvršenja zadatak se proporcijalno povećavaju. S obzirom da periode zadataka moraju ostati nepromenene, nivo iskorišćenosti raste. (Naravno, treba voditi računa da nivo iskorišćenosti ne postane veći od jedan.) Međutim, RMS algoritam nameće dodatno, strože ograničenje u pogledu nivoa iskorišćenosti (desna strana nejednakosti (1)). Desna strana nejednakosti (1) se zove *granična iskorišćenost.* Za *n*=1, granična iskorišćenost jednaka je 1, a sa porastom *n* (broja zadataka) granična iskorišćenost opada i asimptotski teži vrednosti 0.69. Ovo praktično znači da se u cilju garantovanja korektnosti, za dovoljno veliki broj zadataka, žrtvuje oko 30% CPU vremena![2] Šta više, RMS kriterijum ostavlja otvorenim pitanje korektnosti sistema kod koga je nivo iskorišćenja veći od graničnog, a manji od 1. Odnosno, ako je nivo iskorišćenosti veći od granične, ne znači, po automatizmu, da sistem nije korektan. U slučajevima kada za nivo iskoršćenosti dobije vrednost veću od granične, a manju od 1, projektant može da postupi na jedan od sledeća dva načina: (1) upotrebi brži CPU i tako smanji nivo iskorišćenosti ispod graničnog nivoa, ili (2) sprovede dodatnu analizu i dokaže da je sistem zaista korektan, ili ako dokaže suprotno, primeni rešenje (1). Nažalost, dokaz korektnosti u slučaju (2) nije trivijalan.

[2] U realnim uslovima ovaj procenat je još veći, jer se uvek mora računati sa nekom rezervom (kod određivanja vremena izvršenja zadataka uzima se najgori slučaj, slično kod asinhronih zadataka mora se računati na najkraću periodu).

Earliest-Deadline-First Scheduling. Ovo je dinamički algoritam za planiranje kod koga dispečer najviši prioritet dodeljuje zadatku koji je najbliži svom krajnjem roku. Drugim rečima, prioritet zadatka koji čeka da bi bio izvršen vremenom raste. Ova činenica garantuje korektnost sistema, pod uslovom da je *opterećenje* sistema manje od 100%. Pod ukupnim opterećenjem sistema se podrazumeva se:

$$L = \sum_{j=1}^{n} \left(\frac{C_j}{T_j} \right)$$

Ukoliko opterešenje sistema postane veće od 100%, barem jedan zadatak će premašiti krajnji rok, a unapred se ne može odrediti koji će to zadatak biti. S obzirom da je reč o dinamičkom algoritmu za planiranje, neophodno je obezbediti dodatno CPU vreme za rad dispečera.

Planiranje izvršenja zadataka kod soft-RTS.

Za razliku od *hard*-RTS kod kojih je poštovanje krajnjih rokova imperativ, kod *soft*-RTS zahtevi u pogledu krajnjih rokova nisu u toj meri strogi. Neki od kriterijuma za optimizaciju *soft* RTS mogu biti:

- minimalna devijacija vremena odziva;
- minimalan broj prekoračenih krajnjih rokova;
- minimalno prosečno vreme prekoračenja krajnjih rokova;

minimalno vreme prekoračenja krajnjeg roka u najgorem slučaju.

Do zadovoljavajućeg rešenja se uobičajeno dolazi primenom statičkih prioriteta. Projektant određuje vrednost zadatog performansnog kriterijuma na bazi simulacije, a ukoliko nije zadovoljan dobijenim rezultatom, ima mogućnost da promeni prioritete zadataka i ponovi simulaciju.

Kontrola pristupa deljivim resursima

Kod jednoprocesorskog *multitasking* RTS 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. Slično, u slučaju ostalih hardverskih komponenti, moraju biti predviđeni posebni mehanizmi koji će obezbetiti korektnu deljivost. U deljive resurse se mogu svrstati i strukture podataka, kao što su promenljive, baferi, nizovi, kojima pristupa više od jednog zadatka.

Razmotrimo sledeću sitaciju koja se može javiti kod RTS opisanog u PRIMERU 2 (sistem za upravljanje kotlom na tečna goriva). Zadatak TASK1 se startuje periodično i ima funkciju izračunavanja pobudnog signala za ventil na osnovu izmerene i zadate temperature. Pretpostavimo da je vrednost zadate temperature smešetena u promenljivoj T u formatu XY.Z (3 BCD cifre). Zadatu temperaturu unosi operater preko tastature. Unos nove vrednosti sa tastature i ažuriranje promenljive T obrađuje zadatak TASK3. Problem se javlja kada se TASK1 aktivira baš u vremenu dok TASK3 ažurira promenljivu T. Naime, može se desiti da je u trenutku startovanja zadatka TASK1, zadatak TASK3 promenio samo prvu cifru promenljive T, dok su preostale dve ostale nepromenjene. Očigledno, TASK1 dobija pogrešnu vrednost zadate temperature i iz tog razloga generiše pogrešan rezultat. (Kaže se da je promenljiva T, usled prevremenog istiskivanja zadatka TASK3, ostala u *nekonzistentnom* stanju).

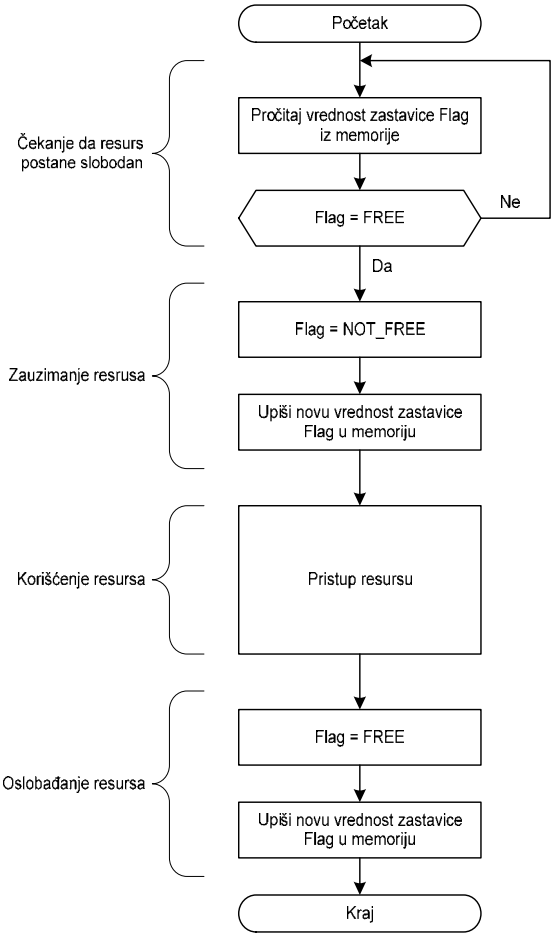
Rešenje ovog problema se sastoji u implementaciji tzv. *uzajamne isključivosti* (*mutual exclusion*), tj. mehanizma koji će obezbediti da deljivom resursu u bilo kom trenutku može da

pristupa najviše jedan zadatak. Međutim, iako jednostavan u svoj definiciji, impementacija mehanizma uzajamne isključivosti skopčana je sa mnogim, često suptilnim, problemima.

Algoritam signalne zastavice

Varijanta 1 (osnovna varijanta; problem bezbednosti)

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 resurs, 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 Sl. 6 prikazan je dijagram toka koji opisuje mehanizam signalne zastavice. Da bi bila ispitana, zastavica mora najpre biti prenesena iz operativne meorije u interni registar procesora. Deo programa koji obezbeđuje pristup deljivom resursu se zove "kritična sekcija" (*critical section*).



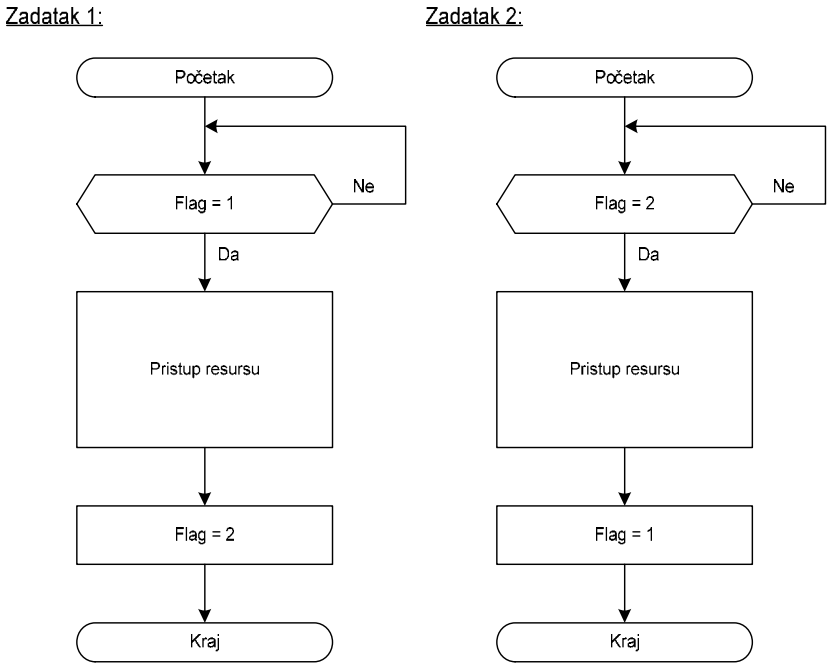
Sl. 6 Osnovna varijanta algoritma signalne zastavice (algoritam nije bezbedan)

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*.

Varijanta 2 (bezbedni algoritam signalne zastavice; problem nametnutog alternativnog pristupa resursu)

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, na oslobodi resurs. Algoritam je opisan dijagramom toka na Sl. 7. 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.

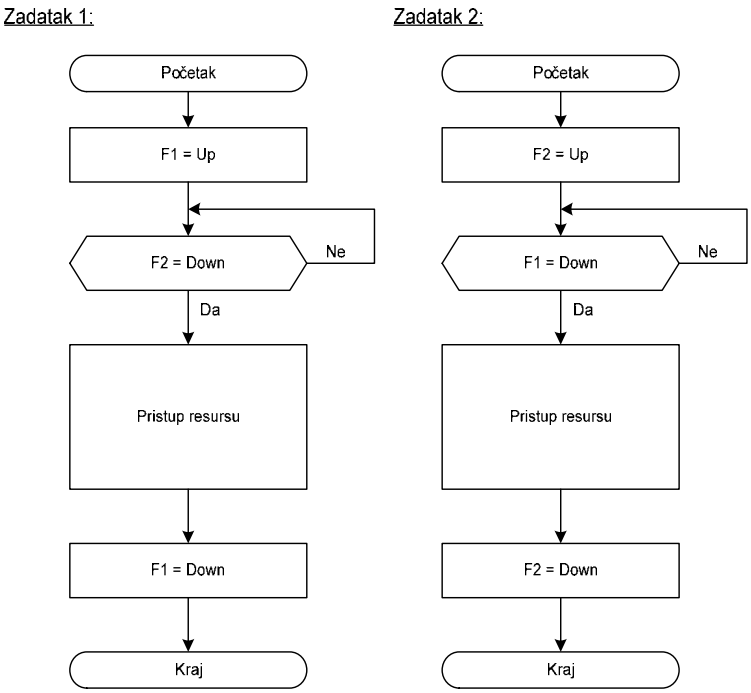


Sl. 7 Bezbedan algoritam signalne zastavice (problem nametnutog alternativnog pristupa resursu)

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.

Varijanta 3 (algoritam sa dve signalne zastavice; problem *dadlock-a* i *starvation*-a)

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. 8). 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.



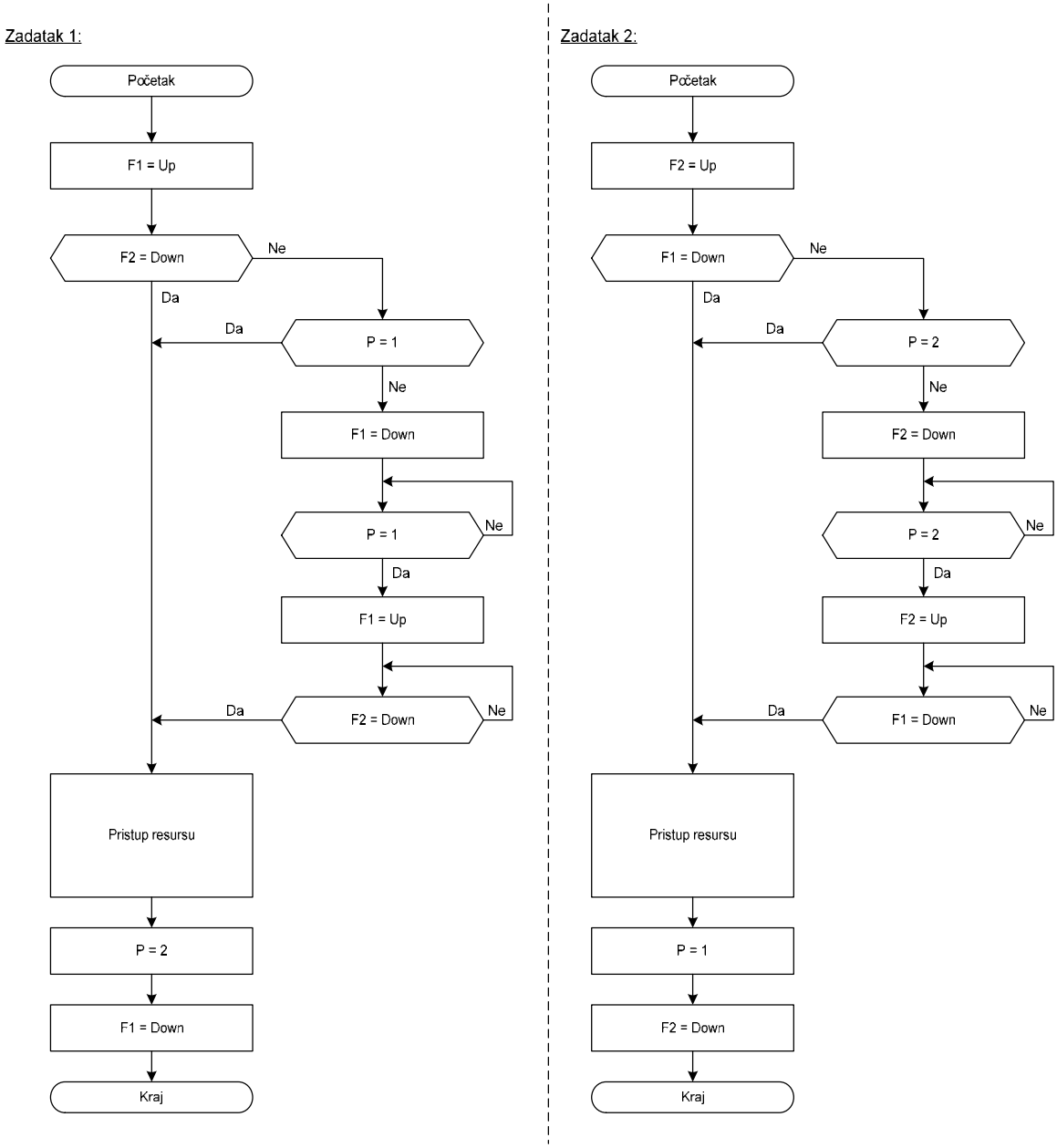
Sl. 8 Algoritam sa dve signalne zastavice (problem "smrtonosnog zagrljaja" i gladovanja)

Međutim, i ovo rešenje poseduje jedan ozbiljan problem, koji je poznat pod nazivom *deadlock* ("smrtni-zagrljaj"). 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 kružna 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 "galdovanje" (*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* RTS.

Varijanta 4 (*Dekkar*-ov algoritam; korektno rešenje)

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 (Sl. 9). 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, algirtam radi na isti načina kao Varijanta 3.



Sl. 9 Dekkar-ov 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 resurs, on najpre menje stanje zastavice prioriteta na P=1. Zadatak 1, 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.

Semafor

Primitivne operacije. Prilikom analize algoritama za kontrolu pristupa deljivim resursima polazi se od pretpostavke da do promene konteksta može doći u bilo kom trenutku u toku izvršenja zadatka. Međutim, u realnosti to nije slučaj. Zadatak se sastoji od niza mašinskih instrukcija koje se izvršavaju od strane CPU-a. Mašinske instrukcije su *nedeljive* (*primitivne* ili *atomizirane*) operacije, u smislu da se njihovo izvršenje ne može prekinuti. To znači da do promene konteksta može doći samo u trenucima kada CPU završava izvršenje tekuće i prelazi na izvršenje naredne mašinske instrukcije.

Test-and-Set operacija. Ključna operacija nad zastavicama koja se koristi kod algoritama za kontrolu pristupa deljivim resursima je oblika:

if *Flag* = FREE ***then*** *Flag* = NOT-FREE;

Ova operacija, poznata pod nazivom "*test-and-set*", najpre testira status zastavice, a zatim, ako je zastavica slobodna, postavlja je u zauzeto stanje. Međutim, ova operacija nije primitivna, s obzirom da se prilikom prevođenja na mašinski jezik, tipično, razlaže na sledeće tri mašinske instrukcije:

```
        MOV   A,Flag     ; prenos vrednosti zastavice Flag u akumulator CPU-a
        JNZ   Lab        ; skok ako je sadrzaj akumulatora različit on nule, tj. Flag=NOT_FREE
        MOV   Flag,#1    ; zauzimanje zastavice, tj. Flag=NOT-FREE
Lab:
```

Očigledno, do promene konteksa može doći kako između prve i druge, tako i između druge i treće instrukcije. U oba slučaja, promena konteksta predstavlja kritičnu aktivnost koja može da dovede do nekonzistentnosti zastavice. Pretpostavimo da je promena konteksta nastupila između prve i druge instrukcije, tj. u trenutku kada je zadatak (recimo Z1) preneo vrednost zastavice u akumulator, ali još uvek nije ispitao njenu vrednost. Neka je pri tome status zastavice *Flag*=FREE. Promena konteksta uključuje čuvanje statusnih informacija zadatka Z1 kao što je sadržaj programaskog brojača i sadržaj internih registara CPU-a, uključujući i akumulator. Neka zadatak koji je preuzeo kontrolu nad CPU-om (recimo Z2), u toku svog izrvšenja promenio status zastavice na *Flag*=NOT-FREE. Prva aktivnost prilikom nastavka izršenja istisnutog zadatka (Z1) je obnavljanje (rekostrukcija) njegovog statusa, što uključuje i punjenje akumulatora sadržajem koga je imao neposredno pre promene konteksta, tj. FREE. Na taj način, zadatak Z1 nastavlja izvršenje (instrukcijom JNZ) sa pogrešnom vrednošću stanja zastavice. Zastavica je nekonzistentna jer oba zadataka Z1 i Z2 "vide" zastavicu na drugačiji način (Z1 smatra da je zastavica FREE, a Z2 da je NOT-FREE). Ovakva situacija ima za posledicu pojavu konflikta prilikom korišćenja deljivog resursa.

Do prethodno opisanog problema ne bi došlo ako bi se u toku izvršenja "test-and-set" operacije onemogućila promena konteksta. Jedan od mehanizama kako se to može postići jeste da se neposredno pre početka "test-and-set" operacije zabrane prekidi, a nakon završetka ove opreacije prekidi ponovo dozvole. Ovakvo rešenje je moguće kod sistema gde se promena konteksta inicira prekidom (*interrupt*-om). Drugo rešenje, koje se sreće kod mnogih

savremenih mikroprocesora, je korišćenje specijalizovane mašinske instrukcije tipa "test-and-set". S obzirom da se radi o mašinskoj instrukciji, koja je po definiciji primitivna, očuvana je kozistentnost zastavice.

Semafor. Semafor je zaštićena promenljiva kojoj se može pristupati isključivo za tu namenu predviđenim primitivnim operacijama, tipa "test-and-set". Razlikuju se dva tipa semafora: *binarni* i *brojački*, ali oba rade na istom principu.

Binarni semafor. Binarni semafor je binarna promenljiva čija vrednost 0/1 ukazuje na takući status pridruženog deljivog resursa (1 - slobodan; 0 - zauzet). Za pristup semaforu koriste se dve primitivne operacije: *Wait()* i *Signal()*. Takođe, svakom semaforu se pridružuje red čekanja zadataka koji su izdali zahtev za pristup resursu, ali nisu dobili pravo pristupa zato sto je resurs zauzet. Zadatak koji želi da pristupi kritičnom resursu poziva operacija *Wait()* neposredno pre ulaska u kritičnu sekciju, a operaciju *Signal*() neposredno posle izlaska iz kritične sekcije:

```
BEGIN
   Programske naredbe;
   Wait(Sem);
   Kriticna sekcija (pristup resursu);
   Signal(Sem);
   Programske naredbe;
END.
```

(Da bi se identifikovao resurs kome je semafor pridružen, semafori se imenuju, a ime semafora se koristi kao parametar operacije *Wait()* i *Signal*().)

Operacija *Wait*(*Sem*) ispituje stanje semafora *Sem* i ako je *Sem*=1, tj. resrus slobodan, postavlja semafor u stanje 0 i vraća upravljanje zadatku koji nastavlja izvršenje ulaskom u kritičnu sekciju. U suprotnom, ako je stanje semafora *Sem*=0, resurs je zauzet, zadatak se suspenduje, tj istiskuje i postavlja u red čekanja pridružen semaforu *Sem*.

```
if (Sem=1) then
    Sem=0;
else
    Suspendovati zadatak;
```

Operacija *Singal*(*Sem*), se poziva prilikom napuštanja kritične sekcije. Znači, u trenutku kada je resrus ponovo slobodan. Operacija *Signal*(Sem) najpre ispituje da li je red čekanja pridružen semaforu *Sem* prazan. Ako u redu čekanja postoje zadaci, zadatku sa vrha reda čekanja upućuje se "signal za buđenje". "Probuđeni" zadatak nastavlja izvršenje i direktno ulazi u kritičnu sekciju. U suprotnom, ako je red čekanja prazan, operacija *Signal*(*Sem*) postavlja semafor u stanje 1, što omogućava prvom sledećem zadatku koji zatraži pristup resursu da dobije pravo pristupa.

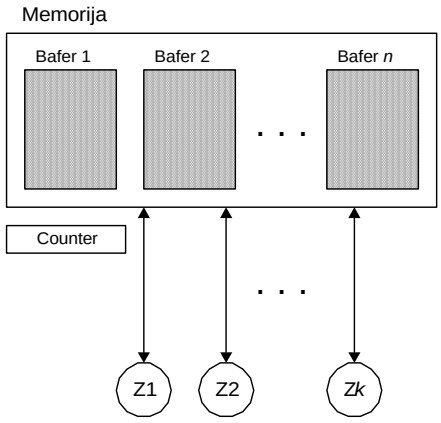
```
if (red cekanja nije prazan)
   "probuditi" zadatak sa vrha reda cekanja;
else
   Sem=1;
```

Treba uočiti da u slučaju da red čekanja nije prazan, operacija *Signal*() ne menja stanje semafora, tj. semafor ostaje u stanju 0 (zauzeto), s obzirom, da će neposredno po okončanju operacije *Signal*() resur biti zauzet "probuđenim" zadatkom.

Brojački (upošteni) semafor.

Pretpostavimo da je deljivi resurs podeljen na *n* delova. Delovi su identiči i svaki od njih obezbeđuje isti tip servisa kao i polazni resurs. Na primer, deljivi resurs može biti memorija koja je podeljena na *n* segmenata (bafera), pri čemu se svaki bafer koristi za privremeno odlaganje podataka (Sl. 10). Svaki od *k* zadataka, Z1, …, Z*k*, povremeno odlaže neke svoje

podatke u bafer, da bi ih sačuvao za neku kasniju fazu obrade. Ukoliko bi se memorija tretirala kao jedinstven resurs, a pristup memoriji kontrolisao uz pomoć semafora, memoriji bi radi odlaganja/preuzimanja podataka uvek mogao da pristupa najviše jedan zadatak, bez obzira što zadatak koristi samo jedan segment memorije. Na taj način dolazi do nepotrebnog usporavanja rada sistema.



Sl. 10 Ilustracija primene brojačkog semafora.

U ovakvim slučajevima koristi se brojački semafor. Za razliku od binarnog semafora, koji se realizuje kao binarna preomenljiva, brojački semafor se implementira kao celobrojna promenliva (tipa **integer**) koja može da uzima vrednosti od 0 do *n*. Inicijalno, semafor je postavljen na maksimalnu vrednost (*n*). Vrednost semafora ukazuje da broj nezauzetih instanci podeljenog resursa. (Na Sl. 10 semafor je označen sa *Counter*, podeljeni resurs je Memorija, a instance podeljenog resursa su: Bafer 1,…, Bafer *n*). Vrednost 0 semafora ukazuje da je resus u potpunosti zauzet. Zadatak, kada želi da pristupi resursu, najpre ispituje da li je resurs dostupan (*Couter* različito od nule) i ako je to slučaj umanjuje vrednost semafora za 1 i nastavlja sa korišćenjem resursa. Kada završi korišćenje resursa, zadatak inkrementira semafor za 1.

Modifikovane procedure *Wait(), Signal()*:

Wait:

if (*Counter*>0) **then**
    Counter = Counter-1;
else
    Suspendovati zadatak;

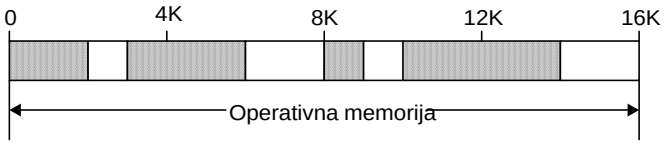
Signal:

if (red cekanja nije prazan)
   "probuditi" zadatak sa vrha reda cekanja;
else
   Counter = *Counter*+1;

Dodela memorije

Upravljanje dodelom i pristupom memoriji predstavlja jedan od najvažnijih zadataka RTOS. Svakom korisničkom zadatku neohodna je momorija za čuvanje potrebnih informacija i podataka, koja mora biti obezbeđena (alocirana) prilikom aktiviranja zadatka. Meorija može biti alocirana statički ili dinamički. Kod statičke alokacije, količina potrene memorije je poznata unapred i dodeljuje se zadatku u fazi startovanja (iniciranja) sistema. Međutim, statička alokacija memorije ima sledeće nedostatke: 1) tačan iznos memorije koja je potrebna zadatku ne može uvek biti poznat unapred (u fazi projektovanja (ili kompajliranja) softvera -

compile-time), što nameće potrebu da se statički alocira maksimalno očekivana količina memorije; 2) U realnim uslovima, kod RTS najveći deo vremena aktivan je samo mali deo od ukupnog broja zadataka, a to znači da je realna potreba za memorijom tipično mnogo manja od ukupnog iznosa memorije koju zahtevaju svi zadaci. Ovi problemi se prevazilaze dinamičkom alokacijom memorije, koja se vrpši u toku rada sistema (*run-time*), tj. u toku izvršenja zadatka. Za ovu namenu RTOS poseduje specijalizovane servise za alokaciju i dealokaciju memorije koji opslužuju. Memorijski prostor koji se koristi za dinamičku alokaciju se zove *heap*. Programski jezici koji podržavaju zahteve za memorijom pomoću operatora tipa *malloc* ili *new*. Ovi zahtevi se u toku izvršenja prevode u sistemske pozive kojima se od RTOS traži da u *heap* pronađe kontinualni deo memorije zahtevane veličine. Kada zadatak više nema potrebu za memorijom koja mu je dodeljena (tipično, prilikom deaktiviranja zadataka), memorija se dealocira (što je podržano operatorima tipa *free* li *delete*). OS vodi evidenciju o trenutno zauzetim i slobodnim memorijskim blokovima. Problem koji se javlja kod dinamičke alokacije memorije poznat je pod nazivom *fragmentacija memorije*. Naime, zbog česte alokacije/dealokacije memorijskih segmenata različite veličine, u *heap* se fragmentira na takav način da se javljaju zauzeti memorijski blokovi proizvoljne veličine razdeljeni slobodnim memorijskim blokovima prizvoljne veličine. Tako, može se javiti situacija da zahtev za alokacijom memorije ne može biti zadovoljen čak i u slučaju da je ukupna količina slobodne memorije u *heap*-u veća od zahtevane (iz razloga što se uvek zahteva kontinualni segment memorije). Ovakva sitacija ilustrovana je na Sl. 11. Osenčene oblasti označavaju zauzete, a neosenčene oblasti slobodne delove memorije. Preptostavimo da se zahteva alokacija 3KB memorije. Ovaj zahtev ne može biti zadovoljen, iako je ukupna količina slobodne memorije veća i iznosi 5KB[3].



Sl. 11 Fragmentacija memorije.

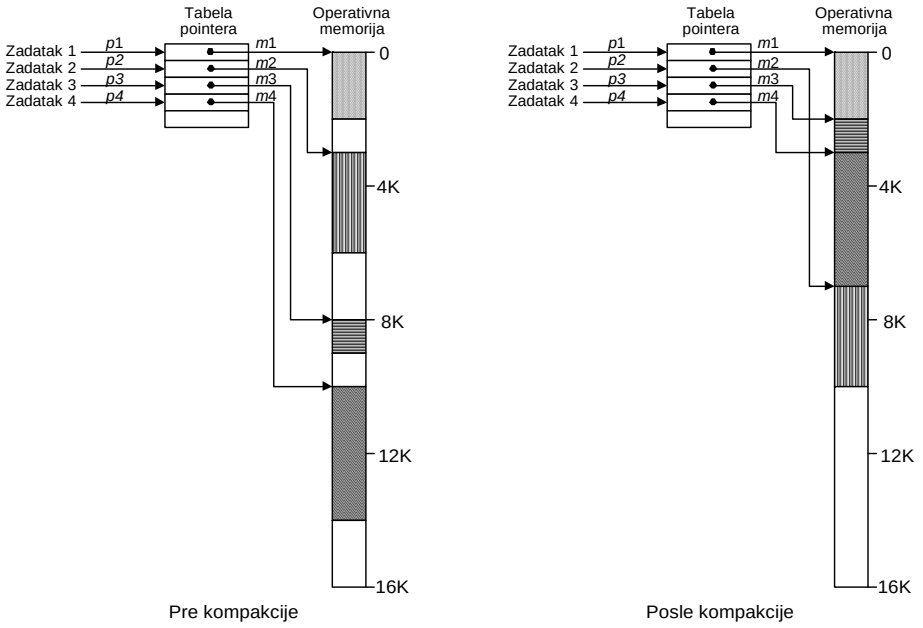
Da bi se minimizovao efekat fragmentacije, RTOS koristi specijalizovane algoritme za dodelu (alokaciju) memorije.

Jedna od strategija za prevazilaženje problema fragmetacije memorije je tzv. *Garbage collection*[4] (GC). GC je proces kompakcije (ili defragmentacije) kojim se *heap* preuređuje tako da se na kraju ovog procesa celokupna slobodna (zauzeta) memorija grupiše u jedan kontinualan segment. GC je zadatak OS-a koji se aktivira periodično ili u nakon zahteva za dodelu memorije koji nije mogao biti opslužen. Kada proces defragmentacije započne, svi aplikacioni zadaci se privremeno "zamrznu", a nakon izvršene defragmentacije povono dozvole. Kod RTOS koji podržavaju defragmentaciju, za pristup memoriji od strane aplikacionih zadataka koriste se mehanizam duplih pointera. Pointer koga poseduje aplikacioni zadatak ukazuje na pointer iz jedne posebene tabele pointera koja je pod kontrolom OS, a koji ukazuje na memoriju dodeljenu zadatku. Ovo omogućava da OS u toku

[3] Sličan problem fragmentacije javlja se i kod *hard* diska nakon višestrukog brisanja starih i snimanja novih fajlova. Međutim, posledice fragmentacije se razlikuju. Za smeštenje fajla na *hard* disk nije neophodan kontinualni meorijski prostor, već se fajl može smestiti u više fizički razdvojenih sektora na disku, što omogućava da se i u slučajevim izražene fragmentacije raspoloživ slobodan memorijski prostor uvek može u potpunosti iskoristiti. Međutim, fragmentacija nepovljno utiče na vreme čitanja/upisa fajlova, s obzirom da prelaz na nesusedni sektor na disku zahteva dodatno vreme za pozicioniranje glave za čitanje/upis.

[4] Garbage Collection znači sakupljanje otpadaka.

defragmentacije manipuliše samo sa pointerima iz tabele, ne menjanjući pri tome pointere koji su vlasništvo aplikacionih zadataka. Ovaj koncept je ilustrovan na Sl. 12.



Sl. 12 Pristup memoriji preko duplih pointera.

GC je moćan koncept, koji u potpunosti rešava problem fragmentacije memorije. Međutim, primenjen kod RTS on ispoljava sledeća dva nedostatka: 1) dupli pointeri povećavaju vreme pristupa memoriji, što dodatno opterćuje CPU; 2) "zamrzavanje" aplikacionog softvera u toku defragmentacije može da dovede do prekoračenja krajnjih rokova vremenski kritičnih zadataka. Ovaj privremeni prestanak rada aplikacionog softvera se dešava u proizvoljnom vremenskom trenutku, a njegovo trajanje se ne može unapred predvideti - drugim rečima, javlja se *nedeterminizam* u radu sistema. Iz tog razloga, tehnika GC se ne primenjuje kod *hard-real-time* sistema.

Drugi mehanizam koji se koristi za rešavanje problema fragmetacije memorije poznat je pod nazivom "*fixed-sized-block heep"* (*heap* sa blokovima fiksne veličine). Kod ovako organizovanog *heep-a,* na svaki zahtev za alokacijom memorije odgovara se dodelom bloka fiksne veličine. S obzirom na to, do fragmentacije nikada ne dolazi. Pri tome, veličina bloka je tako izabrana da zadovoljava sve zahteve za memorijom koji se mogu javiti. Nedostatak ovog pristupa je neefikasno iskorišćenje memorije koje se javlja u slučajevima kada je zahtevana memorija manja od veličine bloka.