

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
Zavod za elektroniku, mikroelektroniku,
računalne i inteligentne sustave

Skripta iz predmeta

Sustavi za rad u stvarnom vremenu

Leonardo Jelenković

Zagreb, 2016.

Predgovor

U računarstvu (koje je prepostavljeno u nastavku) područje obuhvaćeno imenom "sustavi za rad u stvarnom vremenu" (kratica SRSV) je vrlo opsežno. U ovom se predmetu zato i ne pokušava otići u širinu i pokriti cijelo područje (što je možda i nemoguće), već su odabранe i predstavljene neke teme, kratko predstavljene u ovom predgovoru.

Obzirom da se pravi problemi pri analizi, projektiranju, izgradnji i održavanju SRSV-a javljaju u sustavima koji upravljaju s više aktivnosti, tj. u sustavima koji obavljaju više zadataka ili zadatka, glavnina predmeta je posvećena ostvarenju višezadaćnosti u SRSV-ima i rješavanju problema koji se u takvim sustavima pojavljuju. Problemi raspoređivanja zadataka, usklađivanje njihova rada i međusobne komunikacije su glavni problemi. Kako se ti problemi rješavaju prikazano je i korištenjem standardnog POSIX sučelja.

Osim višezadaćnosti, u predmetu se kratko prikazuje problematika analize i programskog ostvarenja upravljačkih zadataka. Pri analizi se mogu koristiti neformalni ili formalni postupci, svaki sa svojim prednostima i nedostacima. Različiti problemi mogu se riješiti prikladnim programskim strukturama. Upravljanje se često ugrađuje u računala manje procesne moći te su jednostavniji algoritmi prikladniji, poput PID upravljanja. Kada se upravljačka funkcija ne može precizno iskazati (zbog nepoznavanja svih ovisnosti i slično), onda se mogu koristiti heuristički pristupi, poput neizrazitog zaključivanja.

Iako je predmet osmišljen na višoj razini apstrakcije i ne ulazi u probleme sklopovlja, ipak su prikazani neki detalji programskih jezika i alata na koje treba pripaziti pri ostvarenju SRSV-a. Operacijski sustavi, koji se mogu koristiti kao podloga SRSV-a, ukratko su opisani, istaknuti su mogući problemi na koje treba obratiti posebnu pažnju. Prikazana su svojstva operacijskih sustava u kontekstu uporabe SRSV-a te preporuke za odabir operacijskog sustava u ovisnosti o zahtjevima.

Sadržaj knjige podijeljen je u poglavљa:

1. Uvod
2. Modeliranje sustava
3. Ostvarenje upravljanja
4. Raspoređivanje poslova
5. Raspoređivanje dretvi
6. Višedretvena sinkronizacija i komunikacija
7. Raspodijeljeni sustavi
8. Posebnosti izgradnje programske potpore
9. Operacijski sustavi

Sadržaj

Predgovor	i
1. Uvod	1
1.1. Primjeri uporabe računala	1
1.1.1. Pisanje dokumenata	1
1.1.2. Reprodukcija audio i video sadržaja	2
1.1.3. Računalom upravljeni kućanski uređaji	2
1.1.4. Lift u zgradi	3
1.1.5. Automatizirano skladište	3
1.1.6. Usporedba sustava	4
1.2. Osnovni pojmovi	4
1.2.1. Ispravnost, greške, zatajenje	5
1.2.2. Logička ispravnost	5
1.2.3. Vremenska ispravnost	6
1.2.4. Sustavi za rad u stvarnom vremenu	6
1.2.5. Podjela SRSV-a	6
1.2.6. Događaji	7
1.2.7. Operacijski sustav	8
1.2.8. Sklopolje za SRSV	8
1.2.9. Determinističko ponašanje	8
1.2.10. Pouzdanost	9
1.2.11. Robusnost	9
1.3. Načini upravljanja	10
1.3.1. Sustavi za nadzor	10
1.3.2. Upravljanje bez povratne veze	11
1.3.3. Upravljanje korištenjem povratne veze	11
1.4. Operacijski sustavi za SRSV-e	12
1.5. Raspoređivanje zadataka u SRSV-ima	12
1.6. Primjeri SRSV-a upravljenih računalom	13
1.7. Problem složenosti	13
1.7.1. Grafički postupci	15

1.7.2. Metoda “podijeli i vladaj”	15
1.8. Zašto proučavati SRSV-e?	16
Pitanja za vježbu	17
2. Modeliranje sustava	19
2.1. Neformalni postupci	19
2.1.1. Skica	19
2.1.2. Tekstovni opis	20
2.1.3. Dijagram stanja	21
2.1.4. Nedostaci neformalnih postupaka	22
2.2. Formalni postupci u oblikovanju sustava	23
2.2.1. UML dijagrami obrazaca uporabe	23
2.2.2. UML sekvencijski i kolaboracijski dijagrami	23
2.2.3. UML dijagram stanja	25
2.2.4. Petrijeve mreže	26
2.2.5. Vremenske Petrijeve mreže	29
2.3. Primjer upravljačkog programa za lift	33
2.4. Proces izgradnje programske potpore	41
2.5. Posebnosti izgradnje SRSV-a	42
2.5.1. Formalna verifikacija	43
Pitanja za vježbu	43
3. Ostvarenje upravljanja	45
3.1. Struktura programa za upravljanje SRSV-ima	45
3.1.1. Upravljačka petlja	45
3.1.2. Upravljanje zasnovano na događajima	49
3.1.3. Jednostavna jezgra operacijskog sustava	50
3.1.4. Kooperativna višedretvenost	52
3.1.5. Višedretvenost	54
3.1.6. Raspodijeljeni sustavi	55
3.2. Ostvarenje regulacijskih zadataka	55
3.2.1. Upravljanje bez povratne veze	56
3.2.2. Upravljanje uz povratnu vezu	56
3.3. PID regulator	57
3.3.1. Proporcionalna komponenta – P	57
3.3.2. Integracijska komponenta – I	58
3.3.3. Derivacijska komponenta – D	58
3.3.4. Diskretni PID regulator	58

3.4. Upravljanje korištenjem neizrazite logike	62
Pitanja za vježbu	68
4. Raspoređivanje poslova	71
4.1. Uvod	71
4.2. Podjela poslova na zadatke	73
4.3. Vremenska svojstva zadataka	75
4.4. Postupci raspoređivanja	76
4.4.1. Statički i dinamički postupci raspoređivanja	76
4.4.2. Postupci raspoređivanja prilagođeni računalu	78
4.4.3. Prekidljivost zadataka	78
4.5. Jednoprocesorsko raspoređivanje	79
4.5.1. Jednoprocesorsko statičko raspoređivanje	79
4.5.2. Jednoprocesorsko dinamičko raspoređivanje	91
4.6. Višeprocesorsko raspoređivanje	93
4.6.1. Višeprocesorsko statičko raspoređivanje	94
4.6.2. Višeprocesorsko dinamičko raspoređivanje	106
4.7. Postupci raspoređivanja i njihova optimalnost	111
4.7.1. Optimalnost prema izvedivosti raspoređivanja	111
Pitanja za vježbu	114
5. Raspoređivanje dretvi	117
5.1. POSIX i Linux sučelja	118
5.2. Raspoređivanje dretvi prema prioritetu	119
5.3. Raspoređivanje prema krajnijim trenucima završetaka	121
5.3.1. Raspoređivanje prema krajnijim trenucima završetka kod Linuxa	122
5.3.2. Raspoređivanje sporadičnih poslova	122
5.4. Raspoređivanje nekritičnih dretvi	125
5.5. Upravljanje poslovima u uređajima napajanim baterijama	128
Pitanja za vježbu	130
6. Višedretvena sinkronizacija i komunikacija	131
6.1. Korištenje vremena u operacijskim sustavima	131
6.1.1. Korištenje satnih mehanizama u operacijskim sustavima	132
6.1.2. Nadzorni alarm	136
6.2. Semafori	137
6.3. Potpuni zastoj	138
6.4. Monitori	139

6.5. Rekurzivno zaključavanje	140
6.6. Problem inverzije prioriteta	141
6.6.1. Protokol nasljeđivanja prioriteta	142
6.6.2. Protokol stropnog prioriteta	143
6.6.3. POSIX funkcije za rješavanje problema inverzije prioriteta	146
6.7. Signali	147
6.8. Ostvarivanje međudretvene komunikacije	149
6.8.1. Zajednički spremnik	149
6.8.2. Redovi poruka	151
6.8.3. Cjevovodi	153
6.9. Korištenje višedretvenosti u SRSV-ima	154
Pitanja za vježbu	155
7. Raspodijeljeni sustavi	157
7.1. Povezivanje računala u mrežu	157
7.2. Model komunikacije	158
7.3. Svojstva komunikacijskog sustava	159
7.4. Primjeri protokola osmišljenih za komunikaciju u stvarnom vremenu	160
7.4.1. Controller Area Network	160
7.4.2. RTP/RTCP	161
7.5. Sinkronizacija vremena u raspodijeljenim sustavima	162
Pitanja za vježbu	165
8. Posebnosti izgradnja programske potpore za SRSV-e	167
8.1. Programske jezice	167
8.1.1. Programske jezike C	167
8.1.2. Programske jezike Ada	167
8.2. Prevoditelj	169
8.3. Problemi s višedretvenošću	170
8.4. Preciznost	171
8.5. Predvidivost trajanja – složenost postupaka	173
8.6. Višeprocesorski i višejezgreni sustavi	175
8.7. Korištenje ispitnih uzoraka	176
8.8. Oporavak od pogreške	176
Pitanja za vježbu	179
9. Operacijski sustavi	181
9.1. Uloga operacijskih sustava	181

9.1.1. Upravljanje vanjskim jedinicama	182
9.1.2. Upravljanje spremničkim prostorom	183
9.2. Operacijski sustavi posebne namjene	185
9.2.1. Operacijski sustav FreeRTOS	185
9.3. Svojstva različitih tipova operacijskih sustava	188
9.3.1. Operacijski sustavi za rad u stvarnom vremenu	189
9.3.2. Operacijski sustavi opće namjene	190
9.3.3. Prilagođeni operacijski sustavi opće namjene	191
9.3.4. Odabir operacijskih sustava	192
Pitanja za vježbu	193
Literatura	195

1. Uvod

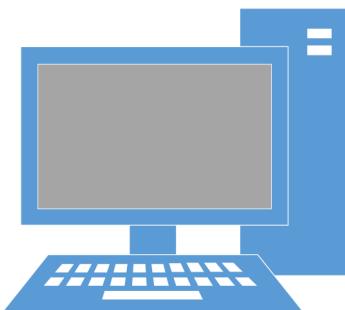
Što su to "sustavi za rad u stvarnom vremenu" – SRSV (engl. *real time systems*) u kontekstu računalnih sustava? Iako svi sustavi rade u vremenu, tj. obavljaju aktivnosti koje se protežu kroz neke vremenske intervale, je li protok vremena ključan za obavljanje tih aktivnosti, spadaju li oni u grupu koju SRSV-a?

1.1. Primjeri uporabe računala

Razmotrimo nekoliko primjera gdje se koristi računalo, kakva su očekivanja od tih sustava u tim primjenama te koje od njih možemo svrstati u kategoriju SRSV-a.

1.1.1. Pisanje dokumenata

Uobičajeni uredski posao jest pisanje raznih dokumenata, pisama i slično u primjerenim programima za obradu teksta, tablica, prezentacija i slika. Uobičajene radnje u takvom poslu jesu otvaranje postojećeg dokumenta ili stvaranje novog, pisanje teksta, uređivanje te spremanje. Napredni će alati imati razne pomoći pri izradi tih dokumenata. Primjerice, ukazivati će na pravopisne greške već tijekom pisanja, prepoznavati posebne unose, primjerice datume, imena i slično te ih ponuditi u uobičajenom formatu i slično.



Slika 1.1. Osobno računalo

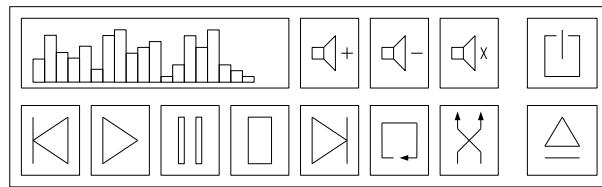
Očekivanja od korisnika u ovakvim sustavima jesu skromna: program treba omogućiti što jednostavniji rad, biti pouzdan, što ovdje prepostavlja da ne zaustavlja svoj rad ("povremeno smrzavanje") ili čak prekida s radom (izlaz iz programa s greškom) u neočekivanim trenucima. Operacije takvih alata nisu vremenski kritične. Neke od njih mogu potrajati i nešto više, primjerice samo otvaranje dokumenta, priprema za ispis. Korisnik je navikao da takve operacije mogu potrajati neko kraće vrijeme (primjerice nekoliko sekundi ili i više ako je dokument velik) te on to neće primati kao loš rad programa. Međutim, jednostavnije operacije moraju biti gotovo trenutne. Primjerice, pisanje teksta prepostavlja da svaki pritisak na neko slovo na tipkovnici bude predstavljen pojavom tog slova u dokumentu gotovo trenutno. "Gotovo trenutno" sa staništa čovjeka je subjektivan pojam, ali možemo pretpostaviti da ako je kašnjenje prikaza u odnosu na pritisak manje od 50 ms čovjek to neće primijetiti. Što ako takvo kašnjenje bude i veće? Ako se ono rijetko događa, korisnik to vjerojatno niti neće primijetiti. Ako se kašnjenje događa češće to će kod korisnika izazvati osjećaj sporosti alata, tj. to će biti ubrojeno u nedostatke alata. U takvom će slučaju korisnik možda potražiti i drugi alat (kada ima izbora).

Međutim, i u takvom sustavu s češćim kašnjenjima ili čak i povremenim zaustavljanjima ili i prekidanju rada ipak se ništa kritično neće dogoditi sa sustavom – nitko neće nastradati ili biti počinjena veća materijalna šteta (ukoliko ne ubrojimo izgubljeno vrijeme korisnika koji mora ponoviti svoj rad).

S staništa vremenskih ograničenja, takav rad zbiva se u stvarnom vremenu, ali bez strogih vremenskih ograničenja te bez većih posljedica u slučaju grešaka, uključujući i obične greške programa i sporosti, tj. kašnjenja u reakciji na korisnikove naredbe.

1.1.2. Reprodukcija audio i video sadržaja

Razmotrimo reprodukciju audio i video sadržaja korištenjem osobnog računala ili posebnog uređaja, primjerice, prijenosnog MP3 uređaja ili uređaja za prikaz sadržaja na televiziji (engl. *set top box*).



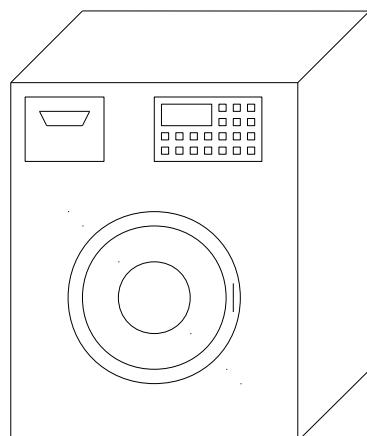
Slika 1.2. Multimedijski uređaj

Kod reprodukcije audio i video sadržaja bitna je vremenska usklađenost reprodukcije i stvarnog protoka vremena. Svaka će se neusklađenost primijetiti u slici ili zvuku. Korisnik će možda podnosititi rijetke pojave grešaka (neusklađene reprodukcije), ali češće greške neće jer takve greške značajno utječu na kvalitetu reprodukcije (kako ju doživljava korisnik). Stoga reprodukcija mora biti vremenski usklađena, a da bi uređaj (ili program) bio koristan za korisnika.

U slučaju greške ili prestanka ispravne reprodukcije, korisnik će ili ponovno pokrenuti program/napravu ili će odustati od njegovog korištenja. Međutim, kao i u prethodnom primjeru, neće biti većih posljedica.

1.1.3. Računalom upravljeni kućanski uređaji

Razmotrimo rad kućanskih uređaja kao što su perilica (rublja, posuđa), mikrovalna i obična pećnica koje su upravljane računalom (nekakvim mikroupravljačem).



Slika 1.3. Perilica rublja

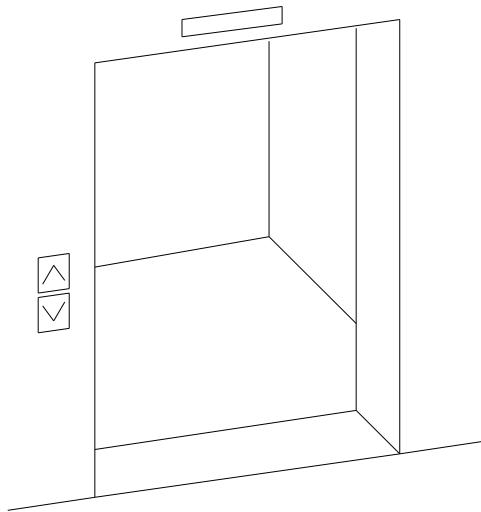
Sva tri navedena uređaja imaju sučelje preko kojeg korisnik odabere program rada te ih pokreće. Korisnik od uređaja očekuje da prati zadani program, tj. da u određenim vremenskim

intervalima sprovodi zadane radnje do okončanja posla. Također, u slučaju nepredviđenih potreškoća, korisnik očekuje prikladno ponašanje uređaja. Primjerice, ukoliko dođe do prekida električnog napajanja te ponovne uspostave, korisnik minimalno očekuje da uređaj ne započne neki drugi posao te da ima mogućnosti ponavljanja zadanog posla ili barem jednostavnog načina prekida. U primjeru perilice to bi pretpostavljalo da ima program koji će primjerice moći samo izbaciti vodu iz perilice prije njezina otvaranja. Drugi neočekivani događaji mogu biti prekid u dotoku vode, previsoka temperatura vode/zraka i slično. Od uređaja se očekuje da mogu primijetiti takve situacije i prikladno reagirati (npr. zaustaviti daljnji rad). Ukoliko uređaj nema prikladne senzore za takve moguće situacije, mogu se dogoditi značajne nezgode. Primjerice, uređaj se može zapaliti i zazvati požar u kućanstvu, kuhinja/kupaonica može biti poplavljena i slično. Te nezgode mogu imati značajne materijalne i ljudske posljedice. Stoga takve uređaje treba pomnije osmisliti, projektirati i ispitati.

Iako takvi uređaji ne trebaju jaču procesorsku snagu jer operacije kojima upravljaju su poprično spore, primjerice sekunda kašnjenja ne predstavlja nikakav propust, vremenska usklađenosnost na većoj skali je ipak potrebna. Također, takvi uređaji moraju biti vrlo pouzdani jer inače mogu izazvati velike štete.

1.1.4. Lift u zgradbi

Lift prevozi ljude i njihove stvari s jednog kata na drugi. Akcije koje upravljačko računalo lifta pokreće uključuju pravovremeno otvaranje i zatvaranje vrata lifta dok on stoji na katu, pomicanje lifta prema gore ili dolje, zaustavljanje lifta na potrebnim katovima, upravljanje tipkama u liftu i na katovima, uključivanje potrebne signalizacije (na kojem je katu, kamo ide, gdje će stati, ...).



Slika 1.4. Lift

Upravljanje liftom jest kritičan posao jer neke greške upravljanja mogu izravno utjecati na ljude koji ga koriste. Iako su akcije koje upravljačko računalo lifta pokreće u granulaciji sekunde ili i više, te akcije raznih elemenata moraju biti usklađene. Primjerice, vrata se ne smiju otvarati prije nego li lift stane, lift ne smije krenuti prije nego li su vrata zatvorena.

1.1.5. Automatizirano skladište

Razmotrimo jedno hipotetsko skladište koje je potpuno pod upravljanjem jednog računalnog sustava koji upravlja utovarnim i istovarnim vozilima koje premještaju proizvode po skladištu, ukrcaju i iskrcaju te proizvode na kamione koji dolaze pred skladište.

Upravljanje takvim složenim sustavom zahtjeva usklađenu koordinaciju svih vozila, njihovih

puteva po skladištu, stavljanju i uzimanju proizvoda s odgovarajuće police ili kamiona na vozilo i s vozila na polici ili u kamion, sve složeno po nekim načelima (primjerice, grupiranje istovrsnih artikala ili optimiranje rasporeda radi bržeg ukrcaja/iskrcaja). Greške u upravljanju mogu dovesti do sudara vozila s preprekom (zidom, kamionom, policom ili drugim vozilom), loše ukrčavanje/iskrcavanje u kojem se ošteti proizvod i slično. Ako u takvom sustavu ne sudjeluju ljudi (izravno u skladištu), greške upravljanja neće imati i ljudske posljedice. Ipak, materijalne posljedice mogu biti nezanemarive. Osim oštećenja proizvoda, vozila, kamiona i inventara šteta je i u odgodi posla, zbog kojeg mogu biti oštećeni svi dionici u lancu. Primjerice, ako je jedan kamion napunjen sat vremena prekasno možda neće svoj teret stići istovariti na brod ili će brod zbog njega morati biti zadržan duže u luci.

Kao i u prethodna dva primjera (kućanski uređaji, lift) i ovaj primjer uporabe računala spada u skupinu kritičnih poslova i sa stanovišta vremena i s posljedicama koje donose greške upravljanja.

1.1.6. Usporedba sustava

U primjerima pisanja dokumenta i reprodukcije multimedije (i sličnim) mogućnost rijetkog наруšавања исправног rada se i ne promatra kao nešto značajno loše za rad sustava. Međutim, u ostalim navedenim primjerima kao i mnogim drugim sustavima to nije tako. Npr. pri upravljanju letjelicom, projektilom, autom, industrijskim procesom, upravljačko računalo mora *uvijek* исправно raditi, bez zastoja, bez obzira na sve ostale poslove i događaje u sustavu, bez obzira na ostale ulaze i stanje sustava. Za neke je sustave dovoljno da se samo u jednom trenutku dogodi greška, da sustav na vrijeme ne daje исправне naredbe, a da se dogodi problem (ispad sustava, kvar, sudar, neispravni proizvodi, ozljede korisnika sustava). Takvi sustavi moraju biti upravljeni programskom potporom za koju kažemo da "radi u stvarnom vremenu" – u pravom trenutku daje prave podatke, naredbe, prema svojoj okolini.

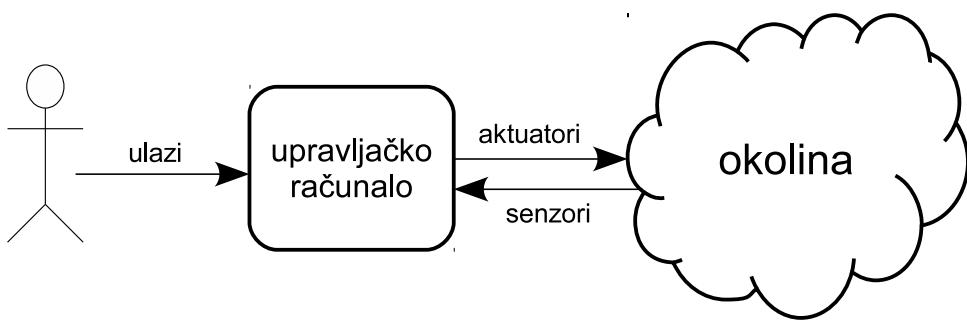
Mnogi sustavi nisu građeni da zadovoljavaju vremensko usklađeno upravljanje te se kod njih niti ne mogu postići takva upravljanja. Primjerice, u osobnom računalu bi mogli povećati prioritet programu koji radi reprodukciju video sadržaja s namjerom da se uvijek izvodi prije svih ostalih programa. Međutim, to nije uvijek dovoljno. Mnoge se operacije obavljaju u okviru operacijskog sustava, kao što je upravljanje ulazno izlaznim napravama, te bi i samo te operacije mogle u nekom kritičnom trenutku mogli zauzeti previše procesorskog vremena a da ostane dovoljno za pravilnu reprodukciju videa (npr. intenzivna komunikacija preko mreže koja pritom zahtijeva i dohvata/pohranu podataka na disk).

Da bi se mogla ostvariti željena svojstva sustava za rad u stvarnom vremenu svi elementi sustava trebaju biti prikladni za to, od sklopovlja, operacijskog sustava do programa i samih korisnika sustava.

1.2. Osnovni pojmovi

Gotovo svi sustavi koji nas okružuju mijenjaju se s protokom vremena. Za neke smo promjene mi odgovorni ili barem sudjelujemo u njima, dok su druge izvan našeg interesa ili moći. Promjene kojima smo mi uzroci događaju se zbog naše izravne upletenosti u njih ili zbog strojeva koje smo postavili da u njima sudjeluju. U nastavku se razmatraju samo oni sustavi koji su upravljeni računalom u obliku programa koji obavlja programirane operacije ili uz upravljanje čovjeka (čovjek upravlja sustavom posredstvom računala). Opća slika takvog sustava prikazana je na slici 1.5.

Računala se ugrađuju u našu okolinu radi njihove mogućnosti preciznijeg i bržeg računanja i reakcije na događaje, kao i radi oslobađanja čovjeka od obavljanja zamornih operacija. Neke od tih primjena su u nekritičnim sustavima (npr. telefon, sat, TV, ...) dok su druge primjene u



Slika 1.5. Sustavi upravljeni računalom

kritičnim sustavima gdje greške nisu dopuštene jer mogu imati osim materijalne štete i mnogo ozbiljnije posljedice (npr. ozljede). Projektiranje sklopovske i programske komponente za jedan i drugi sustav stoga neće biti isto. Kod projektiranja kritičnih sustava koristit će se značajno stroži kriteriji kako u odabiru sklopovskih komponenti tako i u pristupu izradi programske. Poželjno bi bilo i nekriticne sustave projektirati na taj način, ali često bi to dovelo do značajnog povećanja cijene te se pri njihovu projektiranju radi kompromis između kvalitete i brzine izrade (npr. ne koriste se uobičajeni procesi razvoja već se koriste ubrzani procesi koji će prije dati rezultat nauštrb preglednosti, arhitekture, optimalnosti, ispitivanja, ne koriste se skupi izvrsni programeri već jeftiniji i manje stručni).

1.2.1. Ispravnost, greške, zatajenje

Ako sustav obavlja ono što se od njega traži, uz prihvatljivu kvalitetu produkata, sustav je *ispravan*. Kako kvaliteta proizvoda pada tako je i *korisnost* sustava manja.

Zbog raznih okolnosti sustav može doći u *neočekivana stanja* u kojima ne obavlja predviđenu operaciju. Ako su ta stanja kratkotrajna te se sustav iz njih može izvući (pod upravljanjem računala) onda se radi samo o kratkotrajnoj grešci sustava. U protivnom, ako se sustav nakon greške ne može oporaviti radi se o *zatajenju* sustava. Posljedice grešaka i zatajenja mogu biti samo materijalne ili imati posljedice i za ljude u okolini upravljanog sustava.

Uzroci zatajenja mogu biti izvan upravljanog sustava (“ptica je kroz prozor uletjela u stroj i začepila dovod zraka”) ili iznutra. Ako je greška nastala unutar sustava, uzrok može biti sklopovski (“pukla osovina”) ili greška upravljačkog računala (programa). U idućim razmatranjima promatraju se samo greške upravljačkog računala.

Kada program ne radi *ispravno*, kako se greška pokazuje?

1.2.2. Logička ispravnost

Kod projektiranja programske komponente računalnih sustava pred projektante se postavljaju razni zahtjevi. Program treba na osnovu ulaza izračunati valjane (očekivane) izlaze prema postupcima koji su zadani uz problem. Navedeni zahtjev nazivamo *logičkom ispravnošću*. Kod mnogih primjena logička ispravnost je dovoljna, primjerice za:

- program za proračun plaća
- inženjerske proračune (npr. naprezanja, struje/naponi, ...)
- izradu audio/video sadržaja (ne i njihovu reprodukciju)
- obradu teksta.

Svi programi *moraju* davati logički ispravne rezultate! Međutim, kod nekih sustava samo logička ispravnost nije dovoljna.

1.2.3. Vremenska ispravnost

Ispravan rezultat ili naredba vanjskoj napravi često nije dovoljna a da sustav ispravno radi. Dodatno je potrebno da taj rezultat bude pripremljen u pravom trenutku! Vremenska neusklađenost izračuna podatka može dovesti do smanjenja kvalitete, grešaka ili čak do zatajenja.

U primjerima 1.1.1.-1.1.5. već je analizirana potreba vremensko uskađenog upravljanja. Osim što upravljački program/računalo mora generirati potrebne naredbe to mora raditi u pravim trenucima. Ako se naredba daje prerano ili prekasno sustav može snositi posljedice. Možemo reći da osim logičke ispravnosti od ovih se sustava zahtijeva i *vremenska ispravnost*.

1.2.4. Sustavi za rad u stvarnom vremenu

Sustavi za rad u stvarnom vremenu su sustavi kod kojih je osim *logičke ispravnosti* još bitnija *vremenska ispravnost*.

Engleski termin je *real time systems*, kraće *RT systems* ili još kraće *RTS*. Termin je poprilično udomaćen i nalazi se u mnoštvu materijala na hrvatskom jeziku (kao *RTS* ili *RT sustavi*). Ipak, u ovom tekstu se koristi kratica hrvatskog naziva *sustavi za rad u stvarnom vremenu* – *SRSV*.

Iako primjere 1.1.1.–1.1.5. možemo sve svrstati u *SRSV*-e, očito je da će pojedino narušavanje vremenske ispravnosti ipak imati znatno različite posljedice. Zato se često u kontekst *SRSV*-a ubrajaju samo sustavi koji upravljaju stvarnim procesima u našoj okolini, gdje su posljedice grešaka značajne (materijalne ili posljedice po čovjeka). U ovom tekstu se sustavi neće razlikovati, mada je jasno da pri projektiranju takvih sustava treba biti znatno oprezniji nego pri projektiranju igara ili multimedijalnih programa.

1.2.5. Podjela *SRSV-a*

Postoje mnoge podjele *SRSV*-a s obzirom na različite aspekte. Jedna od njih je podjela obzirom na osnovu izvora poticaja za akciju u sustavu. Tako razlikujemo sustave *pobuđivane događajima* (engl. *event triggered systems*) te *sustave upravljane protokom vremena* ili *ritmom okidane sustave* (engl. *time triggered systems*).

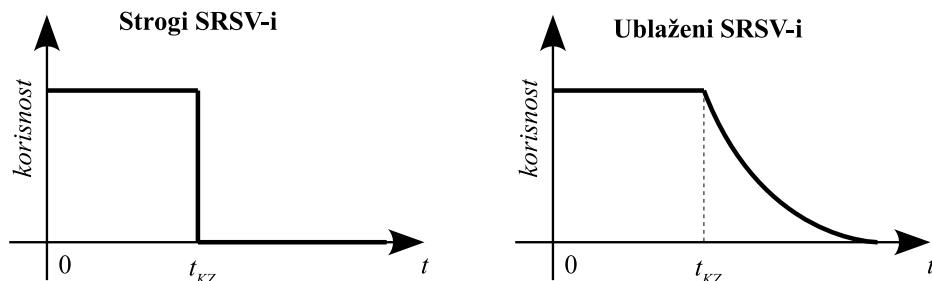
Ipak, osnovna podjela *SRSV*-a zasniva se prema očekivanjima od sustava, tj. posljedicama ne-poštivanja definiranih ograničenja. *SRSV*-i se prema tom kriteriju dijele na:

- *stroge* (engl. *hard RTS*),
- *ublažene* (engl. *soft RTS*) te
- *čvrste* (engl. *firm RTS*).

Kod strogih *SRSV*-a zadana vremenska ograničenja treba strogo poštovati. Ako se i makar jednom to ne postigne sustav zakazuje. Jedan primjer takvih sustava može biti sustav upravljanja, jer svaka greška može biti fatalna. Primjerice, neka u nekoj tvornici montažu nekog uređaja rade robotske ruke na traci. U slučaju da i jedna ruka zakaže i na jednom proizvodu ne napravi ono što treba, može se dogoditi da cijela traka stane, jer iduća robotska ruka ne može nastaviti s radom (može se čak i pokvariti pokušavajući obaviti svoj posao).

Ublaženi *SRSV*-i neće u potpunosti zakazati ako se poneko vremensko ograničenje ne ispoštuje. Primjerice, neka u nekoj proizvodnji računalo upravlja grijanjem gdje je točno zadano koliko grijanje treba biti upaljeno (npr. treba proračunati prema masi objekata). Ako računalo drži grijanje upaljeno malo dulje ili kraće trenutni objekt neće biti optimalne kvalitete. Ako se to događa sporadično, objekti lošije kvalitete izbacit će se iz sustava (u procesu kontrole) i "ukupna" šteta bit će jednaka "samo" gubitku zbog tih (sporadičnih loše grijanih) objekata. Međutim, sustav će i dalje moći raditi. Naravno, ako je broj takvih defektnih proizvoda velik, a iz razloga lošeg upravljanja računala, onda sustav upravljanja nije zadovoljavajući.

Uobičajeno se svojstva sustava procjenjuju prema vremenu reakcije na događaj. Za stroge sustave, reakcija mora biti unutar zadanog, unaprijed poznatog intervala inače dolazi do zatajenja, odnosno, korisnost sustava pada na nulu. Za ublažene sustave dozvoljeno je poneko prekoračivanje zadanog intervala za reakciju, ali svako to prekoračivanje smanjuje korisnost sustava. Slika 1.6. grafički prikazuje (uspoređuje) stroge i ublažene SRSV-e prema vremenima reakcije na događaje.



Slika 1.6. Usporedba strogih i ublaženih SRSV-a prema vremenu reakcije

U novijoj literaturi se spominje i treća grupa – čvrsti SRSV-i, kao sustavi koji dozvoljavaju poneki ispad, ali gdje već nekoliko ispada može biti kobno za sustav. Primjerice, sustav navigacije može ponekad krivo interpretirati ulaze senzora i ne otkriti prepreku. Međutim, ako se uzorkovanje obavlja dovoljno velikom frekvencijom, već će iduća ispravna očitanja i interpretacije spriječiti sudar. U protivnom, ako nekoliko uzastopnih očitanja, odnosno interpretacije tih očitanja budu kriva događa se fatalna greška za sustav (sudar).

1.2.6. Događaji

Jedno od osnovnih obilježja SRSV-a (njegove upravljačke komponente) je njegova pravovremena reakcija na događaje iz okoline kojom računalo kao upravljačka i/ili nadzorna komponenta upravlja. Događaji iz okoline mogu se pojavljivati *periodički*, *aperiodički* ili *sporadično* te *sinkrono* ili *asinkrono* u odnosu na neko mjerilo vremena ili druge događaje.

Periodički događaji javljaju se s istim vremenskim razmakom ili se u istim vremenskim intervalima (periodama) pojavljuje samo jedan takav događaj. Npr. senzor učvršćen na nekoj traci će stvoriti signal svaki puta kada se novi objekt pojavi, što će biti periodički ako se traka periodički pokreće.

Aperiodički događaji su nepravilniji, ali ipak ih se u statističkoj analizi može opisati prosječnom periodom pojave (od koje pojedinačni događaji više ili manje odstupaju). Razmotrimo isti prethodni primjer s trakom. Pretpostavimo da na traci istovremeno radi nekoliko robotskih ruka nad raznim objektima na traci, da objekti nad kojima se obavljaju neki zahvati nisu identični, da za različite objekte pojedina robotska ruka treba različite intervale vremena. Traka se pomiče tek kada su svi roboti završili svoje operacije nad svojim objektom. Obzirom na različita trajanja tih operacija za različite objekte, pomicanje trake bit će nepravilno, ali bi se ipak statistički moglo prikazati s nekim prosječnim vremenima (koja se mogu iskoristiti za analizu rada sustava).

Sporadični događaji su događaji koji se rijetko javljaju (i za koje statističko uvrštavanje u periodičke ne bi imalo smisla, jer bi odstupanja bila ogromna). U prethodnom primjeru s proizvodnjom na traci, sporadični događaj bi npr. bila pogreška jedne robotske ruke i detekcija tog kvara (koji bi npr. uzrokovao da sve robotske ruke u nastavku rada nad istim već defektним objektom preskoče taj objekt u obradi – nad njim ne rade ništa).

Ako se neki događaji obavljaju istovremeno s nekim drugim događajima u računalnom sustavu, onda kažemo da su te dvije skupine događaja *sinkronizirane*. U protivnom događaji su nepove-

zani ili *asinkroni*. Npr. signal koji se javlja svakih 10 ms je sinkroni, signal koji se uvijek javlja u unaprijed poznatim trenucima (ne nužno periodičkim) je također sinkroni (može se uskladiti pokretanje odgovarajućih operacija koje će biti sinkrone tim događajima). Za prethodni primjer s pokretnom trakom možemo reći da su počeci rada robota na traci sinkroni s završetkom pomicanja trake (oni počnu raditi kad novi objekt dođe pred njih). S druge strane, završeci rada pojedinih robota su međusobno asinkroni događaji.

1.2.7. Operacijski sustav

Operacijski sustav (kratica OS) je “skup programa koji olakšava korištenje računala”. Operacijski sustav maskira specifičnosti sklopolja (odgovarajućim upravljačkim programima, engl. *device drivers*) kroz prikladno (unificirano) sučelje prema programima (engl. *application programming interface – API*) i prema korisniku (engl. *graphical user interface – GUI*).

Iz razloga proširivosti, jednostavnosti, prenosivosti i slično, mnogi računalni sustavi (gotovo svi netrivijalni) koriste operacijski sustav. Operacijski sustavi su izgrađeni za određene namjene. Primjerice, operacijski sustav za osobno računalo je građen s namjerom korištenja računala kao uredskog računala (podrška uredskim programima), kao multimedijalni stroj, kao stroj za iganje i slično. Obzirom da nije građen kao stroj za upravljanje vanjskim sustavima on niti nema potrebna svojstva za SRSV, a jedno od najvažnijih jest brza reakcija na događaje uvijek (za stroge SRSV-e), ne u “samo” 99.9% slučajeva.

Osnovni problem operacijskog sustava koji nije za SRSV je u nemogućnosti osiguranja *brze* reakcije na događaje. Potrebna brzina zapravo ovisi o sustavu kojim se upravlja i može se kretati od nekoliko mikrosekundi do nekoliko sekundi ili i dulje (uobičajeno se vrijeme reakcije općenito smatra nekoliko milisekundi kao zadovoljavajuće za prosječne SRSV-e).

1.2.8. Sklopolje za SRSV

SRSV-i se mogu ostvariti na raznom sklopolju. Ipak, sklopolje koje je projektirano s tom namjenom ima znatno bolja svojstva u SRSV okruženju. S druge strane, sklopolje koje je projektirano za sustave s mnoštvo predviđenih operacija, kao što su to osobna računala, prijenosnici, ručna računala, mobiteli, poslužitelji i slično, najčešće nema mehanizme koji osiguravaju vremensku ispravnost. Npr. iako procesor osobnog računala može biti i za red veličine brže od onog koji se koristi za upravljanje nekog kritičnog procesa, vrlo je vjerojatno da se osobno računalo ne bi moglo tamo ugraditi. Nisu problem samo dimenzije. Osobno računalo sastoji se od mnoštva komponenata (s tim je ciljem i projektirano – da bude proširivo). Upravljanje tim komponentama će ponekad oduzeti nepredviđeno vrijeme. Npr. jedan sklop može zbog svojeg sporog rada spriječiti procesor u korištenju sabirnice, a time i posluživanje i upravljanje s drugim komponentama. Ta odgoda u SRSV-ima može biti kobna.

1.2.9. Determinističko ponašanje

Upravljanje sustavom podrazumijeva da se na osnovi trenutnog stanja sustava i poznavanja povijesti zbivanja u sustavu, može donijeti upravljačka odluka za promjenu u željeno buduće stanje. Odluka se potom pretvara u upravljačke signale koji će putem prikladnih uređaja usmjeriti sustav u skladu s odlukama (donesenim, tj. izračunatim u upravljačkom računalu). Da bi navedeno bilo moguće sustav mora biti *predvidiv*, tj. mora unaprijed biti poznato kako će određena akcija koja se pokreće utjecati na cjelokupni sustav, i to za svako moguće stanje sustava. Drugim riječima, ako se sustav nalazi u stanju X (koji opisuje to stanje), onda će on s akcijom a uvijek otići u stanje Y. Za takav sustav kažemo da ima *determinističko ponašanje* (predvidljivo). Sustav koji nije deterministički je *stohastički* (nepredvidljiv).

SRSV-i moraju biti deterministički jer se u protivnom ne bi mogli upravljati. To se odnosi i na

sklopolje i na operacijski sustav i na programe.

Primjerice, ako se ne može odrediti najdulje vrijeme reakcije na točno određeni događaj bez obzira na trenutno stanje sustava, sustav nije deterministički i nije upotrebljiv za SRSV (barem ne za one sa strogim vremenskim ograničenjima). Zato stolno računalo, ni sklopoljem ni operacijskim sustavima nije uporabljivo za SRSV-e.

Da bi računalni sustav bio prikladan za određenu SRSV primjenu, on mora imati odgovarajuće sklopolje, odgovarajući operacijski sustav te odgovarajuće programe, sve pripremljeno da zadovoljava zadana logička i vremenska ograničenja.

1.2.10. Pouzdanost

Sustavi koji svoj posao uvijek obavljaju u skladu s očekivanjima su *pouzdani sustavi*. Postizanje pouzdanosti sustava znači obuhvatiti sve zahteve postavljene pred sustav i njihovo ispravno obavljanje u ostvarenom sustavu. Osim izgradnje pouzdanog sustava, zadatak je inženjera da to pouzdanje prenesu korisniku, da i on vjeruje u ispravan rad sustava, da se pouzda u njegovo upravljanje svojom okolinom. Ostvarivanje povjerenja korisnika nadilazi ova razmatranja, ali opća načela kao što su uvid u metodologije korištene pri ostvarenju sustava, prikaz dokumentacije, prikaz obavljenih ispitivanja će tome sigurno pridonijeti.

1.2.11. Robusnost

Ispravan sustav je ono što kupac očekuje od proizvođača. Međutim, treba predvidjeti da će se tijekom rada pojavljivati određene izvanredne situacije. Npr. zbog dotrajalosti će se neka komponenta pokvariti (npr. neki aktuator). Preporučljivo bi bilo za sve takve komponente za koje se očekuju kvarovi (npr. iz iskustva radnika), da ih se posebno prati, npr. s dodatnim senzorima. U slučaju kvara, tj. detekcije kvara, mogu se tada poduzeti odgovarajuće akcije (npr. zaustaviti proizvodnju da se ne dogodi veća šteta).

Nije realno očekivati da će programska komponenta sustava biti bez greške u svim situacijama. Bilo bi dobro za neke osnovne dijelove upravljačkog programa ugraditi dodatne provjere i dodatno provjeriti da njihovi izlazi budu u dozvoljenim granicama. Npr. određene kritične proračune obavljati na dva načina, pomoću dviju (ili više) metoda. Ako se rezultati poklapaju ili je razlika zanemariva onda koristiti te rezultate. U protivnom zaustaviti sustav i alarmirati osoblje da razriješi problem.

Po potrebi koristiti i druge postupke koji će se primijeniti u slučaju detekcije greške u programu. U SRSV-ima se ne smije dogoditi da program zbog greške jednostavno stane (ili ga operacijski sustav prekine).

Za vrlo kritične sustave može se primijeniti i princip redundantnog upravljanja. Primjerice, umjesto jednog upravljačkog računala mogu se koristiti dva ili više koji rade istu obradu (dovivaju iste ulaze, ili s uvišestručenih senzora). Ako jedno računalo ispadne zbog sklopovske ili programske greške drugo može odmah prihvati upravljanje (i alarmirati osoblje o problemu).

Sustavi koji nisu pod stalnim nadzorom čovjeka, odnosno, sustavi kojima je teško pristupiti mogu imati ugrađen još jedan oblik oporavka od pogreške. Posebnim sklopoljem može se nadzirati ispravnost rada. Kad se primijeti da sustav ne reagira na zahteve tog sklopolja, sustav treba zaustaviti i ponovno pokrenuti (*resetirati*). Navedeni mehanizam naziva se *nadzorni alarm* (engl. *watchdog timer*).

Sustavi koji su napravljeni da mogu nastaviti s radom i nakon raznih (neplaniranih) nepovoljnih događaja mogu se smatrati *robustnim*. SRSV-i koji su u duljim vremenskim razdobljima bez nadzora trebali bi biti građeni i prema kriteriju robusnosti.

1.3. Načini upravljanja

Procesi kojima računalni sustav upravlja mogu biti *kontinuirani* i/ili *diskretni*.

Kontinuirani sustavi se kontinuirano mijenjaju i trebaju kontinuirano upravljanje. Primjerice, upravljanje slavinama u nekom kemijskom postrojenju treba stalni nadzor i podešavanja pročnosti kroz pojedine cijevi, a da bi se u određenim elementima sustava odvijale željene reakcije s potrebnim omjerima različitih smjesa u različitim trenucima vremena.

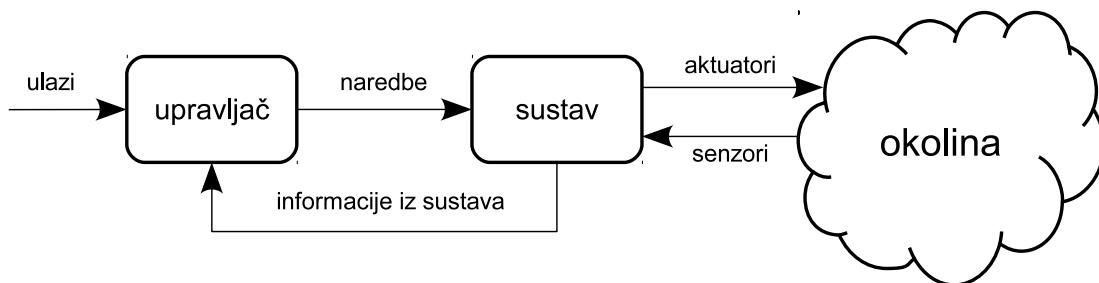
Diskretni sustavi mijenjaju stanje u diskretnim trenucima. Primjerice, upravljanje skretnica nekog željezničkog kolodvora obavlja se ili na intervenciju operatera ili na senzore koji otkrivaju dolazak vlakova ili u točno određenim trenucima uskladenim s redom vožnje.

Većina sustava ima dijelove koje treba kontinuirano upravljati i dijelove koji traže pozornost u diskretnim trenucima. Računalo, kao upravljačka komponenta je u osnovi diskretno, obavlja instrukciju za instrukcijom, jednu za drugom, jednu operaciju za drugom, jedan zadatak za drugim. Ipak, obzirom da računalo to obavlja vrlo velikom brzinom (u usporedbi s čovjekom), ono može prilagoditi frekvenciju kojom zadaje naredbe te tako zadovoljiti i kontinuirane sustave (s većom frekvencijom naredbi i očitanja senzora) i diskretne sustave (kojima npr. upravlja iz prekidnih funkcija).

Okruženja koja razmatramo i čije procese želimo upravljati i nadzirati sastoje se od:

- sustava kojim se upravlja ili koji se nadzire (npr. automobil, robot, stroj)
- upravljač (npr. čovjek, računalo)
- okoline u kojoj se sustav nalazi i koju sustav koristi i mijenja.

Upravljanje se odvija upravljanjem elementima sustava, tzv. *aktuatorima*. Ukoliko je sustav upravljan računalom, onda se upravljački signali koje ono šalje, u aktuatorima pretvaraju u sile kojim oni djeluju na sustav (i okolinu). Slika 1.7. prikazuje poopćeni *upravljački sustav*.



Slika 1.7. Opći upravljački sustav

Interakcija sustava s okolinom je dvosmjerna: sustav mijenja okolinu, ali i preko senzora očitava stanje okoline. Upravljač prima povratne informacije od sustava ali i naredbe od operatera (korisnika). U nastavku se pretpostavlja da je upravljanje ostvareno računalom koje je ugrađeno u sam sustav te se upravljač i sustav prikazuju jednom komponentom – *upravljački sustav*.

Obzirom na odnos upravljačkog sustava i okoline razlikujemo:

- sustave za nadzor
- upravljanje bez povratne veze (engl. *open-loop control*)
- upravljanje s povratnom vezom (engl. *closed-loop control, feedback control*).

1.3.1. Sustavi za nadzor

Sustavi za nadzor prikupljaju podatke iz okoline, obraduju ih te na prikidan način prikazuju operaterima. Primjeri nadzornih sustava su alarmni sustavi, video nadzor (uz računalnu obradu), radarski sustavi, mjerači onečišćenja i slični. Prikupljanje podataka može biti inici-

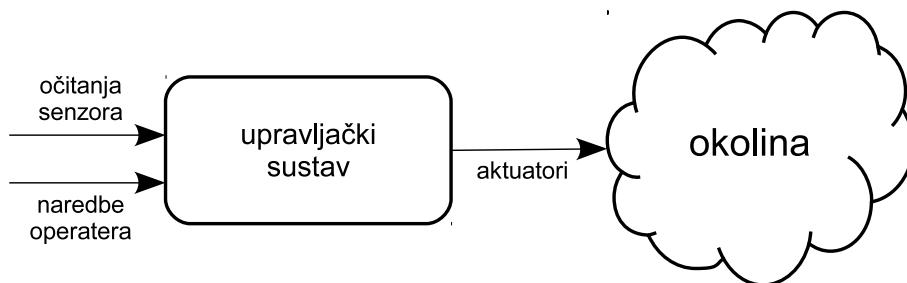
rano iz okoline ili iz nadzornog sustava. Npr. pritisak na senzor stvara signal koji se registrira u nadzornom sustavu (koji tada može pokrenuti alarm na operatorovu zaslонu). Prikupljanje podataka inicirano iz nadzornog sustava je najčešće periodičko, svaki se senzor periodički očitava, obrađuje, eventualno prikazuje i pohranjuje.

1.3.2. Upravljanje bez povratne veze

Upravljanje bez povratne veze, odnosno, upravljački sustav koji ne očitava rezultate svojih upravljačkih odluka donosi odluke na osnovi zahtjeva operatora ili na temelju unaprijed programiranih operacija, bez da prati (za vrijeme upravljanja) kako se njegove odluke sprovode i odražavaju na okolinu. Npr. neki se alatni stroj može programirati da predmet obrađuje na točno određeni način, zadanim brzinama pomicući i alat i predmet obrade. Oblik ulaza te željeni oblik gotova objekta mogu biti ulazi, a program će izračunati sve potrebne radnje, brzinu gibanja komponenata i slično.

Problem ovakvog upravljanja jest što ih se ne smije ostaviti bez nadzora. Npr. ako pri radu alatnog stroja nešto krene krivo, npr. svrdlo pukne, a ne koristi se povratna veza, osim što se proizvod neće oblikovati prema zadanom obliku, stroj se može i dodatno pokvariti. Ipak, za mnoge je sustave ovaj način upravljanja zadovoljavajući, pogotovo stoga što se značajno jednostavnije ostvaruje od upravljanja koje koristi povratnu vezu. Ovakvi se sustavi mogu nadograditi zasebnim senzorima koji će zaustaviti sustav ako se nešto nepredviđeno dogodi (dodatak nije dio upravljanja). Ako računalom izravno upravlja čovjek (npr. preko upravljačke palice) tada čovjek zatvara petlju, tj. on prilagođava upravljanje u skladu s stanjem sustava.

Slika 1.8. prikazuje sustav upravljanja bez povratne veze kojemu su jedini ulazi naredbe operatora te eventualna sporadična očitanja senzora (npr. detekcija početnog stanja). Na osnovu programiranog ponašanja i naredbi operatora sustav vremenski usklađeno šalje naredbe prema aktuatorima.

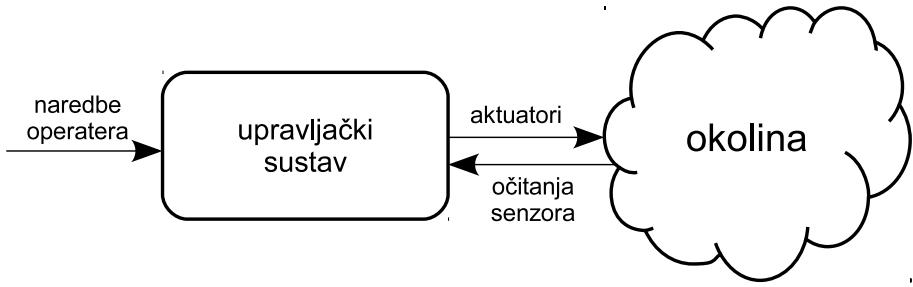


Slika 1.8. Upravljanje bez povratne veze

1.3.3. Upravljanje korištenjem povratne veze

Upravljački sustavi koji koriste povratnu vezu su bitno drugačiji od prethodnih. Slika 1.9. prikazuje načelu shemu takvih sustava. Kod njih upravljanje izravno ovisi i o očitavanju senzora iz okoline (zasnovano je na dostupnosti tih podataka). Očitanje senzora se često radi većom frekvencijom da bi upravljački sustav istom brzinom prilagođavao upravljanje sustavom. Senzori se često ugrađuju u aktuatore da se očita njihovo stvarno stanje i usporedi s izračunatim.

Čovjek je jedan od najboljih primjera sustava za upravljanje s povratnom vezom. Njegov je problem u sporoj reakciji koja se mjeri u desetinkama sekunde, naspram milisekundi i manje (mikrosekunde) koje nude računalno upravljeni sustavi. Čovjek mu može parirati jedino u vrlo složenim sustavima npr. kod kojih treba analizirati sliku i/ili zvuk i kod kojih je dozvoljeno vrijeme reakcije bar nekoliko stotina milisekundi, jer se trenutni algoritmi ipak ne mogu usporediti s čovjekom po mogućnosti prepoznavanja (iako računalo svoju najbolju odluku može donijeti znatno brže).



Slika 1.9. Upravljanje uz povratnu vezu

1.4. Operacijski sustavi za SRSV-e

Pri projektiranju novih SRSV-a kao jednu od prvih odluka koju treba donijeti jest izbor podloge za izgradnju, odnosno, koji će se operacijski sustav koristiti, ako je on uopće potreban (u vrlo jednostavnim sustavima on nije potreban!).

Često je najbolja odluka koristiti neki od komercijalno dostupnih SRSV operacijskih sustava (engl. *real time operating system – RTOS*). Iako takva odluka može podići cijenu sustava, ona može značajno olakšati i ubrzati cijeli proces razvoja. Također, takvo će rješenje vjerojatno biti pouzdanije jer se takvi operacijski sustavi vrlo detaljno ispituju (i već koriste u mnogim drugim sustavima). Nedostatak ovog rješenja, osim cijene, može biti krutost i nemogućnost prilagodbe posebnim zahtjevima sustava koji se izgrađuje. Primjeri komercijalnih operacijskih sustava za SRSV su *VxWorks* i *QNX*.

Alternativa komercijalnim rješenjima su besplatna rješenja kao što je *RTLinux*. Njihov je problem što nemaju korisničku podršku kao komercijalni i često nisu mjerljivi s njima prema svojstvima. Ipak, za ograničene primjene, mogu se i oni iskoristiti.

Treća mogućnost je izgradnja vlastitog sustava (krenuvši "od nule"). Prednosti ovog pristupa su u mogućnosti prilagodbe zadanom problemu i sustavu. Ipak, ovaj pristup treba koristiti jedino kada nema drugog izbora jer zahtjeva značajno više vremena za razvoj i ispitivanja te cijenom može znatno nadmašiti pristupe koji koriste komercijalna rješenja. S druge strane, možda upravo ovo rješenje bude najjeftinije. Sve ovisi o problemu koji se rješava.

1.5. Raspoređivanje zadataka u SRSV-ima

Složeni SRSV-i se ostvaruju korištenjem *višezadaćnosti*. Razlozi su najčešće u mogućnostima ostvarenja upravljanja i u jednostavnosti te izvedbe. Ostali načini bi jednostavno bili suviše složeni, ako bi bili i mogući. Svaka dretva u takvom sustavu obavlja pridijeljeni joj *zadatak* upravljanja pojedinim segmentom sustava. Jednoprocесorski sustavi mogu u jednom trenutku obavljati samo jedan zadatak, tj. izvoditi jednu dretvu. Problem odabira jedne dretve iz skupa dretvi koje su spremne za obavljanje svojih zadataka naziva se problemom *raspoređivanja*.

Uobičajeni principi raspoređivanja dretvi u računalnom sustavu su:

- raspoređivanje prema prioritetu
- raspoređivanje po redu prispjeća
- raspoređivanje podjelom vremena.

Uobičajeno je da se pri raspoređivanju kao prvi kriterij uzima prioritet dretve, a kao drugi (kada prvi ne daje jednoznačno rješenje) red prispjeća (engl. *first-in-first-out – FIFO*) ili podjela vremena (engl. *round-robin – RR*).

Kada se raspoređivanje obavlja prema prioritetu, tada se može dogoditi i istiskivanje trenutno odabrane (*aktivne*) dretve drugom (engl. *preemption*), prioritetnijom. Primjerice, ako se u

obradi prekida dohvate podaci na koje je čekala dretva većeg prioriteta, ona se odblokira i istiskuje trenutnu.

Problemi raspoređivanja u SRSV-ima jesu u specifičnim zahtjevima zadatka, tj. vremenskim ograničenjima postavljenim prema zadacima koje je potrebno zadovoljiti. Problemi su vrlo složeni i zato vrlo opsežno istraživani u znanstvenim krugovima.

Najčešća rješenja složenijih problema raspoređivanja uključuju predimenzioniranje, tj. korištenje bržeg sklopovlja koje će omogućiti veću zalihost procesorske snage i tako smanjiti ili ukloniti problem nemogućnosti raspoređivanja dretvi i u najkritičnijim situacijama. Dretvama se statički pridjele prioriteti prema važnosti zadatka koje obavljaju te ih operacijski sustav raspoređuje prema njima.

1.6. Primjeri SRSV-a upravljenih računalom

Zahvaljujući tehnologiji računala se sve više integriraju u našu okolinu kao *ugrađeni računalni sustavi*. Mnogi od tih sustava imaju značajke i SRSV-a te ih kao takve treba i osmisliti.

Primjeri nekoliko očitijih SRSV-a, koji značajno mogu utjecati na nas i našu okolinu su:

- u vozilima:
 - sustavi pomoći pri kočenju i slični sustavi za sprječavanje proklizavanja
 - upravljanje radom motora radi smanjene potrošnje ili povećane snage
 - sustav održavanja brzine ili razmaka između vozila
 - napredni sustavi detekcije prepreka te samostalnog zaustavljanja vozila
- u vojnoj industriji:
 - za prepoznavanje objekata
 - navođenje letjelica i projektila
 - obradu i prikaz informacija (za pilota, za voditelje timova, ...)
- za upravljanje robotima (robotskim rukama):
 - u medicinske svrhe (operacije, pregledi)
 - u prometu (skretnice, rampe)
 - u industrijskim postrojenjima
 - za upravljanje alatnim strojem
 - za rad na traci (više robota, njihov usklađen rad)
 - za upravljanje (kemijskim) procesom (gdje postoji opasnost za čovjeka)
- za upravljanje većim složenim sustavima:
 - avioni (sakupljanje i obrada podataka te upravljanje) te
 - elektroenergetski sustav, od upravljanje elektranama (i svim njenim procesima) do raspodijele električne energije.

U okviru predmeta neće se razmatrati prethodni složeni sustavi jer bi za njihovo razmatranje trebalo najprije dobro razumjeti njihovu okolinu (npr. trebalo bi proučiti osim načina prikupljanja podataka i njihove utjecaje – trebalo bi razumjeti proces, ne samo s programerskog gledišta).

Ipak, u okviru prikaza postupaka koristit će se dijelovi raznih sustava, primjerice problem upravljanja liftom.

1.7. Problem složenosti

Većina sustava je složeno. Što je sustav složeniji to ga je teže razumjeti i lakše napraviti greške, teže je uočiti odnosi među komponentama, shvatiti način rada, redoslijed obavljanja neke aktivnosti i slično.

Razvoj elektronike omogućio je njenu sve veću integraciju u sve sustave, povećavajući učinkovitost takvih sustava i šireći njihovu uporabljivost. Međutim, upravljanje takvim sustavima postaje sve složenije. Upravljanje mora uzeti u obzir mnoštvo komponenti koje treba uskladiti, odabrati odgovarajuće komunikacijske protokole, kako za komponente koje izravno čine sustav (npr. senzori, aktuatori), tako i za one udaljenije koje pružaju odgovarajuće usluge (npr. servisne informacije, poslužuje zahtjeve klijenata). Pri projektiranju pojedine komponente treba uzeti u obzir ne samo njenu osnovnu namjenu (upravljanja ili nadzora) već i načine na koje se ta komponenta povezuje i integrira s ostatom sustava.

Ako se za primjer uzme automobil, onda se može utvrditi da se upravljanje sve više i više prebacuje na elektroničko, gdje je čovjek preko ulaznih naprava komunicira s računalom, a koje onda upravlja automobilom. Upravljački sustav automobila sastoји se od senzora koji daju informacije te aktuatora koji provode akcije. Primjerice, u senzore se mogu uključiti: volan, pedale (gas, kočnicu kvačilo), ručica mjenjača, ručna kočnica, senzori u motoru (stanje komponenti, kao što su temperatura, tlak), senzori na kotačima, senzori okoline (temperatura, podloga, osvjetljenje, vlažnosti), senzori položaja i smjera (npr. preko GPS signala), senzori u kabini, podaci o svojstvima motora, mape za navođenja i slično. Aktuatori upravljaju: smjerom kretanja, motorom, kočnicama, svjetlima, klima uređajima, zračnim jastucima, alarmom i slično. Svaka od navedenih komponenti je zasebni podsustav sa svojim svojstvima i vremenskim ograničenjima. Gotovo sve komponente su integrirane u automobil, ali mogu se koristiti i informacije iz udaljenih sustava, kao što su GPS i prometne informacije. Ovakav raspodijeljeni sustav je samo jedan primjer današnjih složenih sustava.

Sustav ili samo jedna njegova komponenta koja se nadzire ili kojom se upravlja može biti različite složenosti. Složenost je svojstvo sustava koji treba izgraditi (problema koji treba riješiti) i ne bi trebala biti povećavana predloženim rješenjima. Ako za neki problem postoji nekoliko mogućih rješenja, treba nastojati izabrati *najjednostavnije*, to je jedan od osnovnih ciljeva. Npr. ako se upravljanje može ostvariti samo s jednom jednostavnom petljom, onda tako treba napraviti.

U većini slučajeva, najjednostavnije moguće rješenje je najbolje rješenje.

Iznimno, zbog nekih drugih razloga kao što su sklopovski zahtjevi, bolje mogućnosti proširenja i održavanja, može se odabrati i neko drugo (složenije) rješenje.

U većim projektima složenost je među najvećim problemima. SRSV-i, koji zahtijevaju znatno strože kriterije u gotovo svim pogledima (pogotovo otklanjanje pogrešaka), posebno su osjetljivi na složena rješenja. Zato je od iznimnog značaja da projektanti takvih sustava, kao i oni koji kasnije sudjeluju u njegovoj izgradnji, jako dobro poznaju metode koje se koriste pri analizi te da znaju odabrati prave algoritme za rješavanje problema. "Pravi" algoritam treba biti dostatan za rješavanje problema ali također treba biti što je moguće manje zahtjevan na sklopovlje. Nije uvijek prikladno odabrati najbolji/najprecizniji/najopćenitiji algoritam jer on može značajno poskupiti izradu sustava.

Svaki je sustav poseban, po nečemu različit od drugih. Stoga i rješenja koja se primjenjuju u jednom sustavu ne moraju biti tako uspješna u drugom, ili čak mogu biti neprikladna. Da bi donijeli prave odluke projektanti trebaju poznavati osnovne načine ostvarenja sustava, ali i uobičajene postupke korištene u praksi ("postupke dobre inženjerske prakse"). Najbolje bi bilo kada bi oni već imali iskustva s različitim sustavima te bi usporedbom s onim kojeg treba izgraditi mogli odrediti koji su postupci primjereni. U nedostatku vlastitog iskustva mogu se pogledati i primjeri iz literature.

Za odabir prikladnog postupka potrebno je jako dobro upoznati sustav koji se izgrađuje (i kojemu se dodaje upravljačka komponenta) te način njegova (željena) rada. Postoje mnogi postupci specifikacije sustava i načina njegova rada, uključujući i standardizirane (npr. *UML dijagrami*) i formalne (npr. formule, formalna logika), a čiji je osnovni cilj bolje razumijevanje sustava od strane sudionika u njegovoj izgradnji. Prednost standardiziranih i formalnih načina

specifikacije jest u tome što ih svi jednako interpretiraju i što ih se može iskoristiti za analizu korištenjem prikladnih alata. Prednost grafičkih postupaka jest u jednostavnijem razumijevanju od strane čovjeka, kojemu vizualna slika može znatno pomoći da shvati sustav ili njegove dijelove i način rada.

Postupke koji pomažu u analizi, oblikovanju i izgradnji mogli bi podijeliti prema fazi u kojoj se koriste, prema tome tko ih koristi, prema problemu koji obrađuju, prema arhitekturama kojima su namijenjeni i slično. U poglavlju 2. navedeni su neki od tih postupaka, s naglaskom na primjenu u SRSV-ima.

1.7.1. Grafički postupci

Među poznatije grafičke postupke prikladne i za SRSV-e (pri analizi i oblikovanju) spadaju UML dijagrami:

- *dijagrami obrazaca uporabe i sekvensijski dijagram* za analizu sustava (interakcije, željeno ponašanje),
- *dijagram stanja i dijagram aktivnosti* za analizu mogućeg rada te
- ostali UML dijagrami za detaljnije razmatranje mogućeg ostvarenja i ponekog detalja.

Osim UML dijagrama, za analizu i ispitivanje mogućih rješenja prikladne mogu biti *Petrijeve mreže* i *vremenske Petrijeve mreže*.

Pri pojedinačnoj analizi (početku analize) mogu se koristiti i *neformalne skice*, ali bi one prije ulaska u službenu dokumentaciju (i dijeljenja s ostatkom tima) ipak trebale biti prevedene u neki formalni oblik, a da se izbjegne mogućnost drukčije interpretacije.

1.7.2. Metoda “podijeli i vladaj”

Uobičajeni postupci smanjenja složenosti zasnivaju se na pravilu *podijeli i vladaj*. Ovo je prvo pravilo u temeljnim načelima oblikovanja arhitekture zato jer je i najvažnije. Odgovarajuća podjela će značajno olakšati razumijevanje sustava i smanjiti utjecaj složenosti sustava kao cjeline.

Podjela se može napraviti na mnoštvo načina. Često se i za jedan sustav/problem podjela radi na više načina. Primjerice, operacijski sustav može se podijeliti na slojeve: sloj sklopolja, sloj jezgre, sloj primjenskih programa, ali i na komponente (tj. podsustave): upravljanje napravama, upravljanje spremnikom, upravljanje dretvama, datotečni podsustav, mrežni podsustav itd. Podjele mogu biti ostvarene na razini izvornih kôdova ili na razini izvođenja ili na razini povezivanja s drugim raspodijeljenim programima.

Pravilnom podjelom nastaju dijelovi (slojevi, komponente, razine) koji su zasebno značajno jednostavniji od cijelog sustava. Zato su i čovjeku razumljiviji te ih on može bolje osmislit (npr. bit će brži, manji, smanjit će se broj grešaka). Druga prednost podjele jest što se pojedini dio može pridijeliti zasebnom inženjeru ili timu koji će se njemu posvetiti (analiza i/ili oblikovanje i/ili programiranje). Komunikacija s drugim dijelovima mora biti identificirana i uobličena u dobro definirano *sučelje*.

U ostvarenju upravljanja složenim sustavima podjele se mogu ostvariti i prema *hijerarhijskim razinama* i na *dretve* koje surađujući upravljaju sustavom.

Nedostaci korištenja metode podijeli i vladaj su u sustavima s visokom očekivanom učinkovitošću, gdje dodatni slojevi, razine i slično unose dodatne poslove (npr. svaki sloj ima svoje omotače koje dodaje na poruku prethodnog sloja). U takvim se sustavima može ‘tunelirati’ kroz slojeve, tj. izbaciti neke slojeve ugrađujući potrebne operacije na jednom mjestu (jednom sloju).

1.8. Zašto proučavati SRSV-e?

Ugrađeni SRSV-i, barem oni jednostavniji su svuda oko nas, čak i u jednostavnijim igračkama i kućanskim aparatima. Dobar dio tih sustava koristi mikroprocesor kao osnovnu upravljačku jedinicu koja za upravljanje koristi programe. U budućnosti će se broj takvih sustava samo povećavati te će i potrebe za njihovo projektiranju i izgradnju rasti. Više znanja o takvim sustavima je svakako potrebno inženjerima koji će se u takve projekte upuštati.

Projektiranje i ostvarenje SRSV-a zahtijeva značajno dublje razmatranje problema u usporedbi s aktivnostima za ostvarenje drugih sustava. Posebno su bitne aktivnosti za projektiranje sustava koje uključuju:

- odabir sklopovskih i programske komponenti koje će zadovoljiti zahtjeve uz primjerenu cijenu
- odabir (komercijalno) dostupnog operacijskog sustava ili projektiranje vlastite jezgre
- odabir razvojnih alata (uključujući i programske jezike)
- maksimiziranje otpornosti na greške i povećanje pouzdanosti prikladnim postupcima projektiranja i ispitivanja
- oblikovanje, planiranje i izvođenje ispitivanja tijekom cijelog procesa razvoja sustava.

Predviđanje ponašanja sustava, očitovanje njegova ponašanja, uočavanje *vremena reakcije* te postupci njegova smanjenja su temeljne operacije koje treba provoditi pri izgradnji SRSV-a. Znanje stečeno proučavanjem SRSV-a, odnosno prethodno navedenih aktivnosti će se moći iskoristiti i kod običnih sustava. Stečeno znanje moći će se iskoristiti za izgradnju kvalitetnijih sustava, s manje kôda, veće brzine, pouzdanosti i slično.

Zbog svojstava SRSV-a često se krivo zaključuje o njima. Uobičajene zablude uključuju:

1. SRSV-i su brzi sustavi.
2. Proučavanje SRSV-a svodi se na teoriju raspoređivanja zadataka.
3. Algoritam mjere ponavljanja (engl. *rate monotonic*) rješava problem raspoređivanja u SRSV-ima.
4. Nema potrebe za izgradnjom novih operacijskih sustava za SRSV-e jer već postoji mnoštvo komercijalno dostupnih, pa i besplatnih.
5. Postoji univerzalna, široko prihvaćena metodologija za oblikovanje SRSV-a.

Razlozi zašto prethodne tvrdnje nisu istinite bit će implicitno ugrađene u ostatak knjige.

Pitanja za vježbu 1

1. Koje značajke odvajaju sustave za rad u stvarnom vremenu od ostalih sustava?
2. Što je to logička a što vremenska ispravnost (programa, sustava)?
3. Opisati "stroge", "ublažene" i "čvrste" SRSV-e. ◁
4. Što je najbitnije za SRSV-e koji svoje upravljanje temelje na obradi događaja? ◁
5. Najčešća ocjena nekog sustava s aspekta stvarnog vremena jest u pogledu reakcije na događaje. Kakvi se sve događaji razmatraju te kakva treba biti reakcija SRSV-a?
6. Zašto svaki operacijski sustav nije pogodan za SRSV-e?
7. Zašto bilo kakvo sklopolje nije prikladno za korištenje u SRSV-ima?
8. Opisati atribute: determinizam, pouzdanost, robusnost u kontekstu SRSV-a.
9. Kako se upravlja diskretnim, a kako kontinuiranim sustavima?
10. Usporediti upravljanje koje ne koristi povratnu vezu s onim koje koristi.
11. Koji je osnovni razlog prisutnosti grešaka u programima? ◁
12. Zašto je "problem složenosti" najveći problem pri izgradnji SRSV-a? Koji se postupci koriste radi smanjenja složenosti? ◁
13. Koje su uobičajene zablude o SRSV-ima.

2. Modeliranje sustava

Složenost stvarnih sustava je osnovni razlog korištenja modela. Model je pojednostavljeni prikaz sustava s određene perspektive, zanemarujući neka druga svojstva sustava. Korištenjem modela mogu se lakše razaznati svojstva sustava, uočiti problemi i slično. Ipak, znajući da je model samo pojednostavljeni slika sustava, prije pravog korištenja, stvarni sustav treba ispitati u stvarnom okruženju.

U postupku projektiranja sustava treba odlučiti *kako* nešto napraviti, povezati, ispitati, i slično. Postupci određivanja prikladnih metoda (koje će reći *ovako*) su vrlo različiti i ovise o projektantu, tj. njegovom poznavanju raznih postupaka i rasudivanju o njihovoj prikladnosti. Postupci mogu biti *neformalni* ili zasnovani na nekom standardiziranom, *formalnom* načinu oblikovanja.

U nastavku će se prikazati moguća uporaba oba tipa postupaka na primjeru projektiranja hipotetske upravljačke komponente za *lift u zgradu*.

2.1. Neformalni postupci

Neformalni postupci omogućuju jednoj osobi brži pristup do rješenja korištenjem (samo) njoj prikladne metode. *Skica* koju je projektant sebi na brzinu nacrtao (npr. rukom na papiru) možda će mu omogućiti da bolje vizualizira problem i riješi ga brže nego da primjerice za isto koristi *UML* komunikacijski dijagram za koji će mu trebati puno više vremena da ga nacrtava.

Problemi neformalnih postupaka su upravo u njihovoj neformalnoj strukturi koja se na različite načine može interpretirati i dovesti do problema. Ipak, ako se koriste samo kao pomoćno sredstvo pri rješavanju dijela problema mogu značajno olakšati razumijevanje problema ili predloženog rješenja.

2.1.1. Skica

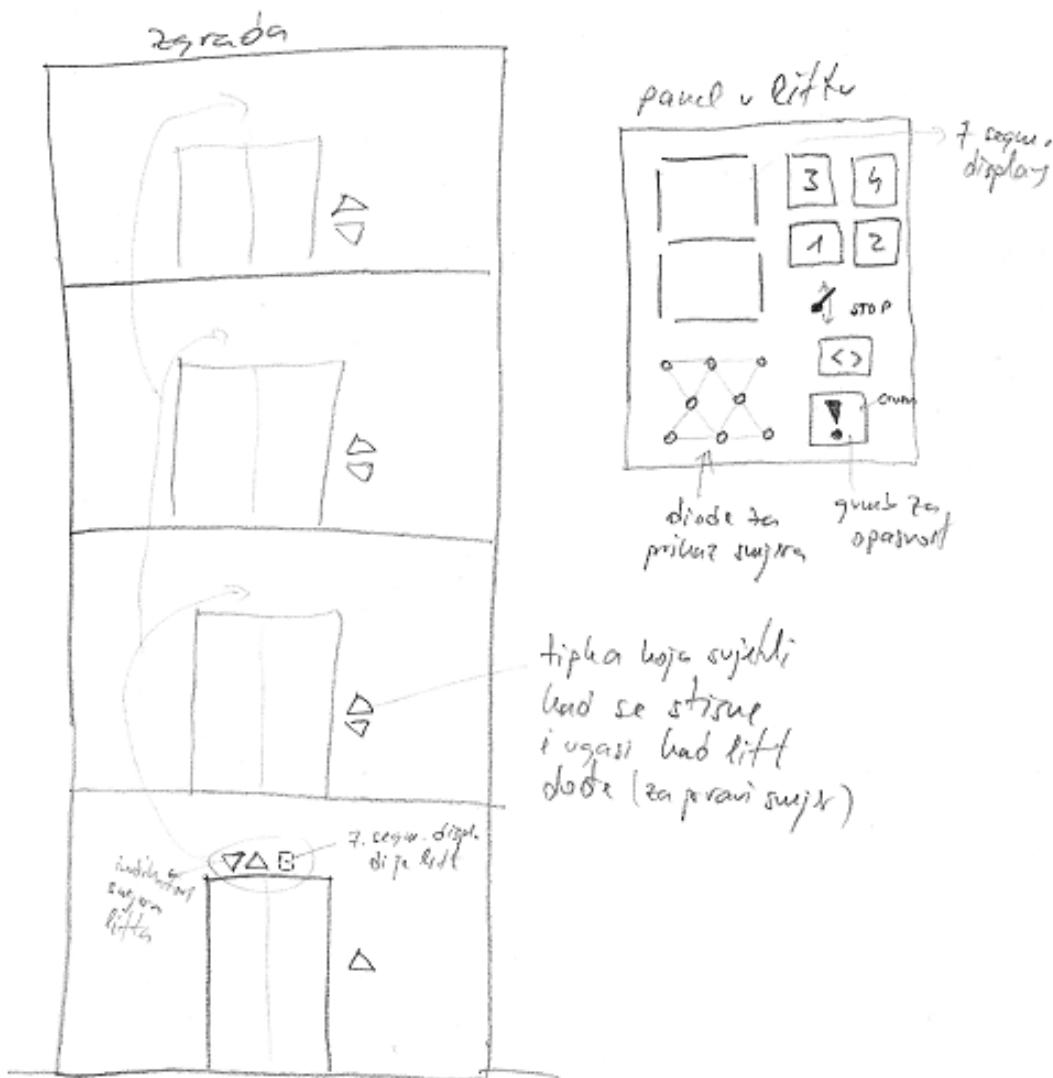
Slika 2.1. prikazuje primjer skice zgrade i sučelja prema liftu, koja može olakšati početno shvaćanje problema. Vrijeme potrebno za izradu ovakve skice je svega par minuta, a njena korisnost za početnu analizu može biti znatna.

Slične potrebe mogu nastati i za druge elemente sustava, za analizu prikladnosti algoritama (izvođenjem na primjeru), skicu stanja sustava u nekom trenutku i slično. Prednosti skice su:

- nastaje u kratkom vremenu
- omogućuje prikaz različitih elemenata (koji se u formalnim oblicima ne stavljuju zajedno)
- nisu potrebni nikakvi alati.

Skica u neformalnom obliku bi trebala biti samo privremeno sredstvo u nekoj fazi razvoja i ne bi smjela biti dio dokumentacije. Zapravo, svim "formalnim" postupcima može prethoditi ovakva skica koja će olakšati izradu formalnih. Npr. u ozbiljnijim projektima prethodna slika će se nacrtati pravim alatima i postupcima (nacrt zgrade s ucrtanim liftom u pravom mjerilu, panel kao elektronička komponenta, ...) i kao takva ući u dokumentaciju (a ne u ovakovom obliku kao na slici 2.1.).

Osim grafičkog prikaza u obliku skice, neformalni postupci obuhvaćaju i druge oblike izražavanja, npr. definiciju problema običnim tekstrom.



Slika 2.1. Skica lifta u zgradi

2.1.2. Tekstovni opis

Početna definicija problema se najčešće zadaje tekstovnim opisom. Npr. za lift, početni opis može biti zadan tekstrom u *prirodnom jeziku* prema primjeru 2.1.

Pri analizi tekstrom zadanog zadatka projektant će si skicirati moguće rješenje, npr. prema slici 2.1. U ovoj fazi analize zahtjeva neki su zahtjevi izravno ucrtani u skicu, a drugi koji su samo načelno zadani su već početno i osmišljeni. Primjerice, indikatori smjera gibanja liftova koji se nalaze na svakom katu su dvije odvojene žaruljice (po katu) te 7-segmentni indikator trenutnog položaja lifta. Na panelu u liftu se također koristi 7-segmentni indikator za prikaz položaja lifta. Međutim, za smjer na panelu se koristi osam dioda jer je možda početna namjera projektanta bila da se i za stanje mirovanja nešto prikaže (npr. crtica).

Već na skici se već površnom analizom mogu uočiti mogući problemi u prvoj ideji te popraviti, također skicom. Npr. korištenje 7-segmentnog indikatora zadovoljiti će zgrade do 10 katova (stajanja), s time da će se za prizemlje morati koristiti oznaka 0 (umjesto npr. PR). Ukoliko zgrada (ili sustav zgrada) za koji se projektira lift neće imati više od 10 katova, i navedeno rješenje može biti zadovoljavajuće.

Primjer 2.1. Primjer tekstovnog opisa

Zadatak: Projektiranje upravljačkog sustava za liftove

Opis:

Projektirati upravljački sustav za liftove koji se ugrađuju u zgrade s minimalno tri kata.

Sučelje prema korisnicima:

Na svakom katu u blizini vrata lifta trebaju biti dvije tipke za poziv lifta prema gore i prema dole (osim na zadnjem katovima gdje je dovoljna jedna tipka). Kad se takva tipka stisne ona treba svijetliti dok lift ne stigne na taj kat, stane na njemu te se namjerava dalje kretati u zahtijevanom smjeru.

Iznad ulaza u lift trebaju biti indikatori smjera kretanja lifta (za gore i za dole) te prikaz trenutnog kata na kojem se lift nalazi.

U liftu se treba nalaziti panel s tipkama i indikatorima: za svaki kat treba postojati zasebna tipka koja će svijetliti ako je stisнута (a označava da će lift stati na tom katu), prikaz trenutnog kata gdje se lift nalazi prikaz smjera gibanja lifta (ništa, gore, dole), sklopka za trenutno (hitno, ručno) zaustavljanje lifta (STOP), tipka zahtjeva za ponovno otvaranje vrata kada lift stoji na katu (<>), tipka zahtjeva za pomoć (npr. kad se lift pokvari i ne ide dalje).

Ponašanje lifta

Kada lift stoji besposlen, on stoji na nekom katu sa zatvorenim vratima. Prvi zahtjev koji tada nađe treba ga pokrenuti. Kada je to pritisak na tipku u liftu ili pritisak na tipku s nekog kata, lift treba otići na traženi kat te otvoriti vrata. Ukoliko zahtjev dođe za vrijeme pomicanja lifta, lift treba stati na traženom katu ako je taj katu na putu kojim lift ide i ako se taj zahtjev može obraditi dalnjim pomicanjem lifta u istom smjeru. Inače se zahtjev poslužuje naknadno, kad lift posluži zahtjeve u trenutnom smjeru.

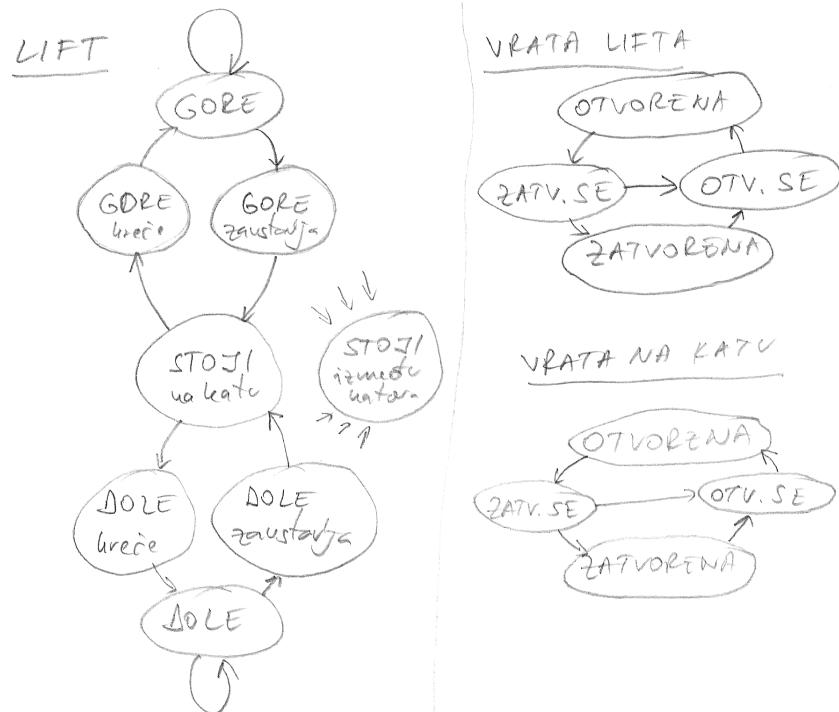
Pritisak na tipku OTVORI treba razmatrati dok lift stoji sa zatvorenim vratima ili su ona u postupku zatvaranja ili su otvorena. U tim slučajevima treba prvo pokrenuti otvaranje vrata, ako već nisu otvorena. Ako su vrata već otvorena, treba produljiti stanje s otvorenim vratima (kao da su vrata tek otvorena).

2.1.3. Dijagram stanja

Iako dijagram stanja posjeduje formalne elemente (stanja, prijelazi), u neformalnom razmatranju na njega se mogu staviti i drugi neformalni elementi. Primjerice, pri razmatranju ostvarenja upravljačke komponente, projektant je možda prvo napravio stanja kao na slici 2.2.

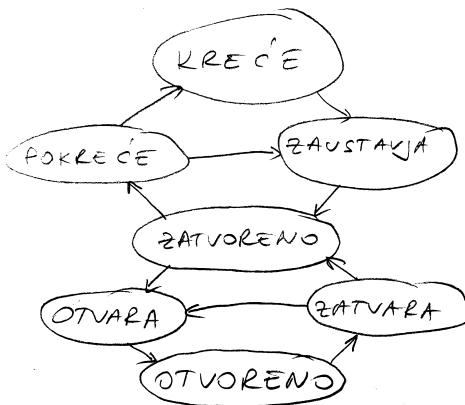
Stanje označeno sa *STOJI između katova* označava da se iz svih stanja može preći u to stanje. Iako se to može nacrtati i strelicama iz svih ostalih stanja, te strelicama iz tog stanja u sva ostala (kad se otpusti hitno stajanje), to bi crtanje učinilo značajno dijagram osjetno složenijim. Hitno stajanje je posebno stanje za koje treba drukčije reagirati u odnosu na sva ostala stanja i događaje. Stoga ima smisla to stanje izdvojiti i ne uključivati ga u dijagram kojemu je osnovna namjena osmisiliti algoritam za upravljanje u uobičajenim okolnostima. Njegovo prisustvo na dijagramu, iako izdvojeno, može poslužiti da se i pri osmišljavanju algoritma za uobičajene okolnosti ipak postupno razmišlja i o problemu hitnog zaustavljanja.

Daljnjom analizom dijagrama stanja sa slike 2.2. moguće je vrlo brzo napraviti analizu, otkriti moguće probleme te napraviti "poboljšani" dijagram. Primjerice umjesto zasebnih stanja za *gore* i *dolje* te zasebnog dijagrama za vrata (kada vrata nisu zatvorena lift mora stajati), može se napraviti novi dijagram, koji informaciju o smjeru kretanja lifta ne pokazuje u dijagramu već u dodatnoj strukturi podataka (tamo će među ostalima biti i trenutni vertikalni položaj lifta).



Slika 2.2. Dijagram stanja lifta, vrata u liftu i vrata na katu

Primjer takvog dijagrama prikazuje slika 2.3.



Slika 2.3. Dijagram stanja lifta, nakon spajanja

2.1.4. Nedostaci neformalnih postupaka

Osim navedenih postupaka (skice, tekstovnog opisa i dijagrama stanja), mogu se koristiti i drugi neformalni postupci (npr. i pseudokôd). Izgradnja jednostavnih nekritičnih sustava može biti zasnovana i na jednostavnoj analizi takvim postupcima. Ukoliko programsku komponentu razvija jedna osoba ili mali broj osoba, i jednostavna skica i opis tekstrom mogu biti dovoljni.

Za kritične sustave i za složenije sustave samo osnovna analiza (npr. neformalnim postupcima) će uglavnom biti nedostatna. Osnovni problem jest naravno složenost i naša nemogućnost da ju savladamo na razini cijelog sustava. Tome upravo služe modeli, da podijele sustav i da prikažu sustav iz pojedinih perspektiva. Neformalni postupci (npr. skica, tekst, dijagram stanja) pomažu u tome, ali niti su dovoljno precizni (ne mogu se svi detalji postaviti na skice) niti se jednakom interpretiraju od različitih projektanata. Čak i ista osoba koja je napravila skicu može ju naknadno na drugi način interpretirati (npr. zbog potrebnih proširenja koji su se pojavili

mjesecima i godinama nakon početne izrade). Neki problemi se neće uočiti sve do kasne faze projekta (kada će biti puno teže prilagoditi sustav) ili će se oni pokazati tek u radu sustava i pri tom prouzročiti ozbiljne probleme. Naknadno ispravljanje sustava (npr. programske potpore) često će uzrokovati narušavanje arhitekture sustava, koja posljedično otežava nadogradnje i održavanja (ukupno znatno podiže cijenu sustava).

Neformalne postupke stoga treba koristiti ograničeno, npr. pri osmišljavanju mogućih rješenja nekih problema (u bilo kojoj fazi razvoja). Pri temeljitijoj analizi treba ih izbjegavati jer oni ne posjeduju potrebnu razinu detalja. Također, zbog neformalnosti, razni sudionici ih mogu različito shvaćati što vodi do problema u njihovoj međusobnoj komunikaciji.

2.2. Formalni postupci u oblikovanju sustava

Za većinu prethodnih neformalnih postupaka postoje odgovarajući formalni, propisani standardi. Iako će i dalje slika sustava koji se upravlja biti od velike koristi, odnosi u sustavu se mogu detaljnije i jednoznačnije prikazati formalnim oblicima, kao što su UML dijagrami.

U detaljnijim analizama i postupcima pronalaska ispravnog rješenja mogu se koristiti i Petrijeve mreže te vremenske Petrijeve mreže, gdje je vrijeme jedan od bitnih pokretača promjena ili je bitno da se poštuju zadana vremenska ograničenja. Za modeliranje ponašanja sustava često se koriste i automati stanja (engl. *Finite State Machines*), ali se oni ovdje ne razmatraju.

U nastavku je prikazano korištenje UML dijagrama i Petrijevih mreža na primjerima vezanim uz problem ostvarenje upravljanja liftom i drugim problemima.

2.2.1. UML dijagrami obrazaca uporabe

Identifikacija usluga/operacija, korisnika usluga i komponenata sustava se može prikazati UML dijagramom obrazaca uporabe (engl. *use case*).

Izrada dijagrama ovisi o tome što se želi prikazati i analizirati. Ukoliko su to operacije upravljačkog računala na zahtjev putnika ili stanja lifta, može se izgraditi UML dijagram obrazaca uporabe kao na slici 2.4.

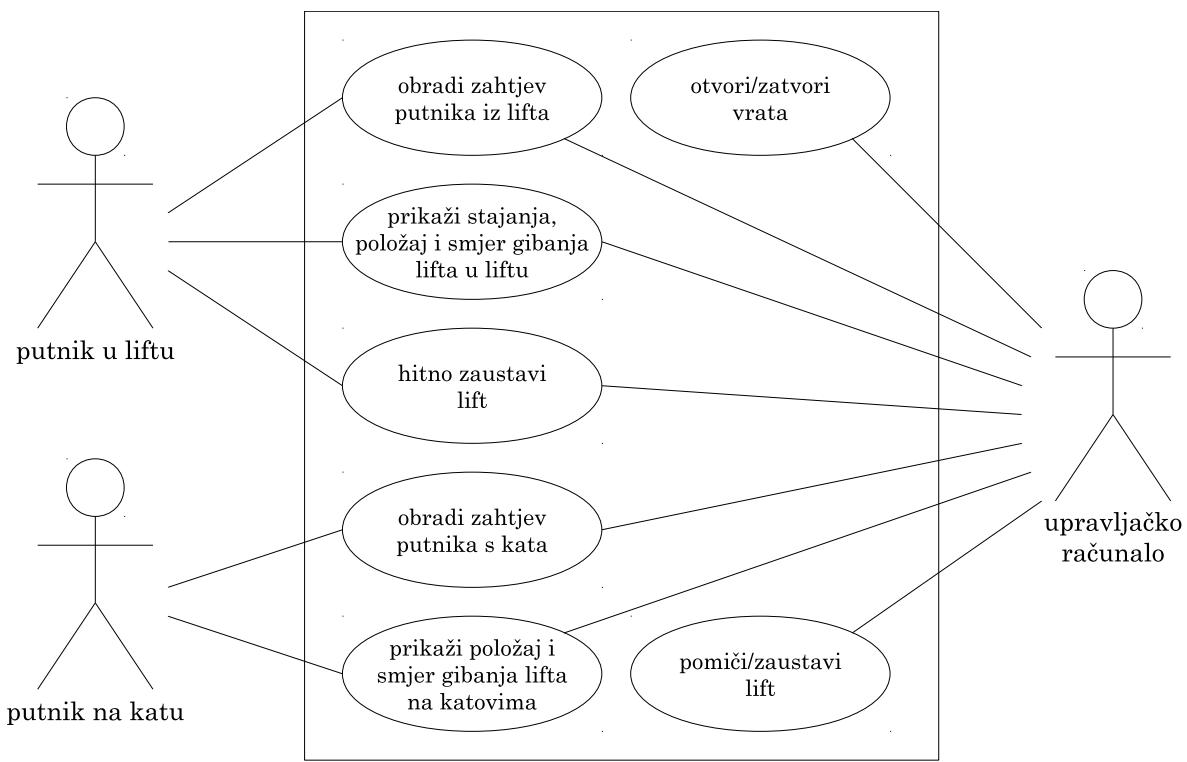
Svaka se operacija može detaljnije opisati tekstom ili nekim drugim dijagramima (npr. sekvenički i kolaboracijski). Primjerice, pomicanje lifta uključuje dohvata trenutnog položaja i stanja lifta te postojećih zahtjeva putnika.

2.2.2. UML sekvenički i kolaboracijski dijagrami

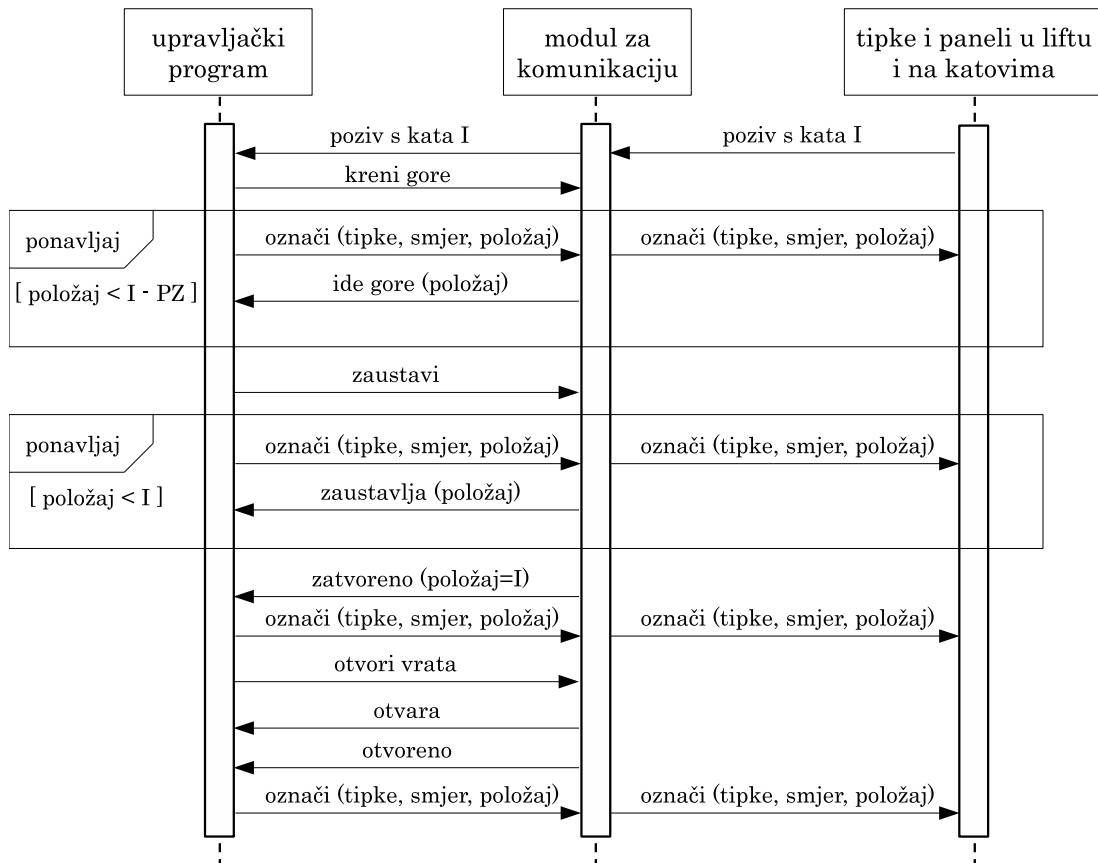
Scenariji slijeda aktivnosti u vremenu se osim prirodnim jezikom i strukturiranom tekstrom mogu prikazati i grafički, primjerice sekveničkim dijagramom. Tko s kim komunicira najlakše je uočiti iz kolaboracijskog dijagrama.

Sekvenički dijagrami se često koriste pri projektiranju i analizi postojećih SRSV-a zbog mogućnosti prikaza vremenskog uređenja događaja i akcija. Primjerice, ako treba analizirati akcije i poruke koje se zbivaju od zahtjeva putnika s jednog kata do dolaska lifta na taj kat, može se izgraditi sekvenički dijagram prema slici 2.5.

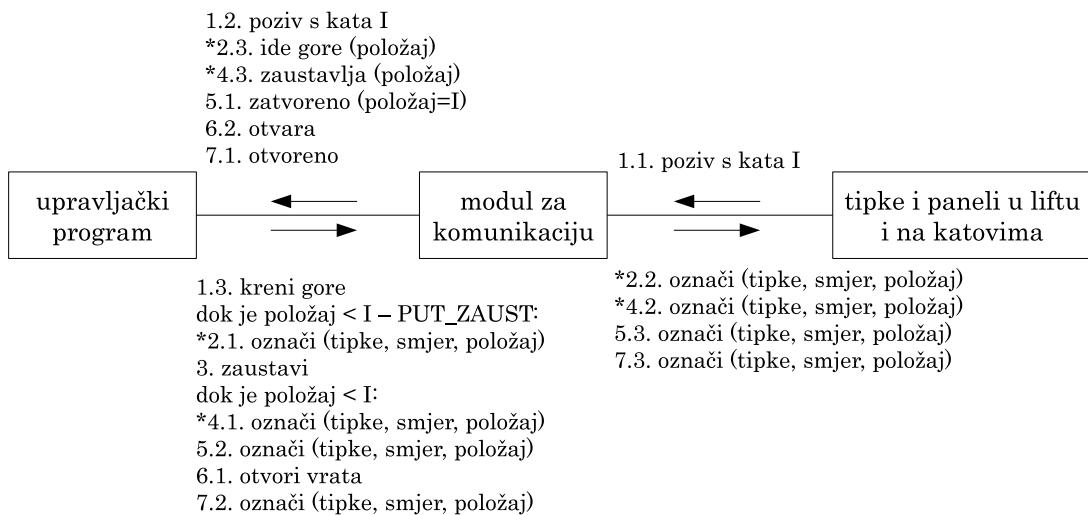
Dijagram obrazaca uporabe, sekvenički i kolaboracijski dijagrami su, osim za postupke analize, prikladni i za izradu početnog dokumenta specifikacije te za pisanje tehničke dokumentacije.



Slika 2.4. UML dijagram obrazaca uporabe



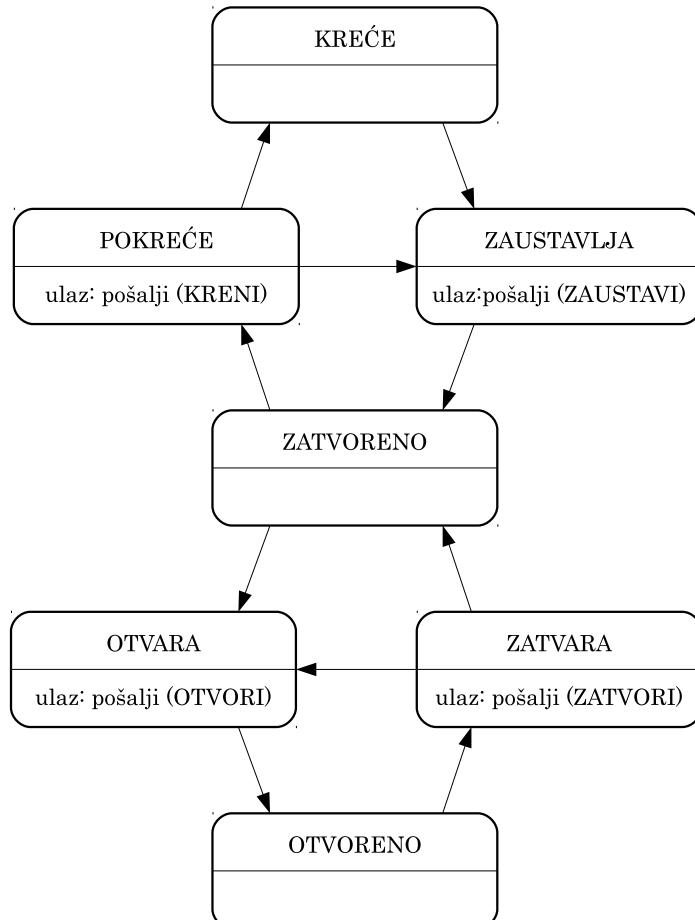
Slika 2.5. UML sekvenčni dijagram



Slika 2.6. Komunikacija među komponentama

2.2.3. UML dijagram stanja

UML dijagram stanja jest proširenje “običnog” dijagrama stanja koji dozvoljava u formalnom obliku detaljniji prikaz uzroka i posljedice pojedinih stanja i događaja. Dijagram stanja za kretanje lifta može se zapisati u formalnom obliku pomoću UML dijagrama stanja (slika 2.7.), pritom dodajući i dodatne informacije, npr. poruke koje treba poslati pri ulasku u pojedino stanje.



Slika 2.7. UML dijagram stanja lifta

Kada su prijelazi većinom uzrokovani dovršetkom određenih procesa, primijereniji dijagram jest dijagram aktivnosti, koji za razliku od dijagrama stanja pruža više mogućnosti za prikaz paralelnih aktivnosti.

UML dijagrami mogu uključivati mnoge dodatne informacije, kao što je navedena poruka pri ulasku u stanje. Većinom su dodatne informacije u UML dijagramu dobrodošle, posebice pri izradi tehničke dokumentacije. Međutim, u procesu analize treba biti pažljiv s dodavanjem detalja jer će oni smanjiti čitljivost dijagrama, a osnovna namjena dijagrama u toj fazi je povećanje razumijevanja i uočavanje mogućih problema u arhitekturi, komunikaciji i slično.

Složenija programska rješenja koja koriste podsustave, komponente i različite programe mogu se vizualizirati dijagramom komponenata te dijagramom ugradnje ako su te komponente na raspodijeljenim računalima.

2.2.4. Petrijeve mreže

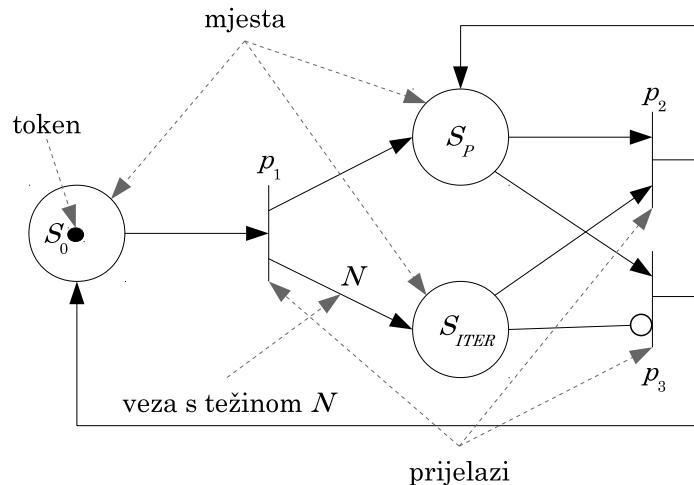
Petrijeve mreže su sredstvo za vizualizaciju i analizu dinamičkih procesa u sustavima s više aktivnih komponenata (procesa, dretvi, poruka, ...). Nazvane su prema Carlu Adamu Petriju koji ih je osmislio i prvi puta koristio 1939. godine. Petrijeve mreže po izgledu su slične usmjerenim grafovima, ali je semantika prijelaza iz stanja u stanje proširena. Petrijeve mreže se koriste u raznim područjima, primjerice kod: paralelnog programiranja, analize podataka, modeliranja procesa, procjene pouzdanosti, oblikovanja programske potpore i simulacije sustava.

Petrijeve mreže sastoje se od tri osnovna elementa:

- mesta (stanja, engl. state/places),
- prijelaza (tranzicija) i
- znački (engl. token) koji prelaze iz jednog mesta u drugo (ili isto) preko prijelaza.

Veze između mjesta i prijelaza te prijelaza i novog mesta su označene vezama. Veze mogu biti *normalne* (težine 1), s pridijeljenom težinom npr. N te *inhibicijske veze* (težine 0). Veze također mogu biti *ulazne* (značke preko njih dolaze u mrežu) te *izlazne* (značke odlaze iz mrežu).

U Petrijevoj mreži može istovremeno postojati više znački, od kojih svaka predstavlja neko sredstvo, komponentu ili slično. Slika 2.8. prikazuje jednu Petrijevu mrežu s naznačenim elemenima mjesta, prijelaza, znački, veze s težinom N te inhibicijske veze. U prikazanom primjeru su stanja i prijelazi označeni simbolima S_0 , S_P i S_{ITER} te p_1 , p_2 i p_3 , ali oni nisu neophodni dio mreže.



Slika 2.8. Primjer Petrijeve mreže

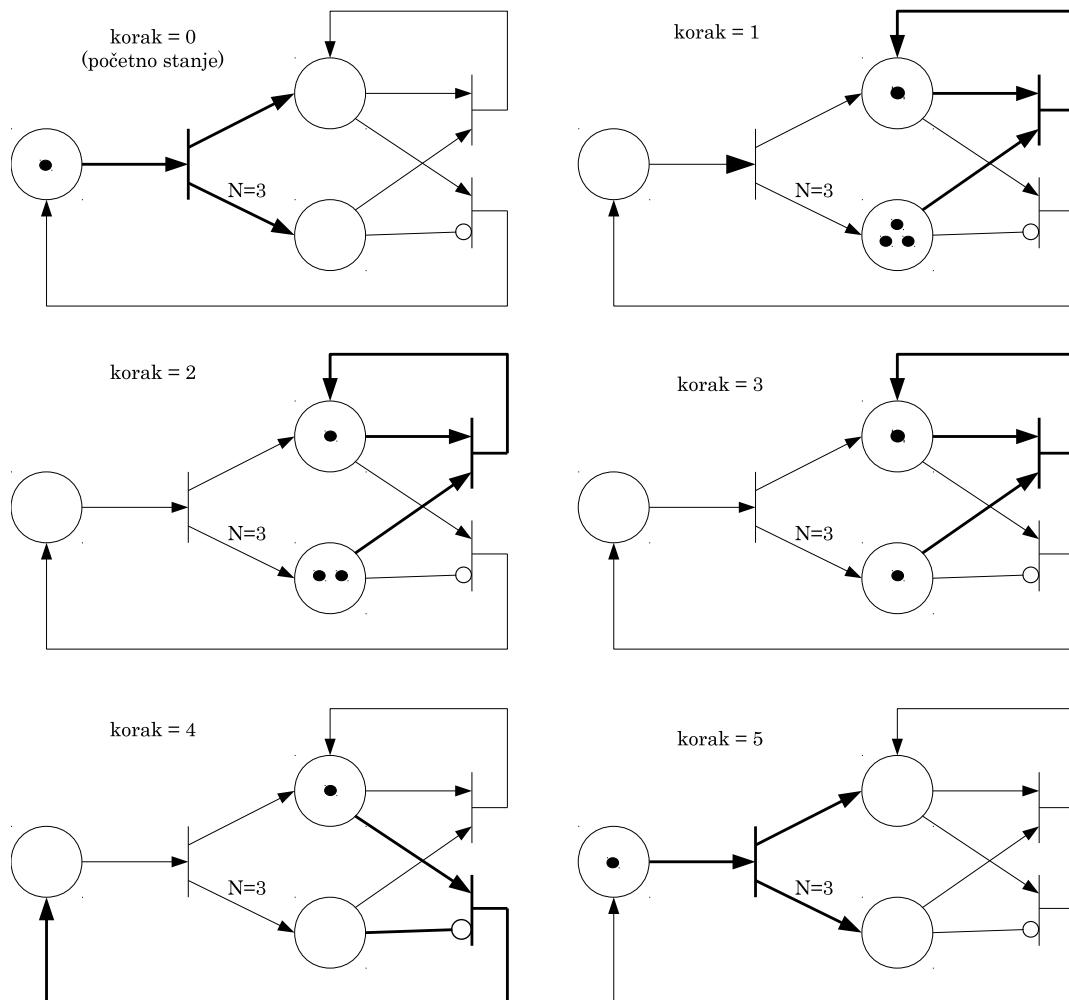
Da bi se pojedini prijelaz aktivirao, tj. obavio prijenos značke, sve ulazne veze (do prijelaza)

moraju biti spremne za prijelaz, tj. odgovarajući broj znački mora biti u polaznim stanjima. Za prijelaz p_1 barem jedna značka mora biti u mjestu S_0 ; za prijelaz p_2 stanja S_P i S_{ITER} moraju imati barem po jednu značku, dok za prijelaz p_3 , koji ima jednu ulaznu inhibicijsku vezu, mjesto S_P mora imati barem jednu značku dok u mjestu S_{ITER} ne smije biti značka. U početku je samo prijelaz p_1 omogućen jer je tamo jedna značka.

Po aktiviranju prijelaza p_1 , jedna će značka otići u mjesto S_P a N znački u S_{ITER} . Aktiviranjem prijelaza p_2 jedna će značka otići u S_P , dok će aktiviranjem prijelaza p_3 jedna značka otići u mjesto S_0 . Ukratko, prijelaz se aktivira kada su sve ulazne veze zadovoljene, a prijelazom se odgovarajući broj znački miče (troši na prijelaz) iz početnih mjesta i zadani broj znački (zadan na izlaznim vezama) se stavlja u odredišna mjesta. Ako veze (ulazne i izlazne) nemaju označku težine, onda se koristi podrazumijevana težina od jedne značke. Prijelaz se može nalaziti samo između mjesta (osim ulaznih prijelaza koji nemaju ulazno mjesto te izlaznih koji nemaju izlazni). Veze (koje su također usmjerene) mogu spajati samo mjesto i prijelaz te prijelaz i mjesto, a nikako mjesto i mjesto ili prijelaz s prijelazom.

Petrijeva mreža na slici 2.8. prikazuje petlju sa N iteracija. Prijelaz p_2 može predstavljati akciju u petlji (ili to može biti ulaz u stanje S_P), p_1 (ili ulaz u S_0) akciju koja joj predhodi te p_3 (ili ulaz u S_{ITER}) akciju nakon nje.

Slika 2.9. prikazuje simulaciju rada Petrijeve mreže kada je $N = 3$. Oni prijelazi koji su omogućeni u tom stanju, koji će uzrokovati promjenu prikazanu na idućoj slici, su podebljani.



Slika 2.9. Primjer rada Petrijeve mreže

Korištenje Petrijevih mreža prikazano je s još nekoliko primjera.

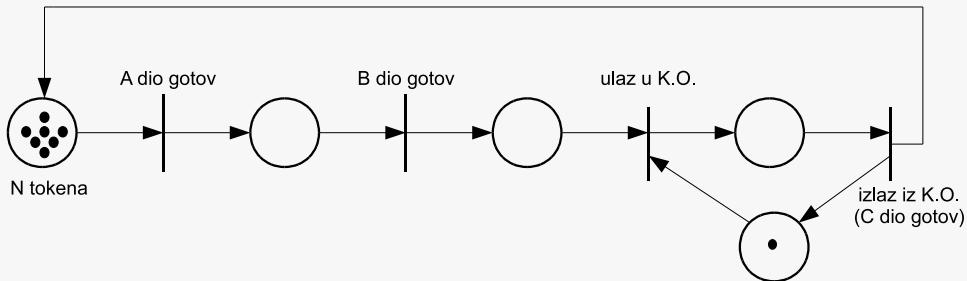
Primjer 2.2. Primjeri s Petrijevim mrežama

Skupina poslova

Zadatak:

Za skup od N dretvi poznato je da obavljaju isti posao koji se sastoji od dijelova A, B i C (svaka dretva radi A dio, pa B dio, pa C dio te ponovno ispočetka A, B, C, ...). Prva dva dijela, A i B, razne dretve mogu obavljati paralelno, ali dio C moraju obaviti međusobno isključivo (u kritičnom odsječku). Modelirati sustav Petrijevom mrežom, gdje svaka značka u početnom mjestu označava po jednu dretvu.

Rješenje:



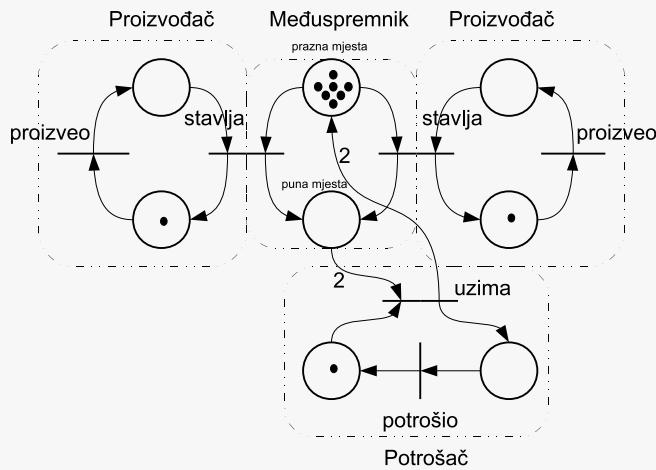
Slika 2.10.

Proizvođači i potrošač

Zadatak:

U sustavu se nalaze ciklički zadaci dva proizvođača i jednog potrošača. Proizvođači stvaraju poruke i stavljaju ih u prazna mjesta međuspremnika (ako je on pun, onda čekaju da se oslobodi jedno mjesto). Potrošač uzima dvije poruke iz međuspremnika te ih procesira (ako nema bar dvije poruke u međuspremniku onda čeka da se poruke pojave). Modelirati sustav Petrijevom mrežom. Veličina međuspremnika je 7 poruka.

Rješenje:



Slika 2.11.

2.2.5. Vremenske Petrijeve mreže

Vremenske Petrijeve mreže omogućuju definiranje vremenskih ograničenja na obavljanje prijelaza.

Primjerice, vremensko ograničenje na prijelaz p_x može se definirati intervalom $[t_1, t_2]$ kao interval u kojem je prijelaz moguće obaviti. Drugim riječima, nije dovoljno da ulazne veze budu omogućene (odgovarajući broj znački u pripadajućim ulaznim mjestima), već to treba biti moguće u zadanim vremenskim intervalima da bi se prijelaz dogodio. Ako se ulazne veze prema prijelazu omoguće prije zadanoj intervala prijelaz se neće obaviti (preko tog prijelaza) do tog intervala. Ako se ulazne veze omoguće nakon intervala, prijelaz se također neće dogoditi.

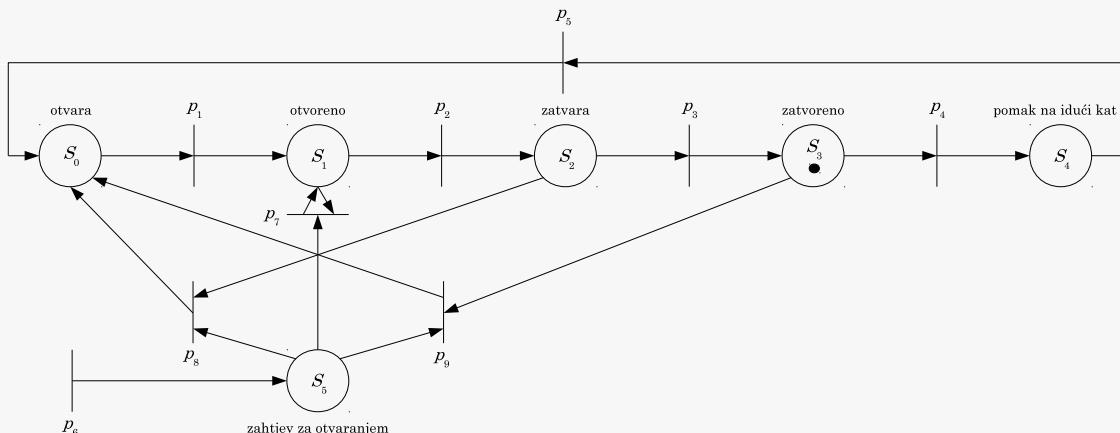
Vremenska ograničenja su dodatna ograničenja u mreži, tj. da bi se prijelaz dogodio i dalje moraju ulazne veze imati potreban broj znački u polaznim mjestima.

Primjer 2.3. Primjeri s vremenskim Petrijevim mrežama

Otvaranje vrata lifta

Zadatak:

Modelirati otvaranje i zatvaranje vrata lifta vremenskom Petrijevom mrežom. Neka je već zadana Petrijeva mreža prema slici 2.12.



Slika 2.12. Stanja vrata lifta (1)

Kada lift stane na katu (S_0) on započinje s otvaranjem vrata što traje 2 sekunde. Nakon 5 sekundi vrata treba zatvoriti. Potom se pričeka još 1 sekunda te se lift pokreće (ako ima idućih zahtjeva za drugim katovima, inače se može ovdje i stati). Ako zahtjev za otvaranjem vrata stigne dok se ona zatvaraju ili već jesu zatvorena, ali lift nije krenuo, vrata treba ponovno otvoriti.

Rješenje:

Vremenska ograničenja su dodatna ograničenja, tj. da bi se prijelaz dogodio tokeni moraju biti na raspolaganju na svakoj ulaznoj strelici u prijelazu.

Za svaki prijelaz moglo bi se definirati vrijeme kad je on moguć. Uobičajeno se kao trenutak mogućeg prijelaza označava dolazak značke u mjesto i to se označava s $time(S_x)$. Tablica 2.1. sadrži vremenska ograničenja za sve prijelaze.

Tablica 2.1. Vremenska ograničenja na prijelaze

prijelaz	ulazna mesta	vremenska ograničenja	opis (prema tekstu zadatka)
p_1	S_0	$time(S_0) + 2$	potrebne su 2 sekunde za otvaranje vrata
p_2	S_1	$time(S_1) + 5$	nakon 5 sekundi započinje se sa zatvaranjem vrata
p_3	S_2	$time(S_2) + 2$	potrebne su 2 sekunde za zatvaranje vrata
p_4	S_3	$time(S_3) + 1$	nakon zatvaranja lift stoji još 1 sekundu
p_5	S_4	–	nije definirano koliko se dugo lift kreće
p_6	–	–	zahtjev za otvaranjem može doći bilo kada
p_7	S_1, S_5	–	ako se zahtjev pojavi dok su vrata otvorena, ponovno se ulazi u stanje S_1 – dodatno vremensko ograničenje nije potrebno (iako se moglo napisati i $time(S_5)$)
p_8	S_2, S_5	–	prekida se zatvaranje i pokreće otvaranje (moglo se napisati i $[time(S_2), time(S_2) + 2]$ ili $time(S_5)$)
p_9	S_3, S_5	–	zahtjev za otvaranjem se pojavit će dok su vrata bila zatvorena (može se označiti i sa $[time(S_3), time(S_3) + 1]$) ili $time(S_5)$)

Vremenska ograničenja su zadana diskretnim trenucima, intervalima ili nisu zadana (–). Vremenska ograničenja za p_7 , p_8 i p_9 proističu iz stanja sustava – prijelazi su mogući tek kad značke dođu odgovarajuća ulazna mjesta. Stoga za navedene prijelaze nisu navedena i vremenska ograničenja (navedena su u opisu).

Kritični odsječak

Zadatak:

Dva ciklička zadatka P_1 i P_2 u svojem izvođenju prolaze kroz kritični odsječak u kojem koriste zajedničke podatke te potom nastavljaju u nekritičnome. Ulazak u kritični odsječak je zaštićen odgovarajućim sinkronizacijskim mehanizmom tako da u njega uvijek može ući samo jedan zadatak. Trajanje kritičnog odsječka za P_1 je u granicama od 5 do 10 jedinica vremena, a za P_2 u granicama od 10 do 15. Trajanje nekritičnog odsječka za P_1 je u granicama od 5 do 25 jedinica vremena, a za P_2 od 20 do 40. Za zadani sustav izgraditi (grafički prikazati) vremensku Petrijevu mrežu te napisati tablicu s opisom prijelaza.

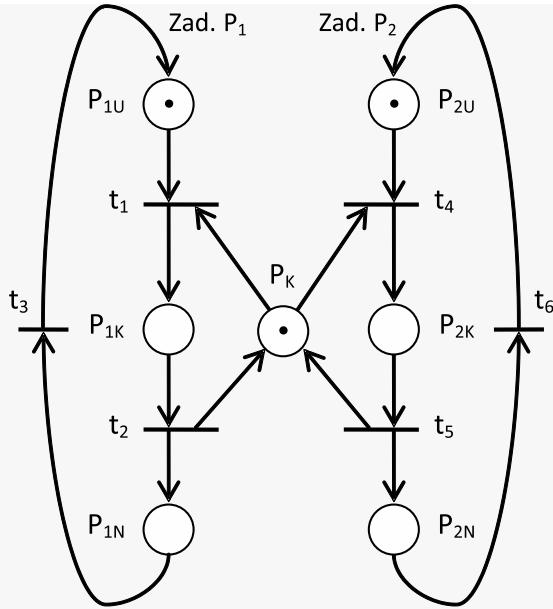
Rješenje:

Stanja su označena sa P_{xU} , P_{xK} te P_{xN} sa značenjima:

P_{xU} – zadatak x čeka na ulaz u kritični odsječak

P_{xK} – zadatak x u kritičnom odsječku

P_{xN} – zadatak x u nekritičnom odsječku



Slika 2.13.

Tablica 2.2. table

prijelaz	ulazna mjesta	vremenska ograničenja
t_1	P_{1U}, P_K	–
t_2	P_{1K}	$[time(P_{1K}) + 5, time(P_{1K} + 10)]$
t_3	P_{1N}	$[time(P_{1N}) + 5, time(P_{1N} + 25)]$
t_4	P_{2U}, P_K	–
t_5	P_{2K}	$[time(P_{2K}) + 10, time(P_{2K} + 15)]$
t_6	P_{2N}	$[time(P_{2N}) + 20, time(P_{2N} + 40)]$

Komunikacija

Zadatak:

U nekom komunikacijskom sustavu postoje dvije strane: klijent (K) i poslužitelj (P). Uspostava komunikacijskog kanala obavlja se tako da klijent pošalje zahtjev (REQ) na koji poslužitelj odgovara (ACK). Korištenjem Petrijevih mreža modelirati postupak uspostavljanja kanala. Prepostaviti da se klijent može naći u stanjima:

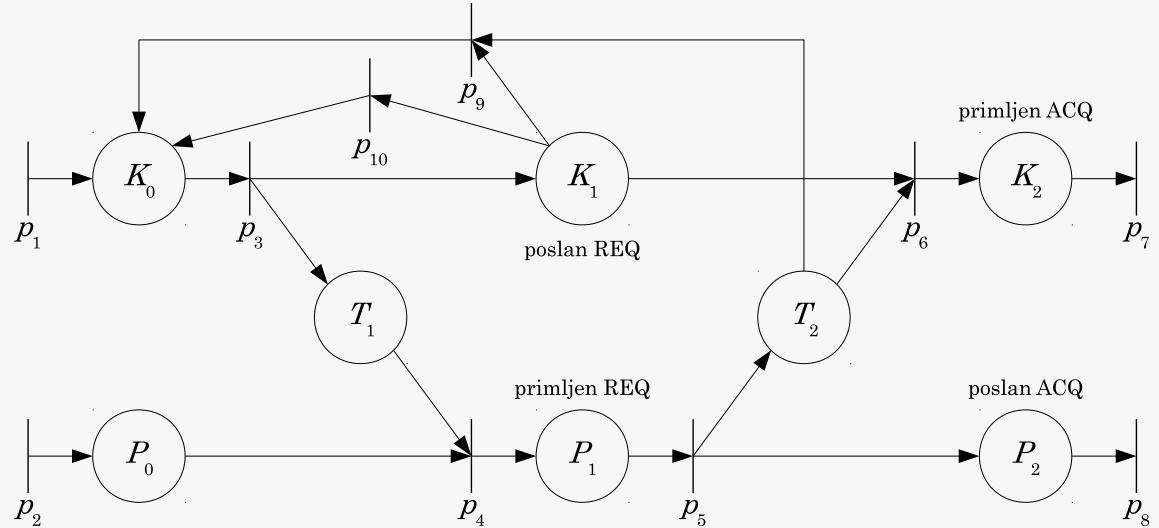
- K_0 – prije slanja zahtjeva REQ (početno mjesto)
- K_1 – nakon slanja zahtjeva REQ, čekanje na odgovor
- K_2 – nakon primitka odgovora ACK – veza uspostavljena.

Poslužitelj se može naći u stanjima:

- P_0 – prije primitka zahtjeva REQ (početno mjesto)
- P_1 – nakon primitka zahtjeva REQ, a prije slanja odgovora ACK
- P_2 – nakon slanja odgovora ACK – veza uspostavljena.

Odgovor na klijentov zahtjev ne smije doći prije $T_1 = 10 \mu s$ (jer za ovaj sustav protok poruka sigurno traje dulje), ali ni kasnije od $T_2 = 500 ms$ ("timeout"). U oba slučaja klijent treba ponoviti slanje zahtjeva REQ (vratit se u mjesto K_0).

Rješenje:



Slika 2.14.

Tablica 2.3.

prijelaz	ulazna mesta	vremenska ograničenja
p_1	—	—
p_2	—	—
p_3	K_0	—
p_4	P_0, T_1	—
p_5	P_1	$< time(P_1) + T_1, \infty >$
p_6	K_1, T_2	$< time(K_1) + T_1, time(K_1) + T_2 >$
p_7	K_2	—
p_8	P_2	—
p_9	K_1, T_2	$[0, time(K_1) + T_1]$
p_{10}	K_1	$time(K_1) + T_2$

Prijelazi $p_1 - p_8$ zbivaju se u očekivanom načinu rada, dok se prijelazi p_9 i p_{10} zbivaju u slučaju grešaka.

2.3. Primjer upravljačkog programa za lift

Nakon analize sustava te početna modeliranja može se krenuti u izgradnju. Neki elementi sustava za lift su već modelirani korištenjem nekih od postupaka (UML dijagramima). Primjerice, UML dijagram stanja na slici 2.7. može poslužiti za upravljanje. Međutim, na njemu nisu označeni razlozi promjene stanja. Da bi to mogli napraviti potrebno je prvo detaljnije opisati stanje sustava, a tek onda osmisliti stanja sustava koja će uzrokovati promjene prema navedenom dijagramu.

Neka se prijelazi definiraju obzirom na stanje sustava opisano ne samo dijagramom stanja nego i položajem lifta, dodijeljenim stajanjima (stani[]) te postavljenim smjerom. U tom slučaju bi se prijelazi mogli detaljnije opisati tablicom 2.4.

Tablica 2.4. Prijelazi među stanjima lifta

Početno stanje	Uvjet prelaska	Novo stanje	Promjene u varijablama
ZATVORENO	stani[trenutni kat] == 1	OTVARA	
	smjer je postavljen	POKREĆE	
	inače	ZATVORENO	
OTVARA	došla poruka da je otvoreno	OTVORENO	stani[trenutni kat] = 0
	inače	OTVARA	
OTVORENO	stani[trenutni kat] == 1	OTVORENO	stani[trenutni kat] = 0
	vrata dovoljno dugo otvorena	ZATVARA	
	inače	OTVORENO	
ZATVARA	stani[trenutni kat] == 1	OTVARA	
	došla poruka da je zatvoreno	ZATVORENO	
	inače	ZATVARA	
POKREĆE	došla poruka da je pokrenut	KREĆE	ažurira se položaj lifta
	stani[idući kat] == 1	ZAUSTAVLJA	ažurira se položaj lifta
	inače	POKREĆE	ažurira se položaj lifta
KREĆE	stani[idući kat] == 1	ZAUSTAVLJA	ažurira se položaj lifta
	inače	KREĆE	ažurira se položaj lifta
ZAUSTAVLJA	došla poruka da je zaustavljen	ZATVORENO	ažurira se položaj lifta
	inače	ZAUSTAVLJA	ažurira se položaj lifta

Osim prijelaza i neke druge odluke i operacije bi trebalo detaljnije razraditi. U nastavku je prikazano jedno konkretnije rješenje upravljanja izraženo pseudokodom, gdje su te ostale operacije opisane i samim pseudokodom i dodatnim komentarima. Najprije su navedene strukture podataka, sučelja te onda sam kôd.

U opisu pojedinih struktura podataka i operacija koriste se neke konstante i makroi. Popis značajnijih je naveden u kôdu 2.1.

Isječak kôda 2.1. Korištene konstante i makroi

```

# stanja lifta
OTVORENO, ZATVORENO, OTVARA, ZATVARA,
POKRECE, ZAUSTAVLJA, KRECE, ZAUSTAVLJEN

SMJER_DOLE(-1), SMJER_NEMA(0), SMJER_GORE(1) # smjer gibanja lifta

# makroi, korišteni samo kada je smjer postavljen (nije SMJER_NEMA):
PREDZNAK(SMJER): SMJER_DOLE => -1, SMJER_GORE => 1
INDEKS(SMJER): SMJER_DOLE => 0, SMJER_GORE => 1
SMJER(INDEX): 0 => SMJER_DOLE, 1 => SMJER_GORE

MANJI_CIJELI(X) # prvi manji cijeli broj;
                  # npr. MANJI_CIJELI(2.7) = 2, MANJI_CIJELI(-0.3) = -1
BLIŽI_CIJELI(X) # bliži cijeli broj;
                  # npr. BLIŽI_CIJELI(2.7) = 3, BLIŽI_CIJELI(-0.3) = 0

```

Neka za pojednostavljeni model lifta u zgradi podatkovna struktura sadrži podatke o stanju lifta, uključujući i zahtjeve korisnika u liftu te podaci o stanju zahtjeva korisnika s pojedinih katova (pozivi liftu), prema kôdu 2.2.

Isječak kôda 2.2. Podatkovna struktura za upravljanje

```

lift
stanje # stanje lifta, prema dijagramu stanja
položaj # decimalna vrijednost - položaj lifta u zgradu
          # od 0 do BROJ_KATOVA-1 (npr. 4.23)
smjer # smjer gibanja lifta: -1, 0, 1
smjer_postavljen # Kad lift stane, je li daljnji smjer već definiran?
stani[BROJ_KATOVA] # Zadana stajanja za lift (u upravljačkom programu).
zahtjev[BROJ_KATOVA] # Zahtjevi za stajanjem na katovima (iz lifta).
poziv[BROJ_KATOVA][2] # Liftu dodijeljeni pozivi izvana (za gore/dole).

tipka_otvori # Je li stisnuta tipka "otvori" (u liftu, kad stoji)?
tipka_zaustavi # Je li stisnuta tipka "zaustavi"?
tipka_nastavi # Je li otpuštena tipka "zaustavi"?
prethodno_stanje # Kad se lift hitno zaustavlja, ovdje se pohranjuje
                   # zatečeno stanje lifta.

čekaj_do # Do kada treba čekati prije iduće akcije,
          # npr. kada treba započeti sa zatvaranjem vrata

vrijeme_zadnjeg_stanja # Kada je zadnji puta ažurirano stanje
                      # (da bi mogli otkriti kad nešto ne radi).

zgrada # Referenca na objekt zgrade, radi brisanja posluženih zahtjeva
       # izvan lifta (pozivi), izravno iz objekta lifta

zgrada
poziv[BROJ_KATOVA][2] # indeks 0 za poziv prema dole, 1 za gore

```

Za izgradnju kôda korišteno je načelo modularnosti pa je kôd podijeljen u dio koji radi operacije posebne za lift, posebne za zgradu te općenite (globalna upravljačka petlja). Sučelja tih modula prikazani su kôdom 2.3.

Isječak kôda 2.3. Sučelje komponenata lift i zgrada

```

lift
    inicijaliziraj () # Koristi se samo pri pokretanju programa.

    # U svakoj iteraciji upravljačke petlje:
    ažuriraj_stanje () # Dohvaća stanje lifta i ažurira svoje podatke te
                        # napravi lokalne odluke, na temelju zadanih zahtjeva.
    pošalji_naredbe () # Na temelju trenutna stanja šalje naredbe liftu
    korak ()           # Pri korištenju simulatora, za simulaciju rada

    # Pomoćne funkcije
    idući_najbliži_kat (kat, udaljenost)
        # Dohvaća najbliži kat na kojem lift može stati (trenutni ako već stoji).
    stajanja_u_smjeru (min, max)
        # Dohvaća najbliži i najdaljni kat na kojem će lift stati (ako ima smjer).

zgrada
    inicijaliziraj () # Koristi se samo pri pokretanju programa

    # U svakoj iteraciji upravljačke petlje
    ažuriraj_stanje () # Dohvaća stanje tipki na katovima (izvan lifta)
    pošalji_naredbe () # Šalje naredbe za označavanje stanja

```

Navedene komponente služe za donošenje odluka upravljanja. Međutim, potrebne su i komponente koje će dohvaćati stanje sustava te slati naredbe. Neka se za komunikaciju s jednim liftom koristi sučelje lift_ui (ulaz/izlaz) a za komunikaciju sa zgradom zgrada_ui. U stvarnim sustavima bi te komponente slale i primale podatke. U simulatoru te komponente komuniciraju sa simulatorom (ili su integrirane s njime). Potrebna sučelja tih komponenata prikazana su kôdom 2.4.

Isječak kôda 2.4. Sučelje komponenata za komunikaciju lift_ui i zgrada_ui

```

lift_ui
    dohvati_smjer ()   # -1, 0, 1
    dohvati_položaj () # [0; BROJ_KATOVA] – realna vrijednost
    dohvati_vrata ()   # stanje vrata: OTVORENO, ZATVORENO, OTVARA, ZATVARA

    # Je li stisnuta neka tipka u liftu?
    dohvati_tipku ( kat ) # Je li stisnuta tipka u liftu za zadani kat?
    dohvati_otvori ()     # Je li stisnuta tipa "otvori" (kada lift stoji)?
    dohvati_zaustavi ()   # Je li stisnuta tipka za hitno zaustavljanje?
    dohvati_nastavi ()   # Je li otpuštena tipka za hitno zaustavljanje?

    # Postavi oznake na panelu u liftu
    postavi_kat ( kat, stanje ) # Upali/ugasi broj za "kat".
    postavi_smjer ( smjer )    # Postavi smjer kretanja.

    # naredbe liftu
    kreni ( smjer ) # Naredba liftu da kreće gore/dole uz pretpostavku
                    # da je lift u tom trenutku zaustavljen.
    nastavi ()      # Naredba liftu da nastavlja s kretanjem u istom smjeru
                    # uz pretpostavku da se lift već kreće.
    stani ()        # Naredba liftu da stane na idućem katu uz pretpostavku
                    # da se lift već kreće.
    otvori ()       # Naredba liftu da otvori vrata.
    zatvori ()      # Naredba liftu da zatvori vrata.
    zaustavi ()     # Naredba liftu da trenutno (hitno) stane.

    korak ()        # Pri korištenju simulatora, za simulaciju rada lifta.

zgrada_ui
    # Je li stisnuta tipka (poziv) izvana na nekom katu?
    dohvati_poziv ( kat, smjer )

```

```
# Postavljanje oznake stanja lifta i poziva na katovima.
postavi_poziv ( kat, smjer, stanje )
postavi_stanje ( lift, kat, smjer )
```

Glavni modul se sastoji od upravljačkog programa s petljom u kojoj se ažurira stanje lifta i zgrade te donose odluke upravljanja. Odluke su podijeljene na dva dijela: odluke koje se mogu donijeti na osnovu stanja lifta i prothodno dodijeljenih zahtjeva (lokalne odluke) te odluke o posluživanju zahtjeva s pojedinih katova (globalne odluke). Prve su (u ovom modelu) ostvarene u komponenti lifta, a druge u glavnem modulu upravljač. Kôd upravljača naveden je u 2.5.

Isječak kôda 2.5. Osnovni upravljački modul

```
# Korišteni objekti:
# - upravljač - osnovni objekt za upravljanje
# - zgrada - objekt za upravljanje zgradom
# - lift[BROJ_LIFTOVA] - objekti za upravljanje liftovima

upravljač:program ()
    # Inicijalizacija
    zgrada_ui = dohvati_sučelje_za_zgradu ( i )
    zgrada.inicijaliziraj ( zgrada_ui )
    za i = 0 do BROJ_LIFTOVA-1 radi
        lift_ui = dohvati_sučelje_za_lift ( i )
        lift[i].inicijaliziraj ( zgrada, lift_ui )

    # Glavna petlja upravljanja
    ponavljaј
        zgrada.ažuriraj_pozive () # dohvati novih zahtjeva
        za i = 0 do BROJ_LIFTOVA-1 radi
            lift[i].ažuriraj_stanje () # ažuriranje i lokalne odluke

        upravljač.dodijeli_zahtjeve () # globalne odluke

        za i = 0 do BROJ_LIFTOVA-1 radi
            lift[i].pošalji_naredbe () # lokalne odluke i slanje naredbi
            zgrada.postavi_stanje_lifta ()

        # Dodatak potreban kada se sustav simulira
        za i = 0 do BROJ_LIFTOVA-1 radi
            lift[i].korak ()
        # po potrebi odgoditi iduću iteraciju za T_KORAK

upravljač:dodijeli_zahtjeve ()
    # Globalne odluke - dodjela zahtjeva iznutra i izvana

    # Zahtjeve iznutra dodijeliti odmah
    za i = 0 do BROJ_LIFTOVA-1 radi
        za j = 0 do BROJ_KATOVA-1 radi
            lift[i].stani[j] != lift[i].zahtjev[j]

    # Zahtjeve izvana treba dodjeljivati najbližim liftovima.
    # Jedan od algoritama, prikazan u nastavku, pretražuje neobrađene
    # zahtjeve dok ne pronađe onaj koji je najbliže nekom liftu. Nakon
    # dodjele tog zahtjeva, traži se idući "najbliži" te tako dok se svi
    # ne "obrade".

    z = kopija (zgrada.poziv) # radna kopija aktivnih zahtjeva

    dok z nije prazan radi
        min_kat = min_smjer = min_lift = 0
        min_udaljenost = BROJ_KATOVA + 1

        za j = 0 do BROJ_KATOVA-1 radi
            za k = 0 do 1 radi
                ako je z[j][k] != 0 tada
```

```

za i = 0 do BROJ_LIFTOVA-1 radi
    # Izračunati udaljenost lifta za ovaj zahtjev.
    # Uzeti u obzir i smjer gibanja lifta, ide li lift i na katove
    # nakon poziva, ako je poziv za suprotni smjer i slično.
    udaljenost = izračunata udaljenost
    ako je udaljenost < min_udaljenost tada
        min_udaljenost = udaljenost;
        min_kat = j;
        min_smjer = k;
        min_lift = i;

    ako je min_udaljenost < BROJ_KATOVA + 1 tada
        # Dodijeliti zahtjev najbližem liftu.
        lift[min_lift].stani[min_kat] = 1;
        lift[min_lift].poziv[min_kat][min_smjer] = 1;
        # Dodatno: postaviti smjer, ako je potrebno.
        # Provjeriti je li navedeni zahtjev već bio prije dodijeljen
        # nekom drugom liftu. Ako jest, onda ga maknuti od tamo i po
        # potrebi maknuti to stajanje onom liftu, ali i dodati neko
        # stajanje tom liftu ako se giba, a više nema stajanja.

    # Označiti zahtjev kao "obrađen" (neovisno o tome je li dodijeljen).
    z[min_kat][min_smjer] = 0;

```

Upravljanje zgradom koristi sučelje zgrada_ui a ostvareno je modulom zgrada, prema 2.6.

Isječak kôda 2.6. Komunikacija sa zgradom

```

zgrada:inicijaliziraj (zgrada_ui)
    zgrada.zgrada_ui = zgrada_ui
    za i = 0 do BROJ_KATOVA-1 radi
        zgrada.poziv[0] = zgrada.poziv[1] = 0

zgrada:ažuriraj_pozive ()
    za i = 0 do BROJ_KATOVA-1 radi
        za j iz {SMJER_DOLE, SMJER_GORE} radi
            zgrada.poziv[i][INDEX(j)] |= zgrada.zgrada_ui.dohvati_poziv(i, j)

zgrada:postavi_stanje_lifta ()
    zgrada.zgrada_ui.postavi_stanje (BLIŽI_CIJELI(lift.položaj), lift.smjer)
    za i = 0 do BROJ_KATOVA-1 radi
        za j iz {SMJER_DOLE, SMJER_GORE} radi
            zgrada.zgrada_ui.postavi_poziv (i, j, zgrada.poziv[i][INDEX(j)])

```

Upravljanje liftom i lokalne odluke za lift ostvarene su u modulu lift prema 2.7.

Isječak kôda 2.7. Upravljanje liftom

```

lift:inicijaliziraj ( zgrada, lift_ui )
    lift.zgrada = zgrada
    lift.lift_ui = lift_ui

    lift.položaj = 0
    lift.stanje = ZATVORENO
    lift.smjer = 0
    lift.smjer_postavljen = 1

    za i = 0 do BROJ_KATOVA-1 radi
        lift.stani[i] = lift.zahtjev[i] = 0
        lift.poziv[i][0] = lift.poziv[i][1] = 0

    lift.tipka_otvori = 0
    lift.tipka_zaustavi = 0
    lift.tipka_nastavi = 0

    lift.prethodno_stanje = 0
    lift.čekaj_do = 0
    lift.vrijeme_zadnjeg_stanja = 0

lift:ažuriraj_stanje ()
    # Dohvaća stvarno stanje lifta, ažurira strukturu podataka

    # 1. Dohvat podataka od elemenata
    stvarni_smjer = lift.lift_ui.dohvati_smjer ()
    stvarni_položaj = lift.lift_ui.dohvati_položaj ()
    stanje_vrata = lift.lift_ui.dohvati_vrata ()
    za i = 0 do BROJ_KATOVA-1
        lift.zahtjev[i] |= lift.lift_ui.dohvati_tipku (i)
    lift.tipka_otvori = lift.lift_ui.dohvati_otvori ()
    lift.tipka_zaustavi = lift.lift_ui.dohvati_zaustavi ()
    lift.tipka_nastavi = lift_ui.dohvati_nastavi ()

    # 2. Provjera ima li smisla očitano + akcije na greške (nije ostvareno)
    # lift.provjeri_smjer ( stvarni_smjer )
    # lift.provjeri_položaj ( stvarni_položaj )
    # lift.provjeri_vrata ( stanje_vrata )
    # Ako nešto ne odgovara, sustav treba zaustaviti ili ugraditi postupak
    # oporavka od pogreške

    # 3. Je li se nešto promijenilo što zahtjeva promjenu stanja lifta
    # (npr. vrata su se otvorila do kraja)?
    lift.položaj = stvarni_položaj
    lift.vrijeme_zadnjeg_stanja = vrijeme_sada ()

    # Upravljanje u kritičnim situacijama, kad se pritisne tipka za hitno
    # zaustavljanje, kao i kad se ona otpusti, nije detaljno obrađeno
    # u ovom kodu, već je korišteno pojednostavljenje.
    ako je lift.tipka_zaustavi tada
        ako je lift.prethodno_stanje == 0 tada
            lift.prethodno_stanje = lift.stanje
            lift.stanje = ZAUSTAVLJEN
    inače ako je lift.stanje == ZAUSTAVLJEN I lift.tipka_nastavi tada
        lift.stanje = lift.prethodno_stanje
        lift.prethodno_stanje = 0
        ako je lift.stanje == KREĆE tada
            lift.stanje = POKREĆE
    ako je lift.stanje == ZAUSTAVLJEN tada
        povratak_iz_funkcije # Daljnje odluke nisu potrebne

    ako je stvarni_smjer == 0 tada

```

```

ako je lift.stanje == OTVARA I stanje_vrata = OTVORENO tada
    lift.upravo_otvorena_vrata ()
lift.stanje = stanje_vrata # OTVORENO, ZATVORENO, ZATVARA, OTVARA
ako je lift.smjer != 0 I lift.smjer_postavljen == 0 tada
    lift.stajanja_u_smjeru ( ima )
    ako je ima != -1 tada
        lift.smjer_postavljen = 1
    ako je lift.stanje == ZATVORENO I lift.smjer_postavljen == -1 tada
        lift.smjer = 0
inače
    # Lift se giba!
    ako je lift.stanje == POKREĆE tada
        lift.stanje == KREĆE # prethodna naredba je bila pokretanje
    inače ako je lift.stanje == KREĆE tada
        # Treba li stati na idućem katu?
        idući_kat = lift.idući_najbliži_kat ( do_idućeg )
        ako je do_idućeg >= PUT_ZAUST_MIN I do_idućeg < PUT_ZAUST_MAX tada
            ako je lift.stani[idući_kat] tada
                lift.stanje = ZAUSTAVLJA

    # Ako lift stoji, treba li produžiti stanje s otvorenim vratima ili ih
    # zatvoriti ili otvoriti?
    ako je lift.stanje == OTVORENO tada
        ako je lift.tipka_otvoriILI lift.stani[lift.položaj] tada
            lift.upravo_otvorena_vrata()
        inače
            ako je "proteklo dovoljno vremena" tada
                lift.stanje = ZATVARA
    inače ako je ( lift.stanje == ZATVORENO I lift.stanje == ZATVARA ) I
        ( lift.tipka_otvoriILI lift.stani[lift.položaj] ) tada
        lift.stanje = OTVARA

    # (dodano =>)
    # Ako su vrata zatvorena ima li zahtjeva koji traže akciju?
    ako je lift.stanje == ZATVORENO tada
        # Ima li posla?
        ako je lift.smjer == SMJER_NEMA tada # Ako nema smjer ...
            lift->smjer = SMJER_DOLE # ... gledaj prvo dole.

    za i = 0 do 2 radi
        lift.stajanja_u_smjeru ( ima )
        ako je ima != -1 tada
            lift.stanje = POKREĆE
            prekini petlju "za"
        inače
            # Nema zahtjeva u tom smjeru.
            # Ima li zahtjeva u suprotnom smjeru?
            ako je lift.smjer == SMJER_DOLE tada
                lift.smjer = SMJER_GORE
            inače
                lift.smjer = SMJER_DOLE

    ako je lift.stanje != POKREĆE tada
        # Niye se pokrenuo - nema zahtjeva u ni jednom smjeru
        lift.smjer = SMJER_NEMA
    # (dodano <=)

lift:pošalji_naredbe ()
lift_ui.postavi_smjer ( lift.smjer )
za i = 0 do BROJ_KATOVA-1
    lift_ui.postavi_kat ( i, lift.stajanja[i] )

ako je lift.stanje == ZAUSTAVLJEN tada # zbog tipke "zaustavi"
    lift_ui.zaustavi()
inače ako je lift.stanje == POKREĆE tada

```

```

lift.ui.kreni()
inače ako je lift.stanje == KREĆE tada
    lift.ui.nastavi()
inače ako je lift.stanje == ZAUSTAVLJA tada
    lift.ui.stani()
inače ako je lift.stanje == OTVARA tada
    lift.ui.otvori()
inače ako je lift.stanje == ZATVARA tada
    lift.ui.zatvori()

# Pomoćne funkcije
lift:idući_najbliži_kat ( (opcionalni argument) do_idućeg )
    # Pomoćna funkcija: prema stanju lifta vraća najbliži idući kat
    # na kojem lift može stati (prema sadašnjem stanju).

ako je lift.stanje iz { ZATVORENO, OTVARA, OTVORENO, ZATVARA } tada
    kat = lift.položaj

inače ako je lift.stanje == KREĆE tada
    ako je lift.smjer == SMJER_GORE tada
        kat = MANJI_CIJELI ( lift.položaj + 1 + PUT_ZAUST_MIN )
    inače
        kat = MANJI_CIJELI ( lift.položaj - PUT_ZAUST_MIN )

inače ako je lift.stanje iz { POKREĆE, ZAUSTAVLJA } tada
    ako je lift.smjer == SMJER_GORE tada
        kat = MANJI_CIJELI ( lift.položaj + 1 )
    inače
        kat = MANJI_CIJELI ( lift.položaj )

ako je zadan_argument ( do_idućeg ) tada
    do_idućeg = kat - lift.položaj
    ako je lift.smjer == SMJER_DOLE tada
        do_idućeg = -do_idućeg

vrati kat

lift:stajanja_u_smjeru ( (opcionalni argumenti) min, max )
    m = M = -1
    ako je lift->smjer != 0 tada
        kat = lift.idući_najbliži_kat ()
        dok je kat >= 0 I kat < BROJ_KATOVA radi
            ako je lift.stani[kat] tada
                M = kat
                ako je m == -1 tada
                    m = kat
                kat = kat + lift.smjer
    ako je zadan_argument ( min ) tada
        min = m;
    ako je zadan_argument ( max ) tada
        max = M;

lift:upravo_otvorena_vrata ()
    lift.stanje = OTVORENO
    lift.stani[lift.položaj] = 0
    lift.zahtjev[lift.položaj] = 0
    lift.čekaj_do = vrijeme_sada() + TRAJANJE_OTVORENA_VRATA

ako je lift.smjer != 0 tada
    # Ima li još zahtjeva u istom smjeru?
    lift:stajanja_u_smjeru ( min, max )
    ako je min != -1 tada
        lift.smjer_postavljen = 1
    inače
        lift.smjer_postavljen = 0

```

```

# Ako je stao radi poziva izvana, tada obrisati taj zahtjev.
ako je lift.poziv[lift.položaj][INDEKS(lift.smjer)] tada
    lift.poziv[lift.položaj][INDEKS(lift.smjer)] = 0
    lift.zgrada.poziv[lift.položaj][INDEKS(lift.smjer)] = 0

inače ako je lift.smjer_postavljen == 0 tada
    # Ne postoji zahtjevi u istom smjeru. Možda u suprotnome?
    lift.smjer = -lift.smjer
    ako je lift.poziv[lift.položaj][INDEKS(lift.smjer)] tada
        lift.poziv[lift.položaj][INDEKS(lift.smjer)] = 0
        lift.zgrada.poziv[lift.položaj][INDEKS(lift.smjer)] = 0
    inače
        # Nema zahtjeva, obrisati smjer.
        lift.smjer = 0

```

Navedeni pseudokod ne uključuje module za komunikaciju ili simulator. U stvarnom sustavu modul za komunikaciju bi koristio komunikacijske protokole za slanje i prihvatanje podataka od strane elemenata sustava. U simuliranom okruženju, simulator treba dati simulirane ulaze kao što je i stanje lifta, ali i simulaciju putnika.

2.4. Proces izgradnje programske potpore

Izgradnja složenih sustava je složen proces. Programska komponenta SRSV-a (programska potpora, engl. *software*) često je vrlo složena te se pri njenoj izgradnji tako i treba ponašati, tj. primjenjivati postupke koji će moći tu složenost nadvladati. Osnovni način rješavanja složenosti je “podijeli i vladaj” te se i postupak izgradnje sustava treba posmatrati kao proces nad kojim se može primijeniti isto načelo.

U nastavku su kratko navedene uobičajene metodologije koje se koriste u procesu izgradnje sustava. Za detaljniji prikaz pojedine od metodologija može se koristiti literatura iz područja programskog inženjerstva (engl. *software engineering*) koja to detaljno opisuje.

Proces izgradnje programske potpore može se podijeliti u nekoliko osnovnih aktivnosti:

1. izrada specifikacije (engl. *requirements*)
2. oblikovanje arhitekture (engl. *design*)
3. izrada komponenata – programiranje (engl. *implementation*)
4. ispitivanje ispravnosti – testiranje
5. održavanje – ugradnja, nadogradnja, evolucija (engl. *maintenance*).

Izrada *specifikacije* uključuje analizu zahtjeva, provedbu studije izvedivosti, izlučivanje zahtjeva te izradu dokumenta specifikacije koji je podloga ugovora za projekt ali i za cijeli proces razvoja, počevši s oblikovanjem arhitekture.

Oblikovanje arhitekture je vrlo bitna aktivnost (većinom najbitnija) jer određuje “kako” sustav treba napraviti te je glavni nositelj konačne kvalitete produkta. Uobičajeno su arhitekture zasnovane na: događajima, protoku podataka, objektima, komponentama, slojevima, repozitoriju podataka i klijent-poslužitelj organizaciji.

Izrada komponenata uključuje programiranje (izgradnju) sustava. Izgradnja treba slijediti arhitekturu te zahtjeve specifikacije.

Ispitivanje kao aktivnost ne bi trebala biti zasebna aktivnost u procesu izgradnje programa već uključena u sve aktivnosti, od izrade specifikacije, oblikovanja arhitekture, programiranje komponenti i njihovoj integraciji do ispitivanje ispravnosti ugrađenog sustava. Osnovni problem ispitivanja složenih sustava jest u tome što se oni ne mogu u potpunosti ispitati – ima previše mogućih stanja sustava. Zato bi ispitivanje trebalo uključiti u sve aktivnosti i komponente sustava koje su znatno manje složenosti od gotovog sustava.

Nakon dovršetka izrade sustava i njegove ugradnje započinje *proces održavanja* koji uključuje naknadne izmjene (ispravke, prilagodbe, nadogradnje). Za sustave s predviđenim dugim životnim vijekom moguće je da održavanje bude višestruko skuplje od samog procesa izgradnje (te i to treba uzeti u obzir pri razmatranju ostvarenja sustava).

Navedene opće aktivnosti procesa izgradnje programske potpore ne moraju se izvoditi kao kraci, najprije prva pa druga itd. Postoje razni modela procesa izgradnje (vodopadni, evolucijski, iterativni, inkrementalni, komponentno oblikovanje, ...) koji su primjenjivi u raznim situacijama. Navedimo par riječi o nekima od njih.

Vodopadni model zahtijeva da svaka aktivnost bude gotova prije nego li se krene sa sljedećom. Vodopadni model je stoga primjenjiv u velikim razvojnim timovima, gdje pojedine aktivnosti obavljaju različite osobe i timovi, te je bitno da pojedina aktivnost bude gotova prije nego li se krene s idućom.

Evolucijski modeli primjenjivi su na sustave kod kojih specifikaciju nije jednostavno napraviti te se započinje s jednim rješenjem koje se popravlja i nadograđuje da zadovolji naknadno uočene zahtjeve i nedostatke. Zbog početno odabranog rješenja bez duble analize te naknadnih dodataka, arhitektura tako nastalih sustava može biti vrlo loša. Međutim, ukoliko se radi o sustavima napravljenim samo za neki događaj (npr. za organizaciju nekog natjecanja) onda se ovim modelom vrlo brzo može doći do rješenja te problem održavanja (koji bi bio skup obzirom na lošu arhitekturu) neće biti prisutan.

Iterativni modeli koriste iteracije radi popravljanja i unapređenja pojedinih elemenata sustava, od specifikacije, arhitekture, oblikovanja komponente do testiranja. Zapravo bi mogli reći da svi modeli uključuju iterativni pristup, barem lokalno, nad pojedinom komponentom. Ovisno o dozvoljenom vremenu za razvoj broj iteracija se prilagođava (povećava ili smanjuje), svakom iteracijom poboljšavajući pojedini segment u razvoju.

Kod *inkrementalnog pristupa* sustav se izrađuje u inkrementima. Prvi inkrement koji se isporučuje naručitelju sadrži osnovnu funkcionalnost, dok idući inkrementi dodaju ostale funkcionalnosti. Prednost inkrementalnog pristupa je u ranoj isporuci proizvoda (manji rizik za neuspjeh projekta) te mogućnosti većeg ispitivanja osnovnih funkcija koje su ostvarene u startu (a ispituju se i u radu).

Oblikovanje zasnovano na *komponentnom pristupu* kreće od premise da se u rješavanju problema uključe već gotove komponente, a da se što je moguće manje izgrađuju nove. Prednosti ovog pristupa su u vrlo kratkom vremenu izgradnje obzirom da se većina komponenata ne treba izgraditi već samo spojiti. Nadalje, gotove su komponente već opsežnije ispitane i samom uporabom u drugim sustavima te se ovim pristupom može povećati pouzdanost sustava.

2.5. Posebnosti izgradnje SRSV-a

Oblikovanje SRSV-a može se obaviti bilo kojim metodologijom, prema procjeni arhitekta, kao i pri izradi drugih sustava. Međutim, zbog potrebe zadovoljenja (strogih) vremenskih ograničenja, u svakom postupku treba birati prikladne metode i komponente i postupke koji će ih moći zadovoljiti te to i provjeriti (ispitati). Primjerice, potrebno je odabratи prikladan programski jezik i alate, potrebno je sprovesti analizu složenosti algoritama i kôda, analizirati sklopovlje i slično. Možda najveća razlika u odnosu na ostale sustave je u postupku ispitivanja koje je za SRSV-e daleko opsežnije i temeljitije.

Proces izgradnje se ne sastoji samo od programiranja. Ako programiranje započne prije nego li se sustav temeljitije analizirao i dokumentirao, izrada programa je znatno teža i sklonija pogreškama i previdima, rezultirajući produktom loše arhitekture kojeg je i teže ispitati i održavati. Zato se niz početnih aktivnosti u izradi SRSV-a, kao ni pri izradi drugih sustava ne smije olako shvatiti i površno obaviti. Grafičke metode, poput skice i UML dijagrama te Petrijevih mreža,

mogu znatno olakšati proces savladavanja problema, njegove analize i pronalaska rješenja.

2.5.1. Formalna verifikacija

Možda najveća razlika u procesu ostvarenja programske komponente za SRSV-e jest u ispitivanju ispravnosti rada. Ispitivanja SRSV-a moraju biti značajno temeljitija. U tom smjeru su i preporuke znanstvenih i stručnih organizacija (npr. IEEE) da se ispitivanje ne vrši samo primjenom ispitivanja korištenjem ulaznih podataka i očekivanih rezultata, već da se koristi *formalna verifikacija*. Naime, zbog praktički beskonačno velike domene (mogućih ulaza) sama ispitivanja zbog vremenskih ograničenja mogu ispitati samo dio te domene, tj. ispitati rad sustava samo u jednom malom dijelu domene. Kad bi se ispravno provela, formalna verifikacija bi korištenjem formalnih postupaka mogla provjeriti ispravnost sustava u cijeloj domeni.

Problem formalne verifikacije jest u njenoj primjeni. Specifikaciju zahtjeva prema sustavu treba pripremiti u formalnom obliku. Izgrađeni sustav treba formalno opisati. Ukoliko formalna specifikacija zahtjeva te formalni model implementacije ne odgovaraju u potpunosti stvarnim specifikacijama i implementacijom, rezultati formalne verifikacije neće biti vjerodostojni (potpuni).

Za pripremu formalne specifikacije te formalnog modela za implementaciju kao i za provedbu postupaka formalne verifikacije potrebna su značajna znanja iz područja formalnih postupaka koju rijetki današnji inženjeri posjeduju.

Područje formalnih postupaka je suviše opsežno i složeno da bi se moglo ukratko prikazati. Zato se zainteresirani upućuju na druge izvore.

Pitanja za vježbu 2

1. Navesti svojstva skice i tekstovnog opisa kao metoda u postupku projektiranja sustava.
2. Zašto nije dobro izradu programske potpore započeti bez pripreme (npr. korištenjem skice sustava, dijagrama stanja i slično)? ◁
3. Navesti osnovne razloge korištenja UML dijagrama: obrazaca uporabe, sekvensijskih, kolaboracijskih i dijagrama stanja. ◁
4. S kojim UML dijagramom se najbolje prikazuje tijek odvijanja neke operacije s pripadnim porukama, pozivima funkcija komponenata i slično, gdje je bitno uočiti vremenski redoslijed?
5. Kada se koriste neformalni postupci, a kada formalni (u kontekstu izgradnje SRSV-a)?
6. Navesti elemente Petrijeve mreže i njihova svojstva.
7. Opisati razliku između vremenskih Petrijevih mreža i običnih Petrijevih mreža. Koje nove mogućnosti nude vremenske mreže?
8. Opisati uobičajene aktivnosti u procesu izgradnje programske potpore.
9. Koje su prednosti korištenja procesa izgradnje programske potpore (bilo kojeg modela, sa svim aktivnostima procesa) naspram postupka koji ide bez toga, tj. kod kojeg se odmah kreće s programiranjem?
10. Ukratko opisati svojstva osnovnih modela procesa izgradnje (vodopadni, evolucijski, iterativni, inkrementalni, spiralni, komponentni).
11. Navesti aktivnosti procesa izgradnje programske potpore i redoslijed njihova izvođenja

u vodopadnom modelu. ◁

12. Navesti aktivnosti procesa izgradnje programske potpore i UML dijagrame koji se koriste u njima te za što se pojedini dijagram koristi u pojedinoj aktivnosti. ◁
 13. U nekom sustavu pri obradi jednog zahtjeva sudjeluje pet dretvi: ulazna (koja zaprima zahtjev), tri radne (koje paralelno rade na tom istom zahtjevu te izlazna (koja radi završnu obradu tek kad su radne dretve gotove i vraća rezultat). Prikazati navedeni sustav Petrijevom mrežom.
 14. Sustav s jednim liftom i dva stajanja (prizemlje i prvi kat) ima sljedeća svojstva: otvaranja vrata (i zatvaranja) traje 2 sekunde, vrata su otvorena minimalno 5 sekundi, pomak lifta (na prvi kat, kao i obratno) traje 10 sekundi. Uz pretpostavku samo vanjskih tipki poziva lifta i bez unutarnjih tipki (ako se pozove, lift dođe otvoriti vrata, zatvoriti vrata i otići na onu drugu poziciju te otvoriti vrata), prikazati lift vremenskim Petrijevim mrežama uzimajući u obzir zahtjeve (pozive liftu).
 15. Sustav s jednim liftom i dva stajanja (prizemlje i prvi kat) ima sljedeća svojstva: otvaranja vrata (i zatvaranja) traje 2 sekunde, vrata su otvorena minimalno 5 sekundi, pomak lifta (na prvi kat, kao i obratno) traje 10 sekundi. Uz pretpostavku samo vanjskih tipki poziva lifta i bez unutarnjih tipki (ako se pozove, lift dođe otvoriti vrata, zatvoriti vrata i otići na onu drugu poziciju te otvoriti vrata), prikazati lift vremenskim Petrijevim mrežama uzimajući u obzir zahtjeve (pozive liftu).
 16. Vremenskom Petrijevom mrežom modelirati ulazak kupaca u trgovinu, tj. automatsko otvaranje vrata pred kupcem. Pretpostaviti da se pred pokretnim vratima nalazi senzor koji će detektirati potencijalnog kupca i poslati signal prema upravljačkom računalu koje tada pokreće otvaranje vrata. Otvaranje i zatvaranje traju po dvije sekunde, a minimalno vrijeme s otvorenim vratima jest pet sekundi. Ukoliko senzor i dalje šalje signale o detekciji kupca vrata trebaju biti otvorena za idućih pet sekundi, tj. započeti će se s zatvaranjem najranije pet sekundi nakon zadnjeg signala. Ako se u postupku zatvaranja detektira novi kupac, potrebno je prekinuti zatvaranje i ponovno otvoriti vrata.
 17. Neka automatska praonica upravlјana je računalom. Sustav se sastoji od tri trake: prve ispred ulaza u praonicu, druge koja vodi auto kroz praonicu do izlaza te treća na izlazu. Vozač se doveze na prvu traku i ugasi auto. Kad auto trenutno u praonici izade, pokrenu se prva i druga traka, dok novi auto ne uđe u praonicu. Potom počinje pranje. Po dovršetku pranja, ako je prethodno oprani auto na izlaznoj traci otišao, aktiviraju se druga i treća traka i odvezu auto van. Trake su opremljene senzorima težine. Opisati sustav Petrijevom mrežom te UML dijagramom stanja (radi ostvarenja upravljanja).
 18. Lift u nekoj zgradi ima dvije brzine kretanja: normalnu, kada prevozi putnike te brzu kada je prazan. Pri pokretanju lifta, on se naprije ubrzava na potrebnu brzinu kretanja te potom nastavlja konstantnom brzinom skoro do odredišna kata, kada započinje sa smanjivanjem brzine do zaustavljanja. Po zaustavljanju započinje s otvaranjem vrata, pa neko vrijeme stoji s otvorenim vratima te ih konačno zatvara. Ako nema novih zahtjeva lift stoji. Za opisani lift napraviti UML dijagram stanja.
-

3. Ostvarenje upravljanja

Zadaća računala u SRSV-ima je nadzor i upravljanje. Za programsko ostvarenje upravljanja treba analizirati problem i odabrat odgovarajući način upravljanja što uključuje odabir prikladne strukture programa, odabir prikladnih upravljačkih algoritama i njihovih parametara. U ovom poglavlju razmatraju se mogući načini upravljanja s obzirom na strukturu kôda i način pokretanja pojedinih aktivnosti. U drugom dijelu poglavlja ukratko se prikazuju osnovni načini ostvarenja automatskog upravljanja sustavom korištenjem PID regulatora te korištenjem neizrazite logike.

Ovo poglavlje (kao i ostala) nije osmišljeno tako da bude referenca u kojoj se nalaze potpuni opisi, već samo pokušava dati pregled mogućih načina upravljanja.

3.1. Struktura programa za upravljanje SRSV-ima

Upravljanje SRSV-ima može se ostvariti na razne načine, ovisno o potrebama i složenosti sistema. Gledano sa stanovišta programskog ostvarenja, struktura programa koja se može koristiti za ostvarenje upravljanja može se svrstati u nekoliko osnovnih kategorija:

- programska petlja, tj. upravljačka petlja
- upravljanje zasnovano na događajima (prekidima)
- korištenje jednostavne jezgre operacijskog sustava (s prekidima i alarmima)
- kooperativna višedretvenost (programska ostvarena višedretvenost)
- višedretvenost (raspoređivanje, sinkronizacija, komunikacija; korištenje OS-a za SRSV)
- raspodijeljeni sustavi (komunikacija, usklađivanje među čvorovima).

Složeniji načini nude više mogućnosti ali i zahtijevaju primjenu složenijih postupaka i složenijeg sklopovlja (skuplji su). U ovom poglavlju razmotrone su mogućnosti navedenih načina upravljanja te su prikazane ideje za njihovo korištenje.

3.1.1. Upravljačka petlja

Za najjednostavnije ugrađene sustave dovoljno je koristiti i najjednostavnije sklopovlje – mikroupravljač (engl. *microcontroller*) koji upravlja sustavom cijelo vrijeme izvodeći jednu *programsku petlju*. U toj se petlji očitavaju ulazni podaci, npr. preko senzora, proračunavaju se potrebne akcije, koje se potom pokreću u definiranim trenucima. Opće ponašanje moglo bi se definirati pseudokôdom 3.1.

Isječak kôda 3.1. Upravljačka petlja

```
upravljanje ()  
    inicijalizacija ()  
    ponavljaј  
        s1 = očitaj_senzor_1 ()  
        ako je potrebna_akcija_1 ( s1 ) tada  
            pokreni_akciju_1 ( s1 )  
        s2 = ... # slično za ostale elemente
```

Ukoliko je za upravljanje potrebno vrijeme ili treba ostvariti odgođene (periodičke) poslove tada se navedeni kod može proširiti prema 3.2.

Isječak kôda 3.2. Upravljačka petlja uz korištenje vremena

```

upravljanje ()
    inicijalizacija ()
    ponavljam
        s1 = očitaj_senzor_1 ()
        t = ažuriraj_vrijeme ()
        ako je potrebna_akcija_1 ( s1, t ) tada
            pokreni_akciju_1 ( s1, t )
        s2 = ...

        ako je ima_odgođenih_poslova ( t ) tada
            pokreni_odgođene_poslove ( t )

```

Navedena struktura može biti dovoljna za mnoštvo sustava, npr. za alarmne sustave i jednostavnije podsustave upravljanja. Složeniji sustavi se također mogu ostvariti na ovaj način, ali će tada izvedba biti složenija. Slijedi nekoliko ideja za ostvarenje upravljanja korištenjem petlje.

Primjer 3.1. Primjeri upravanja zasnovanih na upravljačkoj petlji

Alarmni sustav

U nekom sustavu zadatak upravljačkog programa jest da periodički provjerava senzore te da pri detekciji odstupanja očitane vrijednosti aktivira alarm. Sljedeće rješenje prikazuje jednu mogućnost ostvarenja gdje se u petlji provjeravaju senzori te u slučaju vrijednosti izvan granica aktivira alarm.

```

upravljanje ()
    ponavljam
        t1 = ažuriraj_vrijeme ()
        za i = 1 do N radi
            s[i] = očitaj_senzor (i)
            zapiši_očitanje (i, s[i])
            ako je s[i] < S[i].min ili s[i] > S[i].max tada
                aktiviraj_alarm (i)
        ponavljam
            t2 = ažuriraj_vrijeme ()
        dok je t2 < t1 + period_provjere

```

U idućim primjerima pretpostavljeno je da se stanje elemenata sustava može kontinuirano učitavati te naredbe slati u bilo kojem trenutku, bilo kojom učestalošću. Ukoliko to ne bi bilo tako, u kod bi trebalo dodati dodatne provjere ili odgode.

Ručno upravljana robotska ruka

U nekom sustavu zadatak upravljačkog programa je da stanje upravljačke palice prenese u naredbe aktuatorima. Upravljački program treba preko senzora dohvatiti stanje upravljačke palice, proračunati željeno stanje aktuatora te poslati naredbe koje će pomicati sustav prema tom stanju.

```

upravljanje ()
    ponavljam
        smjer = očitaj_položaj_upravljačke_palice()
        pomak = izračunaj_potreban_pomak ( stanje, smjer )
        pošalji_naredbe_aktuatorima ( pomak )

```

Složenost očitanja senzora i potrebne pretvorbe vrijednosti, kao i izračuna te slanje naredaba je u prikazanome rješenju prebačena u funkcije.

Upravljanje kočenjem

U nekom sustavu zadatak upravljačkog programa je da pri detekciji proklizavanja auta

nasumičnim brzim otpuštanjem i aktiviranjem kočnice smanji ukupni put kočenja (smanji proklizavanje). Idejno rješenje u nastavku prvo provjerava je li kočenje uključeno. Ako jest, onda na osnovu brzina okretanja kotača i stanja pedale za kočenje provjerava je li došlo do proklizavanja. Ako ima proklizavanja, potrebno je proračunati prilagođeno upravljanje kočnicama (koji kotač kočiti, koji privremeno otpustiti). Algoritam proračuna ovisi o mnogo parametara i nije razmatran u ovom primjeru.

```

upravljanje ()
ponavljam
    kočenje = dohvati_stanje_pedale_kočnice ()
    ako je kočenje > 0 tada
        b = očitaj_brzine_svih_kotača ()
        ako je provjeri_proklizavanje ( kočenje, stanje, b ) tada
            t = ažuriraj_vrijeme ()
            stanje = proračunaj_novo_stanje ( kočenje, stanje, b, t )
            za i = 1 do 4 radi
                postavi_stanje_kočnice_za_kotač ( stanje, i )
            inače
                postavi_kočnice ( kočenje )
            inače
                otpusti_kočnicu ()

```

U sustavima kod kojih upravljanje elementima treba izvoditi periodički, gdje svaki element ima različitu periodu potrebno je ažurnije pratiti protok vremena. Sljedeći primjeri prikazuju neke mogućnosti, svaki sa svojim prednostima i nedostacima.

Periodičko očitanje senzora (1)

U nekom sustavu zadatak upravljačkog programa jest periodički očitavati senzore s_1, s_2, s_3 i s_4 s periodama 30 ms, 40 ms, 20 ms te 100 ms, respektivno.

```

N = 4
T[N] = { 30, 40, 20, 100 } # periode (konstante)
t[N] = { 30, 40, 20, 100 } # koliko do iduće aktivacije

upravljanje ()
ponavljam
    t1 = ažuriraj_vrijeme ()
    odgodi = t[1]
    za i = 2 do N radi
        ako je t[i] < odgodi tada
            odgodi = t[i]

    ponavljam
        t2 = ažuriraj_vrijeme ()
        dok je t2 < t1 + odgodi

        za i = 1 do N radi
            t[i] -= odgodi
            ako je t[i] <= 0 tada
                s = očitaj_senzor (i)
                poduzmi_akcije ( i, s )
            t[i] += T[i]

```

Alternativno rješenje za ostvarenje periodičkih poslova moglo bi uzeti najmanji korak kojim bi se sigurno zaustavili u svakom potrebnom trenutku. Za prethodni primjer taj korak iznosi 10 ms, tj. svaka bi odgoda trajala 10 ms nakon koje bi išlo ažuriranje vremena do idućih očitanja senzora ($t[i] == 10 \text{ ms}$) te eventualno i pokretanje očitanja ako je vrijeme dostignuto ($t[i] <= 0$).

Pristup iz prethodna primjera biti će dovoljno precizan jedino ako sve operacije vrlo kratko traju, pogotovo očitaj_senzor i poduzmi_akcije. Kada bi neka operacija trajala

duže, vrijeme odgode bi se povećavalo, ali ne i uračunalo pri određivanju sljedećih izvođenja. U tim slučajevima bolje je koristiti absolutno vrijeme za aktivaciju te uspoređivati ga s trenutnim vremenom. Idući primjer koristi navedeni pristup.

Periodičko očitanje senzora (2)

Korištenjem absolutnog vremena pri određivanju vremena za periodičku aktivacije operacija za isti zadatak iz prethodnog primjera postiže se znatno veća preciznost neovisna o trajanju pokretanja akcija.

```
N = 4
T[N] = { 30, 40, 20, 100 } # periode (konstante)
t[N] # kada je iduća aktivacija

upravljanje ()
    # postavi početna vremena ažuriranja
    t = ažuriraj_vrijeme ()
    za i = 1 do N radi
        t[i] = t + T[i]

ponavljam
    t = ažuriraj_vrijeme ()
    odgodi_do = t[1]
    za i = 2 do N radi
        ako je odgodi_do > t[i] tada
            odgodi_do = t[i]

ponavljam
    t = ažuriraj_vrijeme ()
    dok je t < odgodi_do

    za i = 1 do N radi
        t = ažuriraj_vrijeme ()
        ako je t[i] <= t tada
            s = očitaj_senzor (i)
            poduzmi_akcije ( i, s )
            t[i] += T[i]
```

U prethodna dva primjera relativno jednostavnii zahtjevi (periodičke akcije) zahtijevaju ne tako jednostavno rješenje kada se koristi upravljačka petlja. Kada sustav kojeg treba upravljati sadrži više od jednog elementa čije očitanje ili upravljanje nije sinkronizirano, ostvarenje upravljanja s petljom postaje sve složenije i teže za osmisliiti i ostvariti. Povećana složenost vodi i do povećane mogućnosti grešaka i njihova težeg otkrivanja. Ipak, ponekad je i složenije sustave moguće upravljati programskom petljom, ako se odabere prikladna struktura podataka za zapis stanja svih elemenata sustava.

Upravljanje sustavom s više elemenata

Prepostavimo poopćeni sustav s više elemenata, kod kojih svaki od elemenata može biti u nekom stanju te drukčije reagira na događaje ovisno o stanju u kojem se trenutna nalazi (i eventualnu povijest). Neka se stanje elementa zapisuje u odgovarajuću podatkovnu strukturu. Nadalje, prepostavimo da su akcije koje poduzima upravljačko računalno kratkog trajanja. Npr. akcije su tipa "pročitaj senzor", "proračunaj potrebnu akciju" te "pošalji naredbu". Upravljanje takvim sustavom korištenjem petlje moglo bi izgledati:

```
stanje[N] # stanje za svaki element

upravljanje ()
    ponavljam
        za i = 1 do N radi
            ulazi = očitaj_stanje_i_ulaze_za_element (i)
```

```
t = ažuriraj_vrijeme ()
proračunaj_i_poduzmi_potrebne_akcije ( i, stanje[i], ulazi, t )
```

Kada očitanja, proračuni ili pokretanje akcija ne bi bili kratki, reakcija na neke događaje određene komponente (koji se u ovom načinu otkrivaju očitanjem senzora) bi mogla biti neprihvatljivo duga, a sve zbog čekanja na dovršetka obrade prijašnjih komponenata u lancu.

3.1.2. Upravljanje zasnovano na događajima

Za mnoge sustave upravljanje zapravo predstavlja reakciju na događaje iz okoline. U prethodnom načinu upravljanja smo takve događaje otkrivali očitavanjem senzora. Međutim, ako je neophodna vrlo brza reakcija tada treba ili velikom frekvencijom očitavati senzore ili koristiti druge mehanizme koji će u *trenutku događaja* i reagirati, bez odgađanja. U računalnom sustavu postoji *mehanizam prekida* i njegove obrade koji mogu poslužiti upravo za ostvarenje ovakvog načina upravljanja. Takvi sustavi moraju imati “podsustav za prihvatanje prekida” koji će omogućiti povezivanje događaja koji uzrokuje prekid i akciju koju treba poduzeti (funkciju za obradu prekida).

Prekidni podsustav mora pružati sučelje za povezivanje događaja s funkcijom koja ga obrađuje. Pri inicijalizaciji sustava, korištenjem tog sučelja prekidi se povežu s odgovarajućim funkcijama za obradu (upravljačkim programima naprava, engl. *device drivers*). Dok nema događaja procesor može raditi nešto drugo, npr. upravljati elementima sustava koji se ne javljaju mehanizmom prekida. Stoga se upravljanje korištenjem prekida može smatrati i nadogradnja upravljanja petljom. “Cijena” nadogradnje je dodatno potrebno sklopolje za prihvatanje prekida (složeniji procesori) te programski podsustav za upravljanje prekidima.

Prema potrebi (i mogućnostima) svakom se izvoru prekida može pridijeliti *prioritet* kojim se u slučaju višestrukih prekida ili u slučaju pojave jednog prekida za vrijeme obrade drugog može odlučiti koji će se prije obraditi. Pri preklapanju prekida, odabire se onaj prekid većeg prioriteta, dok se obrada prekida manjeg prioriteta blokira ili privremeno prekida dok se prioritetniji ne obradi. Pri prekidu, kontekst prijašnjeg posla (nekog programa ili obrade prekid manjeg prioriteta) se privremeno pohranjuje na stog s kojeg se i obnavlja po završetku obrade prekida.

U pojedinim dijelovima programa ili obrade prekida može biti potrebno privremeno zabraniti prekidanje. Npr. ukoliko se u nekom segmentu kôda koriste zajedničke strukture podataka (npr. prekidnog podsustava) taj bi se segment trebao zaštititi tako da se započeta operacija obavi do kraja, da struktura podataka ostane u konzistentnom stanju. Prekidni podsustav mora omogućiti mehanizam *zabrane* te ponovne *dozvole prekidanja*.

Dakle, u sustavima koji posjeduju prekidni podsustav upravljanje nekim elementima se može ugraditi u funkcije obrade prekida. Opća slika rada sustava upravljanog prekidima, tj. *pojavom događaja*, sastoji se od niza funkcija koje se pozivaju iz prekida. Upravljanje može biti ostvareno isključivo u prekidnim funkcijama ili se za upravljanje mogu koristiti i drugi principi (kao što je petlja). Česta je kombinacija petlje i obrade prekida, kao u primjeru 3.3.

Isječak kôda 3.3. Upravljanje koje kombinira prekide s drugim oblicima

```
inicijalizacija()
postavi_prekidni_podsustav()
poveži_prekide_s_odgovarajućim_funkcijama()
pokreni_upravljanje() # npr. petljom

obrada_prekida_1 ()
prekidni_dio_upravljanja_napravom_1 ()

obrada_prekida_2 ()
```

```

prekidni_dio_upravljanja_napravom_2 ()

obrada_prekida_3 ()
  prekidni_dio_upravljanja_napravom_3 ()
...

```

Prekidni podsustav je začetak jezgre operacijskog sustava. Dodavanje novih podsustava proširuje mogućnosti upravljanja ali i zahtijeva početno složeniji sustav, sa svojim dodatnim zahtjevima na sklopolje (procesor, spremnik) i programe (programski i podatkovni dio podsustava).

3.1.3. Jednostavna jezgra operacijskog sustava

Nadogradnjom prekidnog podsustava s *podsustavom za upravljanje vremenom*, tj. mogućnošću odgode i postavljanja *alarma* koji se aktiviraju u budućim trenucima, dobiva se *jednostavna jezgra operacijskog sustava*. Za navedene operacije potrebno je brojilo koje sinkrono odbrojava i izaziva prekid po svom dovršetku (dolasku do nule). U obradi prekida brojila pregledava se vrijednost odgode te aktivni alarmi. Ukoliko je odgoda istekla, program nastavlja s radom. Ako je neki alarm istekao, njega se sada aktivira (pozove funkciju zadana pri stvaranju alarma).

Osnovna sučelja podsustava za upravljanje vremenom uključuju:

- odgodu za određeni interval – `odgodi (T)`
- odgodu do određenog budućeg vremena – `odgodi_do (T)`
- stvaranje alarm-a – `postavi_alarm (T1, T2, akcija)`
 - T1 predstavlja odgodu do prve aktivacije alarm-a
 - T2, ako je zadan, definira periodičke alarme, koji će se aktivirati svakih T2 jedinica vremena, nakon početne aktivacije
 - akcija predstavlja funkciju koju treba pozvati pri aktivaciji alarm-a.

U sustavima s ovakvom jednostavnom jezgrom mogu se kombinirati razni mehanizmi upravljanja. Za periodičke poslove mogu se koristiti alarmi, svaki sa svojim budućim trenutkom aktivacije i periodom ponavljanja. Asinkroni događaji mogu se upravljati iz prekidnih funkcija, a svi ostali dijelovi sustava korištenjem upravljačke petlje. Odgoda programa može se ostvariti podsustavom upravljanja vremenom, npr. funkcijom `odgodi (t)` umjesto radnim čekanjem, kao što je korišteno u prethodnim primjerima.

Prethodne konstrukcije:

```

t1 = ažuriraj_vrijeme ()
...
ponavljaј
  t2 = ažuriraj_vrijeme ()
  dok je t2 < t1 + odmak

```

mogu se zamijeniti sa:

```

t1 = dohvati_vrijeme ()
...
odgodi_do ( t1 + odmak )

```

Korištenje relativne odgode sa `odgodi (odmak)` treba izbjegavati zbog mogućeg gomilanja greške jer se ne uzima u obzir trajanje izvođenja programa, koje iako može biti vrlo kratko u jednoj iteraciji petlje ipak postaje značajno nakon mnogo iteracija.

Korištenje jednostavne jezgre, upravljanje se značajno jednostavnije ostvaruje, a moguće je ostvariti i složenija upravljanja koja bez jezgre nisu bila moguća. U nastavku slijedi nekoliko primjera.

Primjer 3.2. Primjeri upravljanja korištenjem prekida i alarma

Periodičko očitanje senzora alarmima

Prethodni primjer s periodičkim očitanjem senzora i reakcijom na njih korištenjem upravljačke petlje može se u sustavu s jednostavnim jezgrom ostvariti postavljanjem četiri alarma prema sljedećem kôdu.

```
T[i] = { 30, 40, 20, 100 } # periode

inicijaliziraj ()
    inicijaliziraj_prekidni_podsustav()

postavi_alarm ( T[1], T[1], obrada_1 )
postavi_alarm ( T[2], T[2], obrada_2 )
postavi_alarm ( T[3], T[3], obrada_3 )
postavi_alarm ( T[4], T[4], obrada_4 )

pokreni_upravljanje_ostalim_komponentama ()

obrada_1 ()
    s = očitaj_senzor (1)
    poduzmi_akcije ( 1, s )

obrada_2 ()
...
...
```

Kada očitanja, proračuni ili pokretanje akcija ne bi bili kratki, reakcija na neke događaje određene komponente (koji se u ovom načinu otkrivaju očitanjem senzora) bi mogla biti neprihvatljivo duga, a sve zbog čekanja na dovršetka obrade prijašnjih komponenata u lancu. U takvoj situaciji bilo bi potrebno razmišljati o dodavanju prioriteta pojedinim funkcijama i traženju sučelja koja to nude.

Upravljanje alatnim strojem

Neka se neki alatni stroj može upravljati programski, korištenjem petlje, uz povremene odgode zadavanja naredbi. Pri radu, zbog raznih razloga (loš program, nekvaliteta alata, tvrdoča predmeta koji se obrađuje i slično) može se dogoditi da se alat pregrije. Za takve situacije postoji senzor koji će mehanizmom prekida signalizirati računalu da treba privremeno zaustaviti rad dok se alat ne ohladi. Sljedeći odsječak kôda može poslužiti kao idejno rješenje upravljanja.

```

upravljanje ()
    poveži_prekid ( PREKID_PREGRIJAVANJA, pregrijavanje )
    inicializacija_svega_ostalog ()
    ...
    # dio koda koji se aktivira po pokretanju operacije
    p = učitaj_naredbe ()
    za i = 1 do p.broj_naredbi radi
        postavi_naredbu ( p.naredba[i] )
        odgodi ( p.trajanje[i] )
        dok je p.očitaj_stanje() != p.očekivano_stanje[i] radi
            odgodi ( p.odgoda[i] )
        ...
    # funkcija koja se poziva iz prekida senzora za pregrijavanje alata
    pregrijavanje ()
        postavi_naredbu ( STANI | HLAĐENJE )
        postavi_alarm ( Th, 0, nastavi )

    # po isteku prethodnog alarme
    # (alat se dovoljno ohladio da može dalje raditi)
    nastavi ()
        postavi_naredbu ( NASTAVI )

```

Prekidi, alarmi i petlja mogu poslužiti za upravljanje mnogim sustavima, čak i vrlo složenima. Ipak, porastom složenosti sustava raste i složenost ostvarenja upravljanja. Ukoliko je složenost prevelika treba razmisliti o korištenju drugih načina, kao što je višedretveno upravljanje.

Osim višedretvenosti upravljanje nezavisnim elementima sustava može se zasnovati na korišteњu struktura podataka koje sadrže stanje pojedinih elemenata. Upravljanje se svodi na petlju u kojoj se ažuriraju te strukture podataka protokom vremena i stanjem naprava (senzora) te po potrebi pokreću odgovarajuće akcije. Stanje elementa sustava kojim se upravlja zapisano je u strukturi podataka te je akcija ovisna o stanju. Prethodni primjer s alatnim strojem jedan je jednostavniji primjer takvog upravljanja. U prethodnom poglavljju je prikazano upravljanje liftovima kroz upravljačku petlju te opsežniju strukturu podataka koja opisuje stanje sustava. U još složenijim sustavima bi se takvo upravljanje još moglo proširiti prekidima i alarmima.

3.1.4. Kooperativna višedretvenost

Proširenje jezgre podsustavom za višedretvenost je značajno proširenje koje velikim dijelom mijenja i ostale podsustave.

Ponekad se jednostavniji oblici višedretvenosti mogu ostvariti i bez podrške jezgre operacijskog sustava, ostvarenjem višedretvenosti unutar samog programa. Potrebne dretve mogu se "ugraditi" u program te prelazak s jedne dretve na drugu (zamjena aktivne dretve) ostvariti izravnim pozivima u programu. Prelaz s jedne dretve na drugu pretpostavlja pohranu konteksta prve i obnovu konteksta druge dretve.

Izravno raspoređivanje dretvi

Zamjena dretva može biti izravna, gdje dretva A izravno predaje kontrolu dretvi B (A traži da ga B zamjeni). Primjerice, preko sučelja `nastavi_dretvu_B()` dretva A prepušta procesor dretvi B. Ovakvo izravno raspoređivanje dretvi mora se programirati u kôd upravljačkog programa. Takva višedretvenost je najjednostavnija za ostvariti, ali i s najviše ograničenja. Idući primjer prikazuje jedno takvo rješenje.

Primjer 3.3. Primjer izravnog raspoređivanja

<pre>dretva A ponavlja obrada_A1 () nastavi_dretvu_B () obrada_A2 () dok je uvjet1 radi nastavi_dretvu_B ()</pre>	<pre>dretva B ponavlja obrada_B1 () nastavi_dretvu_A () obrada_B2 () ako je uvjet2 tada nastavi_dretvu_A ()</pre>
---	---

Naizgled bi se `nastavi_dretvu` mogao zamijeniti skokom. Međutim, uvjeti onemogućuju obične skokove. Mogli bi pamtiti do kuda su stigle pojedine dretve prije skoka, ali to je zapravo dio konteksta dretve, odnosno, to i je neki oblik kooperativne višedretvenosti.

Ponekad bi se, radi boljeg razumijevanja, upravljanje moglo ostvariti na način sličan kao i u primjeru 3.3. iako je moguće i petljom. Npr. ovako se grupiraju naredbe koje se odnose na isti element upravljanja. Na mesta gdje je pojedina naredba ili akcija gotova kontrola se predaje drugim dretvama.

Neizravno raspoređivanje dretvi

Uz dodatnu podatkovnu strukturu (npr. jednostavnu listu) moglo bi se programski ostvariti (izvan jezgre operacijskog sustava) i općenitije upravljanje dretvama, tj. raspoređivanje kod kojeg nije potrebno izravno odrediti iduću dretvu, već se ona određuje iz skupa dretvi (npr. prva iz liste). U takvim sustavima dretve mogu imati i različite prioritete, a dovoljno sučelje može biti `prepusti_procesor()` (engl. `yield()`). Primjer 3.3. bi se u takvom sustavu mogao preinaći u 3.4.

Primjer 3.4. Primjer neizravnog raspoređivanja

<pre>dretva A ponavlja obrada_A1 () prepusti_procesor () obrada_A2 () dok je uvjet1 radi prepusti_procesor ()</pre>	<pre>dretva B ponavlja obrada_B1 () prepusti_procesor () obrada_B2 () ako je uvjet2 tada prepusti_procesor ()</pre>
---	---

Kada bi u sustavu za primjer 3.4. bilo više dretvi, npr. još i dretve C, D i E, tada bi se za svaki poziv prepuštanja procesora iduća dretva odredila uvidom u dodatnu podatkovnu strukturu, tj. odabir iduće dretve nije "programiran" kod svake dretve. Mogućnosti za upravljanje korištenjem neizravnog raspoređivanja su stoga značajno veća nego kod izravnog raspoređivanja.

Treba primijetiti da u oba prikazana načina kooperativne višedretvenosti dretve same odlučuju *kada će* prepustiti procesor drugoj dretvi, tj. odluka *kada* je programirana. Ovakvo raspoređivanje nazivamo i neprekidivim (engl. *non preemptive*) jer trenutno aktivnu dretvu ne može prekinuti neka druga. Naravno da će u sustavima koji imaju prekidni podsustav sami prekidi prekidati izvođenje dretvi, ali će se nakon obrade prekida, kontrola vratiti u istu, prije prekinutu, dretvu, a ne neku drugu, što je možda potrebno u nekim sustavima.

Primjerice, ako prepostavimo da u prethodnom primjeru dretva A obavlja vrlo bitan posao u svom prvom dijelu `obrada_A1`, manje bitan u `obrada_A2` te da `uvjet1` postavlja neka funkcija koja se poziva u obradi prekida. Ukoliko je potrebno da reakcija dretve A treba biti vrlo brza odrada dijela `obrada_A1`, ovakav kooperativni način upravljanja dretvi možda neće biti odgovarajući. Naime, osim što će se u obradi prekida naprave poništiti `uvjet1`, nakon obrade vratit će se u onu dretvu koja je bila prekinuta. Ta dretva može biti i dretva B, te dretva

A neće doći na red dok joj B to sama ne prepusti.

Ovakvo ograničenje može se riješiti jedino ugrađivanjem podrške za višedretvenost u jezgru, odnosno, samo raspoređivanje treba ostvariti u jezgri (uklopiti ga u sve podsustave, kao što je to i prekidni podsustav).

3.1.5. Višedretvenost

Višedretvenost ostvarena u jezgri operacijskog sustava omogućuje ostvarenje i vrlo složenih sustava upravljanja. Primjerice, prekidom uzrokovanim nekom napravom sustav može propustiti trenutno blokirano dretvu koja je upravo čekala na događaj naprave i nastaviti s njenim radom ukoliko je njen prioritet veći od prethodno aktivne dretve (prekinute prekidom). Takve sustave nazivamo *prekidljivim* ili *istiskujućim* (engl. *preemptive*). Dretve u tom sustavu mogu osim u aktivnom i pripravnom stanju (kao i kod kooperativne višedretvenosti) biti i u blokiranom stanju, gdje čekaju na događaj UI naprave, sinkronizacijski mehanizam, protok vremena ili slično.

Obzirom da višedretvenost podrazumijeva raspoređivanje dretvi (prema prioritetu i drugim kriterijima), mehanizme sinkronizacije i međusobne komunikacije dretvi, zaštitu jedne dretve od druge i slično, potreban je "pravi operacijski sustav". Kada se upravljanje ne može ostvariti na jednostavnije načine tada je najbolje odabratи неки od postojećih operacijskih sustava za SRSV koji, pored ostalih mogućnosti, standardno dolaze s podrškom za višedretvenost i pripadne funkcionalnosti.

Višedretvenost se može koristiti na nekoliko načina radi pojednostavljenja upravljanja.

Osnovni način jest da se za pojedine komponente sustava koristite zasebne dretve koje će njima upravljati i tako svu funkcionalnost ostvariti na jednom mjestu ("podijeli i vladaj"). Svaka dretva može imati vlastiti način upravljanja (npr. koji koristi trenutno stanje elementa i sustava, čita potrebne ulaze i podatke o željenom ponašanju te šalje potrebne naredbe).

Za neke probleme ponekad je jednostavnije dijelove upravljanja izravno ugraditi nego li stalno provjeravati ulaze i proračunavati iduće poteze. Primjerice, ako je poznato da od jednog stanja do željenog budućeg stanja sustavom treba upravljati na točno zadani način i da (skoro) ništa što se na ulazu može pojaviti to ne može promijeniti, onda se niz potrebnih naredbi za takvu promjenu stanja može ugraditi u upravljanje dretve. "Ugradnja" ne mora biti isključivo programska – nizom instrukcija, već i korištenjem strukture podataka (npr. lista naredbi više razine). Upravljanje u dretvi se tada pojednostavljuje: ona otkriva takva stanja i ulazi u način rada izvođenja operacije po operacije do dostizanja željenog stanja. Navedeni mehanizam nije ograničen samo na višedretvena ostvarenja.

U nekim se sustavima upravljanje može ostvariti i hijerarhijski. Odluke o operacijama mogu se donositi na najvišoj razini dok se za njihovo sprovođenje trebaju stvarati odluke na nižim razinama. Primjerice, za lift se na najvišoj razini može odlučivati o tome gdje lift treba stati dok se na nižim razinama to sprovodi u slanje poruka, primanje poruka, zadržavanje i slično. Kada se sustav sastoji od više komponenata tada se može jednom dretvom ostvariti najviša razina donošenja odluka o pokretanju osnovnih operacija dok se njihovo sprovođenje može obaviti u drugim dretvama koje upravljaju pojedinim komponentama.

Pri ostvarivanju sustava korištenjem dretvi u okviru operacijskih sustava za SRSV treba pripaziti na raspoređivanje dretvi, sinkronizaciju, komunikaciju, korištenje podataka u zajedničkom spremniku te utrošak vremena koje jezgra koristi za upravljanje dretvama. Navedeni su problemi detaljnije razmotreni u idućim poglavljima.

3.1.6. Raspodijeljeni sustavi

Mnogi SRSV-i se sastoje od više komponenata (ili čvorova, engl. *nodes*) kojima zasebno upravlja vlastiti računalni sustav (npr. jednostavni mikroupravljač). Svaka komponenta "komunicira" s ostalim komponentama neizravno, korištenjem svojih senzora, detekcijom promjena koje su posljedica djelovanja drugih komponenata. Ipak, često je potrebna i izravna komunikacija i razmjena informacija i naredbi. Takva komunikacija treba biti podržana odgovarajućim podsustavom koji ostvaruje potrebne protokole.

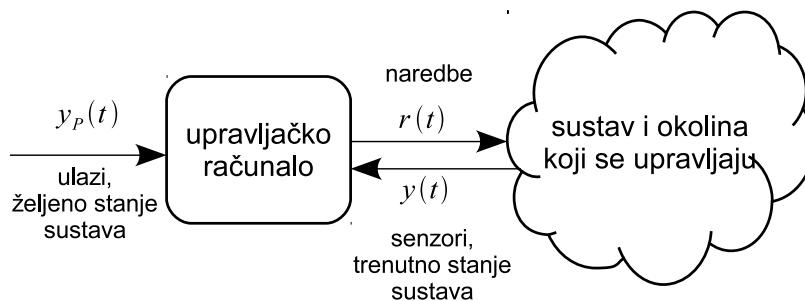
Ostvarenje upravljanja u pojedinoj komponenti raspodijeljenog sustava može biti ostvareno bilo kojim od prethodnih načina. Informacije koje dolaze od drugih komponenata mogu se smatrati kao podaci s ulaznih naprava.

Međutim, posebnost raspodijeljenog sustava je u vremenu odziva. Ulazno-izlazne naprave spojene na jedan računalni sustav su upravljane tim sustavom i njihova se reakcija može poprilično dobro predvidjeti. U raspodijeljenom sustavu vremena odziva pojedinih komponenti mogu biti znatno veća, ali i što je zapravo problem, njihovo trajanje može biti vrlo teško predvidjeti. Posebice je to istaknuto ako su komponente daleko jedna od druge i za komunikaciju koriste nepouzdane kanale ili protokole koji nemaju prikladnu podršku za vremensko usklađenu komunikaciju i signalizaciju (kao što je to Internet ili sustav koji koristi iste protokole). Osnovni problem raspodijeljenih sustava jest u teškom određivanju trajanja pojedinih operacija za najgore moguće slučajeve (što svakako treba razmatrati za SRSV-e).

U sustavu s više komponenata, sa složenim načinom komunikacije, te strogim vremenskim ograničenjima, projektiranje raspodijeljenih sustava postaje izazov (tj. izrazito težak problem).

3.2. Ostvarenje regulacijskih zadataka

Upravljanje mnogih SRSV-a podrazumijeva i *automatsko upravljanje* dijelom podsustava tako da se taj dio ponaša u skladu s očekivanjima. Primjer su sustavi za održavanje brzine, smjera, temperature, tlaka, napona. Svi oni preko senzora očitavaju trenutne vrijednosti $y(t)$ te na osnovu postavljene (željene) vrijednosti $y_P(t)$ i algoritma usklađivanja šalju naredbe $r(t)$ koje će pokušati usmjeriti sustav u postavljeno stanje, prema slici 3.1. Iako su na slici sve vrijednosti navedene kao skalari (jednodimenzionalne), oni mogu biti i drugi složeniji podaci (vektori, matrice, ...). Radi jednostavnosti razmatranja i u nastavku teksta će se koristiti obični skalari.



Slika 3.1. Upravljanje stanjem sustava

Upravljački dio sustava mora reagirati u stvarnom vremenu, tj. očitavati (ili računati) željeno stanje sustava, preko senzora dobiti informacije o trenutnom stanju sustava te izdavati naredbe koje će dovoditi sustav u željeno stanje, tj. smanjiti trenutna odstupanja.

Računala rade u koracima, izvode instrukciju za instrukcijom, proračunavaju i šalju naredbu za naredbom sustavu kojeg upravljaju. Također, očitanja stanja sustava obavljaju se u pojedinim trenucima vremena, diskretnim trenucima, a ne kontinuirano. Kada su ti trenuci dovoljno blizu

jedan drugome (tj. frekvencija očitanja dovoljno velika¹), promjena sustava je mala te se na svaku promjenu može dovoljno brzo reagirati (što i regulator radi). Obzirom da se i očitanja i naredbe izdaju u diskretnim trenucima t_k vrijednosti svih varijabli u tim trenucima mogu se izraziti kao funkcije od t_k , ili i kraće samo dodajući varijablama indeks k . Tako $y_P(t)$ postaje $y_{P,k}$, $y(t)$ postaje y_k , $r(t)$ postaje r_k i slično za ostale vrijednosti i funkcije. Integral i derivacija se aproksimiraju sumom i razlikom susjednih grešaka.

Problemi koje diskretizacija unosi se ovdje neće zasebno razmatrati. Uglavnom, potrebno je da se pripazi na stabilnost algoritma, odnosno, da se korak integracije prikladno odabere u skladu s odabranim postupcima (npr. postupkom numeričke integracije).

3.2.1. Upravljanje bez povratne veze

Ponekad informacije o trenutnom stanju sustava nisu potrebne ili nisu dostupne. Primjerice, pri upravljanju protokom kroz neku slavinu potrebno stanje slavine za željeni protok (uz poznate ostale parametre) se može proračunati. Stvarni će protok biti jednak proračunatom te očitanje stvarnog protoka i nije neophodno.

Za neke složenije sustave, koji su pod stalnim nadzorom (i/ili upravljanjem) čovjeka (npr. preko upravljačke palice), povratna informacija o stanju sustava i nije neophodna obzirom da će čovjek zadavati odgovarajuće naredbe koje će usmjeriti sustav u željenom smjeru.

U oba prethodna primjera upravljačko računalo stvara naredbe samo na osnovu ulaza, tj. podataka o željenom stanju sustava te takva upravljanja nazivamo *upravljanja bez povratne veze*.

Matematički gledano, izlazne naredbe $r(t)$ koje računalo stvara pri upravljanju bez povratne veze su funkcija samo ulaza $y_P(t)$, prethodnog proračunatog stanja $u(t)$ te eventualno i protoka vremena t te se mogu iskazati funkcijски prema (3.1.).

$$r(t) = F(y_P(t), u(t), t) \quad (3.1.)$$

Funkcija F ovisi o problemu i može biti jednostavna ili složena. Za jednostavnija upravljanja funkcija F može ovisiti samo o ulazu y_P i to najčešće linearno te se može zapisati prema (3.2.).

$$r(t) = K_1 \cdot y_P(t) + K_0 \quad (3.2.)$$

I složeniji sustavi se u okolini upravljačke točke uglavnom mogu aproksimirati linearnom funkcijom te se upravljanje i tada može ostvariti jednostavnim mikrokontrolerom.

3.2.2. Upravljanje uz povratnu vezu

Sustavi koji nisu pod stalnim nadzorom čovjeka i koji ne upravljanju procesima za koje se može predvidjeti ponašanje moraju u stvaranju naredbi uzeti u obzir trenutno stanje sustava, tj. moraju koristiti *povratnu vezu*. Ekvivalentno funkciji (3.1.), za sustave s povratnom vezom može se definirati funkcija (3.3.) kojom se izračunavaju naredbe, tj. vrijednosti koje se šalju aktuatorima koji upravljaju sustavom.

$$r(t) = F(y_P(t), y(t), u(t), t) \quad (3.3.)$$

Obzirom da se koristi povratna veza, upravljačko računalo prilagođava svoje izlaze trenutnom stanju sustava i time *regulira* (prilagođava) ponašanje sustava, tj. obavlja funkciju *regulatora*. Regulatori mogu biti ostvareni na razne načine. Za ostvarenja osnovnih regulatorskih zadataka često se koriste *PID regulatori*. Za složenije sustave kod kojih se funkcija upravljanja ne može egzaktno iskazati mogu se koristiti i postupci koji se zasnivaju na *neizrazitom zaključivanju* (engl. *fuzzy logic*). U nastavku su ukratko prikazana načela oba pristupa.

¹Prema Nyquist Shannonovu teoremu frekvencija uzorkovanja treba biti barem dvostruko veća od frekvencije signala kojeg se uzorkuje (njegovog najvišeg harmonika), a da bi se signal mogao u potpunosti obnoviti.

3.3. PID regulator

PID regulator svoje upravljanje zasniva na razlici između željenog stanja $y_P(t)$ i trenutnog stanja $y(t)$, tj. na trenutnom odstupanju $e(t)$. Sve se vrijednosti mogu mijenjati s vremenom te su i naznačene kao funkcije vremena t . Naredbe koje regulator izdaje su stoga funkcije od $e(t)$.

$$e(t) = y_P(t) - y(t) \quad (3.4.)$$

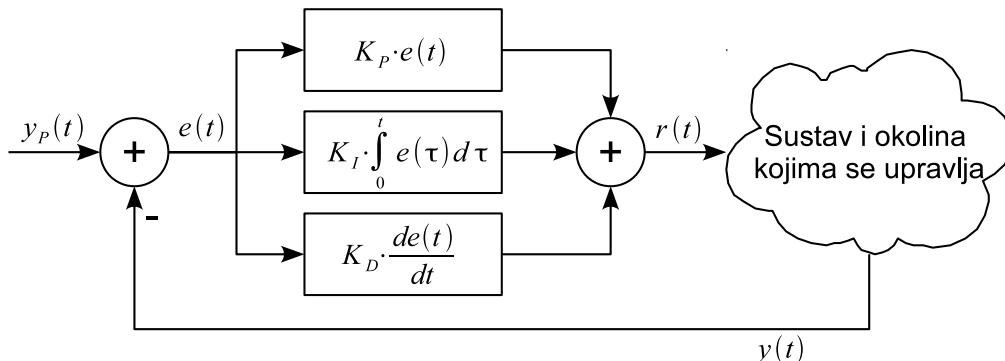
$$r(t) = F(e(t)) = F(y_P(t) - y(t)) \quad (3.5.)$$

PID regulator svoje ime dobiva od tri komponente funkcije F . One su:

- P – proporcionalna komponenta
- I – integracijska komponenta i
- D – derivacijska komponente.

Slika 3.2. prikazuje najjednostavniju izvedbu PID regulatora kod kojeg se sva tri dijela jednostavno zbrajaju prema formuli 3.6.

$$r(t) = K_P \cdot e(t) + K_I \cdot \int_0^t e(\tau) d\tau + K_D \cdot \frac{de(t)}{dt} \quad (3.6.)$$



Slika 3.2. PID regulator

Osim PID regulatora koji sadrži sva tri elementa, za neke se primjene mogu koristiti i samo P ili PI ili PD regulatori sa samo nekim od te tri komponente.

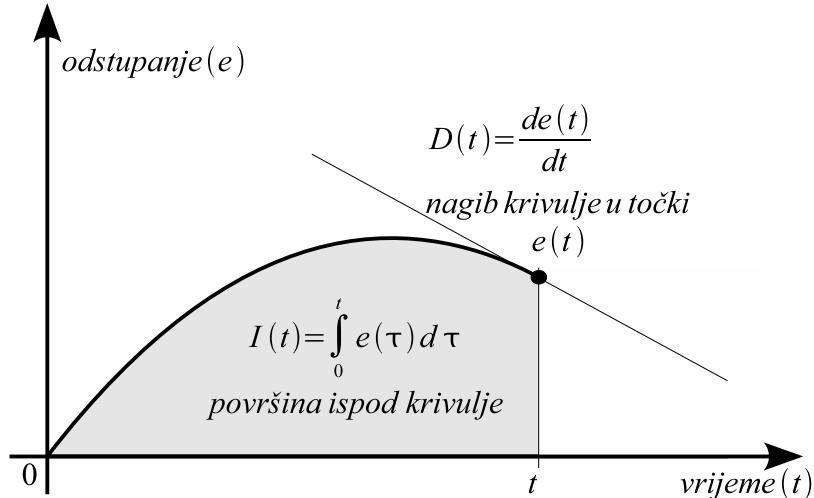
3.3.1. Proporcionalna komponenta – P

Osnovna komponenta regulatora je proporcionalna koja treba ispraviti odstupanje sustava od željenog stanja. Parametar K_P se podešava prema potreboj brzini uspostave željenog stanja. Što je K_P veći to će reakcija biti brža, i obratno, što je K_P manji to će se sustav sporije približavati željenom stanju. Pri odabiru prave vrijednosti tog parametra (a i ostalih) treba koristi modele ponašanja sustava i/ili heuristiku i/ili ispitivanja. Prevelika vrijednost K_P može dovesti do nestabilnosti, dok premala vrijednost ne mora biti dovoljna da se sustav dovede u željeno stanje (zbog otpora u sustavu). Primjerice, ako je K_P preveliki reakcija na grešku može uzrokovati još veću grešku ali drugog predznaka, ova pak još veću itd.

Problemi regulatora koji imaju samo P komponentu se u nekim sustavima javljaju kao nemogućnost dostizanja željenog stanja sustava. Slika 3.4. prikazuje taj problem.

3.3.2. Integracijska komponenta – I

Osnovna zadaća integracijske komponente je da ubrza promjene kada se odstupanje sporo smanjuje (ili se ne smanjuje). Integracijska komponenta zbraja (integrira) prijašnje greške, te ako se greške ne smanjuju, ona vremenom sve značajnije pridonosi povećanju izlaza $r(t)$. Slika 3.3. prikazuje načelo izračuna integracijske komponente na osnovu prijašnjih odstupanja, tj. kao njihov integral.



Slika 3.3. Izračunavanje integracijske i derivacijske komponente

Integracijska komponenta će ponekad riješiti i problem odstupanja od željene vrijednosti koje može nastati uporabom samo P regulatora. Integracijski parametar K_I treba biti pažljivo odabran jer će u protivnom dovesti do istih problema kao i neprikidan parametar K_P .

3.3.3. Derivacijska komponenta – D

Kada reakcija na odstupanja mora biti promptnija (brža) tada se i K_P ili K_I moraju povećavati. Jedan od problema koje to može uzrokovati su oscilacije oko željenog stanja sa sporim prigušnjem. Osnovni zadatak derivacijske komponente jest da smanji oscilacije i da ubrza stabilizaciju sustava. Parametar K_D mora se odabrati tako da donekle smanji brzinu promjene koju stvaraju proporcionalna i integracijska komponenta, ali ne previše, jer bi sustav tada sporo dostizao željeno stanje. Preveliki K_D može dovesti sustav do nestabilnog stanja (preburna reakcija na promjene). Derivacijska komponenta izračunava se kao derivacija odstupanja, prema slici 3.3.

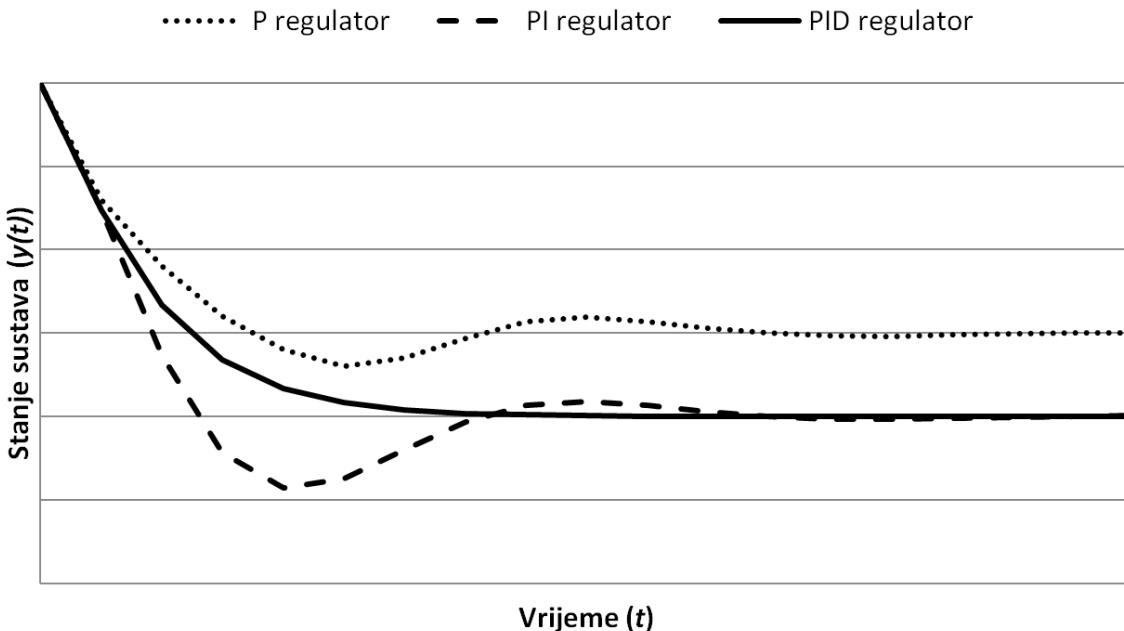
Slika 3.4. prikazuje primjer s utjecajem proporcionalne, integracijske i derivacijske komponente na izlaz $y(t)$. Običan P regulator ne dostiže željeno stanje sustava, PI dostiže, ali najprije oscilira oko njega, dok PID s derivacijskom komponentom prigušuje osciliranje (ili ga u potpunosti eliminira kao na slici).

3.3.4. Diskretni PID regulator

Kod diskretnog PID regulatora koriste se diskretni trenutci t_k (ili samo indeks k) za upravljačke odluke, odnosno označu vremena. Sve varijable/funkcije dobivaju indeks "k" umjesto prijašnjeg "(t)".

Uz korak diskretizacije:

$$T = t_k - t_{k-1} \quad (3.7.)$$



Slika 3.4. Načelni utjecaj komponenata PID regulatora

sve formule za diskretni PID regulator u koraku k su (3.8.)-(3.11.)

$$e_k = y_{P,k} - y_k \quad (3.8.)$$

$$I_k = I_{k-1} + e_k \cdot T, \quad \text{uz } I_0 = 0 \quad (3.9.)$$

$$D_k = \frac{e_k - e_{k-1}}{T}, \quad \text{uz } e_0 = 0 \quad (3.10.)$$

$$r_k = K_P \cdot e_k + K_I \cdot I_k + K_D \cdot D_k \quad (3.11.)$$

Kontinuirani (teoretski) PID regulator ima problema s stabilnošću, tj. s odabirom prikladnih parametara K_P , K_I i K_D za postizanje želenog stabilnog načina upravljanja. Diskretni sustav dodatno muči sama diskretizacija i dodatni problemi zbog toga. U okviru ova prikaza PID regulatora ti se problemi ne razmatraju. Problem odabira i optimiranja parametara PID regulatora su složeni problemi koji su detaljnije obrađeni u drugoj literaturi. U okviru primjera koji slijede korištena je jednostavna metoda pokušaja i promašaja pri odabiru parametara obzirom da su za ponašanje sustava korišteni jednostavniji modeli.

Primjer 3.5. Regulacija brzine PID regulatorom

Razmotrimo diskretni regulator koji upravlja motorom u svrhu postizanja i održavanja željene brzine vozila. Neka upravljana veličina r_k (u trenutku t_k) koju regulator podešava bude promjena vrijednosti gasa g_k . Regulator upravlja samo preko r_k . Sve ostale formule i vrijednosti ovdje služe za simulaciju reakcije sustava, tj. sam PID regulator i dalje koristi iste formule (3.8.)-(3.11.). Neka se u simulaciji sustava promjena gasa računa prema (3.12.).

$$g_k = g_{k-1} + r_k, \quad g_k \in [0, 1] \quad (3.12.)$$

Prepostavimo jednostavan model sustava kod kojeg je moment koji stvara motor izravno proporcionalan trenutno postavljenom gasu te da je sila kojom taj motor djeluje na ubrzanje i usporenje izravno proporcionalna tom momentu, odnosno, trenutnom gasu prema (3.13.).

$$F_{m,k} = K_M \cdot g_k \quad (3.13.)$$

Promjeni brzine vozila neka se odupire sila koja se sastoji od nepromjenjivog dijela F_T i dijela koji je izravno proporcionalan kvadratu trenutne brzine v_k .

$$F_{O,k} = F_T + K_O \cdot v_k^2 \quad (3.14.)$$

Rezultantna sila F_k koja će ubrzavati ili usporavati vozilo ili održavati stalnu brzinu (kada je sila jednaka nuli) iskazana je formulom (3.15.) gdje je m masa vozila te a_k ubrzanje.

$$F_k = m \cdot a_k = F_{m,k} - F_{O,k} \quad (3.15.)$$

Obzirom da se radi o diskretnom sustavu, ubrzanje a_k se izračunava kao omjer promjene brzine i proteklog vremena te je konačna jednadžba gibanja za diskretni sustav (3.16.) gdje je T korak diskretizacije.

$$m \cdot \frac{v_k - v_{k-1}}{T} = K_m \cdot g_k - F_T - K_O \cdot v_k^2 \quad (3.16.)$$

Za usporedbu, jednadžba kontinuiranog sustava bi bila (3.17.).

$$m \cdot \frac{dv(t)}{dt} = K_m \cdot g(t) - F_T - K_O \cdot v^2(t) \quad (3.17.)$$

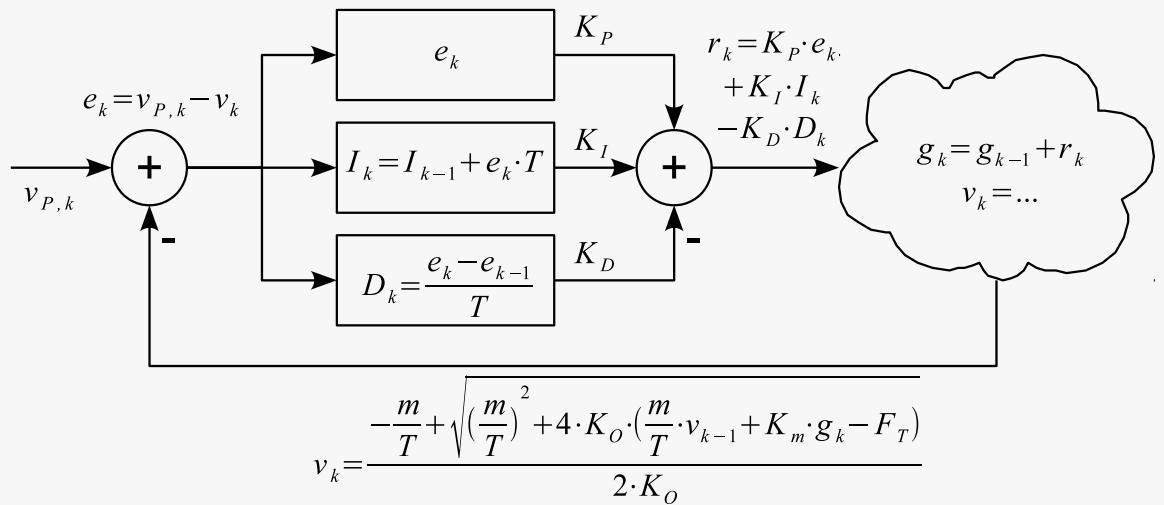
Iz kvadratne jednadžbe (3.16.), odnosno, u sređenom obliku (3.18.), može se izračunati brzina vozila u k -tom koraku, prema opisanom modelu vozila.

$$K_O \cdot v_k^2 + \frac{m}{T} \cdot v_k - \left(\frac{m}{T} \cdot v_{k-1} + K_m \cdot g_k - F_T \right) = 0 \quad (3.18.)$$

Obzirom da su vrijednosti K_O , m i T pozitivne i brzina v_k mora biti pozitivna (ne mijenja se smjer) te se računa prema (3.19.).

$$v_k = \frac{-\frac{m}{T} + \sqrt{\left(\frac{m}{T}\right)^2 + 4 \cdot K_O \cdot \left(\frac{m}{T} \cdot v_{k-1} + K_m \cdot g_k - F_T\right)}}{2 \cdot K_O} \quad (3.19.)$$

Sve navedene formule grafički su prikazane na slici 3.5.



Slika 3.5.

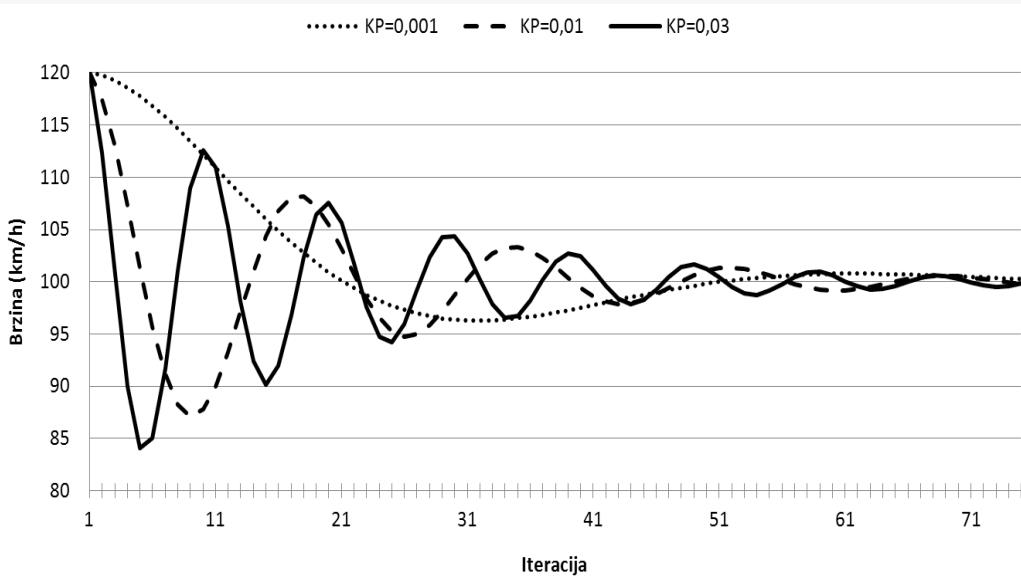
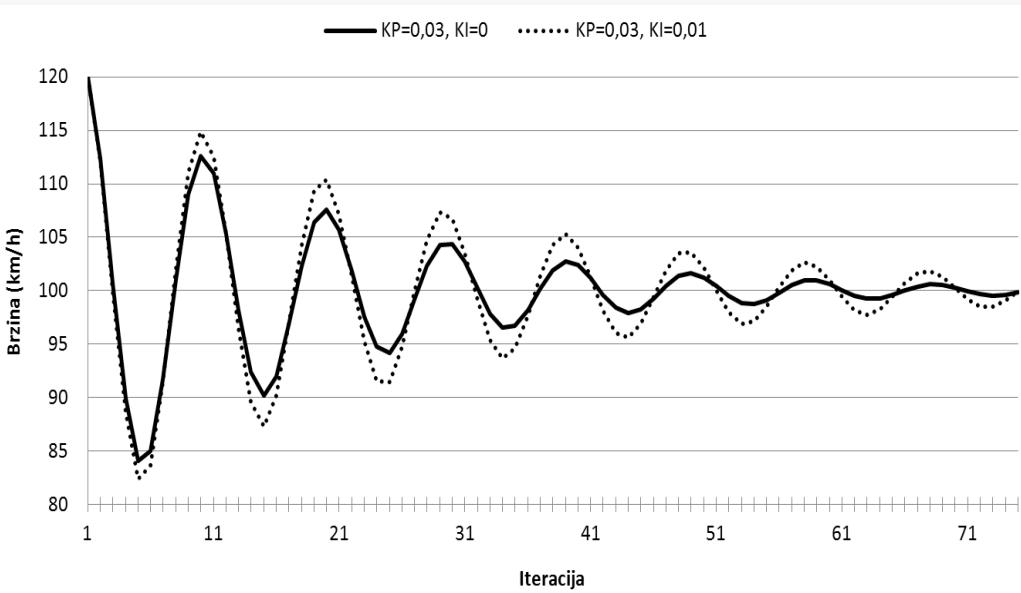
Obzirom da se derivacijska komponenta odupire promjeni, ona je na prethodnoj slici dodana s negativnim predznakom (tako da sam parametar K_D bude pozitivan). Parametri koji modeliraju sustav zadani su u tablici 3.1.

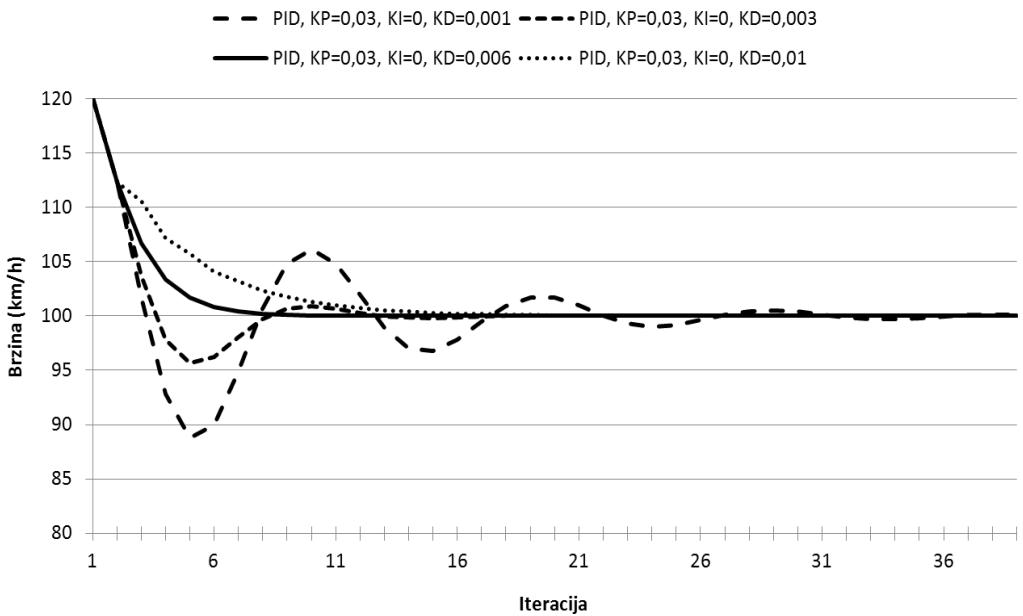
Tablica 3.1.

m (masa vozila)	K_m (snaga motora)	F_T (stalan otpor)	K_O (koef. otpora)
1000 [kg]	40000 [N]	5000 [N]	20 [kg/m]

Parametre K_P , K_I , K_D te T treba podešiti tako da upravljanje bude što bliže očekivanjima. Za korak upravljanja T odabrana je vrijednost 0,1 [s] za sve iduće prikazane regulatore, tj. za različite vrijednosti ostalih parametara.

Slike 3.6.–3.8. prikazuju rezultate dobivene P, PI i PD regulatorima brzine za sustav u kojem je početna brzina 120 [km/h] a postavljena (ciljana) brzina je 100 [km/h].

**Slika 3.6. P regulator****Slika 3.7. PI regulator**



Slika 3.8. PD regulator

P regulator za sve ispitane vrijednosti K_P (osim vrlo malih) dosta oscilira prije dostizanja ciljane brzine. Prihvatljiva odstupanja postižu se tek nakon 70-tak iteracija upravljanja (otprilike 7 sekundi stvarnog vremena).

Dodavanje integracijske komponente (PI regulator) se u ovom primjeru nije pokazalo dobro, barem za ispitane kombinacije K_P i K_I (heuristički odabrane i prilagođavane). Integracijski parametar samo povećava amplitudu oscilacija u ovom primjeru.

Derivacijska komponenta je znatno ubrzala stabilizaciju sustava na željenu brzinu, što se iz slike jasno vidi (PD regulator). Treba uočiti da je vremenska skala za PD regulator skoro dvostruko kraća od one na P i PI regulatorima, tj. da je stabilizacija dvostruko brža nego što to izgleda samom usporedbom grafova. Kada bi vremenska skala bila jednaka prethodnim grafikonima, teže bi se uočila razlika, tj. utjecaj parametra K_D . Kako K_D raste tako se oscilacije prigušuju. Za prikazani sustav oscilacije su potpuno ugušene za $K_D = 0,006$. Daljnjim povećanjem parametra ublažava se postupak postizanja ciljanog stanja (brzine), što u ovom primjeru i ima smisla jer je u protivnom prijelaz s 120 na 100 km/h prenagli (za manje od sekunde!).

3.4. Upravljanje korištenjem neizrazite logike

Sustavi kojima se upravlja mogu biti vrlo složeni. Sukladno tome će biti složena i funkcija kojom treba proračunati naredbe i vrijednosti koje se šalju u okolinu. Ponekad se zbog složenosti procesa kojim se upravlja, funkcija za određivanje izlaznih vrijednosti (naredbi) niti ne da egzaktno iskazati ili funkcija koju koristimo vrlo vjerojatno nije zadovoljavajuća zbog mnogih utjecaja koji nisu u nju uključeni te je teško njihov utjecaj iskazati u obliku formule. Čak i kada je funkcija poznata, njena složenost može zahtijevati korištenje složenijeg procesora umjesto planiranog jednostavnog mikroupravljača. Moguće rješenje za takve sustave jest korištenje neizrazitog zaključivanja (engl. *fuzzy logic*).

Neizrazitost se odražava u:

1. neizrazitom svrstavanju ulaznih vrijednosti u određene skupove
2. korištenje pravila za određivanje skupova kamo pripada izlazna vrijednost
3. postupak izračunavanja izlazne vrijednosti.

Pri definiranju skupova za ulazne i izlazne vrijednosti kao i za zaključivanja koristi se heuristika koja je vrlo blizu čovjekovu načinu razmišljanja i njegovu načinu upravljanja. Pritom se mogu i donekle kršiti stroga matematička pravila (npr. da suma svih vjerojatnosti bude jednaka jedan).

Primjerice, u nekom sustavu tekućina temperature 35 stupnjeva može pripadati u skup TOPLA s vjerojatnošću 0,5 te skupu HLADNA s vjerojatnošću 0,25, dok za skup VRUĆA vjerojatnost iznosi 0. Drugim riječima, ta temperatura ne spada samo u jedan skup, već u više njih s različitom vjerojatnošću. Stoga se i odluka o vrijednosti koju upravljač mora izračunati ne donosi samo na temelju jednog skupa, već korištenjem svih, ali uzimajući u obzir vjerojatnosti pripadnosti (koja je često zadana grafom).

Osim definicija vjerojatnosti pripadnosti za ulazne varijable, za sustav upravljanja moraju se definirati i *pravila zaključivanja* oblika:

$$p_i : \text{ako } (ulaz_a \in U_x) \text{ tada } (izlaz_c \in I_p) \quad (3.20.)$$

$$p_j : \text{ako } (ulaz_a \in U_x) \text{ i } (ulaz_b \in U_y) \text{ tada } (izlaz_c \in I_r) \quad (3.21.)$$

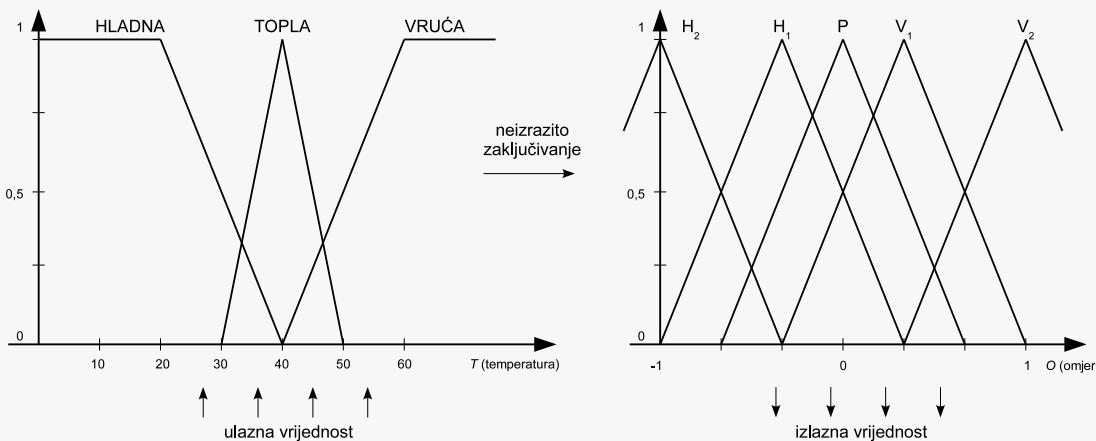
$$p_k : \text{ako } (ulaz_a \in U_x) \text{ ili } (ulaz_b \in U_y) \text{ tada } (izlaz_c \in I_q) \quad (3.22.)$$

Obzirom da ulazi pripadaju u skupove s određenom vjerojatnošću i za izlazne se vrijednosti moraju definirati skupovi i način preslikavanja vrijednosti u njih, u ovisnosti o vjerojatnosti pripadanja ulazne varijable skupu definiranome u pravilu. Jedan od mogućih postupaka neizrazitog zaključivanja – MIN_MAX je prikazan u nastavku u sklopu dva primjera.

Primjer 3.6. Upravljanje temperaturom tekućine

Prepostavimo sustav upravljanja omjerom tople i hladne vode (nepoznatih i promjenjivih temperatura) radi postizanja željene mješavine vode zadane temperature. Iako je ovo jednostavan primjer za koji bi se možda moglo naći i jednostavnije rješenje i sam prikaz neizrazitog zaključivanja je ilustrativan.

Neka se jednim senzorom mjeri temperatura mješavine vode T na osnovu čega upravljač podešava omjer miješanja da bi se postigla ciljana temperatura $T_P = 40^\circ\text{C}$. Neka su skupovi u koje se (s različitim vjerojatnostima) smješta ulazna temperatura T te izlazni skupovi za podešavanje omjera miješanja O prikazani slikom 3.9.



Slika 3.9.

Ulagana temperatura može pripadati skupovima HLADNA, TOPLA i VRUĆA s odgovarajućim vjerojatnostima. Npr. za $T_1=10^\circ\text{C}$ vjerojatnost pripadnosti skupu HLADNA je 1, tj. $\mu(10^\circ\text{C}, \text{HLADNA})=1$; za $T_2=32^\circ\text{C}$ vjerojatnost pripadnosti skupu HLADNA je $\mu(32^\circ\text{C}, \text{HLADNA})=0,4$ te vjerojatnost pripadnosti skupu TOPLA je $\mu(32^\circ\text{C}, \text{TOPLA})=0,2$.

Vrijednost izlazne varijable O svrstava se u skupove od H_2 do V_2 prema slici, tako da varijabla može poprimiti vrijednosti od -1 (kada se propušta samo hladna voda) do 1 (kada se propušta samo vruća voda).

Preslikavanje ulazne vrijednosti preko pripadnosti skupovima (s izračunatom vjerojatnošću) na izlazne skupove i određivanje izlazne vrijednosti radi se preko određenih neizrazitih postupaka. Dio tih postupaka su pravila zaključivanja.

Za zadani primjer pravila zaključivanja bi mogla biti:

- $$\begin{aligned} p_1 &: \text{ako } (T \in \text{HLADNA}) \text{ tada } (O \in V_2) \\ p_2 &: \text{ako } (T \in \text{HLADNA}) \text{ i } (T \in \text{TOPLA}) \text{ tada } (O \in V_1) \\ p_3 &: \text{ako } (T \in \text{TOPLA}) \text{ tada } (O \in P) \\ p_4 &: \text{ako } (T \in \text{TOPLA}) \text{ i } (T \in \text{VRUĆA}) \text{ tada } (O \in H_1) \\ p_5 &: \text{ako } (T \in \text{VRUĆA}) \text{ tada } (O \in H_2) \end{aligned} \quad (3.23.)$$

Pravila p_2 i p_4 mogu biti suvišna (zbog pravila p_1 , p_3 i p_5), ali su ipak uzeta u proračunima zbog prikaza postupka neizrazitog zaključivanja u slučaju složenih predikata.

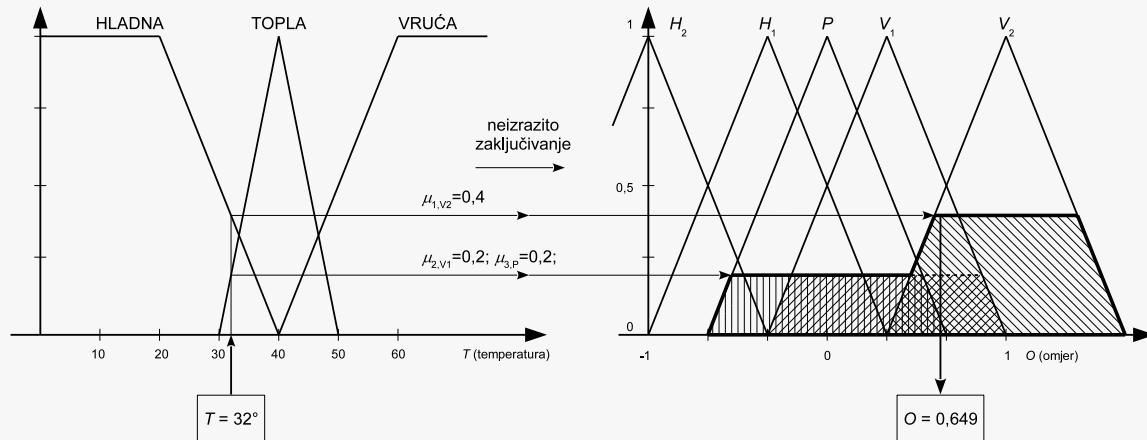
Često korišteni postupak neizrazitog zaključivanja jest MIN_MAX kod kojeg se pravila s predikatima povezanim operatom "i" izračunavaju primjenom minimuma, dok se operator "ili" zamjenjuje maksimumom.

Neka ulazna temperatura za koju želimo izračunati izlaz (omjer) iznosi $T=32^\circ\text{C}$. Tada se vjerojatnosti pripadnosti mogu izračunati i iznose: $\mu(32^\circ\text{C}, \text{HLADNA})=0,4$, $\mu(32^\circ\text{C}, \text{TOPLA})=0,4$ te $\mu(32^\circ\text{C}, \text{VRUĆA})=0$. Taj se postupak često naziva pretvaranje *izrazite vrijednosti* (skalara, fizičke veličine) u *neizrazitu vrijednost* (engl. *fuzzification*).

Primjenom prethodnih pravila dobivaju se vrijednosti:

$$\begin{aligned} p_1 &\Rightarrow \mu_{1,V2}(T) = \min\{0, 4\} = 0,4 \\ p_2 &\Rightarrow \mu_{2,V1}(T) = \min\{0, 4; 0, 2\} = 0,2 \\ p_3 &\Rightarrow \mu_{3,P}(T) = \min\{0, 2\} = 0,2 \\ p_4 &\Rightarrow \mu_{4,H1}(T) = \min\{0, 2; 0\} = 0 \\ p_5 &\Rightarrow \mu_{5,H2}(T) = \min\{0\} = 0 \end{aligned} \quad (3.24.)$$

Svaka se dobivena vrijednost vjerojatnosti pripadnosti (μ) odgovarajućem izlaznom skupu ($H_2 - V_2$), bilježi u tom skupu kao dio površine skupa ispod izračunate vrijednosti vjerojatnosti, prema slici 3.10.



Slika 3.10.

Rezultirajuća površina nastaje spajanjem pojedinačnih. Centroid površine (po izlaznoj vrijabli) je izlazna vrijednost varijable O . Formalno se ona može prikazati integralom:

$$O = \frac{\int_{o_{min}}^{o_{max}} u \cdot f(u) \, du}{\int_{o_{min}}^{o_{max}} f(u) \, du} \quad (3.25.)$$

gdje su o_{min} i o_{max} granice centroida po O osi (horizontalnoj), a $f(u)$ je funkcija gornje granice površine. Postupak se naziva *izračun izrazitih vrijednosti* (engl. *defuzzyfication*).

Umjesto integrala, obzirom da se rezultirajuća površina može rastaviti na trapez i paralelogram, svaki sa svojom površinom P_i i težištem t_i , ukupno težište se može jednostavnije izračunati prema:

$$O = \frac{\sum_i t_i \cdot P_i}{\sum_i P_i} \quad (3.26.)$$

Zadanom metodom izračunata izlazna vrijednost iznosi $O = 0,649$.

Prethodni proračun se ponekad može i pojednostaviti zanemarujući preklapanja. Tada se težište pojedine površine (trapeza) poklapa s težištem skupa, tj. za izlazni skup P je $t_1 = 0$, za V_1 je $t_2 = 1/3$ te za V_2 $t_3 = 1$. Tako izračunate površine te težišta rezultiraju s vrijednošću $O = 0,55$.

Daljnijim pojednostavljenjem, uz pretpostavku da je pojedina površina izravno proporcionalna izračunatoj vjerojatnosti μ_i , izlaz bi se mogao računati prema:

$$O = \frac{\sum_i t_i \cdot \mu_i}{\sum_i \mu_i} \quad (3.27.)$$

što je značajno jednostavnije. U ovom slučaju ovaj postupak daje vrijednost $O = 0,583$. Razlika u odnosu na "pravu" postoji u oba slučaja, ali obzirom da se i tako radi o neizrazitom odlučivanju koje sadrži puno heuristike, mogao bi se i ovaj način primijeniti i za njega optimirati parametri. Navedeni se način, tj. prema formuli (3.27.) koristi u idućem primjeru.

Ovi primjeri (kao i svi navedeni u ovom priručniku) su navedeni s razlogom prikaza mogućnosti pojedinih postupaka. Stoga i nisu detaljno razrađeni i analizirani (optimirani). Primjerice, poнаšanje gornjeg upravljača u stvarnim sustavima bi možda bilo neprikladno jer nije napravljena analiza njegova rada niti u simuliranom okruženju gdje bi se mogla dobiti dinamička svojstva sustava (vrijeme se uopće nije razmatralo, a ono je neophodno radi podešavanja upravljanja u stvarnim sustavima). Zato nije dobro donositi neke općenite predodžbe kako treba izgledati upravljač zasnovan na neizrazitom zaključivanju samo na temelju navedenih primjera.

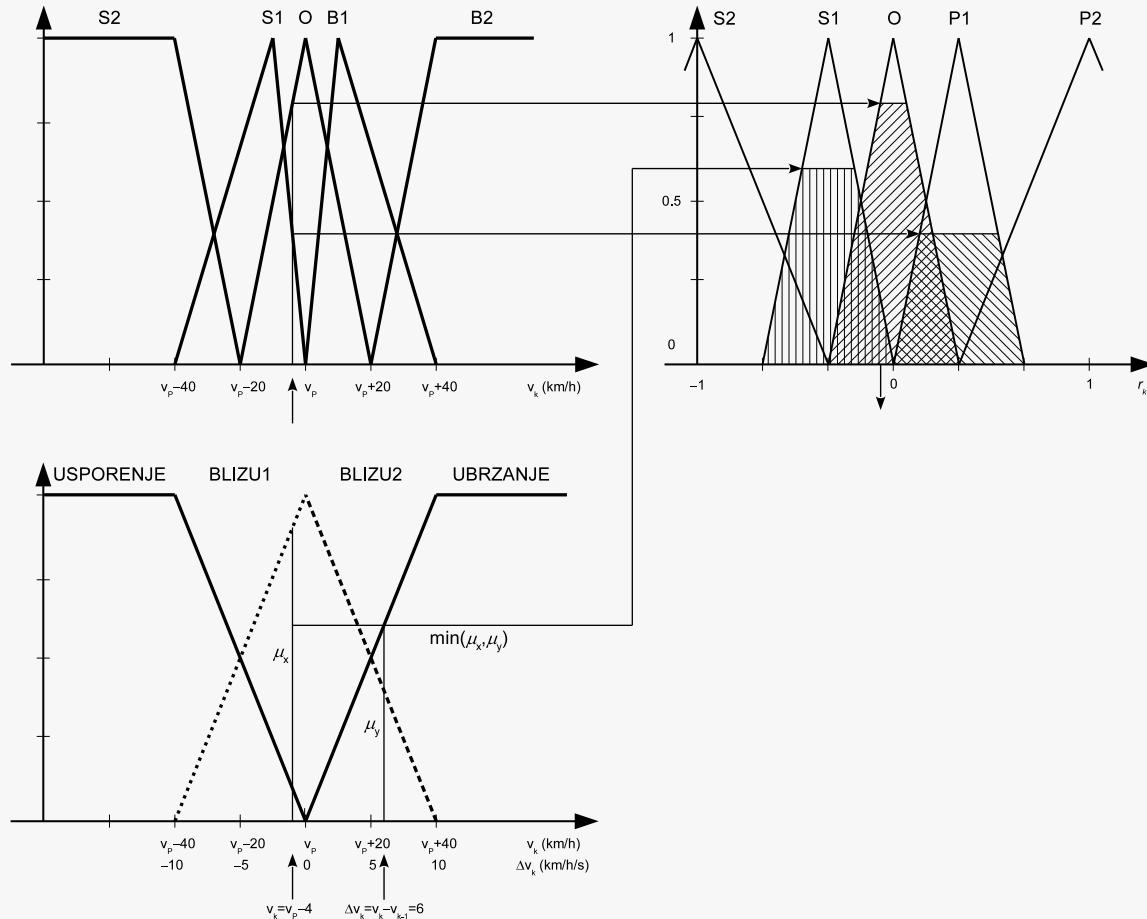
Problemi upravljanja koje smo susretali kod PID regulatora mogu biti prisutni i ovdje ako je upravljač loše projektiran. Primjerice, zbog neke vanjske smetnje možda upravljač ne bi mogao dovesti sustav u željeno stanje (kao i P regulator). Kada bi to bio slučaj onda bi trebalo ili drukčije postaviti ulazne skupove ili dodati neke u blizinu željenog stanja koji će imati snage dodatno gurnuti sustav prema željenom stanju. Uobičajeno se upravljači zasnovani na neizrazitom zaključivanju sastoje od više ulaznih skupova: pet, sedam ili i više.

Problem stabilnosti koji smo kod PID regulatora rješavali D komponentom se kod ovakvog upravljanja neizrazitom logikom često može rješiti samo dobrim postavkama ulaznih i izlaz-

nih skupova. Kada to u nekim slučajevima ne bi bilo tako, mogli bi dodati pravila i skupove kojima bi rješavali samo taj problem. Pogledajmo skicu primjera koji koristi ovakve heurističke, dodatne zahvate nad upravljačem zasnovanome na neizrazitoj logici.

Primjer 3.7. Regulacija brzine neizrazitim regulatorom

Na isti problem kao i u primjeru 3.5. mogla bi se primijeniti neizrazita logika za upravljanje. Skica tog upravljanja je u nastavku. Neka se za upravljanje koristi trenutna brzina v_k i promjena te brzine Δv_k kao ulazi te promjena gasa r_k kao izlaz, uz preslikavanja definirana slikom 3.11.



Slika 3.11.

Skup pravila zaključivanja definirana su formulama (3.28.)

- $p_1 : \text{ako } (v_k \in S2) \text{ tada } (r_k \in P2)$
 - $p_2 : \text{ako } (v_k \in S1) \text{ tada } (r_k \in P1)$
 - $p_3 : \text{ako } (v_k \in O) \text{ tada } (r_k \in O)$
 - $p_4 : \text{ako } (v_k \in B1) \text{ tada } (r_k \in S1)$
 - $p_5 : \text{ako } (v_k \in B2) \text{ tada } (r_k \in S2)$
 - $p_6 : \text{ako } (v_k \in \text{BLIZU1}) \text{ i } (\Delta v_k \in \text{UBRZANJE}) \text{ tada } (r_k \in S1)$
 - $p_7 : \text{ako } (v_k \in \text{BLIZU2}) \text{ i } (\Delta v_k \in \text{USPORENJE}) \text{ tada } (r_k \in P1)$
- (3.28.)

Zadnja dva pravila služe da uspore promjenu sustava kada je pomak prema željenoj vrijednosti prebrz. Prebrz pomak prema željenoj vrijednosti, a kada se sustav već nalazi blizu

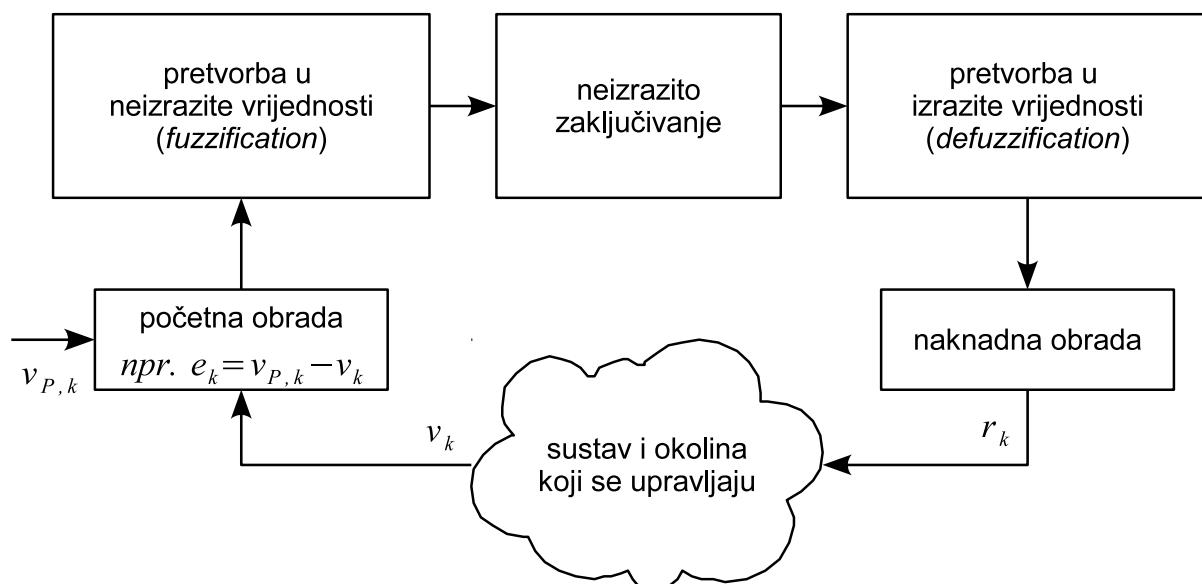
nje, najčešće će rezulirati premašenje željene vrijednosti i daljnji pomak u istom smjeru ali dalje od željene vrijednosti. Navedena pravila su zamišljena kao prigušenje tog premašaja, tj. pomoći u prigušenju oscilacija.

Na slici 3.11. je prikazan i primjer rada regulatora kada su ulazne vrijednosti $v_k = v_P - 4$ (v_P je željena brzina) i $\Delta v_k = 6$. Iz primjera je vidljivo da iako željena vrijednost nije dostignuta, zbog prebrzog primicanja željenoj vrijednosti izlaz (gas) se smanjuje, i to upravo zbog pravila p_6 koje je baš zbog toga dodano u sustav.

Svojstva neizrazitog upravljanja

Neizrazito upravljanje ima svojih prednosti i nedostataka. Najveće su mu prednosti u korištenju pravila koja su čovjeku razumljiva i koja bi on koristio u sličnim situacijama. Neki se složeni sustavi mogu relativno jednostavno upravljati korištenjem aproksimacije i heuristike. Osnovni problem može biti, kao i kod PID regulatora podešavanje parametara, pravila te postupaka zaključivanja (pretvorbe izrazito-neizrazito i obratno za izlazne vrijednosti).

Cjeloviti postupak upravljanja korištenjem neizrazitog zaključivanja može se prikazati slikom 3.12.



Slika 3.12. Upravljanje postupkom neizrazitog zaključivanja

Ponekad treba dodatno pripremiti (obraditi) ulazne vrijednosti, kao npr. izračunavanje odstupanja (na temelju kojeg se proračun obavlja), pretvorba jedinica, skaliranje i slično. Slični postupci mogu biti potrebni i na izlazu, u postupku naknadne obrade.

Projektiranje sustava prepostavlja određivanje skupova za ulazne i izlazne vrijednosti, određivanje pravila zaključivanja te načina zaključivanja (npr. MIN_MAX postupak).

Upravljanje neizrazitom logikom je samo dotaknuto u ovim materijalima. Prije bilo kakvog korištenja ovog načina upravljanja potrebno ga je podrobnije proučiti, primjerice iz literature posvećene tom području.

Pitanja za vježbu 3

1. Struktura programa se odabire prema problemu koji se rješava. Navesti nekoliko uobičajenih struktura programa pogodnih za razne primjene u SRSV-ima te njihova svojstva.
▷
2. Navesti prednosti i nedostatke sustava koji imaju samo prekidni podsustav i podsustav za upravljanje vremenom (trenutno vrijeme, alarmi) u kontekstu SRSV-a.
3. Koji elementi operacijskog sustava su potrebni pojedinim strukturama programa?
4. Zašto se svugdje ne koristi puni operacijski sustav sa svim svojim elementima (operacijska jama koje omogućuje za lakšu izradu programa)?
5. Što su to regulacijski zadaci?
6. Kod upravljanja kontinuiranih sustava radi se diskretizacija u vremenu. Kako odabrati korak diskretizacije?
7. Opisati upravljanje bez povratne veze i s povratnom vezom korištenjem formula.
8. Usporediti upravljanje bez povratne veze i upravljanje koje koristi povratnu vezu (prednosti, nedostaci, primjena).
9. Opisati uporabivost PID regulatora. ▷
10. Od kojih se komponenata PID regulator sastoji? ▷
11. Navesti uporabivost neizrazite logike u kontekstu upravljanja sustavima. Koja su dobra a koja loša svojstva neizrazite logike u tom okruženju?
12. Koja je posebna prednost korištenja neizrazitog upravljača (*fuzzy logic controller*) naspram ostalih oblika upravljanja? ▷
13. Opisati postupak pretvaranja ulazne (izrazite) vrijednosti u neizrazitu (u sustavu upravljanja zasnovanom na neizrazitoj logici) (fuzzification). ▷
14. Kako se koriste pravila zaključivanja (u sustavu upravljanja zasnovanom na neizrazitoj logici) korištenjem MIN_MAX postupka u dobivanju izrazite izlazne vrijednosti. ▷
15. Za neki sustav koji je upravljan mikroupravljačem treba napraviti upravljački program za upravljanje dvaju procesa. Prvi treba obavljati svakih 37 ms (jednom u svakom intervalu od 37 ms) pozivom obrada_1(), a drugi svakih 47 ms (jednom u svakom intervalu od 47 ms) pozivom obrada_2(). Za mjerjenje vremena na raspolaganju je 64-bitovno brojilo na adresi BROJILO koje odbrojava frekvencijom od 1 kHz (i za vrijeme rada sustava neće se premašiti najveća vrijednost koja stane u to brojilo). Obzirom na moguća dulja trajanja obrada (do 5 ms) paziti da se greška u vremenu ne povećava (da se periode ne povećavaju; preporuka koristiti "apsolutna vremena")!
16. Neko računalo treba upravljati s nekoliko aktivnosti. Za aktivnost A treba jednom u 30 ms pozvati a(), za B treba jednom u 50 ms pozvati b() te za aktivnost C jednom u 100 ms pozvati c(). Jednom započeta obrada aktivnosti C (poziv c()) ne smije se prekidati (a() i b() se mogu prekidati). Aktivnost D treba aktivirati iz obrade prekida IRQ=39. Na raspolaganju stoji sučelje: trenutno_vrijeme() (vrijeme u milisekundama), registriraj_prekid(irq, handler), zabrani_prekidanje(), omoguci_prekidanje(). Napisati program za upravljanje (uz prepostavku da funkcije a(), b(), c() i d() postoje), uključujući inicijalizaciju.
17. Prikazati (skicirati) ostvarenje upravljanja uređajem koji na ruci mjeri otkucaje srca, prikazuje trenutno stanje (često, barem svakih pola sekunde) te vibracijom dojavlja

kada je broj otkucaja veći od N . Uredaj (ugrađeni sustav) posjeduje jednostavan mikrokontroler koji je opremljen i jednim 8-bitovnim brojilom koje odbrojava frekvencijom od 1 kHz (ali ne izaziva prekide). Posebnim se sklopom detektira otkucaj srca koji to tada dojavi procesoru prekidnim signalom. Ispis trenutnog pulsa (broja otkucaja u minuti) ispisati funkcijom `ispisi(int puls)` (funkcija postoji!). Vibracija (kratka) se aktivira pozivom funkcije `vibra()`.

18. Neko upravljačko računalo upravlja proizvodnim procesom. Upravljanje se sastoji od reakcije na vanjske događaje ("obradu prekida") te na periodičke akcije. Tri su izvora događaja P_1 , P_2 i P_3 za koje se trebaju pozvati funkcije `p1()`, `p2()` i `p3()` (respektivno: za $P_1 > p1()$, itd.). Dvije su periodičke akcije: prvu `pera_1()` treba obavljati svakih 500 ms (jednom u 500 ms) te drugu `pera_2()` svakih 100 ms (prvi put u 0. ms a kasnije za svakih točno 100 ms ili što je moguće bliže tom trenutku). Riješiti problem upravljanja uz pretpostavku: a) (2) upravljačko računalo ima operacijski sustav (i sva sučelja koja uz to idu) te b) (3) upravljačko računalo nema operacijski sustav, tj. upravljanje ostvariti u upravljačkoj petlji (neka se pojave događaja očitavaju u adresama $ZP1$, $ZP2$ i $ZP3$ – jedinica označava pojavu)
19. PID regulator zadan je parametrima $K_P = 0,1$, $K_I = 0,3$, $K_D = 0,01$ te korakom integracije $T = 0,1$. Ako se reakcija okoline može simulirati formulom $y_{k+1} = y_k + 2 \cdot r_k$ napraviti dva koraka integracije (izračunati y_{k+1} i y_{k+2}). Trenutno stanje sustava je $y_k = 10$, a željeno stanje $y_P = 15$, uz $I_{k-1} = 2$ te $e_{k-1} = 1$.
20. Sustav koji je upravljan PID regulatorom u nekom je trenutku (koraku k) u stanju: $y_P = 100$ (željeno stanje), $y_k = 95$ (trenutno stanje), $e_{k-1} = 7$ (greška u prethodnom koraku) te $I_{k-1} = 5$ (suma grešaka do ovog koraka). Izračunati izlaz iz regulatora r_k , uz: $K_P = 0,5$, $K_I = 0,1$, $K_D = 0,05$ te $T = 0,1$ (T je korak u kojem regulator daje izlaz).
21. U sustavu u kojem se koristi neizrazito zaključivanje postoji pravilo:
 p_1 : ako ($a \in A_1$) ili ($b \in B_2$) tada ($c \in C_3$)
Opisati primjenu tog pravila, ukoliko se koristi MIN_MAX načelo.
22. Održavanje brzine treba ostvariti upravljačem zasnovanim na neizrazitu upravljanju. Pretpostaviti da su brzine s kojima sustav treba raditi u rangu 10-130 km/h, da motor (koji može i kočiti) može raditi s 7 različitim diskretnih snaga (-3, -2, -1, 0, 1, 2, 3). Pokazati rad projektiranog sustava (izračun izlaza) ako je ulazna brzina 50 km/h a postavljena (željena) 80 km/h.
23. Neki kontroler treba na temelju ulaznih vrijednosti a i b izračunati izlaznu i . Poznato je da je izlaz (otprilike) obrnuto proporcionalan ulazu a te (otprilike) proporcionalan ulazu b . Za najmanju vrijednost od a (A_{MIN}) i najveću vrijednost od b (B_{MAX}) izlaz treba biti I_{MIN} i obratno, za $a = A_{MAX}$ i $b = B_{MIN}$ izlaz treba biti I_{MAX} . Ostvariti kontroler korištenjem neizrazitog upravljanja.
24. Postupak grijanja nečega upravlja se sustavom zasnovanom na neizrazitoj logici. Ulaz u sustav je trenutna temperatura (i protok vremena). Idealno zagrijavanje je linearno s promjenom od ΔT stupnjeva u minuti. Kada je promjena manja tada treba povećati snagu grijачa, a kada je veća smanjiti. Skicirati upravljački program koji poziva funkciju `promjena_snage(dt)`, a koja će neizrazitom logikom izračunati kako treba promijeniti snagu. U upravljačkom programu napraviti potrebnu početnu i naknadnu obradu (prije i poslije poziva `promjena_snage(dt)`). Za samo neizrazito zaključivanje navesti grafove s funkcijama pripadnosti i izlaznih vrijednosti, pravila zaključivanja te na primjeru ulaza pokazati (grafički) određivanje izlazne vrijednosti (operaciju `promjena_snage(dt)` ne pisati pseudokodom već neizrazitim zaključivanjem kako je zadano).

4. Raspoređivanje poslova

4.1. Uvod

Računalni sustavi koriste se za upravljanje nekim procesima (iz okoline) ili izvođenjem nekih korisniku potrebnih poslova (zadaća). Upravljanje s više poslova može se programski ostvariti na nekoliko načina. Ukoliko se upravljanje može svesti na obradu događaja koje izazivaju vanjski procesi, tada se sva upravljačka logika može raspodijeliti u procedure koje obrađuju te događaje – u prekidne potprograme. Ako upravljanje traži periodičko očitanje stanja procesa te reakciju na očitanja, upravljanje se može ugraditi u obradu prekida sata ili izvesti programski, na način da se očitavaju svi upravljeni procesi iz istoga kôda (periodičko “prozivanje”).

Navedeni načini upravljanja pogodni su samo za jednostavnije sustave. U složenijim bi sustavima navedeni postupci postali suviše složeni, teško ostvarivi i vrlo teški za održavanje, otkrivanje grešaka, nadograđivanje i slično. Logika upravljanja koja se mora ugraditi u druge podsustave ili “zajedničke” upravljačke programe postaje suviše složena i glavni je problem ostvarenja takvih načina upravljanja.

Kao i gotovo svugdje, osnovni način rješavanja problema složenosti ‘podijeli i vladaj’ može se koristiti i za podjelu složenog upravljačkog programa u manje dijelove – *zadatke* (engl. *task*), od kojih se svaki zadatak brine za jedan vanjski proces. Upravljačka logika zasebnog zadatka je sva na jednom mjestu, samim time je i razumljivija što značajno olakšava i otkrivanje grešaka, ažuriranje i nadograđivanje. Dodavanje novih komponenti u sustav, kao i micanje nekih nepotrebnih, također je jednostavnije.

Odvajanje nezavisnih poslova upravljanja u zasebne zadatke – koji time postaju nezavisne jedinice izvođenja – *dretve*, samo je jedan od razloga potrebe za *višezadaćnim sustavom* (engl. *multitasking*), tj. *višedretvenim sustavom*. Najznačajnija korist višedretvenosti jest u učinkovitosti korištenja sustava. Naime, kad u sustavu postoji više dretvi, jedna dretva može čekati na dovršetak ulazno-izlazne (UI) operacije nad nekom napravom (koju je prije toga zadala – naprava “radi”), druga dretva može korisiti drugu napravu, treći procesor, četvrti drugi procesor kada se radi o višeprocesorskom sustavu¹

Potreba za višedretvenošću može proizlaziti i iz samo jednog posla. U takvom slučaju neki od mogućih razloga mogu biti:

- intenzivni računalni problemi koji se mogu rastaviti na (bar djelomično) neovisne dijelove (zadatke) mogu pomoći višedretvenosti bolje iskoristiti dostupne procesorske jedinice sustava;
- kada jedan posao zahtijeva proračune, ali i korištenje UI naprava, tada se elementi posla mogu odijeliti u one komponente koje vrše proračune i druge koje čekaju na UI naprave (primjerice u video igri neke dretve mogu biti specijalizirane za proračune fizike, mehanike i za grafički podsustav, a druge prihvataju ulaze korisnika, šalju i primaju pakete s mreže i slično);
- kada različite dretve upravljaju različitim elementima sustava koji traže različite načine upravljanja (npr. kontinuirani, periodički, sporadični poslovi, poslovi različita značaja i slično);

¹U ovom se tekstu pod pojmom višeprocesorskog sustava podrazumijeva svaki sustav koji ima više procesorskih jedinica, što obuhvaća i višestruke i mnogostrukе procesore (engl. *multicore*, *manycore*).

- kada je potrebno ostvariti asinkrono upravljanje događajima/zahtjevima (primjerice početna dretva Web poslužitelja svaki novi zahtjev stavlja u red koji obrađuju zasebne (radne) dretve tako da početna dretva može i dalje nesmetano prihvati nove zahtjeve bez obzira o trajanju obrade već pristiglih).

Osnovna ideja višedretvenosti jest u paralelnom radu više dretvi (stvarno paralelnom ili prividno paralelnom, korištenjem načela naizmjenična rada i podjeli procesorskog vremena). Osnovna pretpostavka ovakva načina rada jest postojanje više dretvi. One mogu biti dio istog posla ili pak svaka raditi svoj posao. Operacijski sustav treba omogućiti dinamičko pokretanje poslova – stvaranje novih dretvi, bilo preko sučelja prema korisniku, bilo preko sučelja prema programima koji sami stvaraju dodatne dretve.

Stanje nekog sustava određuje skup dretvi koje se u njemu nalaze i koje obavljaju svoje poslove. Neke od dretvi mogu biti u stanju čekanja (blokirane/zaustavljene dretve), tj. prije nego što nastave s radom moraju pričekati na neki događaj (akciju druge dretve, vanjske naprave ili protok vremena). Primjerice, dretva može čekati na naredbu korisnika, dohvati podatka s diska, istek prethodno zadano vremenskog intervala i slično. Ostale dretve su "pripravne" i mogu se izvoditi na procesoru.

Korisnik pokreće poslove/programe pri čemu operacijski sustav stvara dretve koje ih obavljaju. S korisničke strane to je dovoljno – korisnik u daljem radu treba pratiti rad programa i po potrebi upravljati njegovim radom (unositi tražene podatke, pokretati željene operacije i slično). Po dovršetku posla ili po naredbi korisnika program se zaustavlja – dotične dretve se zaustavljaju i miču iz sustava.

Zadaća operacijskog sustava jest da upravlja dretvama, da im daje procesorsko vrijeme kad im je potrebno i kad je njihov red u odnosu na ostale dretve, da ih miče "na stranu" kada ne trebaju procesorsko vrijeme (kad čekaju na nešto), da ih stvara i dodaje u sustav na zahtjev drugih dretvi i korisnika te da ih miče iz sustava pri završetku njihova rada.

U jednoprocesorskim sustavima u jednom trenutku može biti *aktivna* (izvoditi se na procesoru) samo jedna dretva. Sve ostale moraju čekati da ta završi ili da ju operacijski sustav makne s procesora. U sustavu s N procesora u istom trenutku može biti aktivno N dretvi. Način odabira aktivne dretve (ili N aktivnih) naziva se *raspoređivanje dretvi*.

Pojedina dretva može pripadati sustavu, obavljati potrebne operacije za sam sustav (primjerice obrada prekida, upravljanje ostalim zadacima i sredstvima sustava). Takvu dretvu nazivamo *dretvom sustava* ili *jezgrinom dretvom*. S druge strane dretva može pripadati nekom zadatku koji upravlja određenim vanjskim procesima, ili pak dretva može pripadati programu koji je korisnik pokrenuo, a koji za njega obavlja korisne operacije. Ovakvu dretvu nazivamo *korisničkom dretvom*.

Upravljanje dretvama treba omogućiti izvođenje svih dretvi u sustavu. Način i redoslijed izvođenja treba biti uskladen s važnošću posla koje dretve obavljaju – *prioritetom dretvi*. Korištenje sredstava sustava treba kontrolirati ali i omogućiti njihovo korištenje od strane svih dretvi. U sredstva sustava spadaju: procesorsko vrijeme, spremnički prostor, UI naprave i ostala sredstva sustava, primjerice sinkronizacijski i drugi mehanizmi i objekti.

Upravljanje dretvama mora uzeti u obzir posebnosti pojedinih dretvi. U sustavima za rad u stvarnom vremenu dretve često imaju vremenske okvire u kojima trebaju napraviti zadani posao. Takva ograničenja treba uzeti u obzir prilikom upravljanja dretvama.

Problem raspoređivanja sredstava javlja se u gotovo svakom sustavu. Kako su gotovo svi sustavi upravljeni računalima to postaje problem i u području računarstva. Raspoređivanje sredstava tako da se zadovolje sva ograničenja uz istovremeno postizanje očekivane učinkovitosti ili kvalitete može biti vrlo složen problem. U nastavku se razmatra samo problem *raspoređivanja dretvi*, odnosno, raspoređivanje procesorskog vremena po dretvama sustava.

Raznolikost računalnih sustava zahtijeva razne postupke raspoređivanja te se iz istog razloga

novi postupci neprestano istražuju i usavršavaju. Za različite probleme najčešće se koriste i različiti postupci raspoređivanja. Raspoređivač koji savršeno odgovara jednom tipu problema kod drugog može dati vrlo loše rezultate. Povećanje procesorske snage najčešće daje dovoljno dobre rezultate. Međutim, takvo rješenje poskupljuje gotovi proizvod te ga treba uzeti kao zadnje, ako se problem ne može riješiti drugim postupcima. Ponekad ni zamjena jačim računalom nije dovoljna.

Da bi se bolje razumjelo potrebe raznih sustava i načina raspoređivanja, u okviru ovog poglavlja detaljnije se prikazuju teoretske podloge raspoređivanja poslova, odnosno, razmatra se raspoređivanje zadataka. U idućem, 5. poglavlju, prikazuju se načini raspoređivanja dretvi koji se zaista koriste u operacijskim sustavima.

4.2. Podjela poslova na zadatke

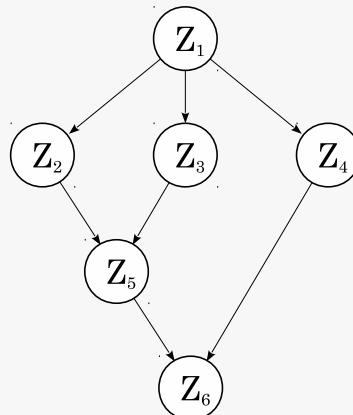
Određeni *posao* (ili zadaća, u engleskoj terminologiji koriste se termini *task* i *job*) može se iz raznih razloga podijeliti na zadatke, kao što je već i opisano u prethodnom poglavlju. Ipak, jedan od najznačajnijih razloga jest radi povećanja učinkovitosti na višeprocesorskim sustavima koji sve više prodiru u razne računalne sustave u koje svakako spadaju u SRSV-i.

Podjelu posla u zadatke² može biti vrlo teško napraviti. Naime, najčešće nam je poznato kojim redoslijedom treba napraviti potrebne operacije da bi se dobio očekivani rezultat zadanog posla. Koje operacije mogu ići paralelno ili koje se operacije mogu razbiti na paralelne zadatke vrlo često nije vidljivo.

Pri postupku podjele poslova u zadatke se često koriste grafičke metode, a da bi se tada vidjelo što ima smisla zaista odvojiti i izvoditi paralelno, a što ne. Česti način prikaza je usmjereni graf u kojemu su čvorovi zadaci dok usmjerene veze predstavljaju uređenje tj. potreban slijed izvođenja zadataka. Graf treba sadržavati samo zaista potrebne veze jer one ograničavaju paralelnost u radu, odnosno, za svaku vezu će biti potrebno ugraditi sinkronizaciju između dva čvora koje veza povezuje.

Primjer 4.1. Primjer podjele posla na zadatke

Neka se posao P može početno podijeliti na skup slijednih zadataka $Z_1 \rightarrow Z_2 \rightarrow Z_3 \rightarrow Z_4 \rightarrow Z_5 \rightarrow Z_6$. U idućem koraku analize neka je ustanovljeno da zadatak Z_1 mora biti gotov prije pokretanja zadataka Z_2 , Z_3 i Z_4 , zadatak Z_5 mora čekati dovršetak Z_2 i Z_3 dok Z_6 zadatke Z_4 i Z_5 . Navedene ovisnosti mogu se prikazati grafički prema slici 4.1.

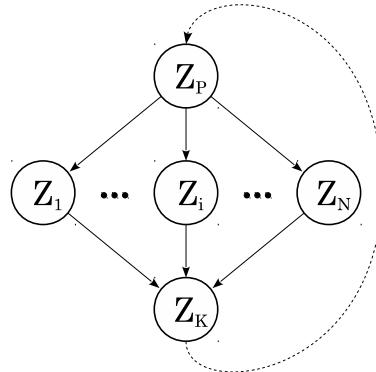


Slika 4.1. Međuvisnost zadataka prikazana grafom

²Neki izvori koriste pojам *zadatak* umjesto *posla* te tada govore o podjeli zadatka na podzadatke.

Zadaci koji se mogu paralelno izvoditi su $\{Z_2; Z_3; Z_4\}$ te $\{Z_4; Z_5\}$. Ako su operacije koje rade navedeni zadaci složeniji, tj. dulje traju, ovakva podjela na zadatke (dretve u izvođenju) ima smisla i pridonijet će bržem završetku posla na višeprocesorskom sustavu. Ako je neki zadatak i sam jako složen, možda ima smisla i njega dodatno rastaviti na manje cjeline (podzadatke) koji se mogu dijelom paralelno izvoditi.

Uobičajena podjela posla na zadatke sastoji se od početnog zadatka koji priprema okolinu (strukture podataka i slično) te pokreće N dretvi koje paralelno obavljaju dio posla. Po njihovu svršetku rezultati se prikupljaju i posao je gotov ili se sve ponavlja s drugim podacima. Slika 4.2. prikazuje takav rastav posla na zadatke.



Slika 4.2. Uobičajena podjela posla na paralelne zadatke

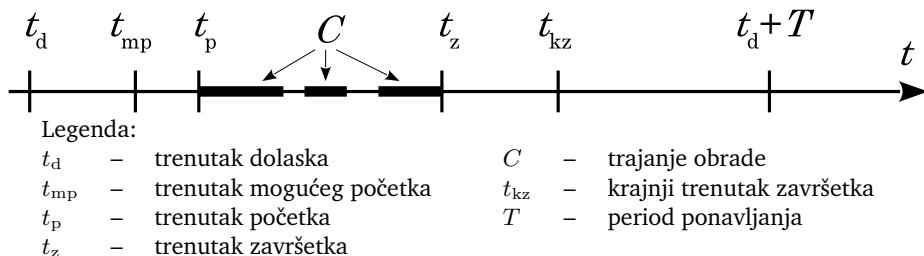
Podjela posla na zadatke treba uzeti u obzir i arhitekturu višeprocesorskog sustava na kojem će se program izvoditi. Sa stanovišta učinkovitosti nije svejedno ima li sustav jedan procesor s više jezgri koje dijele priručni spremnik ili više procesora koji ne dijele priručne spremnike ali dijele zajednički spremnik (engl. *symmetric multiprocessor*) ili više procesora koji u grupama (čvorovima) dijele spremnik (engl. *non-uniform machine architecture*) ili se za proračune koriste posebni uređaji, primjerice procesori na grafičkim karticama (engl. *graphics processing unit – GPU*).

Nadalje, treba uzeti u obzir dodatne operacije koje treba ugraditi u same zadatke radi sinkronizacije zadataka kao i za razmjenu podatka. Stoga treba vrlo dobro poznavati i rad operacijskog sustava te mehanizama sinkronizacije i komunikacije, tj. koliko one same mogu potrošiti procesorskog vremenena te je li se takva podjela uopće isplati. U nekim slučajevima to dodatno vrijeme (engl. *overhead*) može i premašiti vrijeme korisnog izvođenja te uzrokovati da višedreveni program (paralelni) bude sporiji od jednodretvenog (slijednog).

Međuvisnosti među zadacima istog posla treba pri programiranju ostvariti sinkronizacijskim mehanizmima. Više o sinkronizacijskim mehanizmima u 6. poglavljju. U nastavku, u razmatranju raspoređivanja zadataka privremeno će se zanemariti međuvisnost zadataka. Naime, u postupku raspoređivanja razmatraju se samo pripravnih zadataci, a to su zasigurno nezavisni zadaci jer su zavisni blokirani na sinkronizacijskom mehanizmu.

4.3. Vremenska svojstva zadatka

Zadaci mogu biti periodički s unaprijed zadanim vremenima ponavljanja ili sporadični, kao primjerice reakcija na pojavu prekida. I jedni i drugi zadaci imaju neka zajednička svojstva za čije se promatranje mogu definirati određeni vremenski trenuci bitni za njihovo odvijanje. Slika 4.3. prikazuje svojstvene trenutke pri obavljanju jednog zadatka.



Slika 4.3. Svojstveni trenuci u životnom ciklusu jednog zadatka

Zadatak se pojavljuje u sustavu u trenutku t_d .

Obzirom na operacije koje obavlja može biti zahtjevano da svoj posao završi do trenutka označenog sa t_{kz} – krajnji trenutak završetka. Ako zadatak ne završi do t_{kz} sustav snosi posljedice. Stoga je vrlo bitno da se koristi raspoređivanje zadataka koje će svakom zadatku omogućiti završetak do svog t_{kz} . Radi pojedostavljenja ali i radi toga što se tako najčešće i koristi u stvarnim sustavima u idućim razmatranjima promatraju se samo periodički zadaci te se za t_{kz} uzima trenutak idućeg pojavljivanja zadatka.

Iz raznih razloga izvođenje zadatka ne može krenuti u samom trenutku pojave zadatka (npr. radi kućanskih poslova) već tek nakon nekog vremena. Neka trenutak kada izvođenje zadatka može započeti označimo sa t_{mp} (koje može biti i jednak trenutku dolaska – uglavnom tako pretpostavljamo).

Trenutak kada raspoređivač odabere zadatak te kada on zaista započinje svoje izvođenje označeno je sa t_p . Dodatna odgoda može biti prouzrokovana dovršetkom prioritetnijih poslova.

Za vrijeme izvođenja zadatka on može biti i prekidan drugim zadacima većeg prioriteta ili obradom prekida.

U trenutku t_z zadatak završava s pridijeljenim poslom te nestaje iz sustava ili se privremeno zaustavlja do sljedećeg pojavljivanja (ako je zadatak periodički).

Ukoliko je posao zadatka periodički, njegovo sljedeće aktiviranje očekuje se u trenutku $t_d + T$ (T je period ponavljanja zadatka).

Vremenska uređenost događaja sa slike 4.3. mora se očuvati ukoliko se želi stabilan rad sustava, tj. za navedene vremenske trenutke mora vrijediti uređenje (4.1.).

$$t_d \leq t_{mp} \leq t_p < t_z \leq t_{kz} \quad (4.1.)$$

Zadaća operacijskog sustava, ili nekog drugog rješenja koje se koristi umjesto njega, jest omogućiti održavanje navedenog vremenskog uređenja za sve zadatke u sustavu. Drugim riječima, zadaci moraju biti upravljeni tako da svoje izvođenje počinju najranije u trenutku t_{mp} te da sav posao obave najkasnije do t_{kz} . Postoje razni postupci kako to postići, od kojih su neki vrlo jednostavni dok su drugi vrlo složeni. Odabir postupka ovisi o uporabi. Ponegdje će i oni najjednostavniji biti sasvim dovoljni dok će drugdje biti potrebni drugi, različite složenosti.

4.4. Postupci raspoređivanja

Postupci raspoređivanja zadatka (ili dretvi) određuju načine odabira zadatka i njihovo pridjevljivanje procesorima. Postoje mnogi postupci raspoređivanja zadatka, mnogi samo teoretski, ali i mnogi koji se koriste. Postupci se mogu podijeliti prema nekoliko kriterija od kojih su u nastavku podrobnijsi razmotreni sljedeće podjele:

- prema vremenu i načinu donošenja odluka raspoređivanja – prije pokretanja ili za vrijeme izvođenja
- prema postupanju sa zadacima u izvođenju – pušta ih se do kraja ili ih se može i prekinuti
- prema cilnjom računalnom sustavu – jednoprocesorskom, simetričnom i asimetričnom više-procesorskom, spletu (engl. *cluster*), grozdu (engl. *grid*), ...

Poslovi koje izvode spletovi i grozdovi računala često su zahtjevni neinteraktivni proračunski poslovi. Takvim se računalima poslovi "šalju" posebnim naredbama (sučeljem). Raspoređivač poslova u takvim sustavima nastoji učinkovito iskoristiti procesorske mogućnosti, ali i ostala sredstva koja su kritična za takve poslove, posebice spremnički prostor. Stoga se takvi poslovi najčešće raspoređuju pojedinačno, tj. izvode se jedan po jedan od početka do kraja. Jedino u slučaju da jedan posao ne može iskoristiti svu dostupnu procesnu moć (ne posjeduje dovoljnu paralelnost – nema dovoljno dretvi), tada se može izvoditi više poslova. Ovakva raspoređivanja skupnih poslova (engl. *batch jobs*) neće se razmatrati u nastavku jer se za njihovo raspoređivanje koriste uobičajene metode optimiranja (npr. kako iz jedne ploče dobiti što više predmeta zadanih oblika).

4.4.1. Statički i dinamički postupci raspoređivanja

Prema trenutku donošenja odluke ili određivanju vrijednosti koje će se koristiti u postupku raspoređivanja, postupci se mogu podijeliti na:

- statičke i
- dinamičke.

4.4.1.1. Statički postupci raspoređivanja

Kod statičkih postupaka raspoređivanja odluke se donose prije pokretanja sustava (engl. *offline*). Sustav je najprije potrebno analizirati, donijeti odluke o raspoređivanju ili pridjeli obilježja zadacima koja će se koristiti pri raspoređivanju. Pridijeljena obilježja se ne mijenjaju tijekom rada kod statičkih postupaka raspoređivanja.

Odluke mogu biti predstavljene slijedom izvođenja zadatka: koji zadatak u kojem trenutku pokrenuti, kada zadatak zamijeniti drugim i slično, uvažavajući vremenska svojstva zadatka – kad se može koji zadatak pokrenuti te kada mora biti gotov.

Slijed odluka može biti programiran u dodatni upravljački program koji će pokretati, zaustavljati i izmjenjivati zadatke na procesorima, npr. kao na primjeru 4.2.

Postupak ugradnje odluka u raspoređivač jest vrlo učinkovit, ali i poprilično zahtjevan za ostvarenje. Naime, potrebno je jako dobro poznavati sve zadatke, njihova vremenska svojstva te na osnovu toga uobličiti upravljanje. Vrlo često takvi detaljni podaci nisu dostupni.

Drugi pristup statičkog raspoređivanja svodi se na utvrđivanju važnosti operacija koje obavljaju pojedini zadaci. Svakom se zadatku T_i najprije definira prioritet u obliku broja od P_{MIN} do P_{MAX} . Raspoređivač pri donošenju svojih odluka uspoređuje zadatke T_i i T_j prema njihovim prioritetima p_i i p_j . Zadatak većeg prioriteta prije se odabire za izvođenje³.

³Neki sustavi koriste obrnuti pristup značaja same vrijednosti prioriteta – kod njih manja brojčana vrijednost predstavlja veći značaj – veći prioritet.

Primjer 4.2. Primjer statičkog raspoređivanja

Odluke raspoređivanja predstavljene slijedom pokretanja zadataka:

```

...
ne radi ništa do t1
pokreni zadatak Z1
kad Z1 završi ne radi ništa do t2
pokreni Z2
ako je Z2 završio prije t3 tada
    pričekaj do t3
u t = t3:
    ako Z2 još nije gotov tada
        prekini i pohrani kontekst od Z2
    pokreni Z3
kada Z3 završi, nastavi sa Z2, ako nije bio gotov prije
ne radi ništa do t4
...

```

Raspoređivanje u takvom sustavu svodi se na odabir zadatka najvećeg prioriteta. Zadaci se pokreću nezavisno, primjerice kao reakcija na događaje u sustavu (obrade vanjskih događaja mehanizmom prekida) ili periodički. U svakom takvom trenutku poziva se raspoređivač koji među svim pripravnim zadacima uzima onaj najvećeg prioriteta. Kada se u sustavu u nekom trenutku nađe više zadataka istog najvećeg prioriteta, odabir jednog od njih je ili proizvoljan ili se uvode dodatni kriteriji. Npr. ako se u sustavu nalaze tri zadatka s prioritetima $p_1 = 10$, $p_2 = 20$ te $p_3 = 20$, odabir između p_2 i p_3 je ili proizvoljan ili se može koristiti vrijeme kad su ti poslovi došli u red ili neki drugi dodatni kriterij.

Iako se samo raspoređivanje provodi za vrijeme izvođenja (engl. *online*), ovaj pristup ipak svrstavamo u statičke jer su prioriteti statički pridjeljeni prije pokretanja sustava zadataka – zadaci ne mijenjaju svojstva koja se razmatraju pri raspoređivanju tijekom svog izvođenja.

Pridjeljivanje prioriteta zadataka te raspoređivanje prema prioritetu je jedno od najjednostavnijih načina raspoređivanja te se najčešće koristi u praksi (u operacijskim sustavima).

4.4.1.2. Dinamički postupci raspoređivanja

Suprotno statičkim postupcima, kod dinamičkih postupaka raspoređivanja svojstva zadataka koja se koriste pri raspoređivanju se mijenjaju tijekom vremena.

U notaciji prioriteta moglo bi se reći da se prioriteti zadataka mijenjaju vremenom i drugim događajima. Primjerice, dok je trenutak do kada neka operacija mora biti obavljena još daleko u budućnosti, prioritet pripadnog zadataka može biti mali. Kako vrijeme prolazi prioritet tog zadataka raste. Drugi primjer uključuje zadatak trenutno malog prioriteta (prema nekim dodatnim kriterijima), ali koji koristi određenu napravu. Kad se u sustavu pojavi zadatak visokog prioriteta koji treba tu istu napravu, ali koju se ne može prethodnom zadataku oduzeti dok se operacija nad napravom ne obavi do kraja, prethodnom se zadataku može privremeno povisiti prioritet. Tako će zadatak početno manjeg prioriteta dobiti povećanje prioriteta koje će mu omogućiti da prije obavi svoje operacije nad napravom te po njenom oslobođenju omogući nastavak rada zadataka visokog prioriteta.

Dinamički postupci su u pravilu značajno složeniji od statičkih i zahtijevaju dodatne informacije o zadatacima. Dinamički raspoređivači često i sami zahtijevaju nezanemarivo procesorsko vrijeme za izračun podataka na kojima se temelje njihove odluke – traže više “kućanskih poslova” od statičkih (imaju veći *overhead*) te se stoga i znatno rjeđe koriste u stvarnim sustavima. Kada se koriste neki dinamički elementi zadataka onda se ipak odabiru postupci smanjene složenosti za raspoređivanja – teži se $O(1)$ ili u donekle prihvatljivoj $O(\log n)$ složenošću.

Postupci raspoređivanja koji su primjenjivi na sustave zadataka u kojima neki zadaci ne mijenjaju parametre raspoređivanja (npr. prioritet) dok drugi zadaci mijenjaju (npr. prioritet raste protokom vremena) u nekim se literaturama nazivaju *mješovitim raspoređivanjem*.

4.4.2. Postupci raspoređivanja prilagođeni računalu

Postupak raspoređivanja koji je prilagođen jednoprocesorskim sustavima ne mora biti prikladan za višeprocesorske, i obratno. Nadalje, sami višeprocesorski sustavi mogu se podijeliti na simetrične, asimetrične, sa zajedničkim spremnikom, s raspodijeljenim spremnikom (prema procesorskim grupama) i slično (primjerice uvažavajući dijeljene priručne i lokalne djelove spremnika).

Optimalnost višeprocesorskog raspoređivanja je vrlo složena problematika te se, iako postoje mnoge studije na tu temu, ipak uzimaju jednostavniji suboptimalni postupci koji svoje odluke donose brzo.

Dodatni problemi kod višeprocesorskih raspoređivanja mogu nastati zbog istovremenog korištenja zajedničkih sredstava. Stoga i zadatke za takve sustave treba dodatno pripremiti ugradnjom potrebnih sinkronizacijskih i komunikacijskih mehanizama.

Iako su postupci raspoređivanja složeniji za višeprocesorske sustave te kod njih ima više kućanskih poslova i sinkronizacije, procesne mogućnosti takvi sustavi pružaju su vrlo često neophodni za ostvarenje upravljanja u SRSV-ima. Tome u prilog idu i noviji procesori koji su većinom višestruki (engl. *multicore*), čak i za ugrađene primjene.

U idućim poglavljima najprije se razmatra raspoređivanje za jednoprocesorska računala obzirom da je kod njih teže zadovoljiti vremenska ograničenja svih zadataka. Tek potom se razmatra raspoređivanje za višeprocesorska računala.

4.4.3. Prekidljivost zadataka

Postupci raspoređivanja mogu se podijeliti i prema kriteriju mogućnosti prekidanja zadataka u izvođenju na prekidljive (engl. *preemptive*) i neprekidljive (engl. *non-preemptive*). Naime, u nekim primjenama se jednom započeti zadatak ne smije prekidati dok ne završi sa svojim operacijama. Tek po završetku može se odabrat drugi. Drugi postupci dozvoljavaju prekidanje izvođenja zadataka radi obavljanja važnijih operacija, kao što su obrade prekida ili izvođenje zadataka većeg prioriteta.

Prednosti neprekidljivih postupaka jest u smanjenim kućanskim poslovima uzrokovanim spremanjem konteksta prekinutog zadataka i obnavljanjem konteksta novog posla koji prekida pretходni. Ipak, prednosti prekidljivih jesu u znatno bržem odzivu prema hitnim događajima i bitnijim zadacima. Stoga se uglavnom koriste postupci raspoređivanja koji prepostavljaju prekidljive zadatke te će oni biti detaljnije razmatrani u nastavku.

I u sustavima s prekidljivima raspoređivačima može se ostvariti poneki neprekidljivi zadatak – dovoljno je da taj zadatak u željenom neprekidivom odsječku zabrani daljnja prekidanja te se ni sam raspoređivač neće moći pokrenuti dok mu sam zadatak to ponovno ne dozvoli. Taj se pristup koristi u SRSV-ima kod kojih postoje vrlo kritični zadaci čije su operacije bitnije od svega ostalog (čak i od obrada prekida).

4.5. Jednoprocesorsko raspoređivanje

Sa stanovišta SRSV-a, osnovni problem kod jednoprocesorskog raspoređivanja jest kako zadatke rasporediti tako da se zadovolje vremenski zahtjevi svih zadataka (prema slici 4.3.), tj. kako ostvariti da sustav bude "rasporediv". Problem rasporedivosti je detaljnije analiziran u ovom odlomku.

4.5.1. Jednoprocesorsko statičko raspoređivanje

Statičko raspoređivanje zahtijeva analizu sustava zadataka prije pokretanja sustava. Statičko raspoređivanje koje će se razmatrati u nastavku ograničeno je na korištenje statički pridjeljenih prioriteta pojedinim zadacima, prema kriterijima postavljenim pri analizi sustava. Pitanje je kako pridijeliti prioritete zadacima a da raspoređivanje bude zadovoljavajuće, barem za većinu slučajeva.

Radi jednostavnosti analize rasporedivosti, u nastavku se koriste pretpostavke prikazane definicijom 4.1.

Definicija 4.1. Početne pretpostavke za raspoređivanje zadataka

1. Razmatra se konačan skup (sustav) nezavisnih zadataka:

$$\mathcal{S} = \{\mathcal{T}_i = \{C_i, T_i\} \mid i \in \{1..N\}\} \quad (4.2.)$$

2. Svi zadaci su periodički – pojavljuju se u sustavu u trenucima $k \cdot T_i$, gdje je $k \in \mathbb{N}$ i T_i označava periodu zadatka \mathcal{T}_i .
3. Pojedina pojava zadatka \mathcal{T}_i u trenutku $k \cdot T_i$ se označava s τ_i^k .
4. Zadatak τ_i^k može započeti sa svojim izvođenjem odmah po pojavi.
5. Izvođenje zadatka τ_i^k u svakoj periodi ($\forall k$) traje jednako i iznosi C_i .
6. Krajnji trenutak za završetak rada (obilježen sa t_{kz}) zadatka τ_i^k koji se pojavio u periodi k , u trenutku $k \cdot T_i$, je početak iduće periode: $t_{kz} = (k + 1) \cdot T_i$.
7. Zadatak τ_i^k se može prekidati u svom izvođenju.
8. Sustav \mathcal{S} je uređen tako da vrijedi: $T_1 < T_2 < \dots < T_N$, tj. manje indekse imaju zadaci s kraćim periodama.

U definiciji 4.1. se pretpostavlja da svi zadaci imaju različite periode ponavljanja. Kada to ne bi bilo tako, primjerice kada bi dva zadatka imala jednaku periodu, onda bi se oni u dalnjim razmatranjima mogli zamijeniti sa zamjenskim zadatkom čije je izvođenje suma izvođenja oba zadatka.

Neka se prvo razmotre granični slučajevi – kada se sustav zadataka može rasporediti uz zadovoljenje vremenskih uvjeta a kada ne.

4.5.1.1. Procesorska iskoristivost i izvodljivost raspoređivanja

Nikakav postupak raspoređivanja \mathcal{R} neće moći posložiti poslove ako sustav ima previše posla – više nego što može obaviti. Kako provjeriti da li sustav ima previše posla?

Neka je zadan sustav zadataka \mathcal{S} (prema definiciji 4.1.). Svaki se zadatak \mathcal{T}_i iz \mathcal{S} treba obaviti jednom unutar svake svoje periode. Unutar te periode zadatak \mathcal{T}_i treba C_i procesorskog vremena, odnosno, on stvara opterećenje od C_i/T_i prema sustavu (procesoru). Slično je i s ostalim zadacima. Ukupno opterećenje, koje nazivamo i procesorskom iskoristivošću, jest suma opterećenja pojedinih zadataka.

Definicija 4.2. Procesorska iskoristivost

Procesorska iskoristivost za sustav zadataka \mathcal{S} (prema definiciji 4.1.) računa se prema:

$$U_{\mathcal{S}} = \sum_{i=1}^N \frac{C_i}{T_i} \quad (4.3.)$$

Da bi sustav \mathcal{S} mogao biti izvodljiv na zadanom procesoru (za koja su zadana trajanja izvođenja zadataka C_i), procesorska iskoristivost mora biti manja ili jedaka jedan, prema definiciji 4.3.

Definicija 4.3. Nužni uvjet rasporedivosti

Sustav zadataka \mathcal{S} (prema definiciji 4.1.) zadovoljava nužni uvjet rasporedivosti ako vrijedi:

$$U_{\mathcal{S}} \leq 1 \quad (4.4.)$$

Definicija 4.3. (formula 4.4.) vrijedi općenito, za sve postupke raspoređivanja. Međutim, pojedini postupak raspoređivanja može imati i dodatne uvjete na sustav zadataka, a da bi on bio rasporediv tim postupkom.

Najveći problem za raspoređivača će biti kada se svi zadaci iz \mathcal{S} istovremeno pojave u sustavu, tj. kada se počeci njihovih perioda poklope. Takav trenutak se naziva "kritični slučaj" i on se razmatra u postupku provjere rasporedivosti sustava zadataka (on se koristi u svim idućim postupcima).

Definicija 4.4. Kritični slučaj

Za sustav zadataka \mathcal{S} (prema definiciji 4.1.) definira se "kritični slučaj" kao trenutak kada se poklope počeci perioda svih zadataka. Tada se u sustavu nalazi najveći broj zadataka koje procesor treba obaviti (prema redoslijedu koji određuje raspoređivač).

4.5.1.2. Određivanje prioriteta zadataka

Kako dodijeliti prioritete pojedinima zadacima iz sustava \mathcal{S} a da se rasporedivanjem prema prioritetu može rasporediti najviše takvih sustava? Prioritetni raspoređivač će u svakom trenutku odabratи zadatak najvećeg prioriteta, pa tako i u kritičnom slučaju. Primjer 4.3. prikazuje dva moguća načina dodjele prioriteta prema učestalosti pojave zadataka.

Primjer 4.3. Dodjela prioriteta prema učestalosti pojave

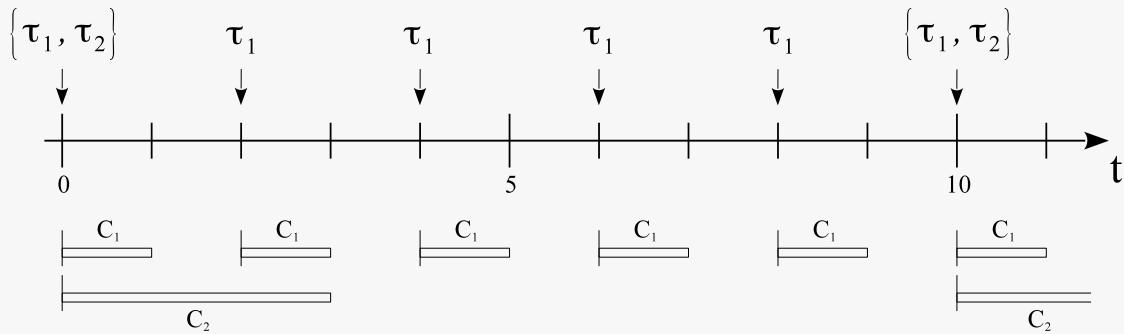
Zadan je sustav s dva zadatka $\mathcal{S} = \{\mathcal{T}_1, \mathcal{T}_2\}$. Neka se prvi zadatak javlja sa $T_1 = 2$ s te neka njegovo računanje traje $C_1 = 1$ s. Drugi zadatak neka se javlja rijeđe, sa $T_2 = 10$ s, ali traje dulje, $C_2 = 3$ s.

Provjerom nužnog uvjeta izvodljivosti rasporedivanja:

$$U_{\mathcal{S}} = \sum_{i=1}^N \frac{C_i}{T_i} = \frac{1}{2} + \frac{3}{10} = 0,5 + 0,3 = 0,8 \leq 1$$

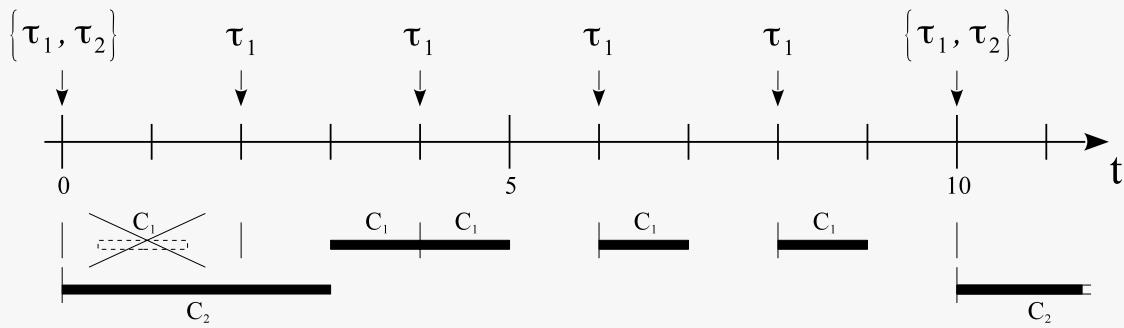
može se ustanoviti da je on zadovoljen. To ne osigurava da je sustav rasporediv korištenjem prioritetnog raspoređivača.

Slika 4.4. prikazuje zadani sustav, periode ponavljanja i potrebna vremena u svakoj periodi.



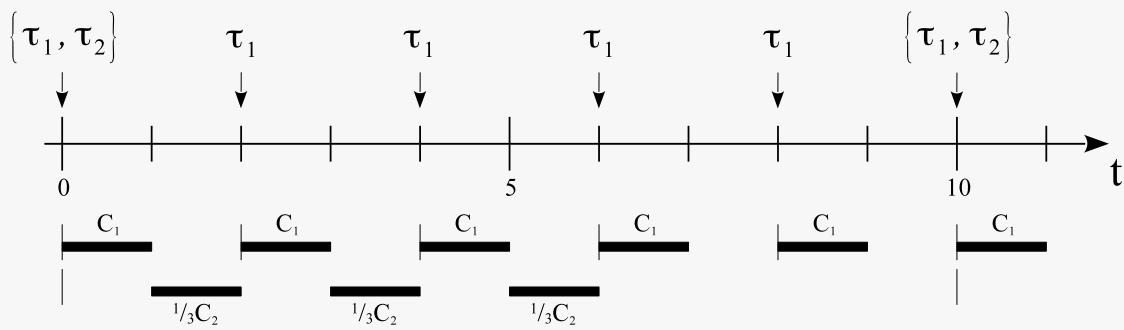
Slika 4.4. Sustav s dva zadatka

Prvi pokušaj dodjele prioriteta neka prioritete dodijeli prema frekvenciji pojavljivanja – zadatku koji se rjeđe javlja daje veći prioritet. Slika 4.5. prikazuje rad prioritetskog raspoređivača kod kojeg je zadatak T_2 dobio veći prioritet.



Slika 4.5. Veći prioritet rijedim zadacima

Iz slike je vidljivo da se prvo pojavljivanje zadatka T_1 nije stiglo izvesti u dozvoljenim granicama. Drugi način dodjeli prioriteta je obrnuti: zadatku koji se češće javlja daje se veći prioritet. Slika 4.6. prikazuje rad raspoređivača u tom slučaju.



Slika 4.6. Veći prioritet učestalijim zadacima

Raspoređivanje uz korištenje staticki dodijeljenih prioriteta postupkom “veći prioritet učestalijim zadacima” uspijeva rasporediti zadatke uz zadovoljavanje njihovih ograničenja.

Osim korištenja perioda zadatka za dodjelu prioriteta, kao što je to korišteno u primjeru 4.3., moglo bi se isprobati koristiti i trajanja. Primjerice, da zadatak s kraćim vremenom izvođenja dobiva veći prioritet. Treća je mogućnosti koristiti neki funkciju koja uzima i periode i trajanja.

Međutim, kada bi se provela temeljita analiza svih ostalih načina došlo bi se do zaključka da je

ipak duljina periode općenito najbolji odabir, tj. da zadacima koji imaju kraću periodu pojavljivanja – učestalije se javljaju, treba pridijeliti veći prioritet, a zadacima koji se rjeđe javljaju treba dati manji prioritet. Navedeni postupak pridjele prioriteta prema učestalosti pojavljivanja zadataka naziva se “mjera ponavljanja” (engl. *rate monotonic scheduling – RMS, rate monotonic priority assignment – RMPA*) i formalno je opisan definicijom 4.5.

Definicija 4.5. Mjera ponavljanja

Postupak pridjele prioriteta zadacima iz skupa \mathcal{S} (prema definiciji 4.1.) korištenjem mjere ponavljanja, zadacima $\{\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_N\}$ će dodijeliti prioritete $\{p_1, p_2, \dots, p_N\}$ (zadatku \mathcal{T}_1 prioritet p_1 , zadatku \mathcal{T}_2 prioritet p_2 , itd.) tako da vrijedi formula (4.5.).

$$p_1 > p_2 > \dots > p_N \quad (4.5.)$$

Raspoređivač koji raspoređuje prema prioritetu treba pozvati svaki puta kada se nešto promijeni u sustavu, tj. kada se pojavi neki zadatak ili kada neki zadatak završi s radom. U svakom trenutku rada raspoređivač će među svim zadacima koji čekaju na obradu za aktivni zadatak odabrati zadatak najvećeg prioriteta.

Mjera ponavljanja (ili “raspoređivanje prema mjeri ponavljanja”) je jedan od najjednostavnijih, ali i najčešće korištenih postupaka raspoređivanja u SRSV-ima.

4.5.1.3. Ograničenja pri raspoređivanju zadataka sa stalnim prioritetima

Raspoređivanje prema mjeri ponavljanja možda postavlja i dodatne uvjete na sustav zadataka, osim nužnog uvjeta prema definiciji 4.3. Prije detaljnije analize ograničenja slijedi nekoliko primjera grafičke provjere rasporedivosti sustava zadataka.

Definicija 4.6. Grafička provjera rasporedivosti

Sustav zadataka \mathcal{S} (prema definiciji 4.1.) bit će rasporediv odabranim postupkom raspoređivanja ako se grafičkom provjerom (simulacijom rada raspoređivača) u kritičnom slučaju potvrdi da su se svi zadaci koji su se pojavili u kritičnom slučaju $s = \{\tau_1^a, \tau_2^b, \dots, \tau_N^z\}$ stigli obaviti do svojih t_{kz} primjenom pravila postupka raspoređivanja, uzimajući u obzir i sve iduće pojave zadataka τ_i^{i+m} koje se zbivaju za to vrijeme (dok se svi iz s ne obave).

Grafički postupak je vrlo jednostavan za ručnu provjeru, ali samo u kratkim primjerima koji se koriste u ovom prikazu. Nad većim sustavima, grafički postupak postaje značajno složeniji – uz graf bilo bi potrebno uvesti dodatne strukture za praćenje stanja svih zadataka (takvi se veći skupovi zadataka neće razmatrati u nastavku).

Primjer 4.4. Grafička provjera rasporedivosti (1)

Zadan je sustav od tri zadataka s periodama i vremenima računanja prema:

$$\begin{aligned} \mathcal{T}_1 : \quad T_1 &= 5 \text{ ms}, & C_1 &= 2 \text{ ms} \\ \mathcal{T}_2 : \quad T_2 &= 15 \text{ ms}, & C_2 &= 5 \text{ ms} \\ \mathcal{T}_3 : \quad T_3 &= 25 \text{ ms}, & C_3 &= 5 \text{ ms} \end{aligned}$$

Provjera rasporedivosti:

a) Nužni uvjet:

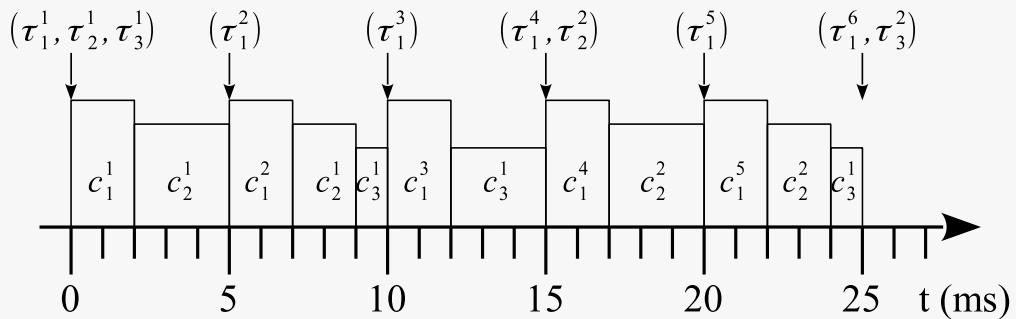
$$U = \sum_{i=1}^3 \frac{C_i}{T_i} = \frac{2}{5} + \frac{5}{15} + \frac{5}{25} = 0,933 < 1$$

Nužni uvjet je zadovoljen.

b) Grafička provjera u kritičnom slučaju

Slika 4.7. prikazuje grafički postupak provjere rasporedivosti. Na slici su označeni trenuci dolaska zadataka sa: τ_{ind}^{br} , gdje ind označava indeks zadatka, a br redni broj pojave tog zadatka.

Prioritetni rasporedivač uvijek odabire zadatak najvećeg prioriteta. Prema postupku mjere ponavljanja zadatak T_1 ima najveći prioritet obzirom da ima najkraću periodu.



Slika 4.7.

Iz slike je vidljivo da svi zadaci uspijevaju završiti svoje izvođenje prije početka svojeg idućeg perioda.

Zaključak: sustav je rasporediv prema postupku mjere ponavljanja.

Primjer 4.5. Grafička provjera rasporedivosti (2)

Zadan je sustav od tri zadatka s periodama i vremenima računanja prema:

$$\mathcal{T}_1 : T_1 = 10 \text{ ms}, \quad C_1 = 5 \text{ ms}$$

$$\mathcal{T}_2 : T_2 = 15 \text{ ms}, \quad C_2 = 5 \text{ ms}$$

$$\mathcal{T}_3 : T_3 = 20 \text{ ms}, \quad C_3 = 1 \text{ ms}$$

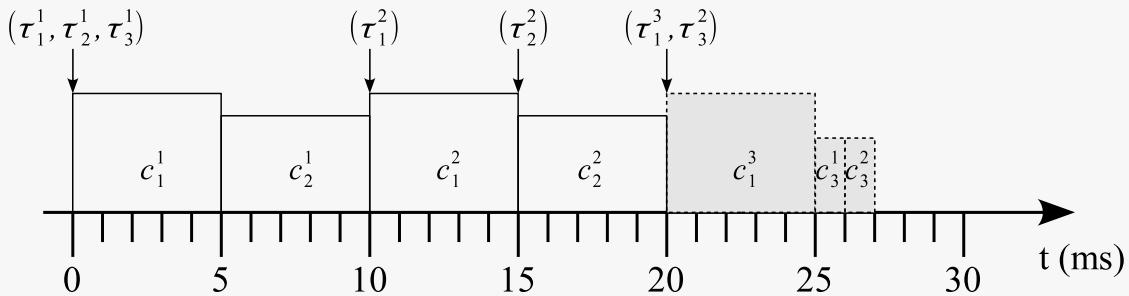
Provjera rasporedivosti:

a) Nužni uvjet:

$$U = \sum_{i=1}^3 \frac{C_i}{T_i} = \frac{5}{10} + \frac{5}{15} + \frac{1}{20} = 0,883 < 1$$

Nužni uvjet je zadovoljen.

b) Grafička provjera u kritičnom slučaju



Slika 4.8.

Iz slike 4.8. je vidljivo da zadatak T_3 ne stigne obaviti svoje prvo pojavljivanje τ_3^1 do svojeg idućeg pojavljivanja τ_3^2 . Time nije ispunjen uvjet dovršetka do svog t_{kz} te samim time sustav zadataka nije rasporediv prema postupku mjere ponavljanja. Otkriće da neki zadatak ne stiže obaviti svoje operacije do zadanih ograničenja zaustavlja provjere – ograničenja su narušena te sustav sigurno nije rasporediv.

Kada t_{kz} ne bi bio postavljen, onda bi obje pojave zadataka T_3 mogle biti obavljene nakon 25. ms, kada je procesor slobodan. Na slici 4.8. je taj dio prikazan zasivljeno (nije dio provjere rasporedivosti – on staje u 20. ms negativnim odgovorom).

Zaključak: sustav nije rasporediv prema postupku mjere ponavljanja.

Svaki periodički zadatak T_i (prema definiciji 4.1.) ima zadan svoj (eksplicitni) trenutak krajnjeg završetka t_{kz} u svakoj svojoj pojavi kao trenutak iduće pojave istog zadataka. Međutim, možda on mora završiti i prije ako u tom trenutku $((k+1) \cdot T_i)$ procesor izvodi zadatak većeg prioriteta.

Definicija 4.7. Implicitni krajnji trenutak završetka – D_i

Za zadatak T_i iz sustava zadataka \mathcal{S} (prema definiciji 4.1.) definira se *implicitni krajnji trenutak završetka* D_i (engl. *implicit deadline*) kao krajnji trenutak u kojem zadatak mora završiti, računajući od pojave tog zadataka u kritičnom slučaju (npr. u $t = 0$).

D_i može biti jednak t_{kz} , ali može biti i manji ako je vrijeme od D_i do t_{kz} popunjeno obradama prioritetnijih zadataka.

Kada je zadatak T_i rasporediv, tada je D_i ili jednak periodu ponavljanja zadataka T_i ili je manji.

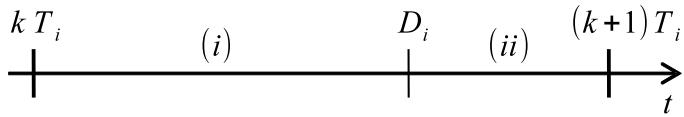
Kada je D_i manji od T_i , tada on može biti jedino jednak trenutku ponovne pojave nekog od prioritetnijih zadataka T_j u intervalu $(k \cdot T_i, (k+1) \cdot T_i)$ uz $j < i$. Takvi priritetniji zadaci u potpunosti koriste interval $[k \cdot T_i + D_i, (k+1) \cdot T_i]$ i ne ostavljaju prostora (vremena) za zadatak T_i .

D_i se traži među točkama raspoređivanja zadataka T_i .

Definicija 4.8. Točke raspoređivanja

U kontekstu razmatranja rasporedivosti sustava zadataka zadanog prema definiciji 4.1. promatranog u kritičnom slučaju, točke raspoređivanja D_i za zadatak T_i su trenuci idućih pojava zadataka iz skupa $\{\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_i\}$ (prioritetniji zadaci od T_i uz sam T_i) u intervalu $(k \cdot T_i, (k+1) \cdot T_i]$ (pojave τ_j^p uz $j \leq i$ te $k \cdot T_i < p \cdot T_j \leq (k+1) \cdot T_i$).

Slika 4.9. prikazuje odnos D_i u odnosu na početak i kraj periode zadatka T_i za $D_i < T_i$. Zadatak



Slika 4.9. D_i – implicitni t_{kz}

τ_i^k stiće će obaviti svoj posao do D_i ako se od početka periode, tj. od pojave zadatka τ_i^k do D_i (u intervalu (i) sa slike 4.9.) stignu obaviti svi prioritetniji zadaci i to onoliko puta koliko se javljaju u tom intervalu te uz to ostane dovoljno vremena da se i zadatak τ_i^k obavi do kraja.

Definicija 4.9. Opći kriterij rasporedivosti

Sustav zadataka \mathcal{S} (prema definiciji 4.1.) bit će rasporediv postupkom mjere ponavljanja ako za svaki zadatak T_i iz skupa \mathcal{S} postoji $D_i \in [0, T_i]$ koji u kritičnom slučaju zadovoljava uvjet (i)^a:

$$(i) : \sum_{j=1}^i \left\lceil \frac{D_i}{T_j} \right\rceil \cdot C_j \leq D_i \quad (4.6.)$$

D_i se odabire među točkama raspoređivanja zadatka T_i .

^aOperator $\lceil x \rceil$ vraća prvi cijeli broj jednak ili veći od x .

Numerički postupak određivanja rasporedivosti zadatka T_i svodi se na pronađazak jednog trenutka D_i među točkama rasporedivosti za koji je zadovoljena nejednakost (4.6.).

Da bi D_i prema definiciji 4.7. bio i implicitni trenutak krajnjeg dovršetka, tada interval $[k \cdot T_i + D_i, (k+1) \cdot T_i]$ (na slici 4.9. označen s (ii)) mora biti popunjeno obradom prioritetnijih zadataka.

Prioritetniji zadatak T_j se u intervalu T_i javlja $\lceil T_i / T_j \rceil$ puta i toliko će se puta obraditi i istisnuti T_i , prvi puta na početku periode, a potom nakon svake pojave unutar periode.

U intervalu od pojave τ_i^k do D_i ($[k \cdot T_i, k \cdot T_i + D_i]$, na slici 4.9. dio označen sa (i)) zadatak T_j javlja se $\lceil D_i / T_j \rceil$ puta. U intervalu od D_i do kraja periode (i) zadatak T_j se javlja $\lceil T_i / T_j \rceil - \lceil D_i / T_j \rceil$ puta. Navedeni izraz uzima u obzir da se prvo pojavljivanje zadatka T_j poklapa s početkom periode. Sličan izraz $\lceil (T_i - D_i) / T_j \rceil$ to ne uzima u obzir i nije uvijek ispravan te se ne koristi.

Vrijeme potrošeno na zadatke većeg prioriteta koji se javljaju u intervalu od D_i do kraja periode, tj. u $[k \cdot T_i + D_i, (k+1) \cdot T_i]$ prikazano je formulom (4.7.)

$$\sum_{j=1}^{i-1} \left(\left\lceil \frac{T_i}{T_j} \right\rceil - \left\lceil \frac{D_i}{T_j} \right\rceil \right) \cdot C_j \quad (4.7.)$$

Treba primjetiti da formula (4.7.) ne uzima u obzir zadatke koji započinju prije D_i ali nastavljaju sa svojim izvođenjem nakon D_i . Međutim, takve zadatke smo već uzeli u obzir nejednakosću (4.6.) koja mora biti ispunjena a da bi D_i bio kandidat za krajnji trenutak završetka, tj. takvi zadaci završavaju prije D_i (nejednakost (4.6.) to zahtijeva).

Da bi svo vrijeme od D_i do kraja periode bilo popunjeno prioritetnijim zadacima, vrijednost prema formuli (4.7.) mora biti veće ili jednako intervalu na slici 4.9. označenome s (ii).

Definicija 4.10. Određivanje implicitna trenutka krajnjeg završetka

Za zadatak \mathcal{T}_i iz sustava zadataka \mathcal{S} (prema definiciji 4.1.), vrijednost D_i jest implicitni krajnji trenutak završetka ukoliko je to najmanja vrijednost koja zadovoljava uvjet (i) iz definicije 4.9. te uvjet (ii) prema formuli (4.8.).

$$(ii) : \sum_{j=1}^{i-1} \left(\left\lceil \frac{T_i}{T_j} \right\rceil - \left\lceil \frac{D_i}{T_j} \right\rceil \right) \cdot C_j \geq T_i - D_i \quad (4.8.)$$

Uvjet (ii) iz definicije 4.10. provjerava da li u intervalu $[D_i, T]$ ima slobodnog procesorskog vremena za zadatak \mathcal{T}_i ili se svo procesorsko vrijeme u tom intervalu troši na zadatke većeg prioriteta.

Primjer 4.6.

Razmotrimo jednostavni primjer s dva zadatka $\{\mathcal{T}_1, \mathcal{T}_2\}$ koji se javljaju svakih $T_1 = 7$ te svakih $T_2 = 10$ jedinica vremena. Neka su trajanja računanja $C_1 = 3$ te $C_2 = 1$.

Pri razmatranju rasporedivosti zadatka \mathcal{T}_1 jedina točka raspoređivanja koja je kandidat za D_1 je $D_1 = T_1$. U njoj su zadovoljena oba uvjeta iz definicija 4.9. i 4.10.

$$\begin{aligned} (i) : \quad & C_1 \leq T_1 \\ (ii) : \quad & 0 \geq T_1 - T_1 \end{aligned}$$

Pri razmatranju rasporedivosti zadatka \mathcal{T}_2 točke raspoređivanja su: $D_2 \in \{T_1, T_2\}$ te se uvjeti provjeravaju prema:

$$\begin{aligned} (i) : \quad & \left\lceil \frac{D_2}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{D_2}{T_2} \right\rceil \cdot C_2 \stackrel{?}{\leq} D_2 \\ (ii) : \quad & \left(\left\lceil \frac{T_2}{T_1} \right\rceil - \left\lceil \frac{D_2}{T_1} \right\rceil \right) \cdot C_1 \stackrel{?}{\geq} T_2 - D_2 \end{aligned}$$

Za $D_2 = T_1 = 7$:

$$\begin{aligned} (i) : \quad & 1 \cdot 3 + 1 \cdot 1 \stackrel{?}{\leq} 7 \quad \checkmark \\ (ii) : \quad & (2 - 1) \cdot 3 \stackrel{?}{\geq} 3 \quad \checkmark \end{aligned}$$

su oba uvjeta zadovoljena, iz čega slijedi da je zadatak rasporediv (pa tako i sustav, obzirom da je i prvi zadatak rasporediv).

Što se dobiva kad bi se uzelo drugu vrijednost za D_2 , tj. $D_2 = T_2 = 10$?

$$\begin{aligned} (i) : \quad & 2 \cdot 3 + 1 \cdot 1 \stackrel{?}{\leq} 10 \quad \checkmark \\ (ii) : \quad & (2 - 2) \cdot 3 \stackrel{?}{\geq} 0 \quad \checkmark \end{aligned}$$

Opet su oba uvjeta zadovoljena. U ovom primjeru se potvrda rasporedivosti dobiva odabirom bilo koje vrijednosti za D_2 .

Prema definiciji 4.7. implicitni t_{kz} je trenutak kada zadatak može završiti. Obzirom da za ovaj primjer vrijeme od 7. do 10. jedinice vremena troši zadatak \mathcal{T}_1 , očito je da implicitni

krajnji trenutak završetka nije $T_2 = 10$, zadatak \mathcal{T}_2 se ne može ni izvoditi ni završiti u tom intervalu (provjera uvjeta (ii) za $D_2 = 7$ je to potvrdila). Preostaje prva točka raspoređivanja $D_2 = T_1 = 7$ kada doista zadatak \mathcal{T}_2 može završiti obzirom da je prije toga je procesor slobodan (\mathcal{T}_1 koristi interval $[0, 3]$ te je interval $[3, 7]$ slobodan za \mathcal{T}_2).

Kada se radi samo provjera rasporedivosti sustava zadatka dovoljno je koristiti definiciju 4.9. (nije potrebno dodatno pronalaziti i implicitni t_{kz}).

Iz primjera 4.6. mogao bi se steći dojam da je dovoljno za D_i uzeti vrijednost T_i . Pri provjeri rasporedivosti općim kriterijem često je najbolje krenuti s tom vrijednošću. Ali ako provjera za nju ne daje pozitivan odgovor, ne smije se još donositi i zaključak o rasporedivosti sustava zadatka već treba ispitivanje ponoviti u ostalim točkama raspoređivanja. Primjerice, kada bi u primjeru 4.6. trajanja bila $C_1 = 4$ i $C_2 = 2$ tada uvjet (i) u $D_2 = T_2 = 10$ neće biti zadovoljen, dok će za $D_2 = 7$ uvjet i dalje biti zadovoljen.

Primjer 4.7.

Prikažimo korištenje općeg kriterija rasporedivosti na istom sustavu kao i u primjeru 4.4.:

$$\mathcal{T}_1 : \quad T_1 = 5 \text{ ms}, \quad C_1 = 2 \text{ ms}$$

$$\mathcal{T}_2 : \quad T_2 = 15 \text{ ms}, \quad C_2 = 5 \text{ ms}$$

$$\mathcal{T}_3 : \quad T_3 = 25 \text{ ms}, \quad C_3 = 5 \text{ ms}$$

a) Rasporedivost zadatka \mathcal{T}_1

Za prvi, najprioritetniji zadatak \mathcal{T}_1 jedina točka raspoređivanja je iduće pojavljivanje tog istog zadataka, tj. provjera se obavlja za $D_1 = 5$ ms.

$$(i) : \quad \left\lceil \frac{D_1}{T_1} \right\rceil \cdot C_1 \stackrel{?}{\leq} D_1 \quad \Rightarrow \quad \left\lceil \frac{5}{5} \right\rceil \cdot 2 \stackrel{?}{\leq} 5 \quad \checkmark$$

Zaključak: prvi je zadatak rasporediv.

b) Rasporedivost zadatka \mathcal{T}_2

Za drugi zadatak \mathcal{T}_2 točke raspoređivanja su: $D_2 \in \{5, 10, 15\}$ ms.

Provjera za vrijednost $D_2 = 5$ ms:

$$(i) : \quad \left(\left\lceil \frac{D_2}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{D_2}{T_2} \right\rceil \cdot C_2 \right) \stackrel{?}{\leq} D_2 \quad \Rightarrow \quad \left(\left\lceil \frac{5}{5} \right\rceil \cdot 2 + \left\lceil \frac{5}{10} \right\rceil \cdot 5 \right) \stackrel{?}{\leq} 5 \quad \times$$

Za prvu vrijednost $D_2 = 5$ prvi uvjet nije zadovoljen, tj. do 5. ms nema vremena za obavljanje prva dva zadataka.

Provjera za vrijednost $D_2 = 10$ ms:

$$(i) : \quad \left(\left\lceil \frac{10}{5} \right\rceil \cdot 2 + \left\lceil \frac{10}{10} \right\rceil \cdot 5 \right) \stackrel{?}{\leq} 10 \quad \checkmark$$

Uvjet (i) je zadovoljen, zadatak \mathcal{T}_2 je rasporediv.

c) Rasporedivost zadatka \mathcal{T}_3

Za treći zadatak \mathcal{T}_3 točke raspoređivanja su: $D_3 \in \{5, 10, 15, 20, 25\}$ ms. Istim postupkom mogli bi provjeriti za sve točke raspoređivanja. U nastavku je dano rješenje samo za zadnju točku raspoređivanje, tj. za $D_3 = 25$ ms (u ostalim točkama zadatak nije rasporediv).

$$(i) : \left\lceil \frac{D_3}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{D_3}{T_2} \right\rceil \cdot C_2 + \left\lceil \frac{D_3}{T_3} \right\rceil \cdot C_3 \stackrel{?}{\leq} D_3 \Rightarrow \left\lceil \frac{25}{5} \right\rceil \cdot 2 + \left\lceil \frac{25}{15} \right\rceil \cdot 5 + \left\lceil \frac{25}{25} \right\rceil \cdot 5 \stackrel{?}{\leq} 25 \quad \checkmark$$

Zaključak: treći je zadatak rasporediv.

Sva tri zadatka su rasporediva te je prema tome i sustav zadataka rasporediv.

Istim postupkom bi se mogao provjeriti i sustav iz primjera 4.5. Rezultat bi trebao pokazati da iako su prva dva zadatka rasporediva, treći nije rasporediv niti u jednoj točki raspoređivanja. U nastavku je ipak prikazan primjer s novim sustavom zadataka.

Primjer 4.8.

Zadan je sustav od četiri zadatka s periodama i vremenima računanja prema:

$$\mathcal{T}_1 : T_1 = 5 \text{ ms}, \quad C_1 = 1 \text{ ms}$$

$$\mathcal{T}_2 : T_2 = 8 \text{ ms}, \quad C_2 = 1 \text{ ms}$$

$$\mathcal{T}_3 : T_3 = 9 \text{ ms}, \quad C_3 = 2 \text{ ms}$$

$$\mathcal{T}_4 : T_4 = 10 \text{ ms}, \quad C_4 = 3 \text{ ms}$$

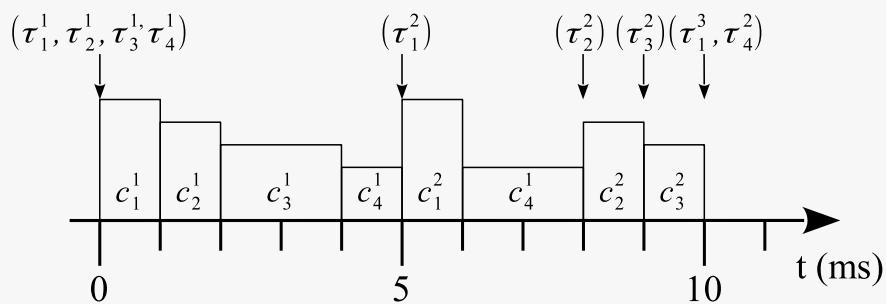
Provjera rasporedivosti:

a) Nužni uvjet:

$$U = \sum_{i=1}^4 \frac{C_i}{T_i} = \frac{1}{5} + \frac{1}{8} + \frac{2}{9} + \frac{3}{10} = 0,847 < 1 \quad \checkmark$$

Nužni uvjet je zadovoljen. Provedimo i grafičku provjeru (za razliku od općeg kriterija, grafička provjera je vrlo brza za male sustave zadataka).

b) Grafička provjera u kritičnom slučaju



Slika 4.10.

Grafička provjera daje potvrdan odgovor o rasporedivosti (postupkom mjere ponavljanja). Općim kriterijom bit će provjeren samo četvrti zadatak.

Za četvrti zadatak T_4 točke raspoređivanja su: $D_4 \in \{5, 8, 9, 10\}$ ms.

Iz slike 4.10. je vidljivo da bi provjere za $D_4 \in \{5, 9, 10\}$ bile neuspješne, odnosno, $D_4 = 8$ je prava vrijednost implicitnog t_{kz} . Prikažimo to i preko općeg kriterija i definicije 4.10.

Uvjet (i):

$$(i) : \left\lceil \frac{D_4}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{D_4}{T_2} \right\rceil \cdot C_2 + \left\lceil \frac{D_4}{T_3} \right\rceil \cdot C_3 + \left\lceil \frac{D_4}{T_4} \right\rceil \cdot C_4 \stackrel{?}{\leq} D_4$$

Uvrštavanjem:

$$(i) : \left\lceil \frac{8}{5} \right\rceil \cdot 1 + \left\lceil \frac{8}{8} \right\rceil \cdot 1 + \left\lceil \frac{8}{9} \right\rceil \cdot 2 + \left\lceil \frac{8}{10} \right\rceil \cdot 3 \stackrel{?}{\leq} 8 \quad \checkmark$$

Uvjet (ii):

$$(ii) : \left(\left\lceil \frac{T_4}{T_1} \right\rceil - \left\lceil \frac{D_4}{T_1} \right\rceil \right) \cdot C_1 + \left(\left\lceil \frac{T_4}{T_2} \right\rceil - \left\lceil \frac{D_4}{T_2} \right\rceil \right) \cdot C_2 + \left(\left\lceil \frac{T_4}{T_3} \right\rceil - \left\lceil \frac{D_4}{T_3} \right\rceil \right) \cdot C_3 \stackrel{?}{\geq} T_4 - D_4$$

Uvrštavanjem:

$$(ii) : \left(\left\lceil \frac{10}{5} \right\rceil - \left\lceil \frac{8}{5} \right\rceil \right) \cdot 1 + \left(\left\lceil \frac{10}{8} \right\rceil - \left\lceil \frac{8}{8} \right\rceil \right) \cdot 1 + \left(\left\lceil \frac{10}{9} \right\rceil - \left\lceil \frac{8}{9} \right\rceil \right) \cdot 2 \stackrel{?}{\geq} 10 - 8 \quad \checkmark$$

Očekivani zaključak: zadatak T_4 je rasporediv.

4.5.1.4. Granice procesorske iskoristivosti

Procesorsku iskoristivost i određivanje njenih granica može se promatrati s dva aspekta:

1. sa stanovišta racionalnog korištenja sredstava i povećanja propusnosti, te učinkovitog korištenja moći (snage) aktivnog sredstva te
2. kao mjeru za utvrđivanje izvodljivosti raspoređivanja (u skladu s danim ograničenjima, koje nameće disciplina raspoređivanja).

Definicija 4.11. Potpuno iskorištenje procesora
--

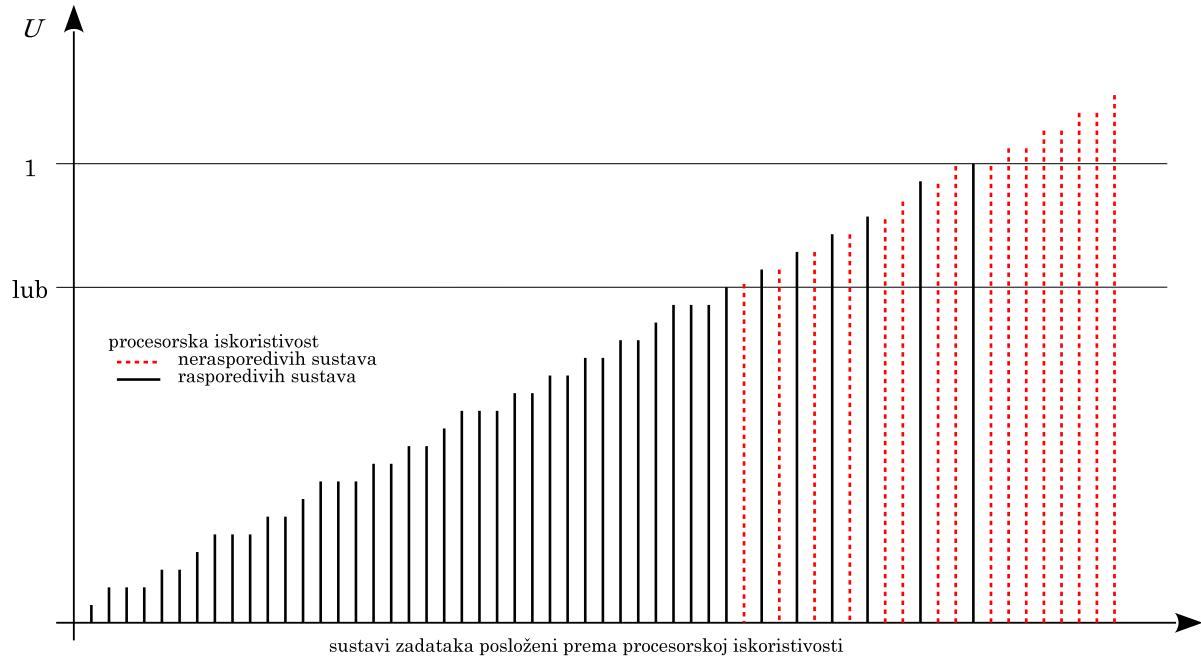
U kontekstu raspoređivanju zadataka skup zadataka *potpuno iskorištava procesor* ako bilo koje povećanje računalnih vremena izvođenja bilo kojeg zadatka uzrokuje da sustav postaje nerasporediv zadanim postupkom.

Važno je shvatiti da potpuno iskorištenje procesora u ovom kontekstu ne mora označavati 100 % iskorištenje. Primjerice, sustav zadataka iz primjera 4.8. u potpunosti iskorištava procesor (u kritičnom slučaju se zadatak najmanjeg prioriteta jedva uspije dovršiti do iduće pojave) – bilo kakvo povećanje vremena izračunavanja vodilo bi u prekoračenje zadanih vremenskih ograničenja.

Definicija 4.12. Najmanja gornja granica faktora procesorskog iskorištenja – lub(U)

Najmanja gornja granica faktora procesorskog iskorištenja (engl. *least upper bound – lub*) je minimalna veličina faktora procesorskog iskorištenja svih mogućih skupova zadataka koji su rasporedivi i koji potpuno iskorištavaju procesor.

Slika 4.11. simbolički prikazuje sve moguće sustave zadataka i njihovu procesorsku iskoristivost. Do granice lub svi su sustavi rasporedivi. Od lub do 1 neki su rasporedivi a neki nisu. Sustavi zadataka koji imaju procesorsku iskoristivost iznad 1 nisu rasporedivi.



Slika 4.11. Primjeri sustava zadataka različitih faktora procesorskog iskorištenja

Kako najmanja gornja granica ($\text{lub}(U)$) podrazumijeva potpuno iskorištenje procesora i uz to jamči izvodljivost (ostvarivost) raspoređivanja u skladu s pridjeljenim prioritetima, to ukazuju na ostvarivost raspoređivanja (engl. *scheduling feasibility*) sustava zadataka koji imaju manji faktor iskoristjenja od tog graničnog. Na taj način $\text{lub}(U)$ predstavlja korisno i važno pomagalo pri utvrđivanju izvodljivosti i iskoristivosti procesora.

Dok za sustave zadataka koji stvaraju manje opterećenje od $\text{lub}(U)$ sa sigurnišću znamo da su rasporedivi, za sustave koji stvaraju veće opterećenje – faktor iskoristivosti veći od $\text{lub}(U)$, ne možemo reći jesu li ili nisu rasporedivi – ne možemo sa sigurnošću reći da nisu rasporedivi, jer mogu biti.

Definicija 4.13. $\text{lub}(U)$ za sustav zadataka koji koristi mjeru ponavljanja

Najniža gornja granica faktora iskoristjenja procesora za sustav od N zadataka \mathcal{S} (prema definiciji 4.1.) raspoređenih sa stalnom pridjelom prioriteta dodijeljenih mjerom (prema definiciji 4.5.) ponavljanja je:

$$\text{lub}(U) = N \left(\sqrt[N]{2} - 1 \right)$$

Dokaz definicije 4.13. prikazan je u [Liu, 1973].

Definicija 4.14. Dovoljan uvjet rasporedivosti kada se koristi mjera ponavljanja

Sustav od N zadataka \mathcal{S} (prema definiciji 4.1.) sigurno se može rasporediti postupkom mjere ponavljanja (prema definiciji 4.5.), ako za njegov faktor iskorištenja procesora $U_{\mathcal{S}}$ vrijedi:

$$U_{\mathcal{S}} \leq \text{lub}(U) \quad \text{tj.} \quad \sum_{i=1}^N \frac{C_i}{T_i} \leq N \left(\sqrt[N]{2} - 1 \right) \quad (4.9.)$$

Definicija 4.14. definira rasporedivost sustava samo ako je $U_{\mathcal{S}}$ u granicama $[0, \text{lub}(U)]$. Kada $U_{\mathcal{S}}$ pada u granice $(\text{lub}(U), 1]$ rasporedivost treba provjeriti drugim kriterijima, primjerice grafičkim ili općim kriterijem rasporedivosti.

Što u sustavu ima više poslova teže ih je rasporediti, iako je njihovo ukupno opterećenje jednako. To prikazuje i definicija 4.13. te tablica 4.1. Ipak "umjereni velik" skup poslova bit će sigurno rasporediv ako operećenje koje on stvara ne prelazi otprilike 0,7 (70 %).

Tablica 4.1. lub(U) za različiti broj zadataka

N	1	2	3	4	5	10	100	1000	10^{10}
lub(U)	1	0,83	0,78	0,76	0,74	0,72	0,696	0,6934	0,6931

4.5.2. Jednoprocesorsko dinamičko raspoređivanje

Dinamički postupci raspoređivanja koriste obilježja zadataka koja se mijenjaju tijekom rada zadataka ili samim protokom vremena ili drugim događajima koji se zbivaju tijekom rada.

Primjerice, ako se neki zadatak u različitim intervalima javlja s različitom učestalošću, on bi u svakom od intervala trebao imati različitu vrijednost prioriteta, a da bi se mogao raspoređivati prema postupku mjere ponavljanja. Dinamička promjena prioriteta morala bi se ugraditi dodatnim postupcima te bi u konačnici globalni sustav raspoređivanja bio dinamički.

4.5.2.1. Raspoređivanje prema krajnjim trenucima završetaka zadataka

Sa stanovišta SRSV-a, najznačajniji postupak raspoređivanja iz kategorije dinamičkih postupaka jest raspoređivanje prema krajnjim trenucima završetaka zadataka – KTZ (engl. *earliest deadline first – EDF, deadline driven scheduling – DDS*).

Definicija 4.15. Raspoređivanje prema krajnjim trenucima završetka

Raspoređivanje prema krajnjim trenucima završetka je postupak raspoređivanja koji u svakom trenutku za izvođenje odabire zadatak koji ima najbliži krajnji trenutak završetka.

Da bi sustav bio rasporediv ovom metodom dovoljno je da vrijedi nužan uvjet rasporedivosti prema definiciji 4.3.

Ukoliko dva ili više zadataka u nekom trenutku imaju isti najbliži trenutak krajnjeg završetka, odabir jednog od njih je ili proizvoljan ili se definiraju dodatni (sekundarni) kriteriji raspoređivanja (npr. red prispijeća, prioriteti).

Raspoređivač se poziva svaki put kada se nešto promijeni u sustavu, tj. kada se pojavi neki zadatak u sustavu ili kada neki zadatak završi s radom.

Raspoređivanje prema krajnjim trenucima završetka ne postavlja dodatne uvjete na faktor isko-

rištenja procesora, tj. ako sustav zadatka zadovoljava nužan uvjet ($U \leq 1$) sustav je rasporediv ovim postupkom (dokaz je u [Liu, 1973]).

Raspoređivanje prema krajnjim trenucima završetka je složenje od raspoređivanja mjerom ponavljanja jer zahtijeva dodatna saznanja o zadacima, tj. njihove trenutke krajnjih završetaka. Zato se ovakvo raspoređivanje rjeđe susreće kao mehanizam u stvarnim operacijskim sustavima.

Primjer 4.9.

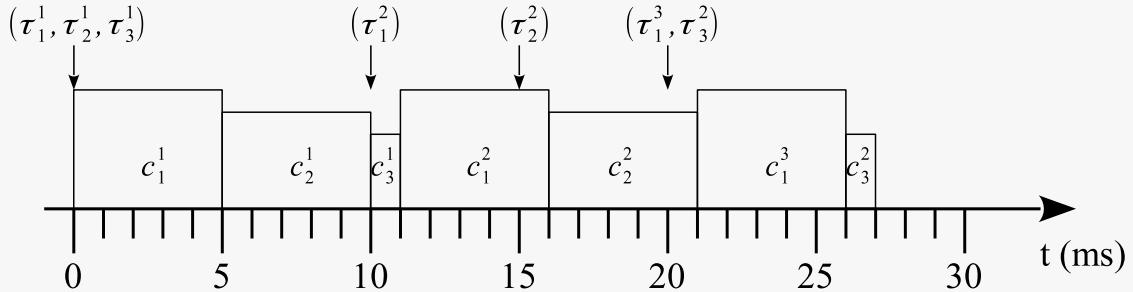
Prikažimo korištenje raspoređivanja prema krajnjim trenucima završetka za sustav zadatka iz primjera 4.5.:

$$\begin{aligned}\mathcal{T}_1 : \quad T_1 &= 10 \text{ ms}, \quad C_1 = 5 \text{ ms} \\ \mathcal{T}_2 : \quad T_2 &= 15 \text{ ms}, \quad C_2 = 5 \text{ ms} \\ \mathcal{T}_3 : \quad T_3 &= 20 \text{ ms}, \quad C_3 = 1 \text{ ms}\end{aligned}$$

Nužan uvjet rasporedivosti:

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} = \frac{5}{10} + \frac{5}{15} + \frac{1}{20} = 0,883 \leq 1 \quad \checkmark$$

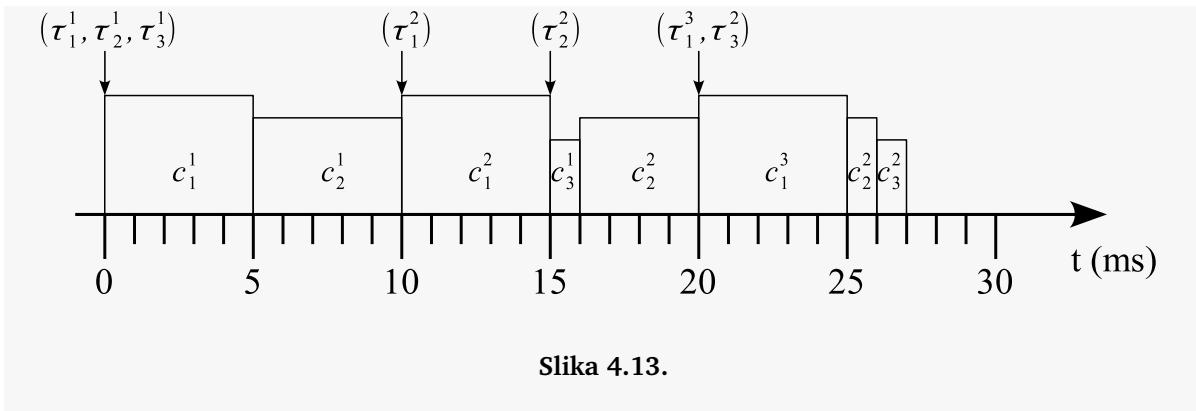
Raspoređivanje prema krajnjim trenucima završetka (grafički), uz sekundarni kriterij prema redu prispijeća:



Slika 4.12.

Točke u kojima se poziva raspoređivač uključuju sam početak, tj. trenutak pojave sva tri zadatka u $t=0$ ms, trenutak završetka prvog zadatka u $t=5$ ms, trenutak završetka drugog i istovremena pojava prvog zadatka (2. put) u $t=10$ ms, trenutak završetka trećeg zadatka i tako dalje. Primijetimo da u $t=10$ ms oba pripravna zadatka, i prvi i treći tada imaju isti krajnji trenutak završetka $t_{kz} = 20$ ms. Prema tome odabir koji će prije krenuti jest proizvoljan. U prikazanom rješenju odabrao se treći zadatak jer on duže čeka u redu pripravnih zadataka.

Ukoliko bi sekundarni kriterij bio prema mjeri ponavljanja (zadaci kraće periode imaju veći prioritet) tada bi raspoređivanje od 10. ms na dalje bilo ponešto drukčije, prema slici 4.13.



Slika 4.13.

4.6. Višeprocesorsko raspoređivanje

Korištenje višeprocesorskih računala u postupcima upravljanja ublažava problem rasporedivosti obzirom na postojanje veće procesne moći. Svi prethodno navedeni postupci primjenjivi su i za višeprocesorska računala, uz male promjene obzirom da će sada biti više aktivnih zadataka (dretvi).

Primjerice, korištenjem postupka mjere ponavljanja na višeprocesorskom sustavu s M procesora, pri raspoređivanju uvijek će se odabrati M zadataka najvećeg prioriteta, a ne samo jedan kao kod jednoprocesorskog raspoređivanja.

Rasporedivost zadataka na višeprocesorskom sustavu je u najgorem slučaju jednaka rasporedivosti u jednoprocesorskom, a u pravilu je povoljnija. Međutim, formule su nešto složenije. Primjeri određivanja lub(U) za postupak mjere ponavljanja na višeprocesorskom računalu opisani su u [Baruah, 2003][Lopez, 2004][Jain, 1994] i sličnim radovima.

Sljedeća dva primjera prikazuju raspoređivanja istih skupova zadataka kao iz prethodnih primjera, ali sada na dvoprocesorskom računalu.

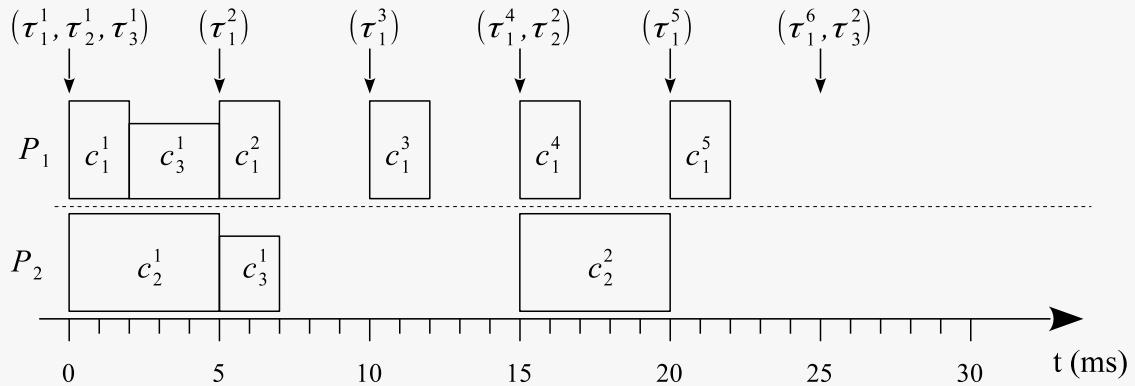
Primjer 4.10.

Prikažimo rad postupka mjere ponavljanja kada se primjenjuje na dvoprocesorskom računalu nad sustavom iz primjera 4.4.

$$\mathcal{T}_1 : \quad T_1 = 5 \text{ ms}, \quad C_1 = 2 \text{ ms}$$

$$\mathcal{T}_2 : \quad T_2 = 15 \text{ ms}, \quad C_2 = 5 \text{ ms}$$

$$\mathcal{T}_3 : \quad T_3 = 25 \text{ ms}, \quad C_3 = 5 \text{ ms}$$



Slika 4.14.

Već se u 7. ms može zaključiti da je sustav rasporediv na dvoprocesorskom računalu (i stati s dalnjom provjerom). Očito je problem olakšan, tj. takvi sustavi mogu zadovoljiti veći skup zadataka.

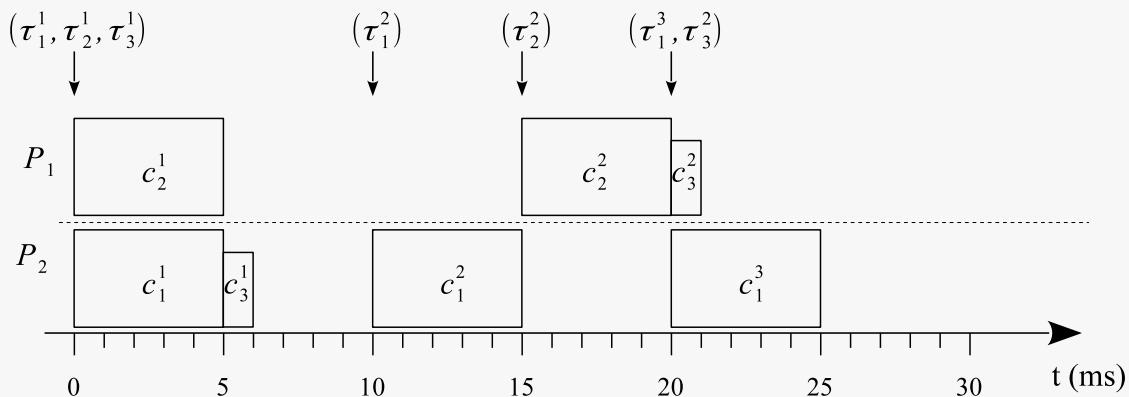
Primjer 4.11.

Prikažimo korištenje raspoređivanja prema krajnjim trenucima završetka za sustav zadataka iz primjer 4.5. na dvoprocesorskom sustavu:

$$\mathcal{T}_1 : T_1 = 10 \text{ ms}, C_1 = 5 \text{ ms}$$

$$\mathcal{T}_2 : T_2 = 15 \text{ ms}, C_2 = 5 \text{ ms}$$

$$\mathcal{T}_3 : T_3 = 20 \text{ ms}, C_3 = 1 \text{ ms}$$



Slika 4.15.

I u ovom je primjeru provjera rasporedivosti mogla stati prije, u 6. ms obzirom da je sustav razmatran u kritičnom slučaju.

Problem raspoređivanja na višeprocesorskim sustavima je ponešto olakšan naspram jednoprocesorskih. Nadalje, višeprocesorski sustav omogućava da se pri rješavanju jednog problema može koristiti paralelnost u izvođenju pojedinog posla. Paralelno izvođenje skupa nezavisnih zadataka je trivijalno. Međutim, ukoliko među zadacima postoje ovisnosti tipa “ \mathcal{T}_i se mora obaviti prije \mathcal{T}_j ”, tada je možda moguće optimirati redoslijed izvođenja zadataka na različitim procesorima tako da cijeli sustav zadataka završi prije. U nastavku se bavimo takvim problemima optimiranja.

4.6.1. Višeprocesorsko statičko raspoređivanje

U višeprocesorsko statičko raspoređivanje spada i već prikazani postupak mjere ponavljanja. Obzirom na njegovu sličnost s raspoređivanjem na jednoprocesorskim sustavima, ovdje se on više neće razmatrati. Postupci koji će se razmatrati u nastavku uključuju optimiranje nad uređenim skupom zavisnih zadataka.

Definicija 4.16. Zavisnost zadataka

Zadaci \mathcal{T}_i i \mathcal{T}_j su zavisni zadaci ako postoji definirani redoslijed njihova izvođenja koji je potrebno poštivati. Redoslijed njihova izvođenja može biti:

$$\mathcal{T}_i < \mathcal{T}_j \quad \text{ili} \quad \mathcal{T}_j < \mathcal{T}_i$$

Operator $<$ označava relaciju “treba se dogoditi prije”, odnosno, označava da zadatak s lijeve strane operatora $<$ mora završiti prije nego li se desni zadatak smije početi izvoditi.

Ukoliko redoslijed njihova izvođenja nije definiran, odnosno, ako se zadaci mogu izvoditi proizvoljnim redoslijedom, pa i paralelno, onda zadaci nisu međusobno zavisni.

Osim slučajeva opisanih u definiciji 4.16. postoji i treća mogućnost. Ukoliko redoslijed njihova izvođenja nije definiran, odnosno, zadaci se mogu izvoditi proizvoljnim redoslijedom, ali ne i paralelno, onda zadaci jesu međusobno zavisni, ali se njihova zavisnost ne iskazuje prema prethodnim relacijama. Takvi se zadaci međusobno trebaju sinkronizirati mehanizmima međusobnog isključivanja. O tome više kasnije.

Definicija 4.17. Skup zavisnih zadataka

Sustav zavisnih zadataka definira se skupom $\mathcal{S} = \{\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_N\}$, vremenima trajanja njihova izvođenja $\{C_1, C_2, \dots, C_N\}$ te skupom zavisnosti:

$$\mathcal{R} = \{(i, j), \quad i, j \in \{1..N\} \quad | \quad i \neq j\}$$

gdje svaki par iz skupa \mathcal{R} definira zavisnost između zadataka \mathcal{T}_i i \mathcal{T}_j prema:

$$\mathcal{T}_i < \mathcal{T}_j$$

uz značenje operatorka $<$ prema definiciji 4.16.

U ovu definiciju ugraditi ćemo i uvjet da nema petlji u lancima zavisnosti, tj. da se u svakom lancu zavisnosti koji proizlaze iz \mathcal{R} u obliku $\mathcal{T}_k < \mathcal{T}_l < \mathcal{T}_m < \dots < \mathcal{T}_n$ jedan zadatak javlja samo jedanput (i svi lanci su konačne duljine).

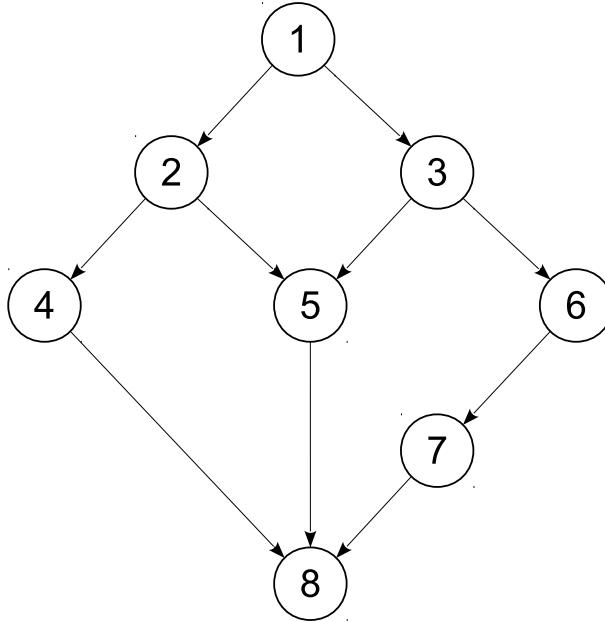
U praksi se često koristi ciklički sustav zadataka kod kojeg postoji jedan početni zadatak (koji jedini može krenuti na početku) te jedan završni zadatak (koji može krenuti tek po završetku svih ostalih). Po završetku završnog zadataka može krenuti iduća iteracija sustava, tj. ponovno početni zadatak. Ovakav se ciklički sustav za jednu iteraciju može analizirati i bez ovisnosti među završnim i početnim zadatakom.

Skup zavisnih zadataka se često prikazuje usmjerenim grafom. Slika 4.16. prikazuje jedan takav primjer.

4.6.1.1. Opće raspoređivanje

Opće raspoređivanje (engl. *general scheduling*) [Nissanke, 1997] je teoretsko raspoređivanje koje razmatra procesore kao poslužitelje koje se može koristiti i samo djelomično, tj. u određenom postotku kroz neke vremenske intervale. Dio procesorske snage koju sustav može dodijeliti zadataku \mathcal{T}_i označava se s α_i , za koju mora vrijediti $0 \leq \alpha_i \leq 1$.

Primjerice, neki zadatak u takvom razmatranju može kroz neki period koristiti 25 % procesora P_1 . Ipak, rezultati i takvog teoretskog razmatranja, tj. raspoređivanja, se naknadno mogu prilagoditi u nešto praktično, npr. zadatak neće dobiti 25 % od P_1 kroz period od 10 sekundi nego će dobiti 100 % od P_1 u intervalu [7, 5; 10].



Slika 4.16. Primjer sustava zavisnih zadataka prikazan usmjerenim grafom

Definicija 4.18. Opće raspoređivanje

Opće raspoređivanje sustava zavisnih zadataka $\mathcal{S} = \{\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_N\}$ na m procesora optimalno raspodijeljuje zadatke po procesorima u svrhu njihova završetka tako da se raspoređivanje razmatra u intervalima Δt_k u kojima vrijedi:

1. podskup zadataka koji se izvode međusobno nezavisni zadaci
2. svaki zadatak \mathcal{T}_i koristi udio procesorske snage $\alpha_{i,k}$ za koji vrijedi:

$$0 \leq \alpha_{i,k} \leq 1$$

3. svi zadaci koji se izvode koriste dio raspoložive procesorske snage:

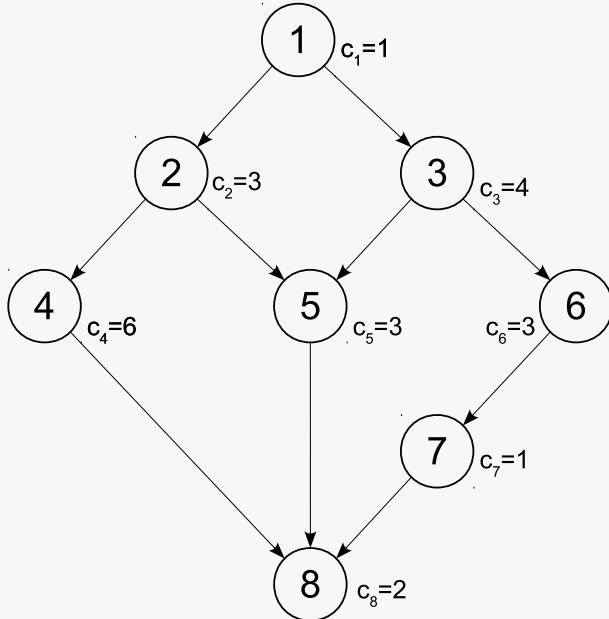
$$\sum_{j=1}^N \alpha_{j,k} \leq m.$$

Opće raspoređivanje nema podlogu u stvarnoj primjeni, ono je hipotetičko i predstavlja samo podlogu za izvođenje raspoređivanja koje će optimalno iskoristiti dane procesore na principima prekidljivog raspoređivanja.

U nastavku u primjerima je korišten pojednostavljeni algoritam ostvarenja općeg raspoređivanja koji uzima u obzir samo dio grafa. Stoga rezultati koji se na ovaj način dobivaju nisu uvijek optimalni, odnosno, najčešće nisu optimalni kada se (pri rješavanju) dođe do neuravnotežnosti u dijelu grafa.

Primjer 4.12.

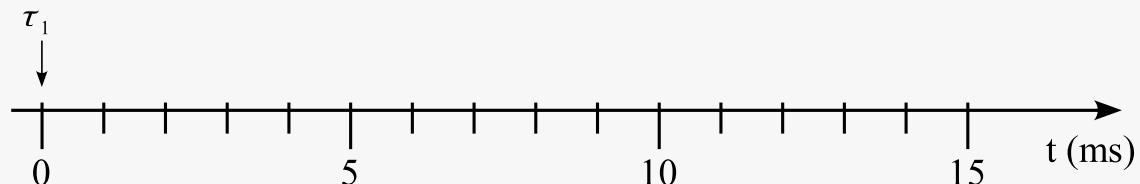
Sustav zadataka zadan je necikličkim računalnim grafom na slici 4.17. Napraviti raspoređivanje sustava zadataka korištenjem postupka općeg raspoređivanja za dva procesora.



Slika 4.17.

Prvi interval koji se razmatra s općim raspoređivanjem započinje pojavom zadatka T_1 . Obzirom da nije zadano vrijeme njegove pojave prepostavlja se $t = 0$. Jedinice vremena nisu zadane te se neće niti koristiti (mogu biti sekunde, milisekunde, ...).

$$1. t_1 = 0$$



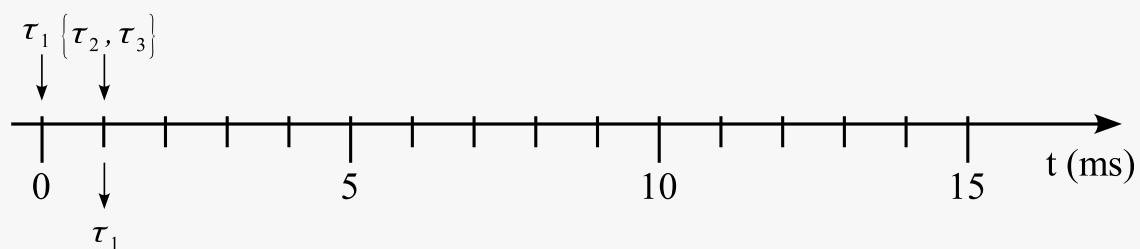
Pripravni zadatak: $T_1(C_1 = 1)$

Obzirom da postoji samo jedan zadatak a dva raspoloživa procesora:

$$\alpha_1 = 1 \Rightarrow T_{z1} = C_1/\alpha_1 = 1 \Rightarrow t_2 = t_1 + T_{z1} = 1$$

Sa T_{zi} označeno je trajanje izvođenja zadatka i uz α_i .

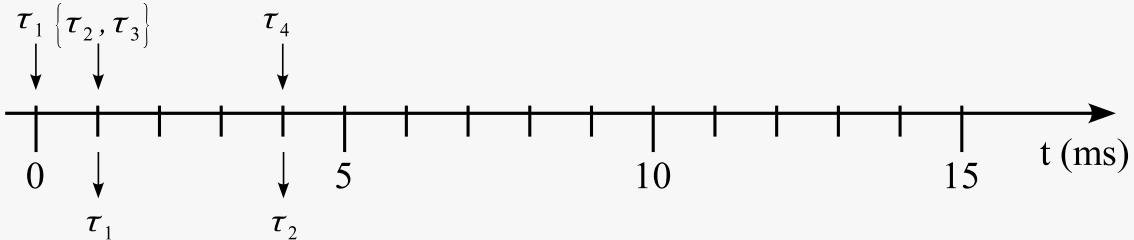
$$2. t_2 = 1$$



Pripravni zadaci: $\mathcal{T}_2(C_2 = 3), \mathcal{T}_3(C_3 = 4)$

$$\left. \begin{array}{l} \alpha_2 = 1 \Rightarrow T_{z2} = C_2/\alpha_2 = 3 \\ \alpha_3 = 1 \Rightarrow T_{z3} = C_3/\alpha_3 = 4 \end{array} \right\} \Rightarrow t_3 = t_2 + \min\{T_{z2}, T_{z3}\} = 4$$

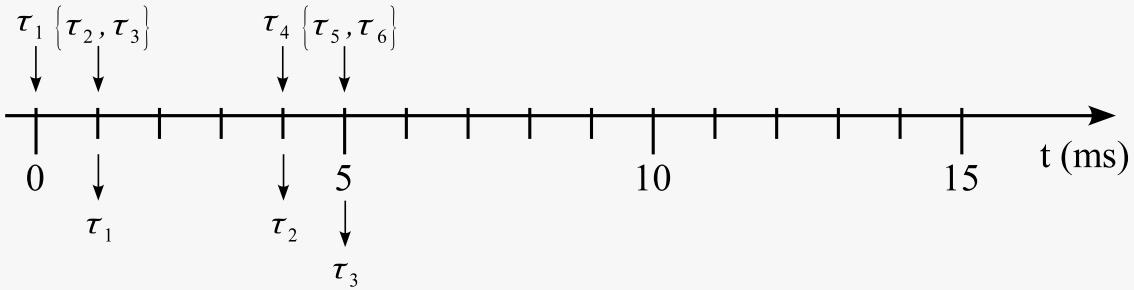
3. $t_3 = 4$



Pripravni zadaci: $\mathcal{T}_3(C_3^{(2)} = 1), \mathcal{T}_4(C_4 = 6)$

$$\left. \begin{array}{l} \alpha_3 = 1 \Rightarrow T_{z3} = C_3^{(2)}/\alpha_3 = 1 \\ \alpha_4 = 1 \Rightarrow T_{z4} = C_4/\alpha_4 = 6 \end{array} \right\} \Rightarrow t_4 = t_3 + \min\{T_{z3}, T_{z4}\} = 5$$

4. $t_4 = 5$



Pripravni zadaci: $\mathcal{T}_4(C_4^{(2)} = 5), \mathcal{T}_5(C_5 = 3), \mathcal{T}_6(C_6 = 3)$

Tri zadatka na dva procesora: kako odrediti raspored, a da bi sustav u najkraćem vremenu sve završio?

Kada bi to bili zadnji zadaci u sustavu, ili kada bi svima slijedio samo jedan zadatak, onda bi omjere udjela procesorskog vremena tako podešili da svi završe u istom trenutku (ako je to moguće), koristeći oba procesora. Primjerice \mathcal{T}_4 bi dobio $\alpha_4 = (C_4^{(2)}/(C_4^{(2)} + C_5 + C_6)) \cdot 2 = 10/11$ (množi se s 2 zbog dva procesora), a ostala dva po $6/11$ udjela u procesorskom vremenu. Međutim, to nisu takvi zadaci.

Uvidom u graf i trenutno stanje, može se primjetiti da zadatak \mathcal{T}_7 slijedi neposredno iza \mathcal{T}_6 , a da njemu kao i zadacima \mathcal{T}_4 i \mathcal{T}_5 slijedi \mathcal{T}_8 . Prethodno razmatranje moglo bi se primjeniti kada bi objedinili zadatke \mathcal{T}_6 i \mathcal{T}_7 u jedan zadatak \mathcal{T}_σ , barem za potrebe izrade rasporeda.

Pripravni zadaci: $\mathcal{T}_4(C_4^{(2)} = 5), \mathcal{T}_5(C_5 = 3), \mathcal{T}_\sigma(C_\sigma = C_6 + C_7 = 4)$

$$\Delta t = \frac{C_4^{(2)} + C_5 + C_\sigma}{2} = \frac{5 + 3 + 4}{2} = \frac{12}{2} = 6 \text{ (potrebno procesorskog vremena)}$$

$$\alpha_4 = C_4^{(2)}/\Delta t = 5/6$$

$$\alpha_5 = C_5/\Delta t = 2/6$$

$$\alpha_\sigma = C_\sigma/\Delta t = 4/6$$

Obzirom da su svi α_i manji od 1 raspored će biti optimalan – oba procesora će biti iskorištena cijelo vrijeme ($\Delta t = 6$). Kada to ne bi bilo tako, kada bi primjerice α_4 po prethodnom proračunu bio veći od jedan (kada bi sustav od tri zadatka bio van ravnoteže – kada bi jedan zadatak imao više posla od zbroja druga dva), tada bi pri optimiranju trebalo postaviti $\alpha_4 = 1$ i s time nastaviti optimizaciju.

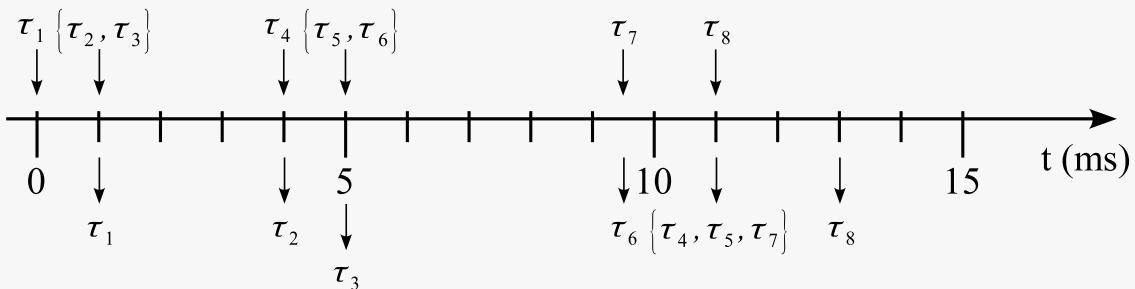
Skup zadataka $\{\mathcal{T}_4, \mathcal{T}_5, \mathcal{T}_\sigma\}$ završit će za $\Delta t = 6$.

Međutim, obzirom da se \mathcal{T}_σ sastoji od \mathcal{T}_6 i \mathcal{T}_7 , prvi od njih \mathcal{T}_6 bit će gotov i prije:

$$T_{z6} = C_6/\alpha_\sigma = 3/(4/6) = 9/2 = 4,5 \Rightarrow t_5 = 9,5$$

Nakon \mathcal{T}_6 u sustav dolazi \mathcal{T}_7 te zajedno s \mathcal{T}_4 i \mathcal{T}_5 završava u $t_6 = 11$.

5. $t_6 = 11$



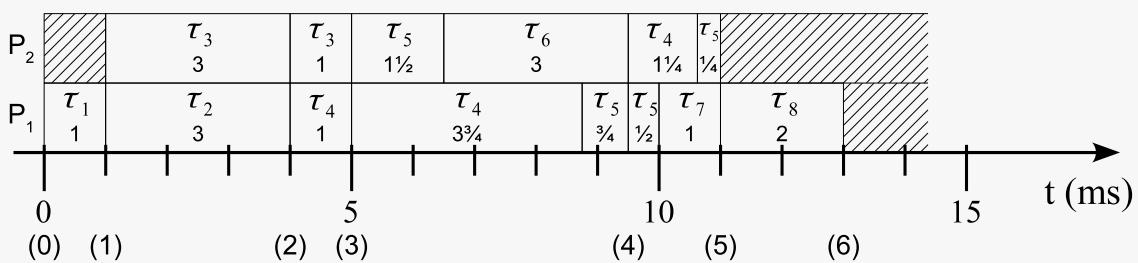
Pripravni zadaci: $\mathcal{T}_8(C_8 = 3)$

$$\alpha_8 = 1 \Rightarrow T_{z8} = C_8/\alpha_8 = 2 \Rightarrow t_7 = t_6 + T_{z8} = 13$$

6. $t_7 = 13$

Završetkom zadnjeg zadataka završava sustav zadataka (prema općem raspoređivanju).

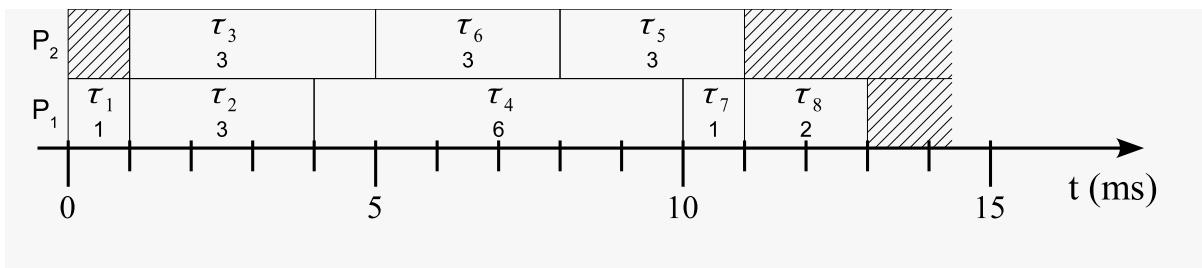
Korištenjem rezultata općeg raspoređivanja može se napraviti stvaran raspored zadataka po procesorima (uz $\alpha_i = 1$ po segmentima) prepostavljajući da se zadaci mogu prekidati (prekidljivo raspoređivanje), kao što to prikazuje slika 4.18.



Slika 4.18.

Do $t = 5$, raspored je jednostavan. U intervalu $[5; 9,5]$ potrebno je smjestiti \mathcal{T}_4 , \mathcal{T}_5 i \mathcal{T}_6 s prethodno izračunatim vremenima za taj interval. Kako će se to napraviti nije predefinirano, može i drugčije nego što je prikazano na prethodnoj slici. Ono na što treba pripaziti je da se osigura da u svakom trenutku na različitim procesorima budu različiti zadaci. Analogno za interval $[9,5; 11]$.

Naravno, kada bi slagali ručno, bez korištenja rezultata općeg raspoređivanja, možda bi došli do jednostavnijeg rasporeda, npr. kao na slici 4.19. Međutim, to je moguće samo u jednostavnijim slučajevima, ne i općenito.

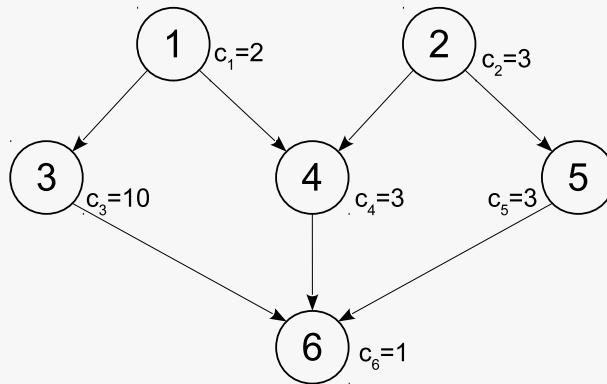


Slika 4.19.

“Ručno slaganje” kao na slici 4.19. nije valjano rješenje u zadacima u kojima se traži korištenje rezultata općeg raspoređivanja.

Primjer 4.13.

Sustav zadataka zadan je necikličkim računalnim grafom na slici 4.20. Napraviti raspoređivanje sustava zadataka korištenjem postupka općeg raspoređivanja za dva procesora.



Slika 4.20.

$$1. t_1 = 0$$

Pripravni zadaci: $\mathcal{T}_1(C_1 = 2)$, $\mathcal{T}_2(C_2 = 3)$

$$\left. \begin{array}{l} \alpha_1 = 1 \Rightarrow T_{z1} = C_1/\alpha_1 = 2 \\ \alpha_2 = 1 \Rightarrow T_{z2} = C_2/\alpha_2 = 3 \end{array} \right\} \Rightarrow t_2 = t_1 + \min\{T_{z1}, T_{z2}\} = 2$$

$$2. t_2 = 2$$

Pripravni zadaci: $\mathcal{T}_2(C_2^{(2)} = 1)$, $\mathcal{T}_3(C_3 = 10)$

$$\left. \begin{array}{l} \alpha_2 = 1 \Rightarrow T_{z2} = C_2^{(2)}/\alpha_2 = 1 \\ \alpha_3 = 1 \Rightarrow T_{z3} = C_3/\alpha_3 = 10 \end{array} \right\} \Rightarrow t_3 = t_2 + \min\{T_{z2}, T_{z3}\} = 3$$

$$3. t_3 = 3$$

Pripravni zadaci: $\mathcal{T}_3(C_3^{(2)} = 9)$, $\mathcal{T}_4(C_4 = 3)$, $\mathcal{T}_5(C_5 = 3)$

$$\Delta t = \frac{C_3^{(2)} + C_4 + C_5}{2} = \frac{9 + 3 + 3}{2} = \frac{15}{2}$$

$$\alpha_3 = C_3^{(2)} / \Delta t = 18/15$$

$$\alpha_4 = C_4 / \Delta t = 6/15$$

$$\alpha_5 = C_5 / \Delta t = 6/15$$

Problem: $\alpha_3 > 1$!

Stavljamo: $\alpha_3 = 1$ a za ostala dva ostaje drugi procesor koji se dijeli na:

$$\Delta t^{(2)} = \frac{C_4 + C_5}{1} = \frac{3 + 3}{1} = 6$$

$$\alpha_4 = C_4 / \Delta t = 3/6 = 1/2$$

$$\alpha_5 = C_5 / \Delta t = 3/6 = 1/2$$

$$\left. \begin{array}{l} \alpha_3 = 1 \Rightarrow T_{z3} = C_3^{(2)} / \alpha_3 = 9 \\ \alpha_4 = 1/2 \Rightarrow T_{z4} = C_4 / \alpha_4 = 6 \\ \alpha_5 = 1/2 \Rightarrow T_{z5} = C_5 / \alpha_5 = 6 \end{array} \right\} \Rightarrow t_4 = t_3 + \min\{T_{z3}, T_{z4}, T_{z5}\} = 3 + 6 = 9$$

4. $t_4 = 9$

Pripravni zadatak: $T_3(C_3^{(3)} = 3)$

$$\alpha_3 = 1 \Rightarrow T_{z3} = C_3^{(3)} / \alpha_3 = 3 \Rightarrow t_5 = t_4 + T_{z3} = 12$$

5. $t_5 = 12$

Pripravni zadatak: $T_6(C_6 = 1)$

$$\alpha_6 = 1 \Rightarrow T_{z6} = C_6 / \alpha_6 = 1 \Rightarrow t_6 = t_5 + T_{z6} = 13$$

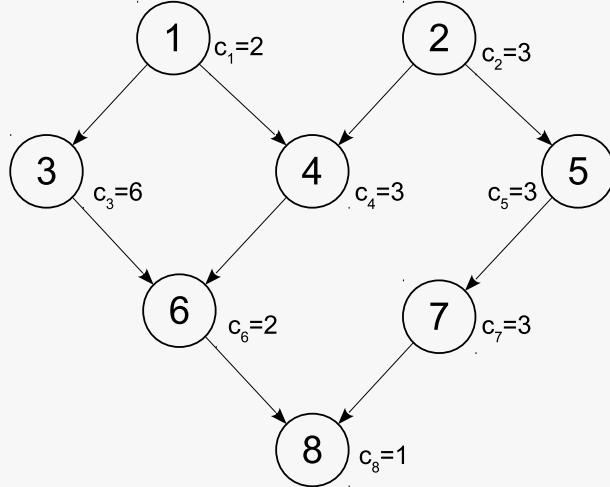
6. $t_6 = 13$

Sustav je prema općem raspoređivanju gotov u $t = 13$.

Zbog neuravnoteženosti u ovom primjeru jedan od procesora miruje i usred izvođenja sustava, a ne samo na početku i/ili kraju kao što je to uobičajeno.

Primjer 4.14.

Sustav zadataka zadan je necikličkim računalnim grafom na slici 4.21. Napraviti raspoređivanje sustava zadataka korištenjem postupka općeg raspoređivanja za dva procesora.



Slika 4.21.

$$1. t_1 = 0$$

Pripravni zadaci: $\mathcal{T}_1(C_1 = 2), \mathcal{T}_2(C_2 = 3)$

$$\left. \begin{array}{l} \alpha_1 = 1 \Rightarrow T_{z1} = C_1/\alpha_1 = 2 \\ \alpha_2 = 1 \Rightarrow T_{z2} = C_2/\alpha_2 = 3 \end{array} \right\} \Rightarrow t_2 = t_1 + \min\{T_{z1}, T_{z2}\} = 2$$

$$2. t_2 = 2$$

Pripravni zadaci: $\mathcal{T}_2(C_2^{(2)} = 1), \mathcal{T}_3(C_3 = 6)$

$$\left. \begin{array}{l} \alpha_2 = 1 \Rightarrow T_{z2} = C_2^{(2)}/\alpha_2 = 1 \\ \alpha_3 = 1 \Rightarrow T_{z3} = C_3/\alpha_3 = 6 \end{array} \right\} \Rightarrow t_3 = t_2 + \min\{T_{z2}, T_{z3}\} = 3$$

$$3. t_3 = 3$$

Pripravni zadaci: $\mathcal{T}_3(C_3^{(3)} = 5), \mathcal{T}_4(C_4 = 3), \mathcal{T}_5(C_5 = 3)$

Supstitucija:

$$\mathcal{T}_{\sigma_1} = \{\mathcal{T}_3, \mathcal{T}_4, \mathcal{T}_6\}$$

$$C_{\sigma_1} = C_3^{(3)} + C_4 + C_6 = 5 + 3 + 2 = 10$$

$$\mathcal{T}_{\sigma_2} = \{\mathcal{T}_5, \mathcal{T}_7\}$$

$$C_{\sigma_2} = C_5 + C_7 = 3 + 3 = 6$$

$$\Delta t = \frac{C_{\sigma_1} + C_{\sigma_2}}{2} = 8$$

$$\alpha_{\sigma_1} = C_{\sigma_1}/\Delta t = 10/8 = 5/4$$

$$\alpha_{\sigma_2} = C_{\sigma_2}/\Delta t = 6/8 = 3/4$$

Iako je $\alpha_{\sigma_1} > 1$ to će se u početku dijelom moći iskoristiti obzirom da se \mathcal{T}_{σ_1} sastoji od dva zadatka koji mogu paralelno.

$$\begin{aligned}\Delta t^{(1)} &= \frac{C_3^{(3)} + C_4}{\alpha_{\sigma_1}} = \frac{5+3}{5/4} = 32/5 \\ \alpha_3 &= C_3^{(3)}/\Delta t^{(1)} = 5/(32/5) = 25/32 \\ \alpha_4 &= C_4/\Delta t^{(1)} = 3/(32/5) = 15/32 \\ T_{z3} &= C_3^{(3)}/\alpha_3 = 5/(25/32) = 32/5 \\ T_{z4} &= C_4/\alpha_4 = 3/(15/32) = 32/5 \\ t_{z3} &= t_3 + 32/5 = 3 + 6,4 = 9,4\end{aligned}$$

Od $t = 9,4$ na dalje bi zadatak \mathcal{T}_6 sam imao na raspolaganju $\alpha_{\sigma_1} = 5/4$ što ne može iskoristiti. Ako drugi podsustav zadataka nije gotov, ovdje se može ponovno napraviti preraspodijela.

$$\begin{aligned}\alpha_5 &= \alpha_{\sigma_2} = 3/4 \\ T_{z5} &= C_5/\alpha_5 = 3/(3/4) = 4 \\ t_{z5} &= t_3 + T_{z5} = 3 + 4 = 7\end{aligned}$$

U intervalu $[7; 9,4]$ zadatak \mathcal{T}_7 koristit će i dalje isti $\alpha_7 = \alpha_{\sigma_2}$.

$$\begin{aligned}\alpha_7 &= \alpha_{\sigma_2} = 3/4 \\ \Delta t^{(2)} &= 9,4 - 7 = 2,4 \\ C_7^{(1)} &= \Delta t^{(2)} \cdot \alpha_7 = 2,4 \cdot (3/4) = 1,8 \\ C_7^{(2)} &= C_7 - C_7^{(1)} = 3 - 1,8 = 1,2\end{aligned}$$

4. $t_4 = 9,4$

Pripravni zadaci: $\mathcal{T}_6(C_6 = 2), \mathcal{T}_7(C_7^{(2)} = 1,2)$

$$\left. \begin{array}{l} \alpha_6 = 1 \Rightarrow T_{z6} = C_6/\alpha_6 = 2 \\ \alpha_7 = 1 \Rightarrow T_{z7} = C_7^{(2)}/\alpha_7 = 1,2 \end{array} \right\} \Rightarrow t_5 = t_4 + \min\{T_{z6}, T_{z7}\} = 10,6$$

5. $t_5 = 10,6$

Pripravni zadatak: $\mathcal{T}_6(C_6^{(2)} = 2 - 1,2 = 0,8)$

$$\alpha_6 = 1 \Rightarrow T_{z6} = C_6^{(2)}/\alpha_6 = 0,8 \Rightarrow t_6 = t_5 + T_{z6} = 11,4$$

6. $t_6 = 11,4$

Pripravni zadatak: $\mathcal{T}_8(C_8 = 1)$

$$\alpha_8 = 1 \Rightarrow T_{z8} = C_8/\alpha_8 = 1 \Rightarrow t_7 = t_6 + T_{z8} = 12,4$$

7. $t_7 = 12,4$

Sustav zadataka završava u $t_7 = 12,4$.

U ovom primjeru zbog pojave neuravnoteženosti u zadacima u dijelu postupka, dobiveno rješenje nije potpuno optimalno (kada bi ručno optimirali mogli bi posložiti zadatke tako da završe do 12. jedinice vremena). Međutim, ovo nije svojstvo općeg raspoređivanja, već pojednostavljenog algoritma za njegovo ostvarenje koje je korišteno u ovim primjerima.

4.6.1.2. Postupak sa stablenom strukturom

Postupak raspoređivanja sa stablenom strukturom jednostavan je grafički postupak primjenjiv na sustave zadataka kod kojih su ovisnosti prikazane grafom koji ima stablenu strukturu, s time da su početni zadaci u listovima stabla, a završni zadatak predstavlja korijen. Svaki zadatak u takvom sustavu ima najviše jednog neposrednog slijednika, osim završnog zadataka koji nema slijednika.

Izrada rasporeda kreće od kraja, od trenutka kada je zadnji zadatak (korijen) završen. Od tog trenutka kreće se unatrag, do trenutka kad on treba započeti a njegovi prethodnici završiti. Isti se postupak dalje primjenjuje za svaki od njegovih prethodnika, pa njihovih prethodnika, sve do listova.

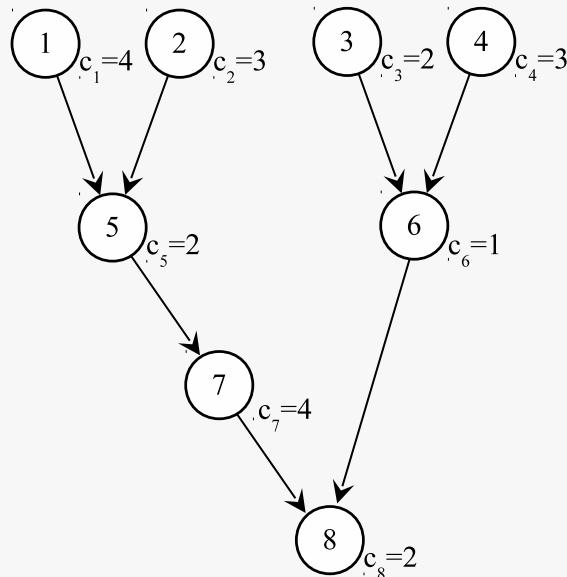
Grafički postupak ne uzima u obzir broj raspoloživih procesora, odnosno pretpostavlja da ih ima dovoljno da se svi paralelni zadaci mogu paralelno izvoditi. Logička visina stabla u tom slučaju predstavlja duljinu najduljeg puta (zbroj trajanja zadataka na putu od lista do korjena).

Stvarno trajanje izvođenja sustava zadataka raspoređenog ovim postupkom može biti dulje. Postupak, slično kao i opće raspoređivanje, pretpostavlja da će svi paralelni zadaci dijeliti raspoloživu procesorsku snagu u istim omjerima (uz ograničenje $\alpha_i \leq 1$). Za zadani broj procesora stoga se može odrediti "stvarna visina" grafa – trajanje pojedinih segmenata, kao i izvođenje cijelog sustava.

Postupak neće uvijek stvoriti optimalno rješenje kao opće raspoređivanje, ali je zato mnogo jednostavniji.

Primjer 4.15.

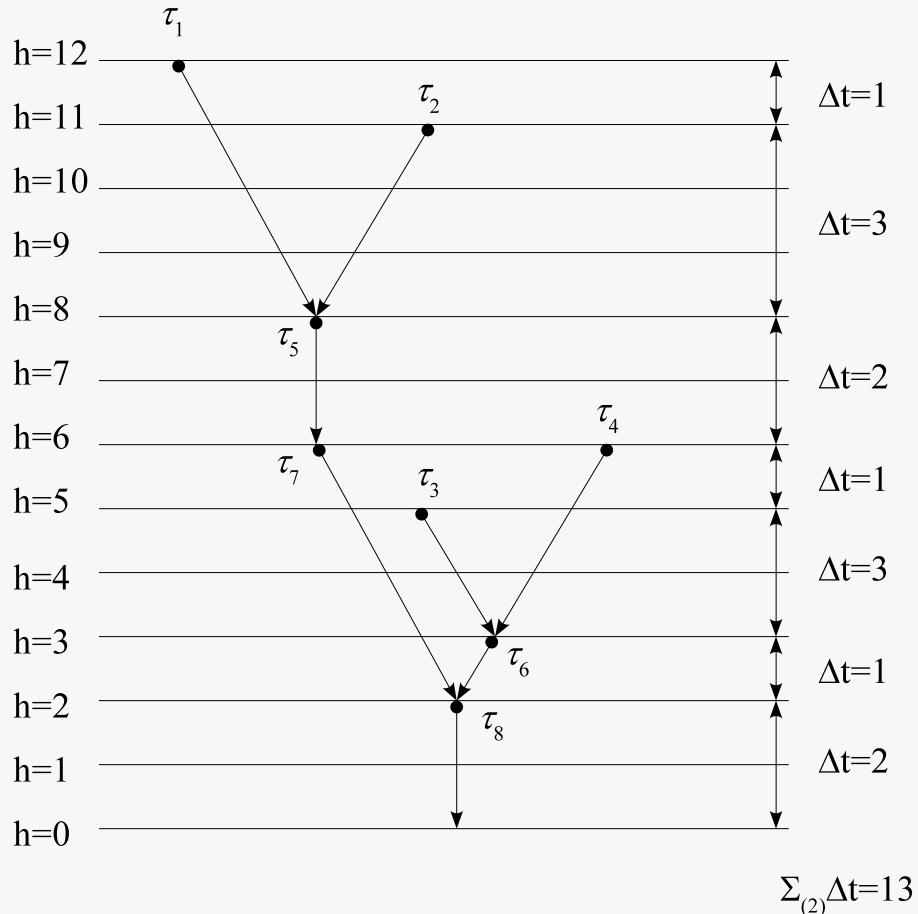
Sustav zadataka zadan je stablenom strukturom prema slici 4.22.



Slika 4.22.

Rješenje raspoređivanja korištenjem postupka sa stablenom strukturom je prikazano slikom 4.23. U rješenju se kreće od trenutka kada su svi zadaci obavljeni do kraja te se vraća prema gore (u stablu). Prvi zadatak kojeg se ucrtava je zadatak T_8 . On traje 2 jedinice vremena. Prije nego li T_8 krene moraju biti gotovi i T_6 i T_7 (te njih crtamo povrh T_8). T_6 traje jednu jedinicu vremena, a T_7 četiri. Sva se ta trajanja koriste pri crtanjtu grafa. S lijeve strane

su označene logičke visine stabla u jedinicama trajanja. Istim postupkom nacrtaju se i ostale grane, sve do listova. Rezultat postupka daje trenutke kada treba pokrenuti pojedini zadatak.



Slika 4.23.

Razmotrimo raspoređivanje na dvoprocesorskom računalu.

Prema prethodnoj slici, samo se u segmentu od $h = 3$ do $h = 5$ pojavljuju tri zadatka (po ovakvom raspoređivanju) te je trajanje izvođenja tog segmenta dulje od visine segmenta: $\Delta t = \Delta h \cdot 3/2 = 2 \cdot 3/2 = 3$.

Ukupno trajanje izvođenja sustava zadataka raspoređenih zadanim postupkom dobiva se zbrajanjem trajanja svih segmenata. Za ovaj primjer ukupno trajanje iznosi 13 jedinica vremena.

Analizom grafa, mogli bi doći do nekih opažanja. Primjerice, iako su početno četiri zadatka neovisna, samo se jedan, T_1 , odabire za izvođenje. Kada bi sve pustili, to ne bi dalo bolje rješenje. Međutim, zadatak T_2 bi mogao krenuti istovremeno, ali će tada završiti prije prvog zadatka.

Ipak, kada bi naknadno "ručno" popravljali ovaj raspored, mogli bi T_3 pokrenuti zajedno sa T_5 te bi tada u svakom segmentu imali najviše dva zadatka i sustav bi mogao biti gotov jednu jedinicu prije. Međutim, "ručno" popravljanje je moguće samo za jednostavne sustave. S druge strane, prikazani postupak raspoređivanja sa stabilnom strukturu može se ostvariti jednostavnim programom i primijeniti na složene sustave zadataka (uz pretpostavku stablaste strukture).

4.6.2. Višeprocesorsko dinamičko raspoređivanje

Postupci dinamičkog raspoređivanja ostvaruju svoju učinkovitost oslanjanjem na relativno jednostavne kriterije koji su tipično vezani na neka obilježja zadataka, kao što su krajnji trenutak završetka (engl. *deadline*) ili hitnost izvođenja zadataka. Cilj im je izbjegći računalno skupo optimiranje rasporeda, tako da sustav može brzo odgovoriti na promjene u okolini. Tome može uvelike pomoći jasno sagledavanje ponašanja okoline, uvid u strategiju planiranja i puno uvažavanje svih ograničenja.

Dinamičko raspoređivanje prema krajnjim trenucima zadataka je identično kao i na jednoprocesorskim sustavima i već je ilustrirano primjerom 4.11. te se neće dodatno analizirati u osnovnom obliku. Međutim, poznавanje t_{kz} zadataka te preostalog vremena potrebnog za izvođenje zadataka $c(t)$ može se iskoristiti za drugi kriterij – kriterij hitnosti započinjanja zadataka, odnosno, koliko se dugo zadatak može još odgoditi prije početka izvođenja, a da se ipak stigne izvesti do kraja prije trenutka t_{kz} . Kriterij se naziva po mogućoj odgodi zadataka – kriterij labavosti.

4.6.2.1. Labavost

Labavost (engl. *slack time*) označava vrijeme koje zadatak može provesti bez da se izvodi, a da se ipak kasnije stigne obaviti prije svog krajnjeg trenutka završetka.

Definicija 4.19. Labavost

Neka je zadatak \mathcal{T} u trenutku t zadan sa $\mathcal{T}(t) = \{C(t), t_{kz}\}$, gdje $C(t)$ predstavlja preostalo potrebno procesorsko vrijeme za završetak zadataka te t_{kz} trenutak krajnjeg dovršetka. Labavost zadataka $\ell(t)$ definira se prema:

$$\ell(t) = t_{kz} - t - C(t) \quad (4.10.)$$

Veličina $\bar{d}(t) = t_{kz} - t$ predstavlja vrijeme do krajnjeg trenutka završetka zadataka (“relativan t_{kz} ”). Preko $\bar{d}(t)$ labavost se definira sa:

$$\ell(t) = \bar{d}(t) - C(t) \quad (4.11.)$$

Labavost se može uzeti kao mjera hitnosti pokretanja zadataka. Što je labavost veća, hitnost je manja, pa se pri nekom raspoređivanju mogu uzimati najprije zadaci najveće hitnosti.

Raspoređivanje prema najmanjoj labavosti (NL, *least laxity first – LLF*) koristi informaciju o krajnjim trenucima završetaka zadataka, ali i o potrebnim vremenima izvođenja zadataka. Obzirom na tu dodatnu informaciju postupak može dati bolji raspored od samog postupka prema krajnjim trenucima završetaka.

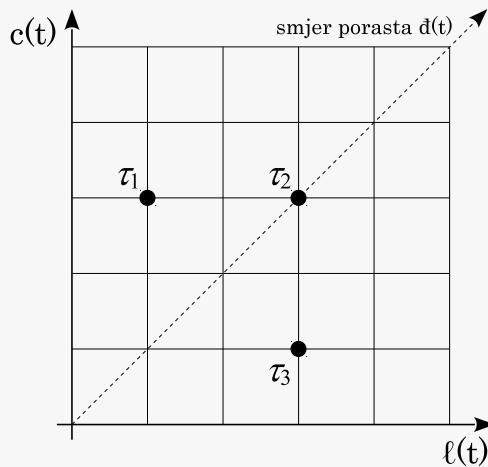
Prema trenutnoj vrijednosti labavosti mogu se donijeti neki zaključci o stanju zadataka:

- $\ell(t) > 0$ – postoji dovoljno vremena za izvođenje zadataka, nije neophodno odmah započeti s njegovim izvođenjem, već se početak još može odgoditi za najviše $\ell(t)$
- $\ell(t) = 0$ – zadatak treba odmah započeti s izvođenjem i ne prekidati izvođenje do njegova završetka, inače se neće stići obaviti do t_{kz}
- $\ell(t) < 0$ – zadatak se više sigurno ne stigne obaviti do t_{kz} (koji je možda već i prošao).

Raspoređivanje primjenom labavosti može se prikazati grafički u ℓ/c prostoru. Zadatak se u grafu ucrtava u trenutku kada se zadatak javlja u sustavu. Labavost se računa prema definiciji 4.19. te se zadatak ucrtava u graf s vrijednostima $(\ell(t), c(t))$.

Primjer 4.16.

Slika 4.24. prikazuje primjer ℓ/c grafa s tri zadatka u trenutku t .



Slika 4.24. ℓ/c graf u trenutku t

Labavost, preostalo vrijeme računanja te vrijeme do krajnjeg trenutka završetka u trenutku t može se očitati i izračunati iz grafa sa slike 4.24. prema:

$$\begin{aligned}\mathcal{T}_1 : \quad C_1(t) &= 3, \quad \ell_1(t) = 1, \quad \bar{d}_1(t) = \ell_1(t) + C_1(t) = 4 \\ \mathcal{T}_2 : \quad C_2(t) &= 3, \quad \ell_2(t) = 3, \quad \bar{d}_2(t) = \ell_2(t) + C_2(t) = 6 \\ \mathcal{T}_3 : \quad C_3(t) &= 1, \quad \ell_3(t) = 3, \quad \bar{d}_3(t) = \ell_3(t) + C_3(t) = 4\end{aligned}$$

Raspoređivač prema labavosti bi prvo odabrao zadatak \mathcal{T}_1 obzirom da je njegova labavost najmanja. Nakon njega bi odabir između \mathcal{T}_2 i \mathcal{T}_3 bio proizvoljan obzirom da je njihova labavost jednaka, ili bi se koristio sekundarni kriterij (primjerice, prema krajnjim trenucima završetka).

Raspoređivač prema krajnjim trenucima završetka bi u početku odabrao \mathcal{T}_1 ili \mathcal{T}_3 obzirom da imaju istu vrijednost \bar{d} (vrijeme do krajnjeg trenutka završetka).

4.6.2.2. Zalihost računalne snage

Uzimajući u obzir labavost pojedinih zadataka iz skupa zadataka, moguće je izračunati zalihost računalne snage za budući interval vremena, tj. koliko se procesorske snage može odvojiti za neke druge poslove a da se razmatrani skup zadataka ipak stigne obaviti do kraja uz poštivanje svih vremenskih ograničenja.

Definicija 4.20. Zalihost računalne snage – $F(x, t)$

U trenutku t zalihost računalne snage u sustavu s m procesora za budući period $[t, t + x]$ iznosi:

$$F(x, t) = m \cdot x - \sum_{\mathcal{T} \in R_1} C_{\mathcal{T}}(t) - \sum_{\mathcal{T} \in R_2} (x - \ell_{\mathcal{T}}(t))$$

gdje R_1 predstavlja skup zadataka koji moraju biti gotovi do trenutka $t + x$, tj.

$$\bar{d}_{\mathcal{T}}(t) = C_{\mathcal{T}}(t) + \ell_{\mathcal{T}}(t) \leq x \quad \forall \mathcal{T} \in R_1$$

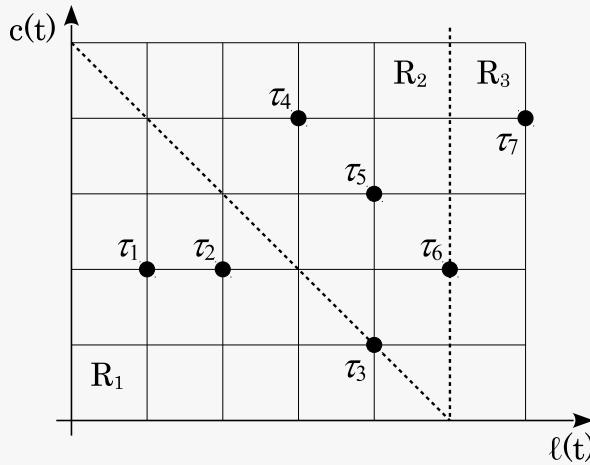
dok R_2 predstavlja skup zadataka koji trebaju djelomično obaviti svoj posao u intervalu $[t, t + x]$, a da bi stigli obaviti završiti prije svog t_{kz} , tj.

$$\bar{d}_{\mathcal{T}}(t) > x \quad \wedge \quad \ell_{\mathcal{T}}(t) \leq x, \quad \forall \mathcal{T} \in R_2$$

Definicija 4.20. kaže da se pri računanju zalihosti za budući period duljine x , od raspoložive računalne snage $m \cdot x$ treba oduzeti vrijeme potrebno za dovršetak svih zadataka koji moraju biti gotovi do $t + x$, ali također i dio vremena za zadatke čija je labavost manja od x (oni mogu propustiti najviše ℓ svog vremena u idućih x jedinica vremena).

Primjer 4.17. Zalihost računalne snage

Slika 4.25. prikazuje primjer ℓ/c grafa s kojeg se može izračunati zalihost računalne snage za interval $[t; t + 5]$.



Slika 4.25. Podjela ℓ/c grafa u područaja pri određivanju zalihosti

Zadaci $\{\tau_1, \tau_2, \tau_3\}$ spadaju u skup R_1 , zadaci $\{\tau_4, \tau_5, \tau_6\}$ u skup R_2 te $\{\tau_7\}$ spada u skup R_3 . Zadaci na granicama područja mogu se pripojiti jednom ili drugom području – jednako utječe na zalihost.

$$\begin{aligned} F(t, 5) &= m \cdot 5 - (C_1(t) + C_2(t) + C_3(t)) - ((5 - \ell_4) + (5 - \ell_5) + (5 - \ell_6)) \\ &= m \cdot 5 - (2 + 2 + 1) - ((5 - 3) + (5 - 4) + (5 - 5)) \\ &= m \cdot 5 - 5 - 3 = m \cdot 5 - 8 \end{aligned}$$

Primjerice, za dvoprocesorski sustav zalihost iznosi: $F(t, 5) = 2 \cdot 5 - 8 = 2$.

4.6.2.3. Raspoređivanje prema labavosti i trenucima krajnjih dovršetaka

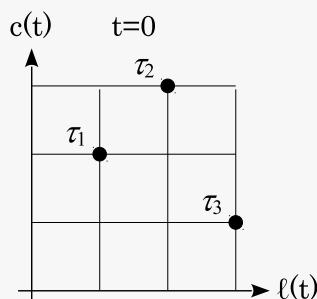
Za sustave za rad u stvarnom vremenu raspoređivanje prema trenucima krajnjih dovršetaka se pokazuje, bar teoretski, kao jedan od najboljih postupaka. Raspoređivanje korištenjem najmanje labavosti implicitno koristi trenutke krajnjih dovršetaka te također spada u istu grupu teoretski najboljih raspoređivača.

Za razliku od raspoređivača preka krajnjim trenucima zadataka koji se treba koristiti u trenucima promjena u sustavu (promjena u pripravnim zadacima), raspoređivač koji koristi labavost bi se trebao pozivati i samim protokom vremena jer se labavost vremenom smanjuje. Ipak, u sljedećim zadacima pretpostavlja se da se raspoređivač izvodi periodički sa zadanim intervalom između dva poziva. Naime, ostvarenja takvog raspoređivača u okviru nekog operacijskog sustava bi također trebala uzeti neki period pozivanja raspoređivača.

Rad oba postupka na istom problemu raspoređivanja korištenjem grafičkog postupka preko ℓ/c grafa prikazan je primjerom 4.18.

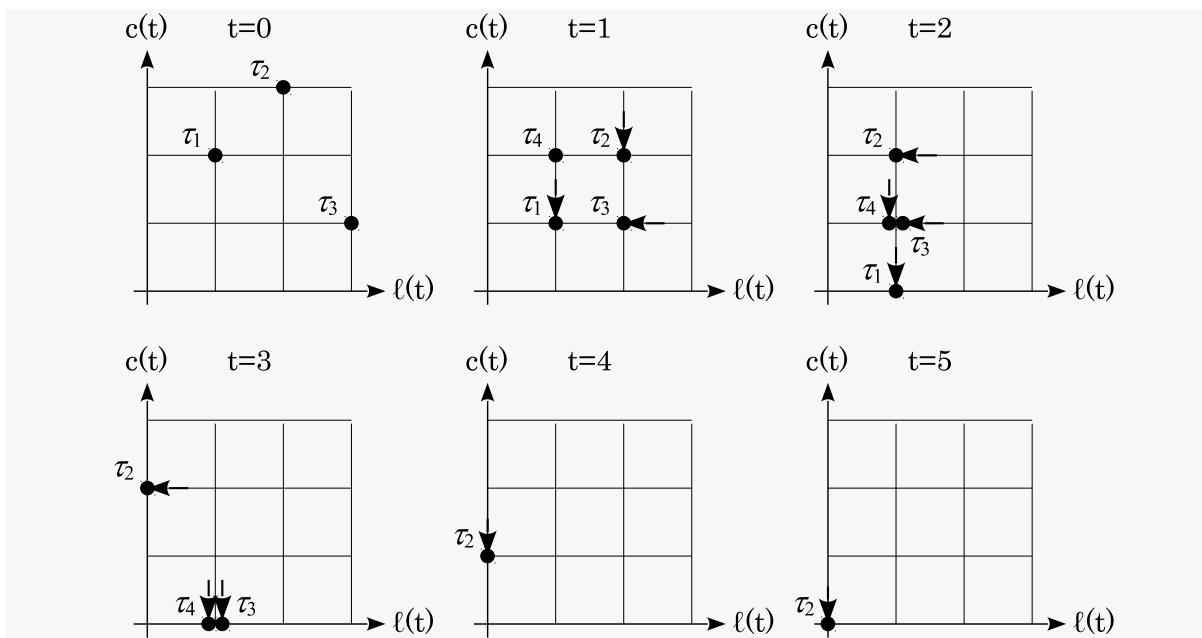
Primjer 4.18.

Sustav zadataka u trenutku $t = 0$ zadan je ℓ/c grafom na slici 4.26. Dodatno je poznato da će se u trenutku $t = 1$ pojaviti još jedan zadatak s potrebnim vremenom računanja od 2 jedinice koji mora biti gotov do $t = 4$. Prikazati rad raspoređivača prema najmanjoj labavosti kao primarnom kriteriju i prema krajnjim trenucima završetaka kao sekundarnom, i obratno. Na raspolaganju je dvoprocesorsko računalo a raspoređivač se poziva nakon svake jedinice vremena.



Slika 4.26. Početno stanje sustava zadataka

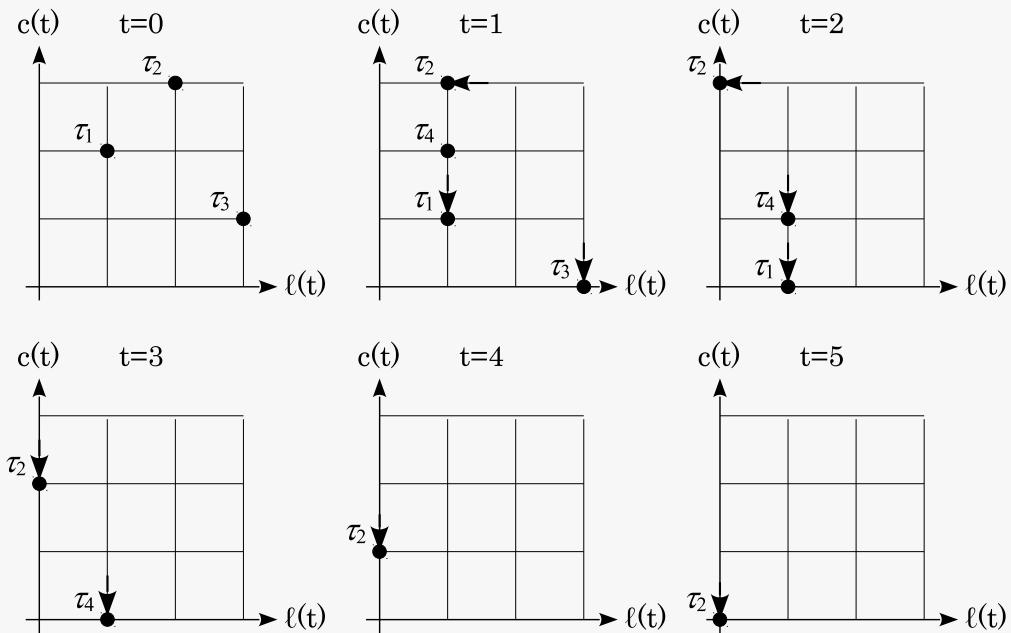
Raspoređivanje prema najmanjoj labavosti za izvođenje najprije odabire zadatke koji su na ℓ/c grafu najbliže osi $c(t)$. Kada više zadataka ima istu labavost, tada se odabir mora napraviti po dodatnom kriteriju. U ovom primjeru dodatni kriterij jest trenutak krajnjeg dovršetka. U grafičkom postupku simulacije rada sustava na ℓ/c grafu, pomak zadataka u prethodnom intervalu prikazan je strelicama. Zadaci odabrani za izvođenje na grafu će se spustiti po vertikalnoj osi, dok će se svi ostali pomakuti lijevo (jedni su se izvodili i smanjili svoj c dok su drugi smanjili svoju labavost).



Slika 4.27. Raspoređivanje prema najmanjoj labavosti

U trenutku $t = 1$ u sustavu, pa time i na grafu pojavljuje se novi zadatak τ_4 . Obzirom da on treba završiti za 3 jedinice vremena od kojih su mu dvije potrebne na procesoru, labavost mu je 1.

Raspoređivanje prema trenucima krajnjih završetaka je slično, jedino je odabir zadataka drukčiji – prvo se biraju zadaci s najbližim trenucima krajnjih završetaka koji su na grafu najbliži pravcu $c = -\ell$. Slika 4.24. prikazuje smjer porasta trenutaka krajnjih završetaka.



Slika 4.28. Raspoređivanje prema trenucima krajnjih dovršetaka

Usporedbom oba postupka raspoređivanja na zadanom primjeru ne uočava se razlika u kvaliteti raspoređivanja.

4.7. Postupci raspoređivanja i njihova optimalnost

Optimalnost postupaka raspoređivanja može se promatrati s nekoliko raznih gledišta. Neki od njih su:

1. zadovoljavanje vremenskih ograničenja zadataka
2. jednostavnost postupka:
 - radi mogućnosti ostvarenja u stvarnim sustavima
 - radi smanjenja procesne snage utrošene na odlučivanje o raspoređivanju i kućanske poslove raspoređivača
 - radi malog broja potrebnih obilježja zadataka koji su dostupni unaprijed ili za vrijeme izvođenja
3. optimiranje dovršetka skupine zavisnih zadataka.

4.7.1. Optimalnost prema izvedivosti raspoređivanja

Optimalnost postupaka za raspoređivanje zadataka odnosi se na prednosti jednih u odnosu na druge, unutar iste klase postupaka. Usaporedba postupaka različitih klasa često nema smisla obzirom da one mogu tražiti poznavanje različitih informacija o zadacima. Postupci raspoređivanja razmatrani u ovom poglavlju mogu se podijeliti u nekoliko klasa:

1. postupci za jednoprocесorske sustave sa statičkom dodjelom prioriteta
2. dinamički postupci za jednoprocесorske sustave
3. dinamički postupci za višeprocесorske sustave.

Definicija 4.21. Optimalan postupak raspoređivanja

Za postupak raspoređivanja se može reći da je optimalan ako se njime može rasporediti bilo koji skup zadataka koji se može rasporediti nekim drugim postupkom iste klase.

Nužan uvjet za rasporedivost sustava zadataka na jednoprocесorskom sustavu jest da ukupni faktor iskorištenja koji stvaraju ti zadaci bude manji od jedan (definicija 4.3.). Obzirom da je isti uvjet dostatan i za raspoređivanje prema krajnjim trenucima završetaka proizlazi da je taj postupak optimalan za jednoprocесorske sustave. Isto vrijedi i za postupak prema najmanjoj labavosti (koji nije zasebno razmatran za jednoprocесorske sustave).

Postupak mjere ponavljanja optimalan je postupak raspoređivanja među postupcima s statičkom pridjelom prioriteta zadacima na jednoprocесorskim sustavima (dokaz u [Liu, 1973]).

Optimalnost glede izvodljivosti raspoređivanja na višeprocесorskim sustavima je mnogo složenija. O samoj izvodljivosti raspoređivanja (najboljim mogućim postupkom) moglo bi se zaključiti prema zalihosti procesne snage.

Definicija 4.22. Izvodljivost raspoređivanja

Nužan i dovoljan uvjet za izvodljivost raspoređivanja sustava zadataka (optimalnim postupkom) jest:

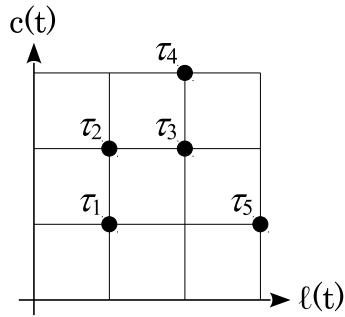
$$F(x, t) \geq 0, \quad \forall x > 0$$

gdje $F(x, t)$ predstavlja zalihost računalne snage u intervalu $[t, t + x]$ koji se izračunava prema definiciji 4.20.

Definicija 4.22. poopćuje nužni uvjet rasporedivosti iz definicije 4.3. (koji vrijedi samo za jednoprocесorske sustave) na višeprocесorske sustave. Postavlja se pitanje postoji li postupak

raspoređivanja koji će za svaki sustav zadataka koji zadovoljava definiciju 4.22. dati zadovoljiv raspored?

Za razliku od jednoprocesorskih sustava, postupak prema krajnjim trenucima završetaka nije optimalan za višeprocesorske sustave. Slika 4.29. prikazuje primjer sustava u ℓ/c prostoru koji postupak prema krajnjim trenucima završetaka ne uspije rasporediti dok postupak prema najmanjoj labavosti to uspijeva.



Slika 4.29. Primjer sustava gdje KTZ ne uspijeva a NL uspijeva rasporediti sustav zadataka

Je li onda možda NL optimalan algoritam? Da bi dobili negativan odgovor dovoljno je pronaći neki sustav koji jest rasporediv nekom metodom, ali ga NL ne uspijeva rasporediti. Jedan takav sustav prikazuje i primjer 4.19.

Dodatni problemi nastaju ako svi parametri potrebni za raspoređivanje nisu unaprijed poznati. Slično je i s problemom nepoznavanja trenutaka pojave zadataka u budućnosti. Navedeni problemi su vrlo česti u stvarnim sustavima. Postupak koji će na osnovu minimalnih informacija napraviti optimalan raspored ne postoji, barem prema sadašnjim saznanjima (a teško da će se to ikad promijeniti). Od prikazanih algoritama, postupak prema najmanjoj labavosti je najbolji, ali on traži puno podataka o svakom zadatku: njegov krajnji trenutak završetka i vrijeme računanja unutar jedne pojave.

Problem raspoređivanja je još uvijek aktualan problem, i nakon pola stoljeća istraživanja. Opće rješenje, "najbolje za sve" još nije pronađeno, ali su zato predložena mnoga druga rješenja koja "dovoljno dobro" (najbolje do sada) rješavaju probleme u pojedinim okruženjima. U stvarnim sustavima, optimalan raspored je gotovo nemoguće postići. Zato se traži najbolji mogući raspored koji praktični algoritmi mogu izračunati u ograničenom (konačnom) vremenu. Ti su algoritmi uglavnom zasnovani na raznim heuristikama ili kombinatoričkom pretraživanju prostora (npr. genetski algoritmi).

U praksi se često kao najjednostavnije rješenje uzimaju prioriteti koji se dodijeljuju zadacima iskustvom arhitekta sustava (koji uzima u obzir mnoga obilježja zadataka, od mjere ponavljanja, bitnosti za sustav u kojem sudjeluju i slično).

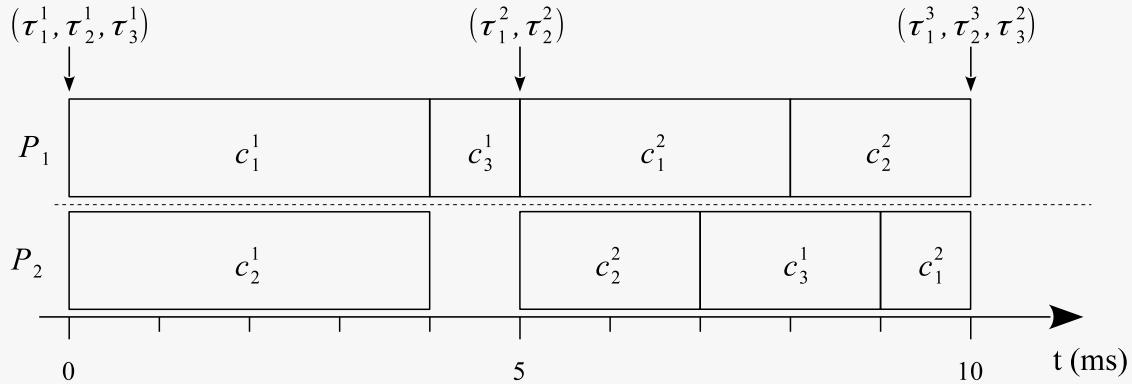
Primjer 4.19. Sustav koji NL ne može rasporediti

Neka je zadan sustav periodičkih zadataka:

$$\mathcal{T}_1 : T_1 = 5 \text{ ms}, \quad C_1 = 4 \text{ ms}$$

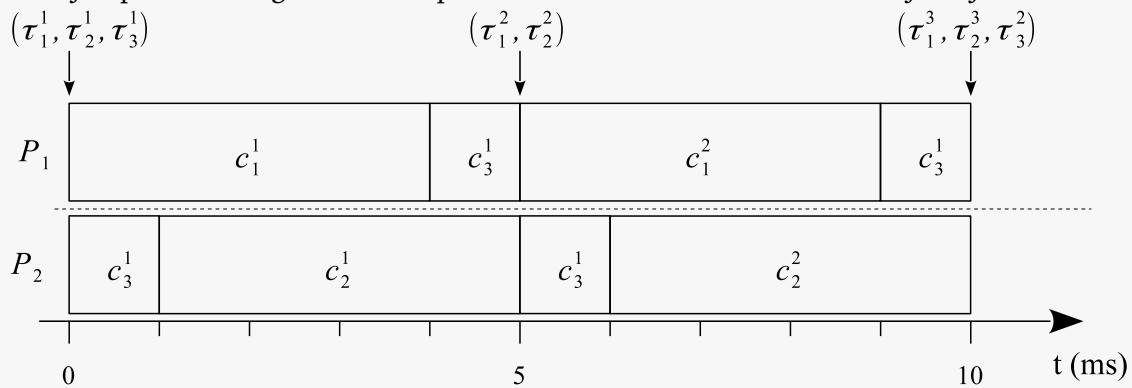
$$\mathcal{T}_2 : T_2 = 5 \text{ ms}, \quad C_2 = 4 \text{ ms}$$

$$\mathcal{T}_3 : T_3 = 10 \text{ ms}, \quad C_3 = 4 \text{ ms}$$



Slika 4.30. NL raspoređivanje uz indeks zadatka kao drugi kriterij

NL (s bilo kojim sekundarnim kriterijem) neće uspjeti rasporediti sustav jer \mathcal{T}_3 neće dobiti dovoljno procesorskog vremena u prvih 10 ms obzirom da tamo ima najmanju labavost.



Slika 4.31. Raspored (ručni) koji zadovoljava sva ograničenja

Pitanja za vježbu 4

1. Navesti razloge korištenja višedretvenosti općenito te u SRSV-ima. ◁
2. Koji su osnovni problemi pri korištenju višedretvenosti?
3. Svaki pojedinačni zadatak ima neka vremenska svojstva. Koja? Kako ona utječu na korisnost zadatka te koje zahtjeve postavljaju prema operacijskom sustavu? ◁
4. Opisati opća svojstva statičkih i dinamičkih postupaka raspoređivanja.
5. Što je to "procesorska iskoristivost" te što se preko nje može zaključiti o rasporedivosti sustava zadataka?
6. Opisati raspoređivanje mjerom ponavljanja (engl. *rate monotonic scheduling*). ◁
7. Opisati korištenje "općeg kriterija rasporedivosti" za provjeru rasporedivosti sustava zadataka kada se koristi raspoređivanje mjerom ponavljanja.
8. Što je to "implicitni trenutak krajnjeg završetka" nekog zadatka?
9. U kontekstu raspoređivanja periodičkih zadataka objasniti razliku između krajnjeg trenutka završetka i implicitna krajnjeg trenutka završetka.
10. Kada se za neki sustav zadataka kaže da "u potpunosti iskorištava procesor" (u kontekstu raspoređivanja tih zadataka nekim postupkom)?
11. Što je to "najmanja gornja granica procesorskog iskorištenja" lub(U)? Koji se zaključci mogu donijeti za sustave za koje vrijedi $U_S \leq \text{lub}(U)$, odnosno, za one koje ista nejednakost ne vrijedi?
12. Opisati raspoređivanje korištenjem postupka prema krajnjim trenucima završetka zadataka (engl. *earliest deadline first*). Navesti dobra i loša svojstva tog postupka. ◁
13. Koje kriterije optimiranja koristi "opće raspoređivanje"?
14. Što je to labavost (u kontekstu zadatka i njegovog raspoređivanja)? Opisati raspoređivanje prema najmanjoj labavosti (engl. *least laxity first*)? ◁
15. Što je to zalihost računalne snage (u kontekstu raspoređivanja)?
16. Koji je nužan i dovoljan uvjet za rasporedivost sustava zadataka na višeprocesorskom sustavu?
17. Za koji postupak raspoređivanja kažemo da je optimalan (u svojoj klasi)?
18. Korištenjem raspoređivanja prema mjeri ponavljanja provjeriti rasporedivost sustava zadataka na jednoprocesorskom sustavu korištenjem:
 - a) procesorske iskoristivosti (nužni uvjet, lub)
 - b) simulacijom (grafičkim postupkom)
 - c) odrediti implicitne trenutke krajnjeg završetka za sve zadatke.

$$\mathcal{T}_1 : T_1 = 10 \text{ ms}, \quad C_1 = 3 \text{ ms}$$

$$\mathcal{T}_2 : T_2 = 15 \text{ ms}, \quad C_2 = 3 \text{ ms}$$

$$\mathcal{T}_3 : T_3 = 20 \text{ ms}, \quad C_3 = 4 \text{ ms}$$

$$\mathcal{T}_4 : T_4 = 30 \text{ ms}, \quad C_4 = 6 \text{ ms}$$

Je li navedeni sustav rasporediv korištenjem raspoređivanja prema krajnjim trenucima završetaka? Pokazati rad tog raspoređivača nad navedenim sustavom zadataka.

19. Korištenjem raspoređivanja prema krajnjim trenucima završetaka simulacijom raspore-

divanja provjeriti rasporedivost sustava zadatka na dvoprocesorskom sustavu.

$$\begin{aligned}\mathcal{T}_1 : \quad T_1 &= 10 \text{ ms}, & C_1 &= 6 \text{ ms} \\ \mathcal{T}_2 : \quad T_2 &= 15 \text{ ms}, & C_2 &= 9 \text{ ms} \\ \mathcal{T}_3 : \quad T_3 &= 20 \text{ ms}, & C_3 &= 10 \text{ ms} \\ \mathcal{T}_4 : \quad T_4 &= 30 \text{ ms}, & C_4 &= 3 \text{ ms}\end{aligned}$$

20. Bez korištenja simulacije (ostalim postupcima: nužni uvjet, lub, ...) ispitati rasporedivost sustava zadatka na jednoprocesorskom sustavu ako se koristi:
- a) raspoređivanje mjerom ponavljanja
 - b) raspoređivanje prema krajnjim trenucima završetaka.

$$\begin{aligned}\mathcal{T}_1 : \quad T_1 &= 10 \text{ ms}, & C_1 &= 2 \text{ ms} \\ \mathcal{T}_2 : \quad T_2 &= 15 \text{ ms}, & C_2 &= 3 \text{ ms} \\ \mathcal{T}_3 : \quad T_3 &= 20 \text{ ms}, & C_3 &= 5 \text{ ms} \\ \mathcal{T}_4 : \quad T_4 &= 30 \text{ ms}, & C_4 &= 3 \text{ ms}\end{aligned}$$

21. Pokazati rad raspoređivanja prema najmanjoj labavosti (LLF) za sustav iz prethodnog zadatka, uz dvostruko veća vremena računanja (redom 4, 6, 10, 6 ms), ako se on izvodi na dvoprocesorskom sustavu (pokazati rad u intervalu 0.–10. ms). Interval poziva raspoređivača (ponovni izračun labavosti) obavlja se svake 1 ms. Sekundarni kriterij kod raspoređivanja je mjera ponavljanja.
22. Zadaci $\mathcal{T}_1 – \mathcal{T}_4$ javljaju se u trenucima 1. ms, 2. ms, 3. ms te 4. ms (respektivno, \mathcal{T}_1 u 1.). Obrada svakog zadatka traje po 3 ms. Svi zadaci (i pojedinačno) moraju biti gotovi do 7. ms. Pokazati rad raspoređivača po najmanjoj labavosti (LLF) nad tim sustavom zadatka ako se koristi dvoprocesorski sustav. Naznačiti posebne trenutke u tom izvođenju (dolasci, odlasci, ...).
23. Korištenjem raspoređivanja prema najmanjoj labavosti te redu prispijeća kao sekundarnom kriteriju, grafički prikazati raspoređivanje sustava zadatka na dvoprocesorskom sustavu u intervalu [0,30].

$$\begin{aligned}\mathcal{T}_1 : \quad T_1 &= 10 \text{ ms}, & C_1 &= 5 \text{ ms} \\ \mathcal{T}_2 : \quad T_2 &= 15 \text{ ms}, & C_2 &= 7 \text{ ms} \\ \mathcal{T}_3 : \quad T_3 &= 20 \text{ ms}, & C_3 &= 10 \text{ ms} \\ \mathcal{T}_4 : \quad T_4 &= 30 \text{ ms}, & C_4 &= 15 \text{ ms}\end{aligned}$$

5. Raspoređivanje dretvi

U prethodnim odjeljcima je problem raspoređivanja teoretski razmatran te je prikazano nekoliko postupaka raspoređivanja.

1. Raspoređivanje prema mjeri ponavljanja, kod kojeg se dretvama statički pridijele prioriteti te se pri radu koristi prioritetni raspoređivač.
2. Raspoređivanje prema krajnjim trenucima dovršetaka te Raspoređivanje prema najmanjoj labavosti, kod kojih raspoređivač dinamički odlučuje o odabiru dretve obzirom na to kada ona mora završiti svoj posao.
3. Opće raspoređivanje, koje se primjenjuje na višeprocesorskim sustavima za optimalan raspored skupa zavisnih zadataka.
4. Raspoređivanje sa stablenom strukturom, koje se primjenjuje na višeprocesorskim sustavima za dobar raspored skupa zavisnih zadataka čije se ovisnosti mogu prikazati stablenom strukturom.

Neki od navedenih postupaka spadaju u klasu statičkih raspoređivača koji odluke o značajkama raspoređivanja (njihove vrijednosti) donose prije pokretanja sustava, dok drugi postupci spadaju u klasu dinamičkih postupaka (značajke koje se koriste za raspoređivanje mijenjaju se tijekom rada zadataka kao i samim protokom vremena).

U praksi se nastoje preuzeti dobra svojstva pojedinih postupaka raspoređivanja, ali tako da sam postupak bude primjenjiv. Obzirom da je vrlo mali broj obilježja zadataka unaprijed poznat, koriste se jednostavniji postupci. Međutim, to ovisi o sustavu koji se razmatra. Negdje su i oni složeniji neophodni, ali ipak rjeđe.

Raspoređivanje značajno ovisi o zahtjevima sustava. Sustavi za rad u stvarnom vremenu samo su jedna skupina sustava u kojoj se koriste računala. U druge skupine možemo ubrojati ugrađene sustave, osobna računala, radne stanice, poslužiteljska računala, prijenosna i mobilna računala (telefoni, tableti i slično). Svojstva zadataka u drugim sustavima nisu jednaka te identični način raspoređivanja dretvi ne mora biti najbolji. Primjerice, multimedijalni program ima svojstva slična zadatacima sustava za rad u stvarnom vremenu: svako kašnjenje može izazvati osjetnu degradaciju kvalitete slike ili zvuka. Slično je i s korisničkim sučeljem: znatnija kašnjenja u reakciji na korisničke naredbe korisniku će umanjiti kvalitetu sustava. Ipak, u oba navedena primjera loše raspoređivanje neće izazvati znatne štete (osim smanjenja kvalitete sustava prema ocjeni korisnika). Operacije koje dulje traju, kao što su matematički proračuni, kompresija podataka i prijenos datoteka, mogu se još malo odgoditi bez umanjenja kvalitete sustava, obzirom da korisnik već očekuje njihovo produljeno trajanje.

Poslove koji se nisu vremenski kritični, koji neće izazvati veće probleme ukoliko zakažu, koji se izvode u običnim sustavima se označuju kao *obični* ili *nekritični*. Poslove koji jesu vremenski kritični, čija greška ili kasna reakcija može imati velike posljedice (koji se izvode u SRSV-ima) se označuju kao *kriticnima*. Operacijski sustavi najčešće omogućuju različite načine raspoređivanja, jedne za kritične poslove i druge za nekritične. Operacijski sustavi pripremljeni za SRSV-e, osim ostalih potrebnih svojstava imaju i vrlo dobru podršku raspoređivanju kritičnih dretvi, dok operacijski sustavi koji nisu posebice pripremljeni za SRSV-e optimiraju neke druge kriterije i u pogledu raspoređivanja poslova.

Pri ostvarenju sustava, jedan zadatak se preslikava u jednu dretvu te se nastavku koristi termin *dretva* umjesto *zadataka*, odnosno, govori se o *raspoređivanju dretvi*. U nastavku su prikazani

uobičajeni načini raspoređivanja dretvi ostvareni u operacijskim sustavima.

5.1. POSIX i Linux sučelja

POSIX je zajednički naziv za skupinu IEEE normi kojima se definira sučelja koja operacijski sustavi trebaju pružati programima, a radi njihove prenosivosti [POSIX]. Prenosivost programa je glavni razlog nastajanja POSIX-a i sličnih normi. U početku su ciljani operacijski sustavi bile razne inačice UNIX-a, ali se kasnije sučelje počelo širiti tako da obuhvaća i sustave za rad u stvarnom vremenu.

POSIX ostvaruju mnogi operacijski sustavi, pogotovo oni predviđeni za sustave za rad u stvarnom vremenu (najčešće radi omogućavanja prijenosa postojećih programa pripremljenih za druge sustave). U kontekstu SRSV-a POSIX definira nekoliko načina raspoređivanja (klase dretvi):

- SCHED_FIFO – prema prioritetu pa po redu prispijeća
- SCHED_RR – prema prioritetu pa kružnom podjelom vremena
- SCHED_SPORADIC – prema prioritetu, sporadični poslovi
- SCHED_OTHER – raspoređivanje nekritičnih dretvi.

Na promjene i proširenje POSIX-a utječu mnogi dionici, posebice oni koji se bave izgradnjom operacijskih sustava. Stoga će u nastavku pored navedenih načina raspoređivanja biti opisani i dodatni načini trenutno ostvareni ‘samo’ u Linux jezgri. Uz gornje načine, uz izuzetak načina SCHED_SPORADIC koji nije ostvaren, u Linuxu su dodatno ostvareni:

- SCHED_DEADLINE – raspoređivanje prema krajnjem trenutku dovršetka
- SCHED_BATCH – raspoređivanje nekritičnih dugotrajnih dretvi
- SCHED_IDLE – raspoređivanje najmanje bitnih dretvi.

U nastavku je najprije prikazano sučelje za postavljanje načina raspoređivanja i dodatnih parametara uz odabранe načine. Obzirom da većina načina koristi prioritete, u sučelja su ugrađene i funkcije koje upravljaju njime.

Isječak kôda 5.1. Postavljanje načina raspoređivanja i ostalih parametara

```
int pthread_setschedparam ( pthread_t thread,
                            int policy,
                            const struct sched_param *param );
```

Isječak kôda 5.2. Postavljanje prioriteta dretvi

```
int pthread_setschedprio ( pthread_t thread,
                           int prio );
```

Ekvivalentne funkcije na razini procesa su sched_setscheduler i sched_setparam (s istim ili ekvivalentnim parametrima).

Ponekad je jednostavnije prije stvaranja nove dretve definirati njene parametre za raspoređivanje.

Isječak kôda 5.3. Stvaranje nove dretve

```
int pthread_create ( pthread_t *thread,
                     const pthread_attr_t *attr,
                     void *(*start_routine) (void*),
                     void *arg );
```

Drugi parametar funkcije (attr) definira atributi za novu dretvu te ga je potrebno postaviti

prema željenim svojstvima nove dretve, prije stvaranja nove dretve.

Isječak kôda 5.4. Postavljanje parametara raspoređivanja za novu dretvu

```
int pthread_attr_init ( pthread_attr_t *attr );
int pthread_attr_setinheritsched ( pthread_attr_t *attr,
                                   int inheritsched );
int pthread_attr_setschedpolicy ( pthread_attr_t *attr,
                                  int policy);
int pthread_attr_setschedparam ( pthread_attr_t *attr,
                                 const struct sched_param *param );
```

Preko `pthread_attr_setinheritsched` se definira da li nova dretva nasljeđuje parametre raspoređivanja od dretve koja ju stvara (kada je drugi parametar `PTHREAD_INHERIT_SCHED`) ili ih treba zasebno postaviti (kada je drugi parametar `PTHREAD_EXPLICIT_SCHED`).

Način raspoređivanja postavlja se preko `pthread_attr_setschedpolicy`. Parametri tog načina (npr. prioritet) postavljaju se preko sučelja `pthread_attr_setschedparam`.

Prioritet dretvi se može promijeniti i pod utjecajem sinkronizacijskih mehanizama. To je objašnjeno uz opis sinkronizacijskih mehanizama u 6.6.

Ako se u sustavu dretve dinamički stvaraju prema potrebama, onda je potrebno razmatrati i njihove završetke. Naime, svaka dretva zauzima dio sredstava sustava. Pri završetku dretve sva zauzeta sredstva se ne oslobođaju automatizmom, već neka ostaju zauzeta. Sredstva se u potpunosti oslobođaju kad je status dovršene dretve preuzet od neke druge dretve (npr. pozivom `pthread_join`) ili ako je dretva pri stvaranju označena kao odvojena (engl. *detachable*). Označavanje dretve kao odvojive se postavlja u atributu `attr` pozivom `pthread_attr_setdetachstate` ili to radi sama stvorena dretva pozivom `pthread_detach`.

Stvaranje dretvi koje se raspoređuju po metodama `SCHED_FIFO`, `SCHED_RR`, `SCHED_SPORADIC` i `SCHED_DEADLINE` zahtijeva povlaštenog korisnika (administratora) budući da njegove dretve mogu u potpunosti istisnuti sve druge dretve, pa čak i one operacijskog sustava.

U nastavku su opisani navedeni načini raspoređivanja definirani gornjim sučeljima.

5.2. Rasporedivanje dretvi prema prioritetu

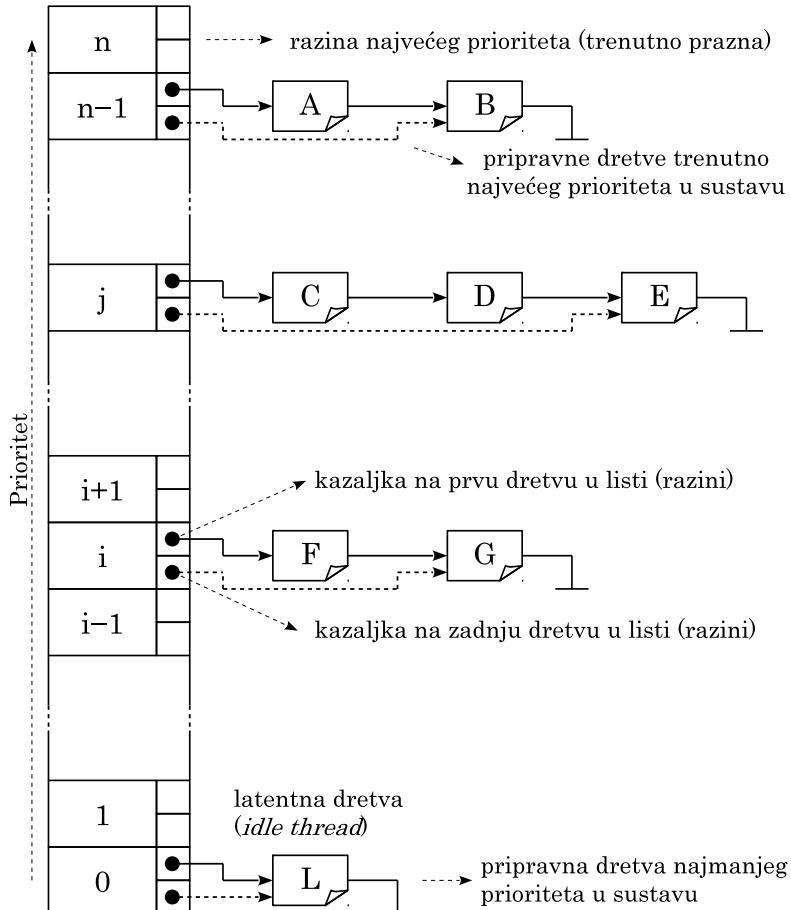
Većina operacijskih sustava pripremljena ili prilagođena za sustave za rad u stvarnom vremenu zapravo koristi samo jedan način rasporedivanja – rasporedivanje prema prioritetu. Zato je način dodjele prioriteta dretvama najčešće prema postupku mjere ponavljanja. Odstupanja od tog postupka se koriste kada različite dretve obavljaju poslove različita značaja. Tada arhitekt sustava može nekim dretvama pridijeliti i veći ili manji prioritet od onog koji bi dretva dobila prema postupku mjere ponavljanja.

Samo rasporedivanje – određivanje trenutne dretve za izvođenje obavlja se pri radu sustava tako da se u svakom trenutku među dretvama spremnim za izvođenje (*pripravne dretve*) odbire ona najvećeg prioriteta (koja tada postaje *aktivna dretva*). Dretve koje nisu spremne za izvođenje (*blokirane dretve*) se ne razmatraju pri rasporedivanju.

Obzirom da postoji mogućnost da u nekom trenutku najveći prioritet nema samo jedna pripravna dretva već više njih, mora se koristiti i dodatni kriterij pomoću kojeg će se odabrati samo jedna dretva. Najčešće korišteni dodatni kriteriji su: prema redu prispijeća (engl. *first in first out – FIFO*) te podjela procesorskog vremena – kružno posluživanje (engl. *round robin – RR*). Obzirom da je prvi kriterij uvijek prioritet, takvi postupci rasporedivanja dobivaju ime prema drugom kriteriju. Tako je i s imenima tih rasporedivača u POSIX-u:

1. `SCHED_FIFO` – rasporedivanje prema prioritetu pa po redu prispijeća
2. `SCHED_RR` – rasporedivanje prema prioritetu pa kružnom podjelom vremena.

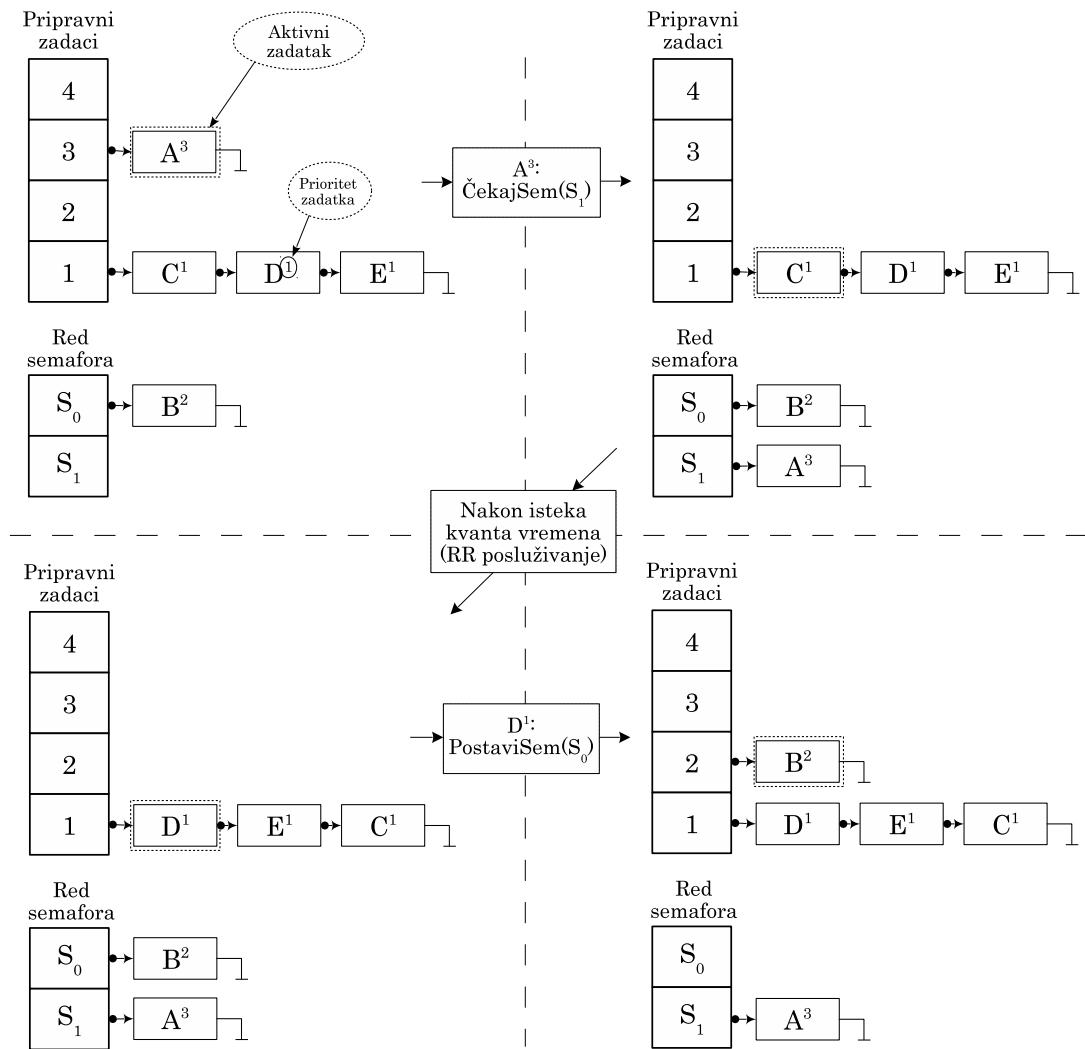
Slika 5.1. prikazuje mogući izgled strukture podataka raspoređivača koji koristi prioritet za raspoređivanje dretvi. Odabir aktivne dretve obavlja se među pripravnim dretvama najvećeg prioriteta, među dretvama A i B. Obzirom da je dretva A prva u listi ona će biti odabrana kao aktivna. Ako je drugi kriterij FIFO dretva A će se izvoditi dok ne završi ili dok se ne blokira. Ako je drugi kriterij RR onda će dretvu A u njenom izvođenju prekinuti raspoređivač nakon što je ona “potrošila” svoj dodijeljeni dio procesorskog vremena (kvant), dretva A bit će stavljena na kraj reda dretvi s istim prioritetom, tj. iza dretva B. Iduća aktivna dretva bit će dretva B.



Slika 5.1. Primjer strukture podataka jezgre za raspoređivanje prema prioritetu

U višeprocesorskim sustavima red pripravnih dretvi (koje se raspoređuju) može biti ostvaren kao na slici 5.1., ali i drugčije. Kod jednog od pristupa se za svaki procesor stvara poseban red pripravnih, prema slici 5.1. Tada postaje potrebno osigurati da se uvijek izvode dretve najveća prioriteta. Ponekad će to zahtijevati “guranje” dretvi u druge redove pripravnih dretvi ili posizanje za dretvama iz redova drugih procesora. Primjerice, kada aktivna dretva A na procesoru I omogući nastavak drugoj dretvi B nastavak svog rada, tada se B može gurnuti procesoru J ako on izvodi dretvu čiji je prioritet manji od dretvi A i B, odnosno i od svih drugih aktivnih dretvi. Slično je i kada neka dretva završi s radom ili se blokira: tada je potrebno odabrati pripravnu dretvu najveća prioriteta među svim pripravnim dretvama – ne samo iz reda procesora koji je sada postao slobodan. U ovom se slučaju radi o “povlačenju” dretve iz reda pripravnih dretvi drugih procesora. Opisani postupci koriste se pri raspoređivanju dretvi u Linux-u.

Slika 5.2. prikazuje primjere raspoređivanja korištenjem prioriteta, kružnog posluživanja te događaja povezanih sa sinkronizacijskim funkcijama.



Slika 5.2. Primjeri raspoređivanja prema prioritetu i podjeli vremena

5.3. Raspoređivanje prema krajnjim trenucima završetaka

Raspoređivanje prema krajnjim trenucima završetaka je rijetko podržani način raspoređivanja dretvi, čak i među operacijskim sustavima za rad u stvarnom vremenu. U sustavima u kojima jest podržano takvo raspoređivanje, mehanizam njegova korištenja je takav da se posebnim sučeljem dretva označi kao periodička, sa zadanom periodom kojom se dretva budi i pokreće, uzimajući u obzir i druge slične dretve i njihove trenutke krajnjih dovršetaka.

Periodičke dretve se mogu prikazati pseudokodom prema primjeru 5.5.

Isječak kôda 5.5. Načelni pseudokod periodičke dretve

```

periodička_dretva
    ponavljam
        odradi_periodički_posao
        odgodi_izvođenje ( ostatak_periode )
    
```

Pseudokod takvih dretvi treba proširiti odgovarajućim pozivima. Uobičajeno sučelje koje nude operacijski sustavi koji podržavaju takvo raspoređivanje prikazano je na primjeru 5.6.

Isječak kôda 5.6. Uobičajena sučelja za raspoređivanje prema krajnjim trenucima završetaka

```

periodička_dretva
označi_periodičnost ( period )
ponavljaј
    čekaj_početak_periode ()
odradi_periodički_posao;

```

Operacijski sustav nudi sučelje koje je u primjeru prikazano funkcijama `označi_periodičnost` i `čekaj_početak_periode`.

5.3.1. Raspoređivanje prema krajnjim trenucima završetka kod Linuxa

Linux jezgra od inačice 3.14 (ožujak 2014.) donosi podršku za ovaj način raspoređivanja pod imenom *Sporadic task model deadline scheduling* s oznakom `SCHED_DEADLINE`. Parametri svake dretve su:

- C – potrebno vrijeme izvođenja unutar periode (`sched_runtime`)
- d – trenutak krajnjeg završetka u odnosu na početak periode (`sched_deadline`)
- T – perioda ponavljanja (`sched_period`)

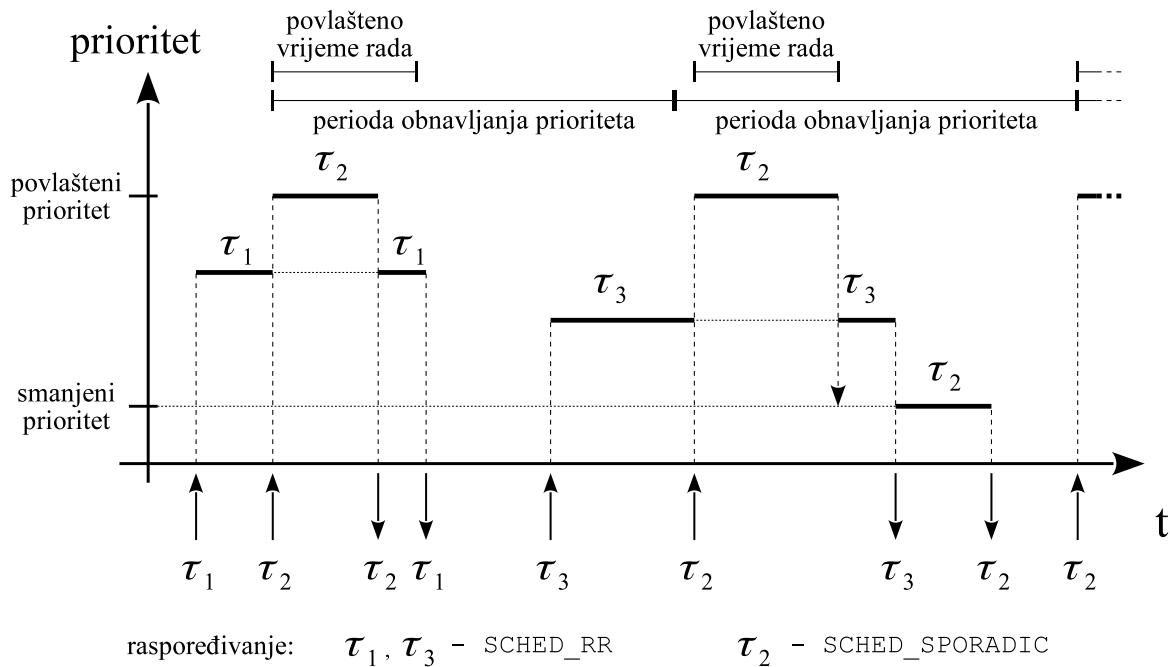
uz ograničenje: $C \leq d \leq T$.

Pri stvaranju nove dretve ovakva načina raspoređivanja provjerava se rasporedivost sustava dretvi koje se raspoređuju prema `SCHED_DEADLINE` uz dodatak opterećenja ove dretve. Ukoliko bi ova dretva narušila rasporedivost, jezgra će odbiti zahtjev za stvaranjem takve dretve ili pretvaranje postojeće u ovaku. Obzirom da dretve koje se raspoređuju sa `SCHED_DEADLINE` istiskuju sve druge dretve, ovakvim jednostavnim provjerama osigurava se rasporedivost svih dretvi koje se raspoređuju ovim načinom, a koje poštuju nametnuta ograničenja vremena izvođenja unutar periode. Da poneka dretva ne bi ugrozila ostale koje se raspoređuju istim `SCHED_DEADLINE` postupkom, dodatno se koristi postupak rezervacije poslužiteljskog vremena (engl. *Constant Bandwidth Server – CBS*). Naime, ako pojedina dretva od početka periode (njena aktiviranja) ‘potroši’ zadano vrijeme izvođenja (a prije blokiranja ili poziva `sched_yield()` kojim se označava kraj računanja u periodi) ona je potrošila svoje ‘rezervirano vrijeme’ u ovoj periodi. Stoga ona u tom trenutku može biti i istisnuta, ako se za to pojavi potreba od ostalih dretvi u sustavu. Obično se to ostvaruje na način da se njen krajnji trenutak završetka poveća za trajanje periode (‘koristi se i njena iduća perioda’) te će na red prije njenog nastavka doći ostale dretve s bližim krajnjim trenutkom završetka. Pregled ovog i ostalih načina raspoređivanja u Linux jezgri detaljnije je opisan na [Linux scheduling].

5.3.2. Raspoređivanje sporadičnih poslova

Osiguranje unaprijed definiranog procesorskog vremena pojedinom poslu trebalo bi omogućiti i sporadično raspoređivanje. Raspoređivanje `SCHED_SPORADIC` je noviji postupak raspoređivanja (vrlo rijetko još podržan u operacijskim sustavima, primjerice [QNX, 6.3]), pripremljen za periodičke dretve, kod kojih dretva tijekom svake svoje periode treba određeno procesorsko vrijeme. Raspoređivač bi joj to vrijeme trebao dodijeliti uz viši prioritet. Ako u tom vremenu ne obavi svoj periodički posao, prioritet joj se smanjuje tako da suviše ne utječe na ostale dretve sustava. Parametri tog raspoređivanja su: period obnavljanja prioriteta (engl. *replenishment period*), povlašteno vrijeme rada (engl. *initial budget*), povlašteni prioritet (engl. *high priority*) te smanjeni prioritet (engl. *lower priority*). Primjer rada ovog raspoređivača prikazuje slika 5.3.

Sporadično raspoređivanje je namijenjeno periodičkim poslovima većeg prioriteta koji uglavnom kratko obave svoje operacije (u svakoj pojavi/periodi). Međutim, ponekada se može dogoditi da posao zahtijeva malo više vremena. Da se u tim slučajevima ne bi narušili ostali poslovi, `SCHED_SPORADIC` će za te dulje obrade smanjiti prioritet dretvi, nakon početnog



Slika 5.3. Primjer sporadičnog raspoređivanja

povlaštenog vremena s većim prioritetom.

Primjer 5.1. Primjer korištenja POSIX-a

U ovome primjeru se koriste načini raspoređivanja SCHED_FIFO i SCHED_RR. Ispis prioriteta dretvi obavlja se unutar kritična odsječka da ne bi došlo do preklapanja u ispisu.

```
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <pthread.h>

#define BROJ_DRETVI 5
static pthread_mutex_t monitor = PTHREAD_MUTEX_INITIALIZER;

static void ispisi_postavke_rasporedjivanja_dretve ( long id )
{
    int nacin_rasporedjivanja;
    struct sched_param prioritet;

    pthread_mutex_lock ( &monitor );
    pthread_getschedparam ( pthread_self(),
                           &nacin_rasporedjivanja, &prioritet );
    printf ( "Dretva %ld: nacin_rasporedjivanja=%d, prioritet=%d\n",
             id, nacin_rasporedjivanja, prioritet.sched_priority );
    pthread_mutex_unlock ( &monitor );
}

static void *posao_dretve ( void *param )
{
    ispisi_postavke_rasporedjivanja_dretve ( (long) param );
    return NULL;
}

int main ()
{
```

```

long min, max, pocetni_prioritet, i, nacin_rasporedjivanja;
pthread_attr_t attr;
pthread_t tid[BROJ_DRETVI];
struct sched_param prioritet;

ispisi_postavke_rasporedjivanja_dretve ( 0 );

/*! Dohvati raspon prioriteta za zadani način rasporedjivanja */
nacin_rasporedjivanja = SCHED_FIFO;
min = sched_get_priority_min ( nacin_rasporedjivanja );
max = sched_get_priority_max ( nacin_rasporedjivanja );

/* Postavi način raspoređivanja i prioritet početnoj dretvi */
pocetni_prioritet = prioritet.sched_priority = (min + max) / 2;
if ( pthread_setschedparam ( pthread_self(),
                             nacin_rasporedjivanja, &prioritet ) )
{
    perror ( "Greska: pthread_setschedparam (dozvole?)" );
    exit (1);
}

ispisi_postavke_rasporedjivanja_dretve ( 0 );

/* Postavke rasporedjivanja za nove dretve */
pthread_attr_init ( &attr );
pthread_attr_setinheritsched ( &attr, PTHREAD_EXPLICIT_SCHED );
nacin_rasporedjivanja = SCHED_RR;
pthread_attr_setschedpolicy ( &attr, nacin_rasporedjivanja );

/* Stvaranje novih dretvi */
for ( i = 0; i < BROJ_DRETVI; i++ ) {
    prioritet.sched_priority = (min + i) % pocetni_prioritet;
    pthread_attr_setschedparam ( &attr, &prioritet );
    if ( pthread_create ( &tid[i], &attr, posao_dretve,
                         (void *) (i+1) ) )
    {
        perror ( "Greska: pthread_create" );
        exit (1);
    }
}

/* Čekaj da stvorene dretve završe s radom */
for ( i = 0; i < BROJ_DRETVI; i++ )
    pthread_join ( tid[i], NULL );

return 0;
}

/* Primjer pokretanja: (na jednoprocесорском sustavu !!!!)
$ gcc scheduling.c -pthread -Wall
$ sudo ./a.out
Dretva 0: nacin_rasporedjivanja=0, prioritet=0
Dretva 0: nacin_rasporedjivanja=1, prioritet=50
Dretva 5: nacin_rasporedjivanja=2, prioritet=5
Dretva 4: nacin_rasporedjivanja=2, prioritet=4
Dretva 3: nacin_rasporedjivanja=2, prioritet=3
Dretva 2: nacin_rasporedjivanja=2, prioritet=2
Dretva 1: nacin_rasporedjivanja=2, prioritet=1
*/

```

Iz ispisa prikazanome uz kod vidi se da se stvorene dretve izvode prema prioritetu: naprije ona prioriteta 5, a najzadnja ona prioriteta 1, iako je redoslijed njihova pokretanja bio obrnut. Međutim, na višeprocesorskim sustavima to ne mora biti tako. Naime, u prikazanome primjeru je posao svake dretve samo ispis parametara raspoređivanja što je vrlo kratko,

kraće od stvaranja nove dretve. Stoga bi ona dretva manjeg prioriteta mogla stići obaviti svoje prije pojave dretve većeg prioriteta koja bi ju inače istisnula.

5.4. Raspoređivanje nekriticnih dretvi

Pri raspoređivanju nekriticnih dretvi mogu se koristiti razna načela:

- pravednost
- veća procesorska iskoristivost
- veći broj završenih dretvi
- minimizacija čekanja u redovima
- kraće vrijeme odziva interaktivnih dretvi
- optimizacija korištenja priručnih spremnika.

Načelo pravednosti omogućuje jednake dijelove procesorskog vremena svim dretvama u sustavu (u skladu njihovim prioritetima). Uz višu iskoristivost procesora, više će se posla obaviti, sustav će biti učinkovitiji. Načelom većeg broja završenih dretvi sustav se brže oslobodi veće broja dretvi, a postiže se tako da se favoriziraju kratke dretve. Minimizacijom čekanja u redovima smanjuje se prosječno vrijeme koje dretve provedu u redu prije nego li su bar dijelom odradile svoj posao. Interaktivne dretve koje obično upravljuju s korisničkim sučeljem ili UI jedinicama utječu na percepciju sustava kao sporog ili brzog. Npr. brza reakcija na zahtjeve korisnika stvara dojam da je sustav brz. U sustavima s više procesorskih jedinki načelo optimiranja korištenja priručnih spremnika nalaže raspoređivaču da nastoji zadržati dretvu na istom procesoru. Naime, u sukcesivnim izvođenjima (nakon zamjene s drugim dretvama) takva dretva može pronaći svoje podatke još uvijek u priručnom spremniku i time smanjiti vrijeme njihovog dohvata pri radu dretve, i općenito gledajući, povećati učinkovitost sustava.

U operacijskim sustavima opće namjene dretve koji nisu vremenski kritične uglavnom se raspoređuju načelom pravedne podjele procesorskog vremena. Dretve i tu imaju atribut prioriteta, ali se prioritet koristi za određivanje koliko će procesorskog vremena te dretve dobiti (izračun se obavlja uzimajući sve pripravne dretve i njihove prioritete).

Uobičajeni princip raspoređivanja koji se koristi za takve dretve naziva se višerazinsko raspoređivanje s povratnom vezom (engl. *multilevel feedback queue – MFQ*). Ciljevi koje to raspoređivanje nastoji ostvariti su:

- dati prednost dretvama s kratkim poslovima
- dati prednost dretvama koje koriste ulazno-izlazne naprave
- na osnovi rada dretve ustanoviti u koju skupinu dretva pripada te ju prema tome dalje raspoređivati.

Duge dretve, tj. dretve koje su procesorski intenzivne, koje trebaju puno procesorskog vremena (koje bi cijelo vrijeme koristile procesor) se žele potisnuti. Naime, takve dretve ionako duže traju, te se očekuje da će kasnije biti gotove te će njihova kratka odgoda zbog izvođenja drugih dretvi manje utjecati na sustav nego odgoda kratkih dretvi, koje, primjerice upravljaju korisničkim sučeljem i čija reakcija mora biti promptna (inače se kod korisnika stvara dojam sporosti sustava).

Sam postupak raspoređivanja koji nastoji poštovati navedena načela može se opisati skupom pravila ponašanja raspoređivača nad skupom dretvi koje su složene prema trenutnim prioritetima u redove prema redu prispijeća (slično slici 5.1.).

- Uvijek se raspoređuje prva dretva iz reda najvećeg prioriteta (najviši neprazni red).
- Procesor se dretvi dodjeljuje za određeni interval vremena (kvant vremena).
- Ako dretva završi u tom intervalu, ona napušta sustav (ne razmatra se više u postupku

raspoređivanja).

- Ako se dretva pri svom izvođenju u tom njoj dodijeljenom intervalu blokira, miče se iz reda pripravnih, ali se kasnije, pri odblokiranju stavlja u isti red pripravnih dretvi iz kojeg i otišla ('zadržava prioritet'), ili ovisno o vremenu provedenom u blokiranim stanju, čak se postavlja i u više redove (podiže joj se prioritet).
- Ako se dretva izvodi cijeli interval i nije gotova niti se blokirala za vrijeme tog izvođenja, onda ju rasporedivač prekida i miče u red niže (smanjuje joj prioritet za jedan).
- Postupak se ponavlja dok god ima pripravnih dretvi i dok one ne dođu do najnižeg reda (reda najmanjeg prioriteta). U tom redu se dretve poslužuju podjelom vremena.
- Kada u sustav dođe nova dretva, ona se stavlja u najviši red (njaprioritetniji red), na kraj tog reda.

Opisani postupak vrlo brzo procjenjuje dretvu: je li ona spada u kategoriju kratkih ili dugih. Ako je kratka, ostaje joj prioritet, a ako je duga prioritet joj pada tijekom izvođenja.

Višerazinsko raspoređivanje s povratnom vezom se ne ostvaruje u operacijskim sustavima upravo prema prikazanim pravilima, ali se sličnim pristupima ostvaruju navedena načela.

U nastavku je ukratko opisano raspoređivanje dretvi u operacijskim sustavima zasnovanim na Linux jezgri te sustavima zasnovanim na porodici operacijskih sustava Microsoft Windows.

Primjer 5.2. Rasporedivači ugrađeni u Linux jezgru

Operacijski sustavi zasnovani na Linux jezgri podržavaju načine raspoređivanja SCHED_FIFO, SCHED_RR i SCHED_DEADLINE (tek od 3.14) za kritične dretve te SCHED_OTHER, SCHED_BATCH (od 2.6.16) i SCHED_IDLE (od 2.6.23) za obične, nekritične dretve. Prioriteti kritičnih dretvi kreću se od 1 do 99 (veći broj označava veći prioritet). Za raspoređivanje nekritičnih dretvi koristi se prioritet 0.

Nekritične dretve će dobiti procesorsko vrijeme tek kada nema niti jedne kritične dretve u redu pripravnih. Iznimno, od Linux jezgre 2.6.25 moguće je rezervirati dio procesorskog vremena i za nekritične dretve. Uobičajeno je to postavljeno na 5 % procesorskog vremena. Navedena rezervacija omogućava administratoru da pokrene zaustavljanje kritične dretve koja ima grešku (npr. beskonačnu petlju i koristila bi svo procesorsko vrijeme).

Dretve koje spadaju u klase SCHED_OTHER, SCHED_BATCH i SCHED_IDLE imaju osnovni prioritet postavljen na nulu (manji od najmanjeg za kritične dretve), ali za međusobnu usporedbu (raspoređivanje) koriste drugu vrijednost, takozvanu *razinu dobrote* (engl. *nice level*) koja se kreće od -20 (najveća) do +19 (najmanja). Dobrota ispod nule može postavljati samo administrator (korisnik *root*). Dobrota dretve utječe na to koliko će procesorskog vremena dretva dobiti u odnosu na ostale dretve različite dobrote. Primjerice, dretva s razinom dobrote q trebala bi dobiti od 10 do 15 % više procesorskog vremena od dretve razine $q + 1$.

Načini raspoređivanja SCHED_OTHER i SCHED_BATCH su slični, jedino što će rasporedivač uvijek pretpostaviti da je dretva u klasi SCHED_BATCH duga dretva i zbog toga biti u nešto lošijem položaju od ostalih dretvi (u klasi SCHED_OTHER).

Dretve u klasi SCHED_IDLE imaju najmanju dobrotu i neće se izvoditi dokle god ima drugih dretvi koje nisu u istoj klasi.

Od inačice Linux jezgre 2.6.23 za raspoređivanje nekritičnih dretvi (SCHED_OTHER, SCHED_BATCH i SCHED_IDLE) koristi se novi rasporedivač naziva *potpuno pravedan rasporedivač* (engl. *completely fair scheduler – CFS*). Korištenjem izračunatog virtualnog vremena koje pripada pojedinoj dretvi i stvarno dodijeljenog vremena, tj. razlike tih vremena izgrađuju se crveno-crna stabla s pripravnim dretvama te se za aktivnu odabire ona dretva kojoj

sustav najviše duguje (s najvećom razlikom).

Za prikaz osnovne ideje CFS-a naveden je pojednostavljeni primjer. Neka se u sustavu u početnom trenutku nalazi pet dretvi $\{D_1, \dots, D_5\}$ iste razine dobrote. Raspoređivač će odabrat jednu, recimo D_1 , i njoj dati kvant vremena T_q . Nakon što taj kvant istekne, raspoređivač će ažurirati virtualna vremena koja pripadaju pojedinim dretvama: u tom intervalu svaka od dretvi dobiva jednak dio virtualnog vremena, tj. $T_q/5$. Obzirom da se samo D_1 izvodila, jedino će se njoj povećati dodijeljeno vrijeme za T_q što će uzrokovati pomak te dretvu u stablu – ona više neće biti zajedno s ostalima već zadnja u ovom trenutku. Zato će raspoređivač u idućem trenutku odabrat neku drugu dretvu $\{D_2, \dots, D_5\}$ kao aktivnu.

Primjer 5.3. Microsoft Windows operacijski sustavi

Raspoređivanje u porodicama operacijskih sustava Microsoft Windows (NT, 2000, XP, 2003, Vista, 7, 8) obavlja se korištenjem prioriteta dretvi. Prioritet dretve računa se na osnovu prioritetne klase procesa i prioritetne razine dretvi, prema tablicama 5.1. i 5.2.

Tablica 5.1. Prioritetne klase procesa

oznaka klase	skraćeno
IDLE_PRIORITY_CLASS	ID
BELOW_NORMAL_PRIORITY_CLASS	BN
NORMAL_PRIORITY_CLASS	N
ABOVE_NORMAL_PRIORITY_CLASS	AN
HIGH_PRIORITY_CLASS	H
REALTIME_PRIORITY_CLASS	RT

Tablica 5.2. Prioritetne razine dretvi

oznaka razine	skraćeno
THREAD_PRIORITY_IDLE	ID
THREAD_PRIORITY_LOWEST	L
THREAD_PRIORITY_BELOW_NORMAL	BN
THREAD_PRIORITY_NORMAL	N
THREAD_PRIORITY_ABOVE_NORMAL	AN
THREAD_PRIORITY_HIGHEST	H
THREAD_PRIORITY_TIME_CRITICAL	TC

Tablice 5.3. i 5.4. prikazuju dodjeljivanje prioriteta na osnovu prioritetnih klasa i razina.

Tablica 5.3. Izračun prioriteta za normalne dretve (ID, BN, N, AN, H)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ID	ID	L	BN	N	AN	H									TC
BN	ID			L	BN	N	AN	H							TC
bN	ID				L	BN	N	AN	H						TC
fN	ID						L	BN	N	AN	H				TC
AN	ID							L	BN	N	AN	H			TC
H	ID									L	BN	N	AN	H,TC	

Prioritet 0 (najniži prioritet) rezerviran je za posebne dretve operacijskog sustava (npr. dretve koje brišu oslobođene stranice u postupku upravljanja spremnikom straničenjem).

Tablica 5.4. Izračun prioriteta za kritične dretve (RT)

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ID	-7	-6	-5	-4	-3	L	BN	N	AN	H	3	4	5	6	TC

Raspoređivanje kritičnih dretvi (klasa `REALTIME_PRIORITY_CLASS`) se ponešto razlikuje od ostalih klasa, odnosno, raspoređivanje je identično prethodno opisanom načinu `SCHED _RR`, uz raspon prioriteta od 16 do 31.

Raspoređivanje nekritičnih dretvi obavlja se korištenjem njihovih prioriteta – dretve najvećeg prioriteta se odabiru za izvođenje, slično `SCHED_RR`, ali uz iznimke.

- Prve iznimke su dretve procesa u klasi `NORMAL_PRIORITY_CLASS` (N) koji je trenutno u fokusu korisnika (engl. *foreground process*) koji dobiva malo povećanje prioriteta (prefiks f u tablici 5.3.) u odnosu na dretve koje nisu u fokusu (prefiks b).
- Druge iznimke su pripravne dretve koje dugo nisu doble ništa procesorskog vremena (izgladnjene dretve). Njima raspoređivač povremeno dodijeljuje dio procesorskog vremena da izbjegne njihovo potpuno izgladnjivanje i također da ublaži problem inverzije prioriteta. Više o problemu inverzije prioriteta u poglavljju 6.6.

5.5. Upravljanje poslovima u uređajima napajanim baterijama

Mnogi su računalni sustavi (računala) napajana iz baterija, bez priključka na izvor stalne energije (na električnu mrežu). U takvim je sustavima osobito bitno osigurati željeni period samostalnog rada. Dok se za prijenosna računala, tablete i pametne telefone to vrijeme mjeri satima, za neke druge to mogu biti dani, tjedni, mjeseci pa i godine! Radi omogućavanja navedena načina rada treba prilagoditi i sklopolje i programe koji će biti i manje zahtjevnici koji će znati iskoristiti posebnosti sklopolja.

Veliki potrošač energije jest procesor jer najčešće on može raditi na najvećoj frekvenciji. Stoga se on pokušava izvesti u posebnoj tehnologiji manje potrošnje, s pojednostavljenim operacijama u odnosu prema procesoru namjenjenome stolnim računalima. Mogućnosti upravljanja takvim procesorom radi uštede energije uglavnom spadaju u sljedeće kategorije:

- prilagodljiva frekvencija rada – manja kada je računalo napajano baterijom ili nema potrebe za većom procesorskom snagom
- zaustavljanje procesora posebnim instrukcijama kada nema nikakva posla (npr. pri izvođenju latentne/idle dretve)
- zaustavljanje nepotrebnih jezgri u višejezgrenome procesoru.

Za ostale komponente računala mogu vrijediti ista načela kao i za procesor: izvedba u tehnologiji manje potrošnje, mogućnost zaustavanja rada naprave te rada u načinu sa smanjenom potrošnjom.

Operacijski sustav treba poznavati mogućnosti naprava i koristiti one načine uštete energije koji naprave pružaju. Ovdje spadaju i isključivanje zaslona, postavljanje u stanje niske potrošnje ili čak i gašenje računala nakon nekog intervala nekorištenja.

Razni tipovi računala se koriste na razne načine.

Prijenosno računalo je zapravo zamjena za stolno te nasljeđuje njegova svojstva i načine korištenja. To zapravo znači da korisnik pokreće neke programe koje operacijski sustav raspoređuje prema utvrđenim kriterijima, primjerice, prema prioritetu.

Pametni telefon i tablet računalo se naizgled koriste na isti način. Međutim, oni su u začetku zamišljeni za jednog korisnika i jedan program s kojim korisnik u jednom trenutku izravno komunicira. Stoga je i sama unutarnja arhitektura tih sustava ponešto drukčija. Program koji komunicira s korisnikom je u tom trenutku najvažniji i njemu treba dati sve potrebna sredstva sustava, dok ostale treba zaustaviti. Obzirom na taj očekivani način rada, programi pisani za takve sustave imaju dio koda koji se izvodi dok je program aktivan (komunicira s korisnikom), dio koda koji se izvodi kada program prestaje biti aktivan, dio koda koji se izvodi kada program ponovno postaje aktivan i slično. Ukoliko program treba izvoditi neke periodičke radnje i dok nije aktivan onda se on mora koristiti posebnim sučeljem operacijskog sustava za izvođenje tih periodičkih aktivnosti. Primjerice, slušanje glazbe zahtjeva periodički dohvata dijela skladbe, obradu tog dijela te prijenos prema zvučnom podsustavu računala koji će ga dalje proslijediti u pravom obliku do zvučnika.

Ostala ugrađena računala manje procesne snage prilagođavaju se svojoj funkciji. Najčešće to nisu računala za opću uporabu i nadogradnju s programima različite namjene, već služe samo za jednu ili do nekoliko namjena. U ovu klasu računala možemo ubrojiti i računala koja se sada nazivaju 'Internet stvari' (engl. *Internet of Things – IoT*), a za koje se predviđa da će u bliskoj budućnosti brojčano znatno nadmašiti sva ostala računala zajedno. Radi osiguravanja dugog rada takvih sustava potrebno je izdvojiti one komponente koje potencijalno troše najviše energije i prilagoditi njihov rad da se potrošnja smanji. S današnje perspektive to su dijelovi računala predviđeni za ostvarenje komunikacije za koje je za sada potrebna najveća energija ukoliko bi željeli stalnu povezanost tih računala s ostalima ('na Internet'). Već i sada postoje nekoliko protokola koji omogućuju komunikaciju koja nije toliko zahtjevna kao dosadašnje tehnologije. Ipak, još uvijek se radi na osmišljavanju još boljih, tj. manje zahtjevnih protokola i tehnologija, koje bi omogućile još dulji rad takvih ugrađenih računala i sa samo povremenom povezanošću s ostalim računalima, npr. čak i ne izravno na mrežnu infrastrukturu već do nje preko ostalih sličnih računala, npr. bežične mreže osjetila (engl. *wireless sensor network – WSN*). Primjerice, ugrađeno računalo koje bi se moglo ugraditi u čovjeka (npr. ispod kože), a koje bi mjerilo temperaturu, krvni tlak, udio neke materije u krvi, pratilo rad srca i slično, trebalo bi biti što manje, u mogućnosti raditi što duže (mjereno u mjesecim/godinama!). S druge strane takvo bi računalo trebalo svoja mjerena u nekim intervalima pokušati proslijediti svoja očitanja prema drugom računalu koje bi pohranilo ta očitanja te možda napravila i neke složenije analize tih podataka. Osnovnu analizu bi možda moglo napraviti i ugrađeno računalo te poslati poruke upozorenja ukoliko su očitani podaci problematični.

Ukoliko se projektira ugrađeno računalo napajano baterijom, pored uobičajenih problema ostvarenja logičke i vremenske ispravnosti treba voditi računa i o samostalnom radu računala u predviđenom periodu njegova rada te koristiti i postupke kojim će se smanjiti potrošnja energije, a da bi se željena samostalnost mogla ostvariti. Odabir redoslijeda izvođenja raznih poslova/operacija će stoga vjerojatno biti različit od do sada opisivanih postupaka koji nisu uzimali u obzir potrošnju energije.

Pitanja za vježbu 5

1. Usporediti postupke raspoređivanja koji se koriste za vremenski kritične poslove s onim poslovima koji nisu vremenski kritični.
 2. Opisati postupke raspoređivanja: SCHED_FIFO, SCHED_RR, SCHED_SPORADIC, SCHED_DEADLINE i SCHED_OTHER. ◁
 3. Opisati kriterije i načela raspoređivanja običnih (vremenski nekriticnih) dretvi prema višerazinskom raspoređivanju s povratnom vezom (engl. *multilevel feedback queue – MFQ*).
 4. Opisati postupke raspoređivanja podržane u operacijskim sustavima Linux te MS Windows.
 5. U sustavu se javljaju dretve A, B, C i D. Dretva A ima najveći prioritet, slijede dretve B i C koje imaju jednaki prioritet, te dretva D koja ima najmanji prioritet. Dretva A se javlja u $t=4$. [s], B u $t=2$. [s], C u $t=5$. [s] te D u $t=1$. [s]. Svaka dretva treba 5 [s] procesorskog vremena. Sustav koristi raspoređivanje SCHED_RR (prioritet pa podjela vremena). Prikazati redoslijed izvođenja dretvi na procesoru dok se sve dretve ne obave do kraja. ◁
 6. U sustavu koji koristi posluživanje SCHED_RR pojavljuju se sljedeće dretve: (id_dretve, vrijeme_pojave, trajanje, prioritet) = { (1, 0, 6, 5), (2, 2, 6, 5), (3, 5, 6, 5), (4, 9, 5, 10), (5, 12, 2, 1) }. Veći broj predstavlja veći prioritet. Kvant vremena iznosi jednu jedinicu vremena (kućanski se poslovi zanemaruju). Prikazati redoslijed izvođenja dretvi na procesoru dok se sve dretve ne obave do kraja.
 7. U nekom trenutku u sustavu se nalazi skup dretvi A, B, C i D. Dretve A i C raspoređuju se prema SCHED_RR a ostale B i D prema SCHED_FIFO. Prioriteti dretvi su: $p_A = 30$, $p_B = 30$, $p_C = 25$ te $p_D = 20$. Pokazati rad sustava dok sve navedene dretve ne završe, uz pretpostavku da svaka od navedenih dretvi treba još pedeset milisekundi te da (za dretve koje ga koriste) kvant vremena iznosi $T_q = 20$ ms. ◁
 8. Neki sustav koristi prioritetno raspoređivanje. Svake sekunde rasporedivač prolazi pripravne dretve te onim dretvama koje još nisu doble procesorsko vrijeme (u zadnjoj sekundi, a pojatile su se u sustavu prije kraja te sekunde) procesor daje po dva kvanta vremena $T_q = 10$ ms. Uz pretpostavke da poslovi pojedinačno nikada ne traju dulje od 50 ms, da u sustavu nikad nema više od 10 poslova (poslovi se dinamički javljaju u sustav), koliko će u najgorem slučaju trajati prekid izvođenja neke dretve (istisnute zbog prioritetnijih)?
-

6. Višedretvena sinkronizacija i komunikacija

Osim problemu raspoređivanja, veliku pažnju treba posvetiti upravljanju vremenom, sinkronizaciji, ostvarenju komunikacije i korištenje zajedničkih podataka i sredstava sustava. U ovom poglavlju ukratko su prikazane neke od mogućnosti korištenja vremena, sinkronizacije i komunikacije, a koje pružaju današnji operacijski sustavi kroz podršku POSIX-a. Uz primjere prikazane u ovom poglavlju, dodatni kratki programi koji prikazuju korištenje navedenih sučelja mogu se pronaći u [Jelenković, 2010].

Potreba za sinkronizacijom proističe iz toga što ima više dretvi koje istovremeno traže korištenje ograničenog broja sredstava sustava (objekata, naprava i slično). Korištenje sredstava treba ograničiti tako da manji broj dretvi istovremeno koriste sredstva, a najčešće da samo jedna dretva istovremeno koristiti jedno sredstvo.

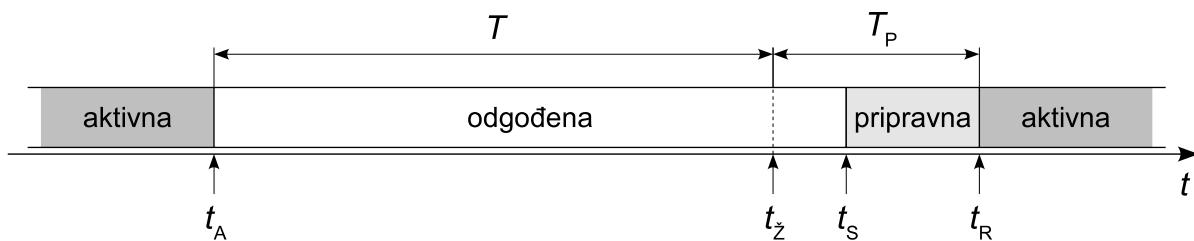
Uobičajeni mehanizmi sinkronizacije dretvi uključuju međusobno isključivanje nad kritičnim odsječkom, semafore, monitore, zaključavanja čitaj/piši, radna zaključavanja (engl. *spinlock*) i barijeru.

Međusobno isključivanje se najčešće ostvaruje sa semaforima i monitorima. U sustavima u kojima programi mogu dobiti vrlo veliku kontrolu nad sustavom, međusobno isključivanje se može ostvariti i onemogućavanjem i naknadnim omogućavanjem prekida (na razini procesora).

6.1. Korištenje vremena u operacijskim sustavima

Vremenska određenost je od presudnog značenja za SRSV-e. U dosadašnjim razmatranjima susreli smo se sa zahtjevima vremenske usklađenosti i kako ju na određene načine ostvariti. Međutim, nismo razmatrali svojstva stvarnih sustava i njihove mehanizme za korištenje vremena, nismo razmatrali probleme poput njihove preciznosti, stabilnosti te kako koristiti vrijeme u raspodijeljenim sustavima.

Razmotrimo primjer odgode dretve za neki željeni interval vremena T prikazan na slici 6.1.



Slika 6.1. Željena i moguća vremena pri odgodi dretve

Dretva je odgodu tražila i dobila u trenutku t_A . Željeni interval odgode T bi dretvu trebao ponovno pokrenuti u trenutku t_Z . Međutim, zbog granulacije sata operacijski sustav će to možda napraviti malo kasnije ili malo prije u trenutku t_S . Iako probuđena u t_S , zbog mogućih drugih događaja u sustavu, primjerice obrade prekida ili prioritetnijih dretvi, dretva može krenuti i kasnije, u trenutku t_R . Od željenog trenutka nastavka rada do stvarnog početka rada može proći vrijeme T_p koje može biti i primjetno veliko u usporedbi sa T . Zbog navedenog potrebno je poznavati svojstva sustava i sučelja koja se koriste za ostvarenje željene vremenske

uskladenosti.

6.1.1. Korištenje satnih mehanizama u operacijskim sustavima

Uz pretpostavku valjanog ostvarenja sustavskog sata u operacijskim sustavima, prije korištenja određenog sučelja ipak treba najprije razmotriti svojstva tog sučelja. Primjerice, je li preciznost zadovoljavajuća? Uobičajeno, sustavi nude satne mehanizme kojima je najmanja jedinica jedna sekunda te mehanizme koji imaju znatno manju granulaciju (milisekunda, mikrosekunda ili nanosekunda). Čak i ovi precizniji mehanizmi trebaju se dodatno provjeriti u ciljanom sustavu, jer iako deklarirana jedinica može biti primjerice nanosekunda, stvarna granulacija može biti značajno veća, pa čak i u rangu milisekundi.

UNIX/POSIX sučelje u granulaciji sekunde, kao što su `sleep`, `time`, `alarm` nam očito nisu prikladni za sustav u kojima je potrebna jedinica vremena ispod sekunde.

Sučelja koja koriste granulaciju ispod sekunde ima više. Neka od njih koriste mehanizam signala za aktivaciju, a druga ne.

6.1.1.1. Dohvat trenutnog vremena, odgoda dretve

Dohvat trenutnog vremena u većoj preciznosti može se obaviti funkcijom:

```
int clock_gettime ( clockid_t clock_id, struct timespec *tp );
```

Parametar `clock_id` određuje sat koji će se koristiti u operaciji. Dva najznačajnija sata su `CLOCK_REALTIME` i `CLOCK_MONOTONIC`. Oba odbrojavaju u taktu stvarnog sata i jedina je razlika što prvi `CLOCK_REALTIME` predstavlja stvarno vrijeme (npr. iz njega se može odrediti da je trenutno vrijeme 12 sati i 35 minuta i 21 sekunda) te se kao takvo može i promijeniti odgovarajućim sučeljem (`clock_settime`) ili to može napraviti neki servis koji povremeno ažurira sat s poslužiteljem točna vremena. `CLOCK_MONOTONIC` se ne može naknadno promjeniti, nakon uključenja računala on uvijek odbrojava te je zato prikladniji za neke procese koje promjena (ažuriranje) sata sustava može navesti na krive zaključke o protoku vremena.

Pri interpretaciji vrijednosti koje funkcija vraća treba uzeti u obzir granulaciju sata.

Odgode izvođenja dretvi mogu se ostvariti sučeljem:

```
int clock_nanosleep ( clockid_t clock_id,
                      int flags,
                      const struct timespec *rqtp,
                      struct timespec *rmtp );
```

Odgoda može biti zadana relativno – za određeno vrijeme, ili absolutno, do određenog vremena. U drugom slučaju je potrebno postaviti zastavicu `TIMER_ABSTIME` u drugom parametru (`flags`).

Kada se dretva probudi prije isteka zadanog intervala, primjerice zbog signala, tada se u zadnjem parametru (ako nije `NULL`) vrati neodspavano vrijeme.

Isečak kôda 6.1. Primjer korištenja satnih mehanizama

```
#include <stdio.h>
#include <time.h>
#include <pthread.h>

#define BROJ_DRETVI 3

static pthread_mutex_t monitor = PTHREAD_MUTEX_INITIALIZER;
static struct timespec t0, tx0;

void timespec_add ( struct timespec *A, struct timespec *B )
```

```

{
    A->tv_sec += B->tv_sec;
    A->tv_nsec += B->tv_nsec;
    if ( A->tv_nsec >= 1000000000 ) {
        A->tv_sec++;
        A->tv_nsec -= 1000000000;
    }
}

void timespec_sub ( struct timespec *A, struct timespec *B )
{
    A->tv_sec -= B->tv_sec;
    A->tv_nsec -= B->tv_nsec;
    if ( A->tv_nsec < 0 ) {
        A->tv_sec--;
        A->tv_nsec += 1000000000;
    }
}

void timestamp ( long id, char *msg, int iter )
{
    struct timespec t;
    pthread_mutex_lock ( &monitor );
    clock_gettime ( CLOCK_REALTIME, &t );
    timespec_sub ( &t, &t0 );
    printf ( "[%02ld:%06ld] Dretva %ld: %s iteracija=%d\n",
             t.tv_sec % 100, t.tv_nsec/1000, id, msg, iter );
    pthread_mutex_unlock ( &monitor );
}

#define BROJAC 30000000ULL /* Prilagoditi brzini procesora */

static void *posao_dretve ( void *param )
{
    long id = (long) param;
    int iter;
    struct timespec iduca_aktivacija, period;
    unsigned long long i;

    /* period = 500 ms * id */
    period.tv_sec = id / 2;
    period.tv_nsec = ( id % 2 ) * 500000000;

    iduca_aktivacija = tx0;

    for ( iter = 0; iter < 100; iter++ )
    {
        timestamp ( id, "POCETAK", iter );
        for ( i = 0; i < id * BROJAC; i++ )
            asm volatile (":::memory");
        timestamp ( id, "KRAJ", iter );

        timespec_add ( &iduca_aktivacija, &period );
        clock_nanosleep ( CLOCK_MONOTONIC, TIMER_ABSTIME,
                           &iduca_aktivacija, NULL );
    }

    return NULL;
}

int main ()
{
    long i;
    pthread_t tid[BROJ_DRETVI];
}

```

```

clock_gettime ( CLOCK_REALTIME, &t0 ); /* vrijeme početka */
clock_gettime ( CLOCK_MONOTONIC, &tx0 );

for ( i = 0; i < BROJ_DRETVI; i++ ) {
    if ( pthread_create ( &tid[i], NULL, posao_dretve, (void *) (i+1) ) )
    {
        perror ( "Error: pthread_create" );
        return 1;
    }
}

for ( i = 0; i < BROJ_DRETVI; i++ )
    pthread_join ( tid[i], NULL );

return 0;
}

/* Primjer pokretanja: (na jednoprocесорском sustavu !!!!)
$ gcc periodic-tasks-1.c -pthread -Wall
$ ./a.out
[00:000140] Dretva 3: POCETAK iteracija=0
[00:000460] Dretva 2: POCETAK iteracija=0
[00:008245] Dretva 1: POCETAK iteracija=0
[00:246640] Dretva 1: KRAJ iteracija=0
[00:418966] Dretva 2: KRAJ iteracija=0
[00:500457] Dretva 1: POCETAK iteracija=1
[00:522505] Dretva 3: KRAJ iteracija=0
[00:613793] Dretva 1: KRAJ iteracija=1
[01:000499] Dretva 2: POCETAK iteracija=1
[01:004525] Dretva 1: POCETAK iteracija=2
...
*/

```

6.1.1.2. Periodički alarm

Sučelje koje koristi periodički alarm, uključujući pomoćne funkcije, definirano je sa:

```

int timer_create ( clockid_t clockid,
                   struct sigevent *evp,
                   timer_t *timerid );
int timer_settime ( timer_t timerid, int flags,
                    const struct itimerspec *value,
                    struct itimerspec *ovalue );
int timer_gettime ( timer_t timerid,
                    struct itimerspec *value );
int timer_getoverrun ( timer_t timerid );
int clock_getres ( clockid_t clock_id,
                   struct timespec *res );
int sigaction ( int sig,
                const struct sigaction *act,
                struct sigaction *oact);

```

Prva funkcija, `timer_create`, samo stvara jedan alarm, ali ga ne aktivira. Funkcijom se određuje po kojem će satu odbrojavati i što će se dogoditi po aktivaciji.

Preko strukture `sigevent` definira se akcija koju treba poduzeti pri aktiviranju alarma (kada zadano vrijeme istekne). Struktura se koristi i općenitije za definiranje akcije na neki događaj, ne samo za alarne već i za signale. Struktura `sigevent` se sastoji od:

- `sigev_notify` – definira način akcije na događaj:
 - `SIGEV_NONE` – nema akcije
 - `SIGEV_SIGNAL` – akcija na događaj je slanje signala `sigev_signo`
 - `SIGEV_THREAD` – akcija na događaj je stvaranje nove dretve koja obrađuje događaj funk-

- cijom `sigev_notify_function` uz parametar `sigev_value`
- `SIGEV_THREAD_ID` – signal se ne šalje pozivajućoj dretvi (koja je postavila alarm) već dretvi definiranoj sa `sigev_notify_thread_id` (proširenje koje donosi *Linux*, nije definirano *POSIX*-om)
 - `sigev_signo` – identifikacijski broj signala koji se šalje (ako se signal šalje kao aktivacija događaja)
 - `sigev_value` – vrijednost (broj ili kazaljka) koja se šalje uz aktivaciju događaja uz signal ili kao parametar funkcije `sigev_notify_function`
 - `sigev_notify_function` – početna funkcija nove dretve stvorene kao reakcija na aktiviranje događaja (uz `sigev_notify==SIGEV_THREAD`)
 - `sigev_notify_attributes` – postavke za novu dretvu
 - `sigev_notify_thread_id` – opisnik dretve kojoj treba poslati signal (ako je akcija na događaj aktivacije slanje signala).

Opisnik stvorenog alarma (u funkciji `timer_create`) sprema se na adresu zadanu varijablom `timerid`. Ista se adresa koristi u idućim funkcijama (nakon stvaranja alarma).

Funkcijom `timer_gettime` alarm se aktivira. Struktura `itimerspec` definira vrijeme prvog pojavljivanja (aktiviranja) alarma (`value->it_value`) te naknadna periodička ponavljanja (`value->it_interval`). Svako aktiviranje alarma popraćeno je slanjem definiranog signala (u strukturi `sigevent` poziva `timer_create`) procesu (ili druge radnje, prema postavljenom ponašanju sa `timer_create`).

Vrijeme do iduće aktivacije alarma može se dohvatiti sa `timer_gettime`.

Ako je aktivacija alarma ostvarena signalom, moguće je da zbog raznih razloga (npr. zbog prioritetnijih dretvi) prethodni signal još nije prihvачen i obrađen do trenutka pojave novog signala (idući interval). Obzirom da se standardni signali ne gomilaju, operacijski sustav će po procesu zapamtiti samo jedan signal istog tipa koji čeka na obradu. Funkcijom `timer_getoverrun` može se provjeriti je došlo do takvog prekoračenja u prihvatu i obradi signala, te ako je, koliko puta.

Iako sučelje dozvoljava definiranje vremena u granulama od nanosekunde, stvarna granulacija može biti znatno grublja. Informacije o granulaciji, tj. rezoluciji korištenog sata u danom sustavu mogu se dohvatiti funkcijom `clock_getres`.

Akciju na signal (uobičajeni način aktivacije) treba definirati odgovarajućim sučeljem, primjerice sa `sigaction`, koje ima najviše mogućnosti.

Isječak kôda 6.2. Primjer korištenja periodičkih alarma

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <signal.h>

/* makro koji pojednostavljuje poziv funkcija uz provjeru grešaka */
#define WARN      0
#define STOP      1
#define CALL(ACT,FUNC,...)          \
do {                                \
    if ( FUNC ( __VA_ARGS__ ) ) { \
        perror ( #FUNC );         \
        if ( ACT == STOP )       \
            exit (1);           \
    }                                \
} while(0)

static int act[2] = { 0, 0 };

static void dretva_za_alarm ( union sigval val )
```

```

{
    printf ( "Alarm %d [%d]\n", val.sival_int, ++act[val.sival_int-1] );

    return;
}

int main ()
{
    timer_t timer1, timer2;
    struct sigevent event;
    struct itimerspec period;
    struct timespec t;

    event.sigev_notify = SIGEV_THREAD;
    event.sigev_notify_function = dretva_za_alarm;
    event.sigev_notify_attributes = NULL;

    event.sigev_value.sival_int = 1;
    CALL ( STOP, timer_create, CLOCK_REALTIME, &event, &timer1 );

    event.sigev_value.sival_int = 2;
    CALL ( STOP, timer_create, CLOCK_REALTIME, &event, &timer2 );

    period.it_value.tv_sec = period.it_interval.tv_sec = 1;
    period.it_value.tv_nsec = period.it_interval.tv_nsec = 0;
    CALL ( STOP, timer_settime, timer1, 0, &period, NULL );

    period.it_value.tv_sec = period.it_interval.tv_sec = 2;
    CALL ( STOP, timer_settime, timer2, 0, &period, NULL );

    t.tv_sec = 10;
    t.tv_nsec = 0;
    nanosleep ( &t, NULL );

    return 0;
}

```

6.1.2. Nadzorni alarm

SRSV-i se projektiraju i ispituju znatno strože od ostalih sustava. Često su oni ugrađeni u druge sustave te su stoga građeni tako da mogu dugotrajno raditi bez vanjske intervencije. Ipak, sustavi bez i jedne greške su vrlo rijetki. Razlog je naravno složenost sustava i nemogućnost potpunog ispitivanja. Neočekivani ulazi i situacije, kao primjerice kratkotrajni strujno/naponski poremećaj također uzrokuju probleme.

Što napraviti kada sustav ipak zataji, odnosno, kako izgraditi sustav da se oporavi od zatajenja? Postoji nekoliko mogućnosti.

Prva je da on sam stane i čeka intervenciju nadležnog nadzornika koji će ispitati uzroke zatajenja i ponovno pokrenuti sustav, ako to ima smisla. Pritom će uzrok zatajenja iskoristiti za ispravljanje greške u programu (ili isti dojaviti razvojnom timu). Jedan od mogućih nedostataka ovog pristupa je u vremenu rekacije nadzornika, koje se može mjeriti i satima ili čak i danima.

Drugi pristup zasniva se na vjerojatnosti da svaki kvar koji dovodi sustav do zatajenja se javlja vrlo rijetko, u sklopu poklapanja vrlo malo vjerojatnih događaja koji nisu predviđeni pri izgradnji sustava. Moguće je da se slična situacija neće još dugo ponoviti. Zato je jedan od uobičajenih postupaka pri zatajenjima u računalnom sustavu, sustav ponovno dovesti u početno stanje i pokrenuti (engleski termin *reset* se u našem jeziku udomaćio kao “resetirati”). Naravno da neće sva zatajenja biti moguće riješiti na taj način, ali mnoga hoće. Postavlja se pitanje kako usta-

noviti da je sustav zatajio i kako ga zaustaviti, dovesti u početno stanje i ponovno pokrenuti? Jedan od načina je korištenjem nadzornog alarma koji je spojen na RESET ulaz računala.

Nadzorni alarm (engl. *watchdog timer*) je brojilo koje odbrojava od zadane vrijednosti do nule, u nekom definiranom tempu. Ispravan upravljački program (dretva) će periodički poništavati nadzorni alarm i na taj način ga prisiliti da njegovo odbrojavanje ponovno krene od početne vrijednosti. Takvim poništavanjem dretva signalizira ispravni rad. Ako se nešto neočekivano dogodi (dretva se blokira, završi u beskonačnoj petlji ili neka druga programska greška u ovoj ili nekoj drugoj dretvi ili operacijskom sustavu), tj. ako sustav zataji, brojilo nadzornog alarma odbrojat će do kraja. Kada se to dogodi, nadzorni alarm će postaviti električni signal koji je spojen s RESET ulazom procesora te će se sustav ponovno pokrenuti (resetirati).

Ostvarenje nadzornog alarma zahtjeva analizu i odabir periode u kojima nadzorni alarm treba signalizirati (poništiti) od strane programa koji se nadzire. Također, signalizaciju treba ugraditi u sam program, tj. treba odabrati dijelove kôda u koji treba ugraditi pozive za signalizaciju nadzornog alarma.

Nadzorni alarm se ostvaruje kao zasebni sklop u računalu ili pak kao dio procesora.

Izvedbe nadzornog alarma mogu biti i složenije, npr. prema [Murphy, 2000]. Ponekad je bitno otkriti i prerana poništavanja ako je poznat minimalni interval koji treba proteći.

Primjer 6.1. Primjer korištenja nadzornog alarma

```
uint16 volatile *brojilo = (uint16 volatile *) 0xFF0000;

int main ()
{
    inicializacija();

    for (;;)
    {
        *brojilo = 10000;
        ocitaj_stanje_senzora();
        izracunaj_i_posalji_naredbe();
        zapisi_stanje_sustava();
    }
}
```

Nadzorni alarm bi se za manje zahtjevne sustave mogao i programski ostvariti korištenjem satnih mehanizama operacijskog sustava. Međutim, takvi nadzorni alarmi imaju značajna ograničenja u korištenju. Signali, koji se koriste za njihovu aktivaciju mogu biti programski isključeni za dotični proces te nikakve reakcije na alarm neće biti u takvom slučaju.

6.2. Semafori

Semafor je vrlo jednostavan sinkronizacijski mehanizam jezgre operacijskog sustava uz koji su pridijeljeni podaci (u jezgri): vrijednost semafora i red za blokirane dretve na tom semaforu. Semafor je "prolazan" ako mu je vrijednost veća od nule, a neprolazan ako mu je vrijednost jednaka nuli. Semafor može zbog toga brojati koliko u nekom trenutku ima nečega na raspolaganju. Dok god ima tih sredstava, dretve koje pozivaju *ČekajSemafor* će proći taj semafor i pritom smanjiti njegovu vrijednost za jedan te (nakon poziva *ČekajSemafor*) uzeti (koristiti) tražena sredstva. Kada vrijednost tog semafora postane nula, sve iduće dretve će se istim pozivom blokirati na tom semaforu. Pozivom *PostaviSemafor* vraća se sredstvo te se odblokira prva blokirana dretva na tom semaforu, ili ako nema takvih dretvi, vrijednost semafora se povećava

za jedan. Primjer sinkronizacije semaforima prikazan je u nastavku.

Primjer 6.2. Sinkronizacija jednog proizvođača s jednim potrošačem

```
dretva proizvođač
ponavljaј
    p = proizvedi_poruku ()
    ČekajSemafor (prazna)
    međuspremnik[ulaz] = p
    ulaz = (ulaz+1) mod N
    PostaviSemafor (puna)
```

```
dretva potrošač
ponavljaј
    ČekajSemafor (puna)
    r = međuspremnik[izlaz]
    izlaz = (izlaz+1) mod N
    PostaviSemafor (prazna)
    potroši_poruku (r)
```

Početna vrijednost semafora prazna mora biti postavljena na N (veličina međuspremnika), a početna vrijednost semafora puna na 0. Pomoćne varijable ulaz i izlaz početno trebaju imati vrijednost 0.

Osnovno sučelje za rad sa semaforima (prema POSIX-u) sastoji se od:

```
int sem_post ( sem_t *sem );
int sem_wait ( sem_t *sem );
int sem_trywait (sem_t *sem );
int sem_timedwait ( sem_t *sem, const struct timespec *max_wait );
```

Sučelje `sem_trywait` je neblokirajuća inačica poziva za čekanje, koja će u slučaju da se semaforu vrijednost ne može smanjiti za jedan vratiti grešku kao povratnu vrijednost te pritom neće blokirati dretvu.

Ograničeno blokiranje pruža sučelje `sem_timedwait` koje će nakon isteka zadanog vremena dretvu odblokirati (ako se u međuvremenu nije odblokirala uobičajenim načinom, pozivom `sem_post` od strane neke druge dretve).

Drugo sučelje za rad s ponešto drukčijim semaforima ponešto drukčijih mogućnosti uključuje: `semget`, `semop` te `semctl`.

Semafori su gotovo najjednostavniji sinkronizacijski mehanizam te su pogodni za jednostavnije potrebe. Međutim, zbog toga su neprikladni za sinkronizacije kojima se štiti više od jednog sredstva. Korištenje u takvima situacijama može dovesti do problema potpunog zastoja.

6.3. Potpuni zastoj

Potpuni zastoj (engl. *deadlock*) je stanje sustava dretvi u kojemu su sve dretve blokirane te ne postoji mogućnost njihova nastavka rada. Potpuni zastoj može nastati u sustavu s više dretvi od kojih svaka u nekom trenutku svog izvođenja traži više od jednog sredstva, a svako je sredstvo zaštićeno zasebnim sinkronizacijskim objektom.

Najjednostavniji primjer uključuje dvije dretve i dva sredstva. U takvom sustavu može se dogoditi da svaka dretva zauzme po jedno sredstvo te se blokira na zahtjevu za drugim (koje ima druga dretva). Najčešći primjer koji se spominje u ovom kontekstu jest problem pet filozofa smislen od strane Edsgera Dijkstre 1965. te kasnije korišten u gotovo svakoj literaturi te tematike (npr. [Budin, 2011]).

Primjer 6.3. Sinkronizacija više proizvođača i više potrošača, s greškom

Neka se u pokušaju proširenja rješenja iz primjera 6.2. na više proizvođača i više potrošača, doda dodatni semafor `ko`, s početnom vrijednošću 1 koji će spriječiti više istovremenih pristupa međuspremniku.

```
dretva proizvođač
ponavljam
    p = proizvedi_poruku ()
    ČekajSemafor (prazna)
    ČekajSemafor (ko)
    međuspremnik[ulaz] = p
    ulaz = (ulaz+1) mod N
    PostaviSemafor (ko)
    PostaviSemafor (puna)
```

```
dretva potrošač
ponavljam
    ČekajSemafor (ko)
    ČekajSemafor (puna)
    r = međuspremnik[izlaz]
    izlaz = (izlaz+1) mod N
    PostaviSemafor (prazna)
    PostaviSemafor (ko)
    potroši_poruku (r)
```

Zbog različitog dodavanja tog semafora kod proizvođača i potrošača (greškom), može se dogoditi da jedan potrošač prođe semafor `ko` (i pritom ga postavlja u neprolazno stanje) te se i sam blokira na semaforu `puna` (u početku je međuspremnik prazan). Nakon toga niti jedan potrošač neće moći ući u svoj kritični odsječak i staviti poruku u međuspremnik (i pozvati `PostaviSemafor`) te će se sve dretve blokirati – nastaje potpuni zastoj.

U prethodnom primjeru se potpuni zastoj može izbjegći opreznim programiranjem. Međutim, općenito se problem potpunog zastaja ne može riješiti na taj način. Zato je preporuka da se u slučaju potrebe složenijeg sinkronizacijskog mehanizma ne koriste semafori već monitori.

6.4. Monitori

Sinkronizacijski mehanizam monitora omogućava da se problemi dostupnosti sredstava izražavaju proizvoljno i programski ispituju izvan jezgre, a ne kao kod semafora u jezgri operacijskog sustava korištenjem vrijednosti semafora. Zato se pri korištenju monitora treba definirati i dodatna struktura podataka koja će pratiti stanje sustava (raspoloživost sredstava).

Monitor se ostvaruje sučeljem jezgre operacijskog sustava:

- ulazak u monitor (`Uđi_u_monitor`),
- izlazak iz monitora (`Izađi_iz_monitora`)
- blokiranje unutar monitora (`Čekaj_u_red_uvjeta`)
- propuštanje blokiranih dretvi (`Oslobodi_iz_reda_uvjeta`).

Pokažimo primjenu tih funkcija na rješavanje istog problema sinkronizacije i komunikacije dretvi proizvođača i potrošača.

Primjer 6.4. Sinkronizacija proizvođača i potrošača s monitorom

```
dretva proizvođač
ponavljam
    p = proizvedi_poruku ()
    Uđi_u_monitor (m)
    dok je ( br_poruka == N )
        Uvrsti_u_red_uvjeta ( prazna, m )
        međuspremnik[ulaz] = p
        ulaz = (ulaz + 1) mod N
        br_poruka = br_poruka + 1
    Oslobodi_iz_reda_uvjeta ( puna, m )
    Izađi_iz_monitora (m)
```

```
dretva potrošač
```

```

ponavlja
Uđi_u_monitor (m)
dok je ( br_poruka == 0 )
    Uvrsti_u_red_uvjeta ( puna, m )
    r = međuspremnik[izlaz]
    izlaz = ( izlaz + 1 ) mod N
    br_poruka = br_poruka - 1
    Oslobodi_iz_reda_uvjeta ( prazna, m )
    Izadi_iz_monitora (m)
    potroši_poruku (r)

```

Umjesto brojača broja slobodnih i punih mjesta u međuspremniku koji su u primjeru sa semaforima bili dio semafora, sada je potrebno dodatnim varijablim `br_poruka` pratiti koliko se poruka nalazi u međuspremniku. Ta se varijabla ispituje nakon ulaska u monitor, a prije korištenja međuspremnika (za stavljanje nove poruke ako ima mjesta, te za uzimanje poruke, ako međuspremnik nije prazan).

Uvjet koji se ispituje unutar monitora, a koji će služiti za odluku da li dretvu pustiti dalje ili ju blokirati, može biti vrlo složen (u prethodnom primjeru je to samo ispitivanje jedne varijable). Zbog toga se i u složenim sinkronizacijskim problemima monitori na isti način koriste – samo je uvjet ispitivanja drugi. Potpuni zastoj je stoga mnogo jednostavnije izbjegći.

Osim već osnovnih funkcija za ostvarenje monitora, kao i kod sučelja za semafore i kod monitora imamo dodatne mogućnosti u obliku neblokirajućih „čekaj“ funkcija i vremenski ograničenih blokiranja. Popis najbitnijih sučelja je:

```

int pthread_mutex_init      ( pthread_mutex_t *mutex,
                               const pthread_mutexattr_t *attr );
int pthread_mutex_lock     ( pthread_mutex_t *mutex );
int pthread_mutex_unlock   ( pthread_mutex_t *mutex );
int pthread_cond_init      ( pthread_cond_t *cond,
                           const pthread_condattr_t *attr );
int pthread_cond_wait      ( pthread_cond_t *cond,
                           pthread_mutex_t *mutex );
int pthread_cond_signal    ( pthread_cond_t *cond );
int pthread_cond_broadcast ( pthread_cond_t *cond );
int pthread_mutex_trylock  ( pthread_mutex_t *mutex );
int pthread_mutex_timedlock ( pthread_mutex_t *mutex,
                           const struct timespec *abstime );
int pthread_cond_timedwait ( pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime );

```

6.5. Rekurzivno zaključavanje

Što napraviti ako dretva koja je već zaključala semafor ili monitor opet pokuša zaključati isti objekt (semafor ili monitor)? Ponekad je takvo ponašanje donekle opravdano. Npr. neka se iz početne funkcije koja je ušla u monitor pozivaju druge funkcije koje i same imaju zaštitu od paralelnog pozivanja korištenjem istog monitora (zato jer se pozivaju i izvan prve funkcije, tj. izvan monitora), kao u primjeru:

<pre> funkcija_1 () Uđi_u_monitor (m) nešto_radi funkcija_2() još_radi Izadi_iz_monitora (m) </pre>

<pre> funkcija_2 () Uđi_u_monitor (m) nešto_drugo_radi Izadi_iz_monitora (m) </pre>

Izlaziti iz monitora da bi se u njega opet ušlo može, osim nepraktičnosti, biti i loše rješenje (logički neispravno). Problem se može riješiti na nekoliko načina. Jedan od njih je i korištenje podrške za rekurzivno zaključavanje koje nude neki sinkronizacijski mehanizmi.

Takva podrška je predviđena POSIX standardom za mehanizam monitora, odnosno, za pozive `pthread_mutex_lock`/`pthread_mutex_unlock`. Prilikom stvaranja (inicijalizaciji) takvog monitora potrebno je pozivom `pthread_mutexattr_settype` postaviti tip (`type`) `PTHREAD_MUTEX_RECURSIVE` atributu kojim se inicijalizira objekt zaključavanja.

Drugi način rješavanja prethodnog problema zahtjeva promjenu u izvornim kôdovima tako da se operacije u pseudokôdu označene sa `nešto_drugo_radi` ostvare u zasebnoj funkciji koja nije zaštićena monitorom te da se poziva iz obje funkcije, tj. iz `funkcija_1` i `funkcija_2` (unutar monitora).

6.6. Problem inverzije prioriteta

U ugrađenim sustavima određenost, pouzdanost i jednostavnost mnogo su značajniji od prosječne učinkovitosti. Dretve u ugrađenim sustavima se međusobno razlikuju po važnosti, odnosno prioritetu. Pri dodjeli procesora dretva većeg prioriteta ima prednost ispred one manjeg prioriteta. Iako je takav ili slični slučaj i u ostalim sustavima, odnosno, ostalim operacijskim sustavima koji nisu namijenjenih SRSV-ima, kod njih je prioritet uglavnom nešto što određuje koliko će pojedina dretva dobiti procesorskog vremena, a ne kada će ta dretva postati aktivna. U operacijskim sustavima za rad u stvarnom vremenu i za ugrađene sustave jasno je definirano da kada dretva višeg prioriteta postaje spremna za izvođenje ona istiskuje dretvu nižeg prioriteta koja se trenutno izvodi. Međutim, i u takvim se sustavima događaju slučajevi kada dretva nižeg prioriteta zaustavi izvođenje dretve višeg prioriteta, odnosno, dolazi do problem *inverzije prioriteta*.

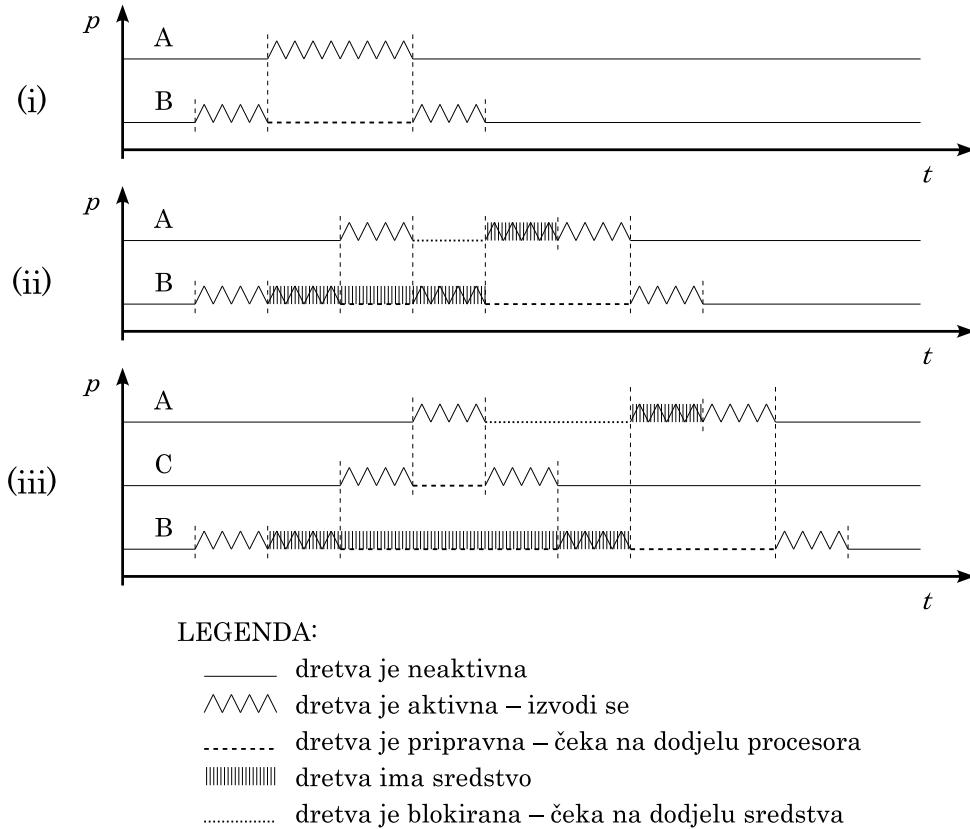
Problem inverzije prioriteta nastaje kada dretva višeg prioriteta za nastavak rada treba sredstvo koje je zauzela druga dretva nižeg prioriteta. Slika 6.2. prikazuje tri slučaja koja se mogu pojaviti u višedretvenom sustavu.

Slika 6.2.(i) prikazuje uobičajeno ponašanje kada su u sustavu dvije dretve različitog prioriteta. Aktiviranjem prioritetnije dretve, manje prioritetna se istisne, tj. makne s procesora. Na slici, čim dretva A postane spremna istisne dretvu B, koja ima manji prioritet.

Slika 6.2.(ii) pokazuje sličnu situaciju, ali dretva A i B tijekom rada koriste zajedničko sredstvo i to međusobno isključivo (npr. zaštićeno binarnim semaforom). Dretva manjeg prioriteta B za vrijeme svog rada, dok je A bila neaktivna, zauzela je sredstvo. Kasnije se dretva A aktivirala te odmah istisnula dretvu B. Međutim, u jednom trenutku dretvi A za nastavak rada treba sredstvo koji dretva B još nije otpustila (binarni semafor je neprolazan). Dretva A se zaustavi te dretva B, kao jedina dretva u sustavu nastavlja s radom. Problem koji je nastao naziva se problem inverzije prioriteta jer dretva manjeg prioriteta radi, dok dretva većeg prioriteta čeka na nju. Međutim, čim dretva B oslobodi zauzeto sredstvo, dretva A se otpusti i odmah zauzima sredstvo i nastavlja s radom (primjerice kad dretva B pozove *PostaviSemafor*).

Slika 6.2.(iii) prikazuje situaciju kad u sustavu osim dretvi A i B postoji i treća dretva C prioriteta većeg od dretve B, ali manjeg od dretve A. Kao što se vidi iz grafa, ovakva dretva može dodatno odgoditi izvođenje dretve A. U sustavima s više dretvi vrlo je teško procijeniti koliko se dretva A može odgoditi. Zato je problem inverzije prioriteta vrlo opasan za sustave za rad u stvarnom vremenu.

Metode koje se najčešće koriste u slučajevima problema inverzije prioriteta ne rješavaju sam problem nego ublažavaju njegove posljedice. To rade tako da se dretvi koja je zauzela sredstva potrebna prioritetnije dretvi (korištenjem sinkronizacijskog mehanizma) omogući što brži rad do oslobođanja dotočnih sredstava. Dvije najpoznatije takve metode su:



Slika 6.2. Problem inverzije prioriteta

- protokol nasljeđivanja prioriteta (engl. priority inheritance protocol) i
- protokol stropnog prioriteta (engl. priority ceiling protocol).

Važna pretpostavka za oba protokola je da dretve nakon što sredstvo zauzmu isto će i oslobođiti nakon nekog vremena.

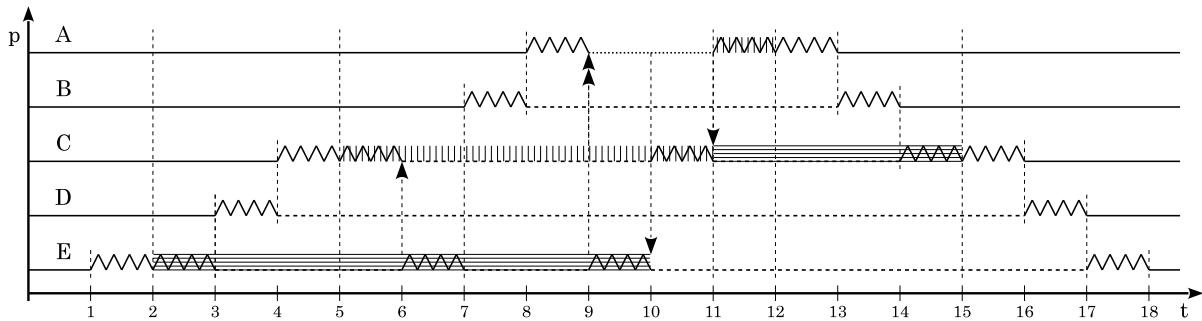
6.6.1. Protokol nasljeđivanja prioriteta

Protokol nasljeđivanja prioriteta privremeno podiže prioritet dretvi manjeg prioriteta kada ona blokira dretvu većeg prioriteta – dretva manjeg prioriteta nasljeđuje prioritet blokirane dretve. Nasljeđivanje se događa u trenutku blokiranja dretve većeg prioriteta (npr. kada ona pozove *ČekajSemafor*).

Pri otpuštanju sredstva, dretvi se prioritet vraća na prijašnju vrijednost te konačno dretva višeg prioriteta zauzima sredstvo i nastavlja s radom. Povećanje prioriteta može biti i tranzitivno: najprije se poveća prioritet dretve koja drži sredstvo, a potom i prioritet dretvi koja blokira prethodnu radi drugog sredstva, itd. Kao i prethodnu funkciju i ovu mora slijediti poziv raspoloživača koji će uzeti u obzir nove prioritete.

Primjer rada protokola nasljeđivanja prioriteta prikazan je na slici 6.3.

Uz protokol nasljeđivanja prioriteta veže se zanimljivost iz svemirske misije na Mars robotske letjelice *Mars Pathfinder* [Jones, 1997] koja je bila upravljana operacijskim sustavom [VxWorks]. Nakon uspešnog slijetanja i prvih nekoliko dana rada, na sustavu upravljanja počeli su se pojavljivati neobjašnjivi i učestali prekidi. Kako je sustav napravljen da prilikom detekcije greške prekida rad i ponovo pokreće cijeli sustav (mehanizmom nadzornog alarma) sve je do određene mјere i dalje radilo. Greška koja je za nekoliko dana pronađena te potom i uklonjena jest u protokolu nasljeđivanja prioriteta, koji je bio greškom isključen. Problem je nastao kada je prioritetnoj dretvi na dulje vrijeme bio uskraćen pristup sabirnici što je sklop za nadzor inter-



1. E, kao jedina pripravna dretva započinje s radom
2. E zauzima sredstvo S_1 (vodoravne crte)
3. D započinje s radom i istiskuje E jer ima najveći prioritet (u tom trenutku)
4. C započinje s radom i istiskuje D jer ima najveći prioritet
5. C zauzima sredstvo S_2 (okomite crtice)
6. C treba S_1 koji je E zauzeo, E nasljeđuje prioritet od C i nastavlja s radom (s prioritetom od C)
7. B započinje s radom i istiskuje E jer ima najveći prioritet
8. A započinje s radom i istiskuje B jer ima najveći prioritet
9. A treba S_2 koji je C zauzeo, C nasljeđuje prioritet od A, E nasljeđuje prioritet od C i nastavlja s radom (s prioritetom od C)
10. od A)
11. E oslobađa S_1 , vraća mu se početni prioritet, C zauzima S_1 i nastavlja s radom (s prioritetom od A)
12. C oslobađa S_2 , vraća mu se početni prioritet, A zauzima S_2 i nastavlja s radom
13. A završava, B nastavlja s radom (ima najveći prioritet)
14. B završava, C nastavlja s radom (ima najveći prioritet)
15. C oslobađa S_1 i nastavlja s radom
16. C završava, D nastavlja s radom (ima najveći prioritet)
17. D završava, E nastavlja s radom (jedina dretva)
18. E završava

Slika 6.3. Primjer korištenja protokola nasljeđivanja prioriteta

pretirao kao kritičnu grešku te je zaustavio i ponovo pokrenuo sustav. Uključivanje protokola nasljeđivanja prioriteta riješilo je problem.

Protokol nasljeđivanja prioriteta ne rješava problem potpunog zastoja već se za sprječavanje i rješavanje istog moraju upotrijebiti dodatni algoritmi ili postupci.

6.6.2. Protokol stropnog prioriteta

Za protokol stropnog prioriteta postoje dvije inačice:

- izvorni protokol stropnog prioriteta i
- pojednostavljeni protokol stropnog prioriteta.

Pojednostavljeni protokol ima još nekoliko naziva: izravni protokol stropnog prioriteta (engl. *immediate ceiling priority protocol*), protokol zaštite prioritetom (engl. *priority protect protocol*) kod POSIX standarda te oponašanje protokola stropnog prioriteta (engl. *priority ceiling emulation*) kod programskog jezika Java.

6.6.2.1. Izvorni protokol stropnog prioriteta

Izvorni protokol ima za cilj i izbjegavanje nastajanja potpunog zastoja. Načelna ideja protokola jest da kada dretva želi zauzeti neki semafor tada ona mora imati veći prioritet od najveća stropna prioriteta nekog od već zauzetih semafora. Ako nema takav prioritet, onda joj se ne dopušta da zauzme semafor iako je on možda i u prolaznom stanju. Definicija protokola koja slijedi zasniva se na opisu iz [Rajkumar, 1991].

Osnovne prepostavke:

1. Razmatra se skup dretvi $\{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_N\}$ koje u dijelovima svog izvođenja koriste razna sredstva zaštićena binarnim semaforima $\{S_1, S_2, \dots, S_M\}$. Za svaku je dretvu unaprijed poznato koje semafore će možda trebati tijekom rada, odnosno, za svaki je semafor poznato koje ga dretve mogu koristiti. Za svaki se semafor izračunava stropni prioritet $p(S_i)$ tako da se odabere najveća vrijednost među prioritetima dretvi koje koriste taj semafor.

2. Koristi se prioritetno raspoređivanje – pripravna dretva najvećeg prioriteta se izvodi.

Rad protokola:

1. Dretva \mathcal{D}_i poziva $\check{\text{CekajSemafor}}(S_x)$

- a) Niti jedan semafor nije zauzet – svi su prolazni:
 - poziv na $\check{\text{CekajSemafor}}(S_x)$ neće blokirati dretvu \mathcal{D}_i (poziv radi uobičajeno).
- b) Semafor S_x je već zauzet od dretve \mathcal{D}_j :
 - poziv na $\check{\text{CekajSemafor}}(S_x)$ blokira dretvu \mathcal{D}_i
 - ukoliko je $p(\mathcal{D}_j) < p(\mathcal{D}_i)$, dretva \mathcal{D}_j nasljeđuje prioritet od $\mathcal{D}_i : p(\mathcal{D}_j) = p(\mathcal{D}_i)$.
- c) Semafor S_x nije zauzet, ali neki drugi semafori jesu:
 - neka S^* označava semafor najvećeg stropnog prioriteta $p(S^*)$ koji je zaključan od strane dretve \mathcal{D}_k (vrijedi $p(S^*) \geq p(\mathcal{D}_k)$)
 - ukoliko je $p(\mathcal{D}_i) > p(S^*)$ poziv $\check{\text{CekajSemafor}}(S_x)$ neće blokirati dretvu \mathcal{D}_i
 - ukoliko je $p(\mathcal{D}_i) \leq p(S^*)$ poziv $\check{\text{CekajSemafor}}(S_x)$ će blokirati dretvu \mathcal{D}_i te ukoliko je $p(\mathcal{D}_k) < p(\mathcal{D}_i)$, dretva \mathcal{D}_k nasljeđuje prioritet od $\mathcal{D}_i : p(\mathcal{D}_k) = p(\mathcal{D}_i)$.

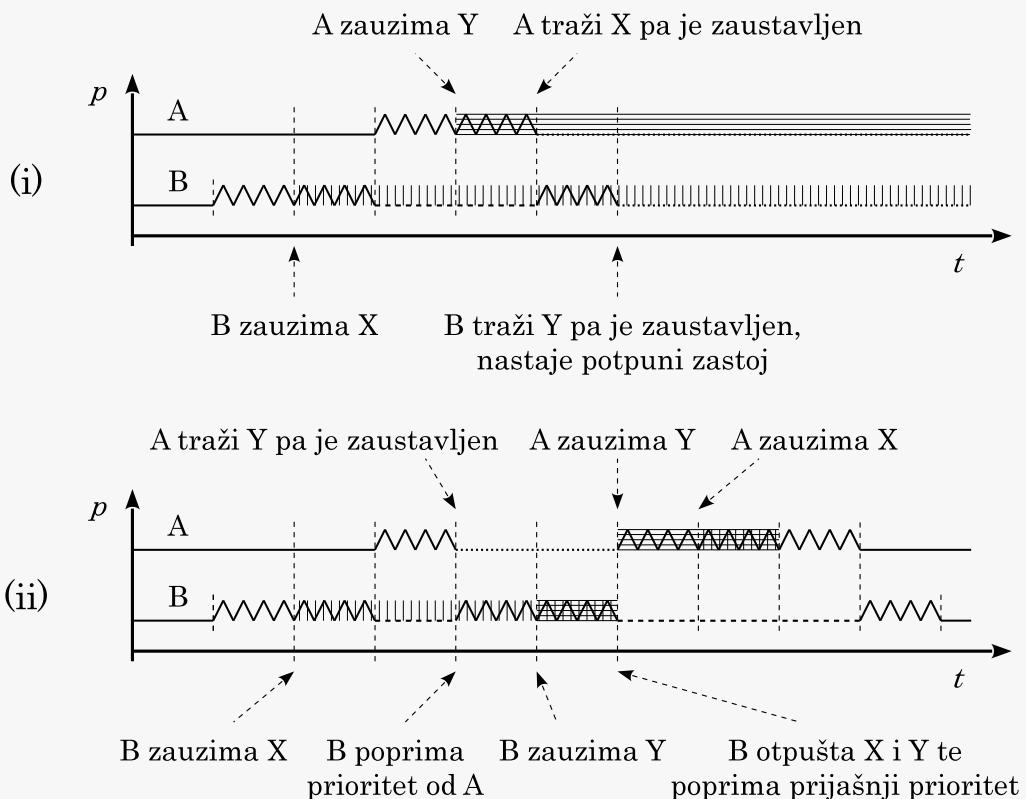
2. Dretva \mathcal{D}_i poziva $\text{PostaviSemafor}(S_x)$.

- a) Funkcija $\text{PostaviSemafor}(S_x)$ obavlja svoju uobičajenu operaciju (propušta prvu blokirajuću dretvu ako takva postoji ili postavlja semafor u prolazno stanje) te
- b) ukoliko je dretvi \mathcal{D}_i bio povećan prioritet radi korištenja semafora S_x , prioritet joj se vraća na prijašnju vrijednost.

Analizom ovog protokola može se ustanoviti sličnost s protokolom nasljeđivanja prioriteta. I kod ovog se protokola prioritet zaustavljene dretve nasljeđuje u trenutku zaustavljanja. Osnovna je razlika što se u nekim slučajevima dretvi neće dozvoliti zaključavanje semafora iako je on slobodan, a zbog mogućeg zaustavljanja dretvi većeg prioriteta nad drugim semaforima. Na prvi pogled to zaustavljanje izgleda nepotrebno, ali ono zapravo pridonosi rješavanju problema potpunog zastoja, barem osnovnih oblika potpunog zastoja. U slučajevima gdje je problem potpunog zastoja značajan i često se pojavljuje, ovaj protokol može biti dobar odabir.

Primjer 6.5. Izbjegavanje potpunog zastoja

Slika 6.4. prikazuje primjer pojavljivanja potpunog zastoja (i) kao i njegovog izbjegavanja korištenjem protokola stropnog prioriteta (ii).



Slika 6.4. Primjer potpunog zastoja i kako ga stropni protokol izbjegava

Primjer 6.6. Rad izvornog protokola stropnog prioriteta

Prepostavimo da u nekom sustavu imamo 4 dretve, svaku zadanu s prioritetima i sredstvima koje koristi u dijelovima svog izvođenja prema tablici 6.1. Analizom tih podataka mogu se odrediti stropni prioriteti za sredstva prema tablici 6.2.

Tablica 6.1. Sustav zadataka

Dretva	Prioritet	Sredstva
\mathcal{D}_1	20	S_1, S_2
\mathcal{D}_2	15	S_2, S_3
\mathcal{D}_3	10	S_3, S_4
\mathcal{D}_4	5	S_1, S_3

Tablica 6.2. Stropni prioriteti

Sredstvo	Stropni prioritet
S_1	20
S_2	20
S_3	15
S_4	10

Prikažimo primjer mogućeg odvijanja događaja u sustavu. Neka su sredstva semafori i neka su svi početno postavljeni u vrijednost 1.

1. \mathcal{D}_1 se prva pojavljuje i poziva $\check{C}ekajSemafor(S_3)$

Obzirom da niti jedan semafor nije još zaključan (ne postoji S^*), a i S_3 je prolazan,

poziv neće blokirati te će dretva \mathcal{D}_4 nastaviti s radom.

- \mathcal{D}_2 se pojavljuje te kao dretva najvećeg prioriteta istiskuje \mathcal{D}_4 te se izvodi.
 - U nekom trenutku \mathcal{D}_2 poziva: $\text{\textit{ČekajSemafor}}(S_2)$
 - S^* (zaključani semafor najvećeg prioriteta) je S_3 sa stropnim prioritetom 15
 - \mathcal{D}_2 nema veći prioritet od 15 (ima samo 15), pa se blokira
 - \mathcal{D}_4 zbog toga (blokiranje \mathcal{D}_2) poprima prioritet od \mathcal{D}_2 (15) i nastavlja s radom
 - \mathcal{D}_1 se pojavljuje i poziva $\text{\textit{ČekajSemafor}}(S_1)$
 - S^* (zaključani semafor najvećeg prioriteta) je S_3 sa stropnim prioritetom 15
 - \mathcal{D}_1 ima veći prioritet od 15 (ima 20), pa prolazi – zaključava S_1 i nastavlja s radom
 - S^* je sada S_1
 - \mathcal{D}_1 se blokira na nekom drugom redu (npr. čeka dovršetak neke ulazno-izlazne operacije)
 - \mathcal{D}_4 nastavlja s radom
 - \mathcal{D}_4 otpušta sredstvo, tj. poziva $\text{\textit{PostaviSemafor}}(S_3)$
 - Semafor S_3 se otključava
 - \mathcal{D}_4 poprima prijašnji prioritet, tj. 5
 - Provjerava se mogućnost propuštanja blokiranih dretvi, tj. dretve \mathcal{D}_2 :
 - Obzirom da je stropni prioritet od $S^* = S_1$ i dalje veći od prioriteta \mathcal{D}_2 , \mathcal{D}_2 ostaje blokirana.
 - \mathcal{D}_4 nastavlja s radom, te nakon nje i druge dretve.

Izvorni protokol stropnog prioriteta prepostavlja mogućnost da će dretva D_1 trebati i S_2 te ne dozvoljava D_2 da ga zauzme, obzirom da je D_1 već zauzeo S_1 . Međutim, također treba primijetiti da dretva koja je zauzela semafor ne bi trebala odgađati svoj rad prije nego li otpusti semafor, tako da bi se vrlo rijetko trebala pojavljivati situacija u kojoj D_2 čeka na dretvu koja se neće izvoditi odmah nakon blokiranja D_2 .

6.6.2.2. Pojednostavljeni protokol stropnog prioriteta

Kod pojednostavljenog protokola stropnog prioriteta dretvi se odmah pri zauzimanju sredstva podigne prioritet na unaprijed izračunatu stropnu vrijednost. Na taj se način za vrijeme korištenja nekog sredstva dretvama povećava prioritet da bi one što prije završile s njegovim korištenjem i oslobostile ga za dretve višeg prioriteta. Protokol je jednostavniji za ostvarenje od prethodnog, ali je po pitanju učinkovitosti lošiji. Dretva nižeg prioriteta zauzećem određenog sredstva dobiva veći prioritet i istiskuje prioritetnije dretve čak i onda kada od dretvi višeg prioriteta ne postoji potreba za sredstvom.

6.6.3. POSIX funkcije za rješavanje problema inverzije prioriteta

POSIX definira mogućnost korištenja protokola nasljeđivanja prioriteta te pojednostavljenog protokola stropnog prioriteta na mehanizmu monitora, tako da se prije inicijalizacije monitora postave odgovarajuće zastavice attributa `attr` kojim se on inicijalizira.

Sučelje za postavljanje protokola za problem inverzije prioriteta je:

```
int pthread_mutexattr_setprotocol ( pthread_mutexattr_t *attr,  
                                    int protocol );
```

Protokol se odabire drugim parametrom. Mogućnosti su:

- PTHREAD_PRIO_NONE – bez korištenje ijednog protokola
 - PTHREAD_PRIO_INHERIT – korištenje nasljeđivanja prioriteta

- PTHREAD_PRIO_PROTECT – korištenje stropnog prioriteta.

Kada se koristi zadnja opcija, PTHREAD_PRIO_PROTECT, tada treba definirati i stropni prioritet pridijeljen monitoru funkcijom:

```
int pthread_mutex_setprioceiling ( pthread_mutex_t *mutex,
                                    int prioceiling,
                                    int *old_ceiling );
```

6.7. Signali

Signali su jedan od načina asinkrone komunikacije među dretvama, ali i način s kojim se dretvama mogu “dojaviti” razni događaji koje otkriva operacijski sustav. Dretve koje prime signal mogu tada (u tom trenutku) reagirati, primjerice privremeno prekinuti s trenutnim poslom (nizom instrukcija) te pozvati funkciju za obradu tog događaja. Po obradi događaja dretva se vraća prijašnjem poslu (nastavlja gdje je stala prije nego li je bila prekinuta).

Dretva koja prima signal na njega može reagirati tako da:

- prihvati signal i obradi ga zadanom funkcijom (postavljenoj pri inicijalizaciji prihvata signala)
- prihvati signal i obradi ga prepostavljenom funkcijom (definiranom u bibliotekama koje koristi program)
- privremeno ignorira signal (signal ostaje na čekanju)
- odbacuje signal (ne poduzima nikakve akcije na njega, niti ga pamti).

Sučelje za signale uglavnom spominje procese, a ne dretve. Međutim, za očekivati je da će se u skoroj budućnosti više pažnje posvetiti dretvama i da će i ona sučelja koja su definirana standardom biti sve više ostvarivana u operacijskim sustavima.

Osnovne funkcije za upravljanje signalima su:

```
int sigaction ( int sig,
                const struct sigaction *act,
                struct sigaction *oact);
int kill ( pid_t pid, int sig );
int pthread_kill ( pthread_t thread, int sig );
int raise ( int sig );
int sigqueue ( pid_t pid, int signo, const union sigval value );
```

Poziv `sigqueue` jedini omogućava slanje signala uz koje ide i dodatna informacija. Za prihvatanje takvih signala funkcija za obradu mora imati dodatne parametre, npr. prema:

```
void obrada_signala ( int signum, siginfo_t *info, void *context );
```

ili takve signale dohvaćati funkcijom:

```
int sigwaitinfo ( const sigset_t *set, siginfo_t *info);
```

Mnogi mehanizmi se zasnivaju na signalima. Primjerice, signali se koriste za ostvarivanje odgode izvođenja, periodičko pokretanje, dojavu promjena na ulazno-izlaznim napravama.

Ipak, pri korištenju signala na (starijim) sustavima koji nisu predviđeni za rad u stvarnom vremenu, treba biti oprezan, zato jer mnoge funkcionalnosti koje definira POSIX ne moraju biti do kraja ostvarene. Nadalje, signali prekidaju neke jezgrine funkcije, tj. ukoliko je dretva bila blokirana nekom jezgrinom funkcijom (npr. odgoda, sinkronizacija, komunikacija s UI napravama) moguće je da će signal upućen takvoj dretvi prekinuti to blokirano stanje (nakon obrade signala). Za detalje je potrebno pogledati upute uz svaku jezgrinu funkciju zasebno.

Isječak kôda 6.3. Primjer korištenja signala

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <pthread.h>

void obrada_signal ( int sig, siginfo_t *info, void *context )
{
    int j;

    printf ( "[Obrada signal] zapocela obrada za signal %d", sig );
    if ( info != NULL && info->si_code == SI_QUEUE )
        printf ( " [.sival_int = %d / .sival_ptr = %p]\n",
                 info->si_value.sival_int, info->si_value.sival_ptr );
    else
        printf ( "\n" );

    for ( j = 1; j <= 5; j++ ) {
        printf("[Obrada signal] obradujem signal %d: %d/5\n", sig, j);
        sleep(1);
    }

    printf("[Obrada signal] zavrsena obrada za signal %d\n", sig);
}

static pid_t pid_roditelja;
static pthread_t opisnik_glavne_dretve;

void *posao_dretve ( void *p )
{
    union sigval v;

    sigset(SIGUSR1, &posao_dretve);
    sigadd(SIGUSR2, &posao_dretve);

    v.sival_ptr = NULL;

    sleep(1);
    printf ( "[Dretva] saljem SIGUSR1 (glavnoj dretvi)\n" );
    pthread_kill ( opisnik_glavne_dretve, SIGUSR1 );

    sleep(3);
    printf ( "[Dretva] saljem {SIGUSR1, 1} (procesu)\n" );
    v.sival_int = 1;
    sigqueue ( pid_roditelja, SIGUSR1, v );

    sleep(10);
    printf ( "[Dretva] saljem {SIGUSR2, 2} (procesu)\n" );
    v.sival_int = 2;
    sigqueue ( pid_roditelja, SIGUSR2, v );

    return NULL;
}

int main ()
{
    struct sigaction act;
    sigset(SIGUSR1, &posao_dretve);
    pthread_t opisnik_dretve;
    siginfo_t info;

    pid_roditelja = getpid();
}
```

```

opisnik_glavne_dretve = pthread_self();

/* stvori dretvu koja salje signale */
pthread_create ( &opisnik_dretve, NULL, posao_dretve, NULL );

/* postavi da se signal SIGUSR1 obradjuje funkcijom */
act.sa_sigaction = obrada_signala;
sigemptyset ( &act.sa_mask );
sigaddset ( &act.sa_mask , SIGUSR1 );
act.sa_flags = SA_SIGINFO;
sigaction ( SIGUSR1, &act, NULL );

/* maskiraj signal SIGUSR2 (dohvaćaj ga sa sigwait) */
sigemptyset ( &sigmask );
sigaddset ( &sigmask, SIGUSR2 );
pthread_sigmask ( SIG_BLOCK, &sigmask, NULL );

printf ( "[Glavna dretva] cekam na SIGUSR2\n" );

while ( sigwaitinfo ( &sigmask, &info ) == -1 )
    perror ( "sigwaitinfo" ); /* interrupted with SIGUSR1? */

printf ( "[Glavna dretva] primljen signal %d", info.si_signo );
if ( info.si_code == SI_QUEUE )
    printf ( " [.sival_int = %d / .sival_ptr = %p]\n",
            info.si_value.sival_int, info.si_value.sival_ptr );
else
    printf ( "\n" );

pthread_join ( opisnik_dretve, NULL );

return 0;
}

```

6.8. Ostvarivanje međudretvene komunikacije

Međudretvena komunikacija najčešće se obavlja korištenjem zajedničkog adresnog prostora procesa, ali i drugim mehanizmima kao što su redovi poruka i cjevovodi. Korištenje zajedničkog adresnog prostora podrazumijeva korištenje sinkronizacijskih mehanizama za ostvarivanje kritičnog odsječka, npr. semafora ili monitora. Redovi poruka i cjevovodi su zasebni mehanizmi sa zasebnim sučeljima.

6.8.1. Zajednički spremnik

Sve dretve istog procesa dijele cijeli adresni prostor tog procesa – dretve istog procesa mogu razmjenjivati podatke preko varijabli i objekata u tom procesu. Međutim, dretve različitih procesa za ostvarenje istog načina komunikacije trebaju korištenjem operacijskog sustava uspostaviti dio spremnika koji će biti dostupan dretvama oba procesa – zajednički spremnički prostor (engl. *shared memory*). Sučelja za to postoje u većini sustava i zasnivaju se na raznim načelima. Jedan od takvih načela jest preslikavanje datoteka u radni spremnik procesa, kod kojeg se najprije stvori posebna vrsta datoteke i nju mapira u radni spremnik svih procesa koji žele zajedno komunicirati.

Skup sučelja za ostvarenje navedena načina korištenja zajedničkog spremnika sastoji se od:

```

/* stvaranje objekta zajedničkog spremnika */
int shm_open ( const char *name, int oflag, mode_t mode );

/* brisanje objekta zajedničkog spremnika */
int shm_unlink ( const char *name );

/* postavljanje veličine datoteke (pa i zajedničkog spremnika) */
int ftruncate ( int fildes, off_t length );

/* preslikavanje (mapiranje) datoteke u adresni prostor procesa */
void *mmap ( void *addr, size_t len, int prot, int flags,
             int fildes, off_t off );

/* odvajanje preslikanog dijela spremnika od adresnog prostora procesa */
int munmap ( void *addr, size_t len );

```

Isječak kôda 6.4. Primjer korištenja zajedničkog spremnika

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <fcntl.h>

struct dijeljeno {
    int a, b;
};

#define NAZIV_ZAJ_SPREMNIKA "/fibonacci" /* napravljeno u /dev/shm/ */
#define VELICINA sizeof (struct dijeljeno)
#define BROJ_PROCESA 10

void posao_procesa ( int proc_id )
{
    int id;
    struct dijeljeno *x;

    sleep ( proc_id ); /* svaki novi proces kreće sekundu kasnije */

    id = shm_open ( NAZIV_ZAJ_SPREMNIKA, O_CREAT | O_RDWR, 00600 );
    if ( id == -1 || ftruncate ( id, VELICINA ) == -1 ) {
        perror ( "shm_open/ftruncate" );
        exit(1);
    }
    x = mmap ( NULL, VELICINA, PROT_READ | PROT_WRITE, MAP_SHARED, id, 0 );
    if ( x == (void *) -1 ) {
        perror ( "mmap" );
        exit(1);
    }
    close ( id );

    if ( proc_id == 0 ) {
        x->a = 0;
        x->b = 1;
    } else {
        x->b += x->a;
        x->a = x->b - x->a;
    }
    printf ( "[%d] %d\n", proc_id+1, x->b );

    /* zadnji proces briše zajednički spremnik */
    if ( proc_id == BROJ_PROCESA-1 ) {
        munmap ( x, VELICINA );
        shm_unlink ( NAZIV_ZAJ_SPREMNIKA );
    }
}

```

```

    }
    /* ako se ne obriše, segment ostaje zauzet */
}

int main ( void )
{
    int i;

    for ( i = 0; i < BROJ_PROCESA; i++ )
        if ( !fork() ) {
            posao_procesa (i);
            exit (0);
        }
    for ( i = 0; i < BROJ_PROCESA; i++ )
        wait (NULL);

    return 0;
}

```

6.8.2. Redovi poruka

Komunikacija porukama vrlo često je osnovni način komunikacije među dretvama u operacijskim sustavima za SRSV-e. Takvi sustavi omogućuju slanje poruka izravno dretvama, tj. uz svaku se dretvu stvara i red poruka pridružen toj dretvi (u koji se smještaju poruke upućene toj dretvi).

Komunikacija redom poruka prepostavlja da je stvoren red poruka (određena struktura podataka) te da dvije dretve koje ga žele koristiti mu mogu pristupiti. Sučelje, dakle, uključuje stvaranje reda, dohvati već stvorenoga reda, slanje poruke u red te prihvati poruke iz reda. Svaka poruka, osim korisnog sadržaja može biti označena i tipom poruke (tj. prioritetom kod POSIX-a).

Osnovna sučelja za rad s redom poruka su:

```

mqd_t mq_open ( const char *name, int oflag );
mqd_t mq_open ( const char *name, int oflag,
                mode_t mode, struct mq_attr *attr );
int mq_send ( mqd_t mqdes, const char *msg_ptr,
              size_t msg_len, unsigned msg_prio );
ssize_t mq_receive ( mqd_t mqdes, char *msg_ptr,
                     size_t msg_len, unsigned *msg_prio );

```

Sučelje `mq_open` s više parametara mora se koristiti pri stvaranju reda, dok se ono kraće može koristiti ukoliko red već postoji. Parametar `msg_prio` kod `mq_send` definira prioritet poruke. Red poruka je složen prema prioritetu - pri čitanju iz reda uzima se prva poruka – ona najveća prioriteta.

Isječak kôda 6.5. Primjer korištenja reda poruka

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <mqqueue.h>
#include <sys/wait.h>
#include <fcntl.h>

#define NAZIV_REDА "/msgq_example_name"
#define MAX_PORУKA_U_REDУ 5
#define MAX_VELICINA_PORУKE 20

int proizvodjac ()
{
    mqd_t opisnik_redа;
    struct mq_attr attr;
    char poruka[] = "primjer sadrзaja";
    size_t duljina = strlen (poruka) + 1;
    unsigned prioritet = 10;

    attr.mq_flags = 0;
    attr.mq_maxmsg = MAX_PORУКА_U_REDУ;
    attr.mq_msgsize = MAX_VELICINA_PORУKE;
    opisnik_redа = mq_open (NAZIV_REDА, O_WRONLY | O_CREAT, 00600, &attr);
    if ( opisnik_redа == (mqd_t) -1 ) {
        perror ( "proizvodjac:mq_open" );
        return -1;
    }
    if ( mq_send ( opisnik_redа, poruka, duljina, prioritet ) ) {
        perror ( "mq_send" );
        return -1;
    }
    printf ( "Poslano: %s [prio=%d]\n", poruka, prioritet );

    return 0;
}

int potrosac ()
{
    mqd_t opisnik_redа;
    char poruka[MAX_VELICINA_PORУKE];
    size_t duljina;
    unsigned prioritet;

    opisnik_redа = mq_open ( NAZIV_REDА, O_RDONLY );
    if ( opisnik_redа == (mqd_t) -1 ) {
        perror ( "potrosac:mq_open" );
        return -1;
    }
    duljina = mq_receive ( opisnik_redа, poruka, MAX_VELICINA_PORУKE,
                           &prioritet );
    if ( duljina < 0 ) {
        perror ( "mq_receive" );
        return -1;
    }
    printf ( "Primljeno: %s [prio=%d]\n", poruka, prioritet );

    return 0;
}

int main ( void )
{
    proizvodjac ();
}
```

```

sleep (1);
if ( !fork() ) {
    potrosac ();
    exit (0);
}
wait (NULL);

mq_unlink ( NAZIV_REDА );
return 0;
}

```

Funkcije slanja i primanja mogu biti blokirane, ako je red pun ili prazan. Zato postoje i dodatne funkcije s ograničenim vremenom blokiranja (`mq_timedsend` i `mq_timedreceive`).

6.8.3. Cjevovodi

Za razliku od poruka gdje u redu poruka postoji granulacija podataka – jedinica podataka je poruka (zadana zaglavljem), kod cjevovoda takve granulacije nema. Novi podaci se nadodaju na kraj starih. Zbog nepostojanja dodatnog zaglavljia cjevovodi su pogodniji kada treba prenijeti veću količinu podataka.

Rad s cjevovodima je gotovo identičan radu s datotekama. Iz njih se može čitati i u njih se mogu upisivati podaci. Cjev ima dvije strane: ulaznu i izlaznu. Svaka od njih ima svoj opisnik koji je identičan opisniku datoteke te se s njima radi i kao s datotekama. Dapače, cjevovod može imati i ime u datotečnom sustavu (imenovani cjevovod). U komunikaciji među dretvama cjevovod se može stvoriti i dinamički, pozivom `pipe`, bez da mu se dodaje ime u datotečnom sustavu (neimenovani cjevovod).

Sučelje za rad s cjevovodima su ista kao i za rad s datotekama (uz par iznimaka), uz dodatak sučelja `pipe` koje stvara novi neimenovani cjevovod. U datotečnom sustavu cjevovod se može napraviti naredbama (ili funkcijom iz programa) `mknod` i `mkfifo`.

Osnovno sučelje za rad sa cjevovodima uključuje:

```

int pipe ( int fildes[2] );
int mknod ( const char *path, mode_t mode, dev_t dev );
int mkfifo ( const char *path, mode_t mode );
int open ( const char *path, int oflag, ... );
int close ( int fildes );
ssize_t write ( int fildes, const void *buf, size_t nbyte );
ssize_t read ( int fildes, void *buf, size_t nbyte );

```

Isječak kôda 6.6. Primjer korištenja imenovanog cjevovoda

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/wait.h>

#define NAZIV_CIJEVI "/tmp/pipe_example"
#define N 50

void citac ()
{
    int fd, i, sz;
    char buffer[N];

    fd = open ( NAZIV_CIJEVI, O_RDONLY ); /* čekaj da i pisač otvara */

```

```

    if ( fd == -1 ) {
        perror ( "citac:open" );
        exit (1);
    }
    printf ( "citac otvorio cijev\n" );

    i = 0;
    memset ( buffer, 0, N * sizeof(char) );
    while( (sz = read ( fd, &buffer[i], N * sizeof(char) - i ) ) > 0 )
        i += sz;
    printf ( "Primljeno: %s\n", buffer );

    exit (0);
}

int main ()
{
    int fd, i;
    char *poruka[] = { "abcd", "ABC", "12345", "X", NULL };

    if ( mkfifo ( NAZIV_CIJEVI, S_IWUSR | S_IRUSR ) )
        perror ( "pisac:mkfifo" ); /* možda već postoji */

    if ( fork() == 0 )
        citac ();

    sleep (1); /* i čitač će zbog ovoga čekati! */
    fd = open ( NAZIV_CIJEVI, O_WRONLY ); /* čekaj da i čitač otvara */
    if ( fd == -1 ) {
        perror ( "pisac:open" );
        return 1;
    }
    printf ( "pisac otvorio cijev\n" );

    for ( i = 0; poruka[i] != NULL; i++ )
        write ( fd, poruka[i], strlen (poruka[i]) );
    close (fd);

    wait (NULL);

    unlink ( NAZIV_CIJEVI );

    return 0;
}

```

Načelo cjevovoda se koristi i pri komunikaciji s udaljenim računalima, kada se kao prijenosni protokol koristi spojna usluga (engl. *transmission control protocol – TCP*).

6.9. Korištenje višedretvenosti u SRSV-ima

Operacijski sustavi (barem oni namijenjeni SRSV-ima) imaju mogućnosti koje se mogu iskoristiti za ostvarenje SRSV-a. Ipak, pri ostvarivanju takvih sustava korištenjem dretvi posebnu pažnju treba posvetiti:

- raspoređivanju dretvi – odabrati prikladne postupke raspoređivanja i njihove parametre
- načinu sinkronizacije – odabrati prikladne mehanizme i paziti na mogućnost pojave potpunog zastoja, inverzije prioriteta, rekurzivnog zaključavanja
- načinima komunikacije – svojstva odabranih mehanizama (blokirajući ili ne, veličina poruke, trajanje prijenosa podataka, način obavještavanja odredišne dretve, ...)
- korištenju podataka u zajedničkom spremniku – zaštititi sinkronizacijskim mehanizmom
- trošku poziva jezgrinih funkcija – u sustavima u kojim se vrlo često pozivaju jezgrine funk-

cije i gdje se često obavlja raspoređivanje dretvi (npr. poslovi dretvi između sinkronizacijskih funkcija su vrlo kratki) utjecaj poziva jezgrinih funkcija i (kućanskih) poslova koji se pritom obavljaju, postaje primjetan i može predstavljati problem ako nije uračunat pri osmišljavanju sustava.

Višedretvenost je moćan koncept, ali ga treba oprezno koristiti.

Pitanja za vježbu 6

1. Opisati uzroke grešaka u korištenju vremena preko sučelja operacijskih sustava.
2. Navesti operacije povezane s upravljanjem vremenom koje nude operacijski sustavi. ◁
3. Koja je razlika između satova označenih sa `CLOCK_REALTIME` i `CLOCK_MONOTONIC`?
◁
4. Opisati svrhu te korištenje nadzornog alarma. ◁
5. Navesti osnovne mehanizme za sinkronizaciju dretvi?
6. Što je to potpuni zastoj? Kad se može pojaviti? Kako se izbjegava?
7. Što je to rekurzivno zaključavanje i zašto je ponekad potrebno?
8. Na primjeru prikazati nastajanje problema inverzije prioriteta. ◁
9. Opisati izvorni protokol stropnog prioriteta te njegovo pojednostavljenje.
10. Opisati protokol nasljeđivanja prioriteta. Prikazati rad protokola nad primjerima. ◁
11. Nad kojim sinkronizacijskim mehanizmima definiranih POSIX normom su podržani protokoli stropnog prioriteta te protokol nasljeđivanja prioriteta?
12. Opisati mehanizam signala. Koji se problemi mogu javiti u korištenju signala (a na koje treba pripaziti, odnosno, što treba istražiti kod operacijskog sustava i njegovog upravljanja signalima).
13. Opisati osnovna načela i svojstva međudretvene komunikacije korištenjem mehanizama zajedničkog spremnika, reda poruka, cjevovoda i signala. Kada koristiti pojedine mehanizme?
14. Od svih međudretvenih komunikacijskih mehanizama, koji se najčešće koristi u SRSV-ima? Zašto? ◁
15. Osim uobičajenih Čekaj*/Pročitaj*, Postavi*/Pošalji* sinkronizacijskih i komunikacijskih funkcija s uobičajenim ponašanjem, koje se dodatne operacije i proširenja zahtijevaju od takvih funkcija u kontekstu korištenja u SRSV-ima?
16. Što se sve može dogoditi s dretvom pri izvođenju koda (očekivano i neočekivano)?

```
struct timespec t;
t.tv_sec = 0; t.tv_nsec = 100000; /* 100 mikrosekundi */
clock_nanosleep ( CLOCK_REALTIME, 0, &t, NULL ); /* odgodi za t */
```

17. U nekom sustavu javljaju se dretve: D_1 u $t_1 = 0$ ms, D_2 u $t_2 = 3$ ms te D_3 u $t_3 = 6$ ms. Svaka dretva sastoji se od tri dijela posla: A, B i C. Izvođenje A dijela traje 2 ms, B dijela 3 ms te C dijela 2 ms. Prije izvođenja B dijela dretve trebaju zauzeti semafore: dretva D_1 semafor S_1 , dretva D_2 semafor S_2 , dretva D_3 semafor S_1 . Dretva D_3 ima najveći prioritet, dok dretva D_1 ima najmanji. Ukoliko se koristi protokol nasljeđivanja

prioriteta pokazati izvođenje zadanih dretvi, tj. što procesor radi u pojedinom trenutku, dok sve dretve ne završe sa svojim poslovima.

18. Za neki sustav poznati su događaji pokretanja dretvi, njihovi poslovi i trajanja.

U $t = 1$ pokreće se dretva D_1 . Nakon 1 ms rada zauzima semafor S_1 . Nakon još 2 ms rada (uz semafor S_1) zauzima semafor S_2 . Nakon još 1 ms rada (uz semafeore S_1 i S_2) otpušta oba semafora. Nakon još 1 ms rada dretva završava. Ukupno, dretva treba 5 ms procesorskog vremena.

U $t = 3$ pokreće se dretva D_2 . Nakon 1 ms rada zauzima semafor S_2 . Nakon još 2 ms rada (uz semafor S_2) zauzima semafor S_3 . Nakon još 1 ms rada (uz semafeore S_2 i S_3) otpušta oba semafora. Nakon još 1 ms rada dretva završava. Ukupno, dretva treba 5 ms procesorskog vremena.

U $t = 5$ pokreće se dretva D_3 . Nakon 1 ms rada zauzima semafor S_3 . Nakon još 2 ms rada (uz semafor S_3) zauzima semafor S_1 . Nakon još 1 ms rada (uz semafeore S_1 i S_3) otpušta oba semafora. Nakon još 1 ms rada dretva završava. Ukupno, dretva treba 5 ms procesorskog vremena.

Dretva D_3 ima najveći prioritet, slijedi D_2 dok D_1 ima najmanji prioritet. Prikazati rad sustava dretvi na jednoprocesorskom sustavu koji koristi prioritetno raspoređivanje:

- a) i nikakve druge protokole (niti nasljeđivanje prioriteta niti stropne protokole)
- b) i ako koristi protokol nasljeđivanja prioriteta
- c) i ako koristi jednostavniju inačicu protokola stropnog prioriteta
- d) i ako koristi izvorni protokol stropnog prioriteta.

19. Dretve $D_1 - D_4$ koriste semafor S po 100 ms, ali ne odmah po pokretanju već nakon 50 ms rada (svaka dretva najprije nešto radi 50 ms pa onda hoće semafor S za idućih 100 ms). Nakon otpuštanja semafora dretve rade nešto još 50 ms (dakle ukupno $50+100+50$, svaka dretva). Dretva D_2 javlja se prva u $t = 0$ ms. Slijedi D_3 u $t = 100$, D_4 u $t = 100$ te D_1 u $t = 150$. Prioritet dretvi određen je njenim indeksom: D_4 ima najveći a D_1 namanji prioritet. Ukoliko se dretve izvode na dvoprocesorskom sustavu i koristi se protokol nasljeđivanja prioriteta (uz prioritetni rasporedivač), pokazati rad sustava.

20. U nekom jednoprocesorskom sustavu izvode se dretve A, B, C i D . Dretva A ima najveći prioritet, slijedi dretva B , pa C te D koja ima najmanji. Dretve koriste tri sredstva koja su zaštićena semaforima S_1, S_2 i S_3 . Sve dretve mogu trebati bilo koje sredstvo u svom izvođenju. Ponekad, dretva koja već ima jedno sredstvo može tražiti i drugo, ali ne i treće (prije traženja trećeg otpušta sva zauzeta). Pretpostavka je da se potpuni zastoj neće dogoditi. Ako sustav koristi raspoređivanje prema prioritetu te protokol nasljeđivanja prioriteta, koliko se najviše (u najgorem slučaju) može zaustaviti dretva A zbog inverzije prioriteta? Opisati scenarij u kojem se to događa.
-

7. Raspodijeljeni sustavi

Razmatranje raspodijeljenih SRSV-a zahtjeva dodatne analize veza između raspodijeljenih računala (čvorova sustava) te načina komunikacije.

Općenito je komunikacija vrlo bitna za SRSV-e jer je za njih neophodna pouzdana vremenska usklađenost. Komunikacija može biti sinkrona, kada se podaci šalju i primaju usklađeno s nekim izvorom takta (ili drugim dogadjajima), ili asinkrona, kada se poruke šalju kad se za njima pojavi potreba. Nadalje, podaci koji se prenose mogu biti slani u cjelini ili pak podijeljeni na blokove, koji se u ovom kontekstu nazivaju paketima. Drugi način, korištenjem paketa je češći u nekritičnim sustavima.

Komunikacija se može podijeliti na komunikaciju računala s ulazno-izlaznim jedinicama (periferijom) i komunikaciju između različitih računala. Komunikacija računala s periferijom je zapravo dio protokola komunikacije s raznim uređajima, primjerice RS232 i USB te se neće posebno razmatrati u nastavku.

Komunikacija između različitih računala obavlja se korištenjem različitih protokola. Najpoznatiji među njima je svakako protokolni slog Internet, često predstavljen i kraticom TCP/IP koja označava dva najznačajnija protokola u tom slogu: TCP (engl. *transmission control protocol*) na prijenosnom sloju i IP (engl. *internet protocol*) na mrežnom. TCP/IP se sastoji od pet slojeva:

- aplikacijski (primjeri: HTTP, POP3, SMTP, DNS)
- prijenosni (primjeri: TCP, UDP)
- mrežni (primjeri: IP (IPv4, IPv6), ICMP, ARP)
- podatkovni (primjeri: ethernet, Wi-Fi, PPP)
- fizički (ovisi o podatkovnom, npr. CSMA/CD, RS232).

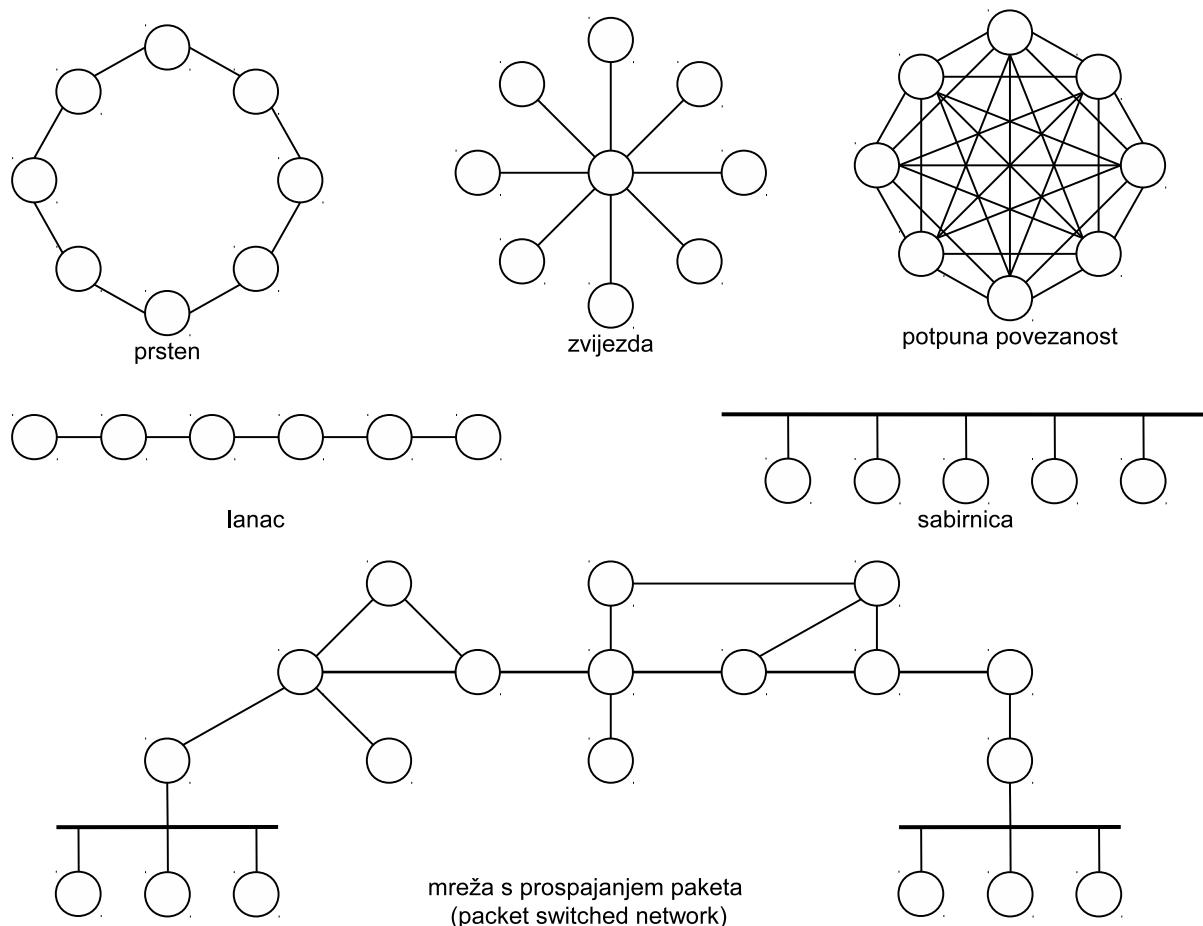
Zadnji sloj, fizički, se često i ne navodi kao dio protokolnog sloga, već ga se smatra dijelom podatkovnog.

TCP/IP je suviše složen da bi se ovdje detaljnije razmatrao. On je izgrađen da bude robustan te zato ima redundancije u vezama koje se mogu iskoristiti pri većem opeterećenju i pri ispadu pojedinih veza. Međutim, nije građen s namjerom korištenja u SRSV-ima te u tom kontekstu ima loša vremenska svojstva.

7.1. Povezivanje računala u mrežu

Način povezivanja među računalima raspodijeljenog sustava ovisi o korištenim protokolima, fizičkoj raspodjeli računala, potrebnim svojstvima i slično. Primjerice, veze mogu biti ostvarene centralizirano, gdje postoji glavno računalo koje upravlja komunikacijom, ili može biti decentralizirano, gdje su svi sudionici u komunikaciji ravnopravni.

Komunikacijska infrastruktura može također biti izvedena na razne načine, svaki sa svojim prednostima i nedostacima, prema slici 7.1. Primjerice, komunikacija "svatko sa svakim" pruža najviše mogućnosti, ali je znatno skupljia od svih ostalih. Kao i svugdje, i ovdje je potrebno napraviti kompromisno rješenje koje će ipak zadovoljiti zahtjeve. Kada se veza između dva računala (čvora) ne uspostavlja izravno već preko drugih računala, onda se najčešće koristi komunikacija s proslijedivanjem paketa.



Slika 7.1. Primjeri načina spajanja računala u mreži

Razni SRSV-i koriste razne načine povezivanja. Pri razmatranju nekog sustava treba uzeti u obzir kašnjenja koja korištena mreža stvara i jesu li ona prihvatljiva. Odabir ovisi o potrebama i procjeni projektanta sustava. Kod SRSV-a potrebno je uzeti sigurnosnu granicu, tj. predimenzionirati sustav. Korištenjem standardne mrežne opreme smanjuje se cijena sustava i olakšava razvoj i povezivanje, ali se vjerojatno kvare vremenska svojstva.

Komunikacijski medij može biti parica, koaksijalni kabel, optičko vlakno i slični materijali, ili se komunikacija može obavljati i bežično (tamo gdje udaljenost i smetnje nisu zapreka). Protokoli koji se koriste na komunikacijskim medijima mogu biti zasnovani na načelima:

- podjele vremena – svaki čvor dobije svoj dio vremena na mediju
- korištenja značke (tokena) kao oznake prava pristupa čvora mediju
- pojedinačno korištenje zajedničkog medija uz praćenje pristupa mediju (CSMA/CD, CAN).

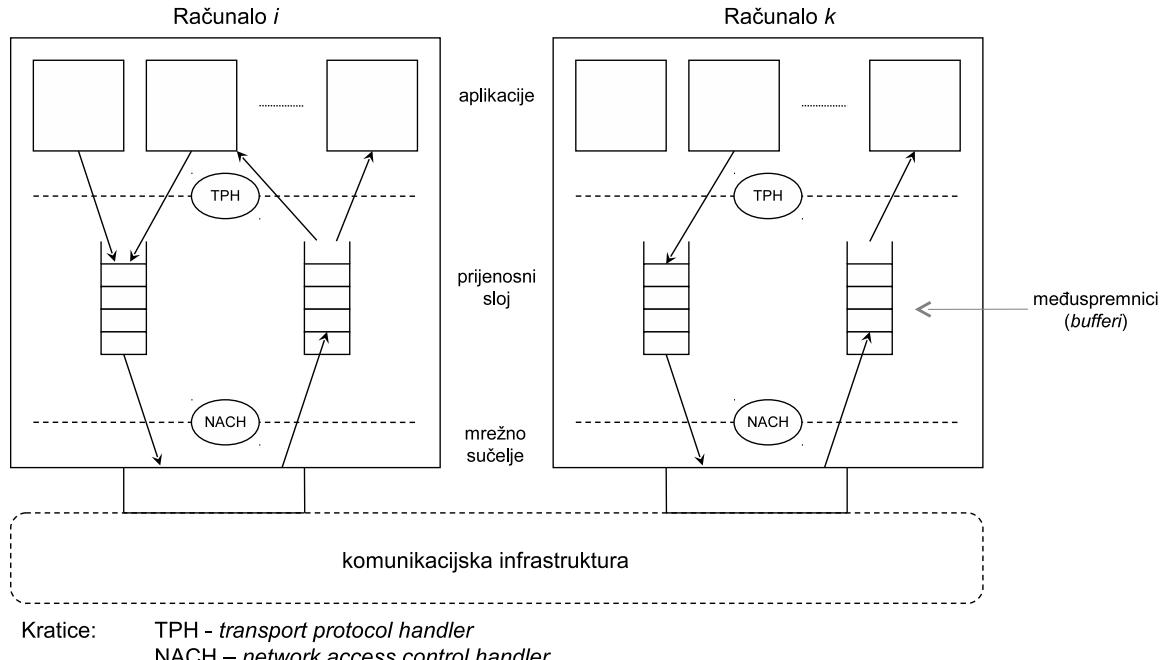
Svaki od navedenih načela ima svojih prednosti i nedostataka.

7.2. Model komunikacije

Osim fizičkih komponenata, za komunikaciju je potrebna i programska potpora. Iako ona može biti dio programa (u vrlo jednostavnim slučajevima), najčešće se koristi mrežni podsustav operacijskog sustava. Mrežni podsustav ostvaruje protokolni slog i omogućuje korištenje komunikacijskog sustava kroz svoje sučelje.

Slika 7.2. grafički prikazuje model komunikacije koji uključuje i procese i operacijski sustav i

komunikacijsku infrastrukturu. U ovakvom slojevitom sustavu, svaki sloj može obavljati dodatnu podjelu podataka koju je dobio od višeg sloja, omatati te dijelove dodatnim zaglavljima, koristiti redove za privremenu pohranu i slično, te obratne radnje kod primitka paketa od nižeg sloja, kada radi i provjeru ispravnosti. Sve te radnje traju, i u zbroju sa radnjama ostalih čvorova koji sudjeluju u prijenosu mogu narušiti svojstva potrebna u SRSV-ima.



Slika 7.2. Model komunikacije u raspodijeljenom sustavu

7.3. Svojstva komunikacijskog sustava

Pri razmatranju komunikacijskog sustava za uporabu u SRSV treba pogledati njegova svojstva. Posebice treba pripaziti na:

- kašnjenje (engl. *delay*)
- brzinu/propusnost mreže (engl. *throughput, bandwidth*)
- varijacije u kašnjenju (engl. *delay jitter*)
- postotak prekasno pristiglih paketa (engl. *miss rate*)
- postotak izgubljenih paketa (engl. *loss rate*)
- postotak pogrešaka pri prijenosu (engl. *invalid rate*)
- veličinu paketa
- dimenzije mreže (udaljenosti).

Kašnjenje

Jedno od najbitnijih svojstava u okruženju SRSV-a je kašnjenje i definira koliko vremena prolazi od slanja do primanja paketa (na aplikacijskoj razini). Preveliko kašnjenje može uzrokovati kasnu reakciju pojedinog računala te prekoračivanje zadanih vremenskih ograničenja. Uzroci kašnjenja su u mediju prijenosa (npr. ograničenje brzine svjetlosti), ali u značajnoj mjeri i svojstvima računala koja šalju, primaju i prenose podatke, koji obavljaju operacije stavljanja poruka u redove, obrade poruka (čitaju zaglavja, provjeravaju ispravnost).

Propusnost

Propusnost mreže definira koliko podataka se može prenijeti u jedinici vremena. Propusnost je među važnijim svojstvima uobičajenih računalnih mreža. Međutim, sa stanovišta SRSV-a, propusnost je bitnija jedino u multimedijskim primjenama, radi kvalitete prijenosa slike i zvuka u stvarnom vremenu. U pogledu upravljanja sustavima, propusnost je manje bitna jer su upravljačke poruke uglavnom kratke i nije potrebna velika propusnost već je bitnije vrijeme pristizanja poruke do želenog čvora, tj. potrebno je kratko vrijeme kašnjenja.

Propusnost i kašnjenje ne moraju biti proporcionalni. Velika propusnost ne mora značiti i kratko kašnjenje (iako to često i jest). Primjerice, čitanje podataka s diska može doseći velike brzine (i do stotinjak MB u sekundi), ali za dohvrat pojedinačnog (nasumičnog) podatka treba ponekad čekati i desetak milisekundi i više.

Varijacije u kašnjenju

Samo kašnjenje, koje može biti izračunato kao prosječna vrijednost kašnjenja svih paketa, nije dovoljna za analizu primjenjivosti u SRSV-ima. Ako je kašnjenje vrlo promjenjivo, čak i ako se duža kašnjenja javljaju rijetko, komunikacijski sustav ne mora biti prihvatljiv, pogotovo za stroge SRSV-e.

Greške u prijenosu

Prekasno pristigli paketi, kao i oni koji su izgubljeni ili imaju grešku, zahtijevaju retransmisiju – polazni čvor ih mora ponovno poslati, osim u posebnim slučajevima kada je povremeni gubitak prihvatljiv (kao kod prijenosa slike i zvuka). Takvu grešku treba prvo ustanoviti: pošiljalac će ustanoviti da nije primio odgovor na poslati paket tek nakon isteka zadanog vremena čekanja na dogovor, i tek će tada ponovno poslati paket (ako je takav protokol). Greška zato uzrokuje značajnija kašnjenja i time narušava vremenska svojstva komunikacije.

Unutarnja struktura komunikacijskog sustava je složena tematika te se ovdje neće u nju ulaziti i raspravljati o razlozima kašnjenja, grešaka i drugih problema koji se mogu pojaviti.

7.4. Primjeri protokola osmišljenih za komunikaciju u stvarnom vremenu

Prikažimo ukratko svojstva dva protokola, jednog na nižoj razini (podatkovnoj) i jednog na aplikacijskoj razini.

7.4.1. Controller Area Network

Protokol koji se često koristi u sustavima upravljanja jest *Controller Area Network* (CAN). CAN je značajno jednostavniji od primjerice TCP/IP zasnovane mreže, ne koristi slojeve unutar sebe. Kod CAN-a se koristi zajednički medij (sabirница), brzine do 1 Mbit/s i najveće duljine do 50 metara. Svi čvorovi spojeni na tu mrežu mogu i slati i primati poruke. Poruke imaju definirani format, ali mogu biti promjenjive duljine. Umjesto adresiranja polazišta i odredišta, svaka poruka započinje s identifikatorom, 11-bitovnim brojem. Pojedini čvorovi čitaju i koriste poruke s određenim identifikatorom, a ostale ignoriraju. Kao i kod CSMA/CD protokola, čvor prije slanja osluškuje sabirnicu i tek kad je ona slobodna započinje sa slanjem. Kada se dogodi da dva čvora istovremeno započinju sa slanjem poruka, onda se koristi svojstvo sabirnice i dominantnog bita – bita 0. Naime, obzirom da svaka poruka započinje identifikatorom, on se koristi za oznaku prioriteta poruke. Čvor koji šalje manje prioritetu poruku primijetit će da je na sabirnici njegov identifikator nadjačan prioritetnijom porukom te će u tom trenutku

zaustaviti slanje i prepustiti sabirnicu prioritetnijoj poruci koja će biti poslana do kraja bez prekidanje i greške.

Primjer 7.1.

Pretpostavimo da dva čvora istovremeno započinju sa slanjem svojih poruka, prvi s identifikatorom 57, a drugi sa 43. Obzirom da manji broj predstavlja veći prioritet, druga poruka treba biti prioritetnija. Binarno, identifikatori su:

$$57_{10} = 000001\underline{1}1001_2 \quad (7.1.)$$

$$43_{10} = 000001\underline{0}1011_2 \quad (7.2.)$$

Prvi čvor će tek pri slanju 7. bita primjetiti da poslano ne odgovara očitanom na sabirnici te će on tada prestati slati, dok će drugi čvor svoju poruku poslati do kraja.

Poruke se, osim početnog identifikatora, sastoje od upravljačkog zaglavja i podataka. Poruke po tipu mogu biti podatkovne, zahtjevi za podacima (engl. *remote*), poruke o greškama (engl. *error*) te poruke preopterećenja (engl. *overload*).

CAN ima malu iskoristivost prijenosnog medija, ali zato osigurava isporuke uz poštivanje vremenskog ograničenja.

Osim prioriteta ugrađenog u protokol, CAN primjenjuje i druge postupke zbog kojih ostvaruje svojstva prikladna za upravljačke sustave. Nabrojimo najvažnije:

- prioritetna shema poruka
- jamstvo da kašnjenje neće biti iznad definiranog
- prilagodljivo podešavanje (nije čak ni neophodno)
- slanje prema više čvorova (engl. *multicast*)
- osiguravanje konzistencije podataka
- ugrađeni postupci otkrivanja grešaka i njihovo signaliziranje
- automatizirana retransmisija kao opcija
- razlikovanje privremene greške od kvarova u čvorovima (koji se tada mogu i isključiti).

CAN sabirnica je zbog svojih svojstava vrlo popularan izbor za komunikaciju u upravljačkim sustavima (primjerice automobilima), iako je patentirana i potrebno je kupiti licencu za uporabu.

7.4.2. RTP/RTCP

Prijenos multimedije nije kritičan posao i uglavnom se ništa tragično neće dogoditi ako bude problema u prijenosu. Međutim, taj prijenos posjeduje svojstva SRSV-a jer se mora obavljati u stvarnom vremenu i (skoro) svaka se greška primjećuje. Ako dotok podataka nije pravilan primjetit će se problemi u slici i zvuku.

TCP kao prijenosni protokol nudi pouzdanu uslugu prijenosa podatka, sam se brine za probleme pri prijenosu (izgubljeni paketi, paketi s greškama i slično) i višim slojevima nudi pouzdanu uslugu prijenosa podataka. Međutim, ima vrlo malo mogućnosti za upravljanje vremenskim svojstvima prijenosa, čime u sustav unosi nedeterminističko ponašanje bez mogućnosti upravljanja u vremenski kritičnim situacijama.

Prikladnim postupcima se i nad lošim komunikacijskim kanalom (s puno grešaka) može ostvariti prihvatljiva komunikacija koja može zadovoljavati i neke primjene za SRSV-e, ali ne s TCP-om.

Mnogo bolji izbor za uporabu u SRSV-ima jest UDP (engl. *user datagram protocol*). On ne pruža

pouzdan prijenos podataka, ali upravo stoga omogućuje znatno više kontrole koja se ostvaruje iznad njega. Automatske retransmisije ugrađene u TCP u SRSTV-ima zapravo smetaju. Kada dođe do problema, mi ga želimo rješiti na prikladan način, što ne mora uvijek biti retransmisija kao kod TCP-a. Primjerice, kod multimedije, možda se poneka slika (engl. *frame*) može i izbaciti, bez potrebe retransmisije, a da kvaliteta videokonferencije i dalje bude zadovoljavajuća. Retransmisija bi možda previše zadržavala video signal i time smanjila mogućnosti komunikacije u stvarnom vremenu.

RTP (engl. *real time protocol*) i RTCP (engl. *real time control protocol*) osmišljeni su upravo za prijenos multimedije, za videokonferencije, za interaktivne multimedijalne aplikacije, za raspodijeljene simulacije. Za svoje ostvarenje koriste UDP. RTP prenosi podatke dok RTCP služi za upravljanje prijenosom. Svaka komponenta multimedije se zasebno prenosi, u zasebnom RTP kanalu koji koristi zasebnu pristupnu točku (engl. *port*) u mrežnom podsustavu za RTP i zasebnu pristupnu točku za RTCP.

RTP/RTCP omogućava dinamičko dodavanje sudionika u komunikaciji, kao i dinamičku promjenu svojstava komunikacije (svojstva audio i video signala koji se prenose). Osim čvorova koji su izvori podataka i čvorova koji podatke primaju, u sustavu mogu biti i dodatni pomoćni čvorovi koji ili samo prenose podatke ili koji ih pritom i prilagođavaju (mikseri, translatori).

RTCP koristi zasebni kanal po svakom RTP kanalu. Preko RTCP se šalju upravljački paketi koji opisuju kvalitetu primljenog sadržaja (engl. *reception quality*). Izvori podataka (i mikseri i translatori) na osnovu tih podataka zaključuju o mogućim problemima u prijenosu te prilagođavaju idući promet preko RTP kanala. Svaki čvor bi trebao povremeno slati izvješća preko RTCP kanala koji uobičajeno sadrže informacije o učinkovitosti prijenosa: koliko se paketa izgubi, kolika su vremena kašnjenja i varijacije tog kašnjenja i slično. Da RTCP ne bi zaguošio komunikacijsku vezu, za njega se rezervira samo mali dio propusnosti (npr. do 5%). Učestalost slanja RTCP paketa ovisi i o veličini grupe, svojstvima mreže i broju sudionika u komunikaciji.

Pri prijenosu multimedije, zvuk i slika se šalju odvojenim kanalima. U podatke (sekvence) koji se prenose preko tih kanala se ugrađuju identifikatori izvora, vremenske oznake, redni brojevi i opis formata. U jednoj sjednici (engl. *multicast group*) može biti i do nekoliko tisuća sudionika.

7.5. Sinkronizacija vremena u raspodijeljenim sustavima

U raspodijeljenim sustavima, osim osiguranja lokalne vremenske ispravnosti, potrebno je osigurati i zajedničku (globalnu) vremensku ispravnost sustava.

Na razini raspodijeljenog sustava, ponekad je dovoljna razina ispravnosti vremenska uređenost događaja – da se uvijek može uspostaviti koji se događaj dogodio prije, a koji poslije. U tu svrhu mogu se koristiti odgovarajući protokoli, primjerice raspodijeljeni Lamportov algoritam, ili njegova pojednostavljenja (protokol Ricarta i Agrawala) ili slični.

Često samo uspostava relacija između događaja nije dovoljna, već je potrebno vremenski usklađeno upravljati raznim elementima sustava iz različitih čvorova, tj. računala u tim čvorovima. Međutim, satovi u tim čvorovima ne moraju biti uvijek usklađeni, odnosno, čak i ako jesu u nekom trenutku usklađeni, zbog nepreciznosti satnih mehanizama oni će se u budućnosti pomaknuti.

Usklađivanje sata može se obaviti korištenjem vanjskog izvora sata, primjerice preko nekog poslužitelja koji daje "koordinirano svjetsko vrijeme" (engl. *Coordinated Universal Time – UTC*) te se to vrijeme uskladi s lokacijom čvora (Hrvatska je u zoni UTC+1). Preciznije usklađivanje može koristiti izvor sa satelita, primjerice GPS-a.

Ako se usklađivanje obavlja preko mreže, kao što je to Internet, postavlja se pitanje kako riješiti probleme koje će sama mreža unijeti. Naime, kada bi prijenos podataka bio trenutan, tada bi vrijeme koje se dobije od poslužitelja odmah mogli postaviti u sat klijenta. Međutim, vrijeme

potrebno paketu s informacijom o točnom vremenu da stigne do klijenta nije jednako nuli. Štoviše, to vrijeme može biti različito u različitim trenucima obzirom da paket može putovati raznim granama mreže, a i brzina pojedinih grana može se vremenom mijenjati (primjerice zbog različita opterećenja mreže). Kako izmjeriti kašnjenje i uzeti ga u obzir pri usklađivanju sata klijenta sa satom poslužitelja?

Jedan od jednostavnih načina sinkronizacije vremena s udaljenim poslužiteljem može se ostvariti uz pretpostavku simetrične veze, tj. veze kod koje je trajanje puta paketa u jednom i drugom smjeru isto. Neka se pri komunikaciji za sinkronizaciju sata klijenta sa satom poslužitelja koriste sljedeći zapisi o vremenima, koji se dodaju u podatke sinkronizacije:

1. ${}^k t_1$ – vrijeme slanja zahtjeva, po satu klijenta (satu kojeg treba uskladiti)
2. ${}^p t_2$ – vrijeme primitka zahtjeva, po satu poslužitelja (satu s kojim se treba uskladiti)
3. ${}^p t_3$ – vrijeme slanja odgovora, po satu poslužitelja
4. ${}^k t_4$ – vrijeme primitka odgovora, po satu klijenta.

Prefiksi ‘k’ i ‘p’ u vremenima označavaju vremena izražena u klijentskim i poslužiteljskim satovima.

Prvo i zadnje vrijeme je očitano sa strane klijenta po njegovu satu, a drugo i treće na strani poslužitelja po njegovu satu. Ako pretpostavimo da satovi jednak precizno odbrojavaju u zadanom intervalu (osim što nisu usklađeni po apsolutnoj vrijednosti, jedna sekunda jednak traže u oba sustava), onda možemo izračunati:

- kašnjenje mreže (engl. *round trip delay*):

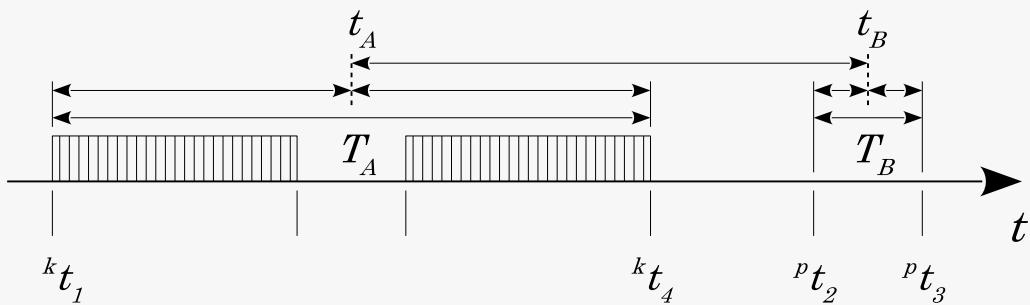
$$d = T_A - T_B = ({}^k t_4 - {}^k t_1) - ({}^p t_3 - {}^p t_2) \quad (7.3.)$$

- razliku u satovima (engl. *offset*):

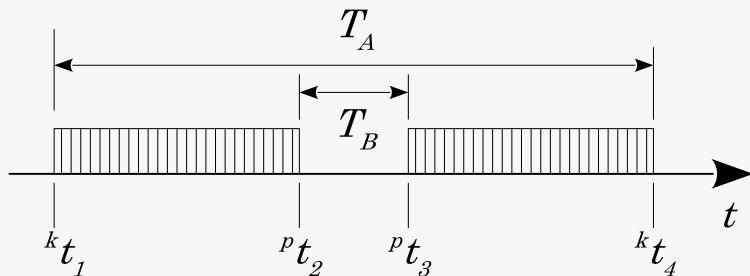
$$c = t_B - t_A = ({}^p t_2 + {}^p t_3)/2 - ({}^k t_1 + {}^k t_4)/2 \quad (7.4.)$$

Naime, ako od proteklog vremena ($T_A = {}^k t_4 - {}^k t_1$) oduzmemo vrijeme obrade zahtjeva na poslužitelju ($T_B = {}^p t_3 - {}^p t_2$) ostaje vrijeme puta. Ako je trajanje puta paketa sa zahtjevom jednak trajanju puta paketa s odgovorom, što je pretpostavljeno, onda možemo zaključiti da bi se sredine intervala vremena na klijentskoj strani ($t_A = {}^k t_1 + {}^k t_4)/2$) i na poslužiteljskoj strani ($t_B = {}^p t_2 + {}^p t_3)/2$) poklopile kada bi satovi bili usklađeni. Ako satovi nisu usklađeni, razlika u sredinama tih intervala jest odstupanje u satovima. Da bi klijent uskladio svoj sat sa poslužiteljskim satom, po primitku odgovora od poslužitelja on na svoj sat mora dodati c .

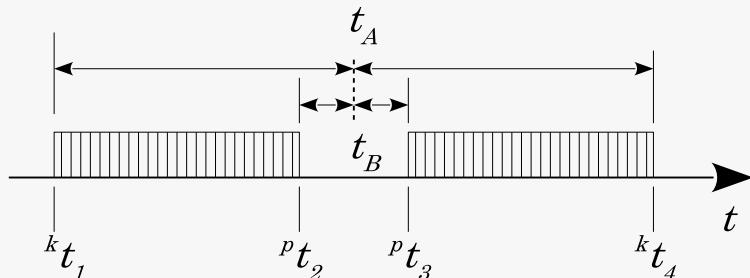
Primjer 7.2. Neusklađeni sustavi



Slika 7.3.

Primjer 7.3. Usklađeni sustavi

Slika 7.4.



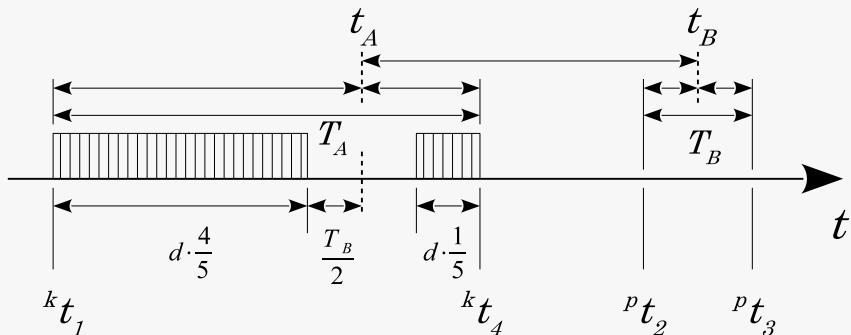
Slika 7.5.

Primjer 7.4. Nesimetrična veza

Neka je slanje zahtjeva četverostruko sporije od primanja odgovora (puta paketa u tom smjeru), tada bi se t_A koji se koristi u prethodnom primjeru u formuli 7.4. za računanje razlike u satovima sada računao prema:

$$t_A = {}^k t_1 + 4/5 \cdot d + T_B/2 \quad (7.5.)$$

Trajanje prijenosa dijeli se u omjeru 4:1 te je prvi dio 4/5 te vrijednosti, na što se još zbraja pola intervala T_B da bi došli do iste točke koja odgovara t_B kod poslužitelja. Navedeni primjer prikazuje slika 7.6.



$$d = T_A - T_B \Rightarrow t_A = {}^k t_1 + d \cdot 4/5 + T_B/2 \Rightarrow c = t_B - t_A \quad (7.6.)$$

Slika 7.6.

Pitanja za vježbu 7

1. Navesti slojeve protokolnog sloga Internet. ◁
2. Koji su razlozi nepouzdanosti (loših vremenskih svojstava) mreže koja se temelji na Internetu i njegovim protokolima?
3. Zašto TCP nije dobar za primjenu u SRSV-ima? Zašto se umjesto njega preferira UDP i nad njime izgrađuju komunikacijski kanali (protokoli)? ◁
4. Navesti moguće izvedbe spajanja računala u mrežu. Opisati svojstva pojedinih načina.
5. Koja se svojstva komunikacijskog sustava razmatraju pri analizi tih sustava? Koja od njih su osobito bitna za SRSV-e? ◁
6. Opisati mogućnosti sabirnice CAN koje se koristi za izbjegavanje kolizije na zajedničkom mediju? ◁
7. Što je to RTP/RTCP? Za što se koristi?
8. Na primjeru prikazati sinkronizaciju sata na čvoru klijenta sa čvorom poslužitelja.
9. U nekom sustavu čvor I pristupa čvoru J preko asimetrične veze: slanje poruka je pet puta sporije od primanja. Ukoliko je lokalno vrijeme slanja poruke u čvoru I 10 te primanja odgovora 50, dok je vrijeme primanja poruke u čvoru J 30 i vrijeme slanja 40 (u lokalnom vremenu čvora J) odrediti vrijeme prijenosa podataka od čvora I do J i obratno te razlika među satovima ($t_I = t_J + \text{razlika}$).
10. Satelit A treba uskladiti svoj sat sa satelitom B. Udaljenost među satelitima jest 300 000 km. Opisati postupak sinkronizacije i pokazati njegov rad na primjeru ako je u tom početnom trenutku sat na A imao vrijednost 13:13:13.111111 a sat na B 13:13:13.111222. Obrada poruka traje 0,0001 s.

8. Posebnosti izgradnja programske potpore za SRSV-e

Osim poznavanja postupaka/algoritama, pri samoj izradi programa (programiranju) treba uzeti u obzir i mogućnosti programskog jezika koji se koristi, svojstva alata s kojim se sustav izgrađuje, mogućnosti procjene ili mjerjenja trajanja pojedinih dijelova programa na stvarnom sustavu.

8.1. Programske jezice

Svi se programske jezici mogu koristiti za izgradnju SRSV-a (u pravom okruženju, uz prikladan operacijski sustav i sklopovlje). Ipak, svaki od njih imaju svoje prednosti i nedostatke. Sa stanovišta SRSV-a bitno je kako dobro poznavati svojstva programskega jezika i načina njihova prevođenja ili interpretiranja. Također, vrlo je bitna podrška za vremenski usklađeno upravljanje koje omogućuje programski jezik ili njegova proširenja.

Najčešće korišteni jezik za izradu SRSV-a jest C (barem za upravljačke komponente) zbog postojeće baze programa i programera. Ipak, i drugi, se programske jezice mogu koristiti, počevši od uobičajenog izbora za obične programe, poput C++, C# i Java, ali i drugi, primjerice skriptni i slični jezici koji se interpretiraju (Perl, Python).

Osim programskega jezika opće namjene, za SRSV izgrađeni su dodatni programske jezici kao što je Ada.

8.1.1. Programske jezike C

Za programiranje na niskoj razini, gdje je potrebno upravljati mikroupravljačem (mikrokontrolerom) ili koristiti posebne sklopove, najprikladniji odabir jesu C i asembler. Vrlo pažljivom uporabom programi pisani u njima mogu biti i značajno učinkovitiji (brži). Ipak, obzirom da su to jezici niske razine, izrada programske podrške može biti dugotrajnija nego korištenjem programskega jezika više razine.

Programski jezik C je jedan od najstarijih te ima vrlo mnogo njegovih proširenja (biblioteka), a neke od njih i za podršku SRSV-u. U prethodnim poglavljima prikazana su neka sučelja koja nudi POSIX za programske jezike C. Slična sučelja i proširenja postoje i za neke druge programske jezike (npr. Real-Time za Javu).

U programskom jeziku C gotovo je sve dozvoljeno. Primjerice, svaki dio spremnika se jednostavno dohvata i mijenja. Zato je C vrlo moćan, ali i potencijalno opasan zbog mogućih grešaka koje se lako previde a teško otkriju.

8.1.2. Programske jezike Ada

Rad na osmišljavanju programskega jezika Ada započeo je krajem '70-tih i početkom '80-tih godina prošlog stoljeća kao projekt američkog ureda za obranu (DoD), ali se njegov razvoj i dalje nastavlja (Ada95, Ada2005). Ada je proširenje Pascala (i drugih) sa svrhom da bude jezik za korištenje u ugrađenim sustavima i SRSV-ima. Zato Ada ima ugrađene mehanizme u sam

jezik koji značajno pomažu u takvim okolinama. Primjerice, tu spadaju:

- podrška za rad u stvarnom vremenu (u smislu upravljanja vremenom u zadacima)
- upravljanje dretvama (zadacima, *task* u Adi), od njihove sinkronizacije, komunikacije, raspoređivanja raznim strategijama (Ada ima vlastiti rasporedjivač)
- podrška radu ulazno-izlaznih naprava.

Sljedeći primjer prikazuje neke mogućnosti Ade za upravljanje vremenom i dretvama. Program prikazuje rad dva zadatka ping i pong (uz glavni zadatak koji počinje u zadnjem bloku begin /end).

Isječak kôda 8.1. Primjer programa u Adi

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure ping_pong_primer is

    task ping is
        entry pong_ping;
    end ping;

    task body ping is
    begin
        loop
            select
                delay 1.0;
                Put_line("ping");
            or
                accept pong_ping do
                    Put_Line("ping");
                    end pong_ping;
                    delay 1.0;
                    pong.ping_pong;
                end select;
            end loop;
        end ping;

    task pong is
        entry ping_pong;
    end pong;

    task body pong is
    begin
        loop
            accept ping_pong do
                Put_Line("pong");
                end ping_pong;
                ping.pong_ping;
            end loop;
        end pong;

begin
    delay 3.5;
    ping.pong_ping;
end ping_pong_primer;
```

Podrška za vremensko usklađivanje i višedretvenost je ugrađena u programski jezik. Zadatak se definira tipom *task*, a svi definirani zadaci automatski se pokreću s glavnim zadatkom. Odgoda, sinkronizacija i reakcija na događaje dio su programskog jezika. Primjerice, zadatak ping konstrukcijom select sa delay u jednoj grani i accept u drugoj, odgađa zadatak ping do jedne sekunde. Ako se unutar te sekunde ne pojavi zahtjev za sinkornizacijom na točki pong_ping odabire se prva grana select-a, tj. ispisuje se "ping".

Obzirom na namijenjenu uporabu Ada ima vrlo strogu provjeru tipova podataka, kako prilikom prevođenja izvornog kôda tako i pri radu. Na primjer, ako je varijabla x definirana da prima vrijednosti između 5 i 10 tada pokušaj stavljanja neke druge vrijednosti neće biti moguće: ako prevoditelj može to ustanoviti javit će grešku pri prevođenju, ali će i u izvršnu inačicu programa ugraditi kôd koji dinamički otkriva pokušaj stavljanja vrijednost izvan dozvoljenog opsega te izazva iznimku (engl. *run-time checks*).

8.2. Prevoditelj

Programi u višim programskim jezicima (tu se uključuje i C) prevode se u instrukcije korištenjem prevoditelja (ili generatora kôda). Osim poznavanja korištenih programskega jezika, treba znati i kako se taj program prevodi te kako iskoristiti prevoditelj za izgradnju boljeg koda i pronalaženje mogućih grešaka.

Prevoditelj će prilikom rada prikazati greške u programu koje on može otkriti. Međutim, to su samo greške u sintaksi programskega jezika. Logičke greške on ne može otkriti. Ipak, prevoditelji imaju mogućnosti odabira različite razine povratnih informacija preko takozvanih upozorenja (engl. *warnings*). Takva upozorenja mogu biti nebitna ako što je to npr. varijabla koja je definirana, ali nije korištena. Međutim, ponekad ona mogu biti pokazatelji da je nešto krivo ili bi moglo biti krivo. Primjerice, ako je upozorenje da je vrijednost variable postavljena, ali se ne koristi, to može biti pokazatelj da možda nešto nedostaje – kod u kojem se ta vrijednost koristi. Stoga je preporuka uključiti sve moguće razine upozorenja koje prevoditelj može dati. Takva upozorenja je najbolje odmah i rješavati. U protivnom bi se pri prevođenju moglo pojaviti puno upozorenja od kojih su neka i bitna, a možda ih korisnik ne primijeti. Ako se svako takvo upozorenje rješava (otklanja izmjenama u kôdu), tada bi svako novo upozorenje nastalo novim izmjenama kôda odmah bilo vidljivo. Prevoditelj gcc ima mnoštvo zastavica za upravljanje razinama upozorenja. Primjerice, zastavicom `-Wall` aktiviraju se sve razine upozorenja, a zastavicom `-Werror` se s upozorenjima postupa kao i s greškama, prisiljavajući korisnika da ih ispravi (makne).

Nadalje, pri prevođenju treba računati i na dodatne poslove koji su potrebni a možda nisu izravno vidljivi iz kôda te optimizacije kôda radi brzine ili veličine programa i problemi koji zbog toga mogu nastati.

Iako se u višem programsku jeziku neke radnje ne vide, na njih treba računati. Primjerice, pri pozivanju funkcije koristi se stog i to za prijenos parametara u funkciju, za pohranu konteksta te za lokalne variable funkcije.

Pri pojavi prekida spremi se kontekst prekinut dretve. Pri povratku iz prekida obnavlja se kontekst. Ako se jezgrine funkcije pozivaju mehanizmom prekida, što je uobičajeno za obične operacijske sustave (ali ne uvijek u operacijskim sustavima za rad u stvarnom vremenu), onda su pohrana i obnavljanje konteksta sastavni dio takvog poziva. Ponekad ti kućanski poslovi mogu značajno narušiti učinkovitost sustava u odnosu na planirano, pogotovo ako se jezgrine funkcije vrlo često pozivaju, a čemu može biti uzrok presitna zrnatost podjele posla po zadacima (te zbog toga učestala sinkronizacija).

Brzina izvođenja programa kao i njegova veličina u najvećoj su mjeri definirani samim programom (programiranjem), tj. posljedica rada programera. Ipak, mogućnosti današnjih prevoditelja u optimiranju kôda ne treba podcijeniti. Uobičajeno je da prevoditelji imaju nekoliko razina optimiranja koji se uključuju ili isključuju pri prevođenju posebnim zastavicama. Primjerice, kada se koristi gcc tada zastavicom `-O` određujemo razinu optimiranja, od osnovne razine `-O1` do potpune optimizacije sa `-O3`. Ili se optimiranje može provesti korištenjem kriterija manje veličine programa, zastavicom `-Os`. Razina `-O0` isključuje optimiranja.

Najveća razina optimiranja prevoditelja uključuje i pretragu za funkcijama koje će se ugraditi u kôd programa na sva mesta gdje se pozivaju i time izbjegći dio kućanskih poslova. Ipak, tu su

mogućnosti i znanje prevoditelja ograničena i uputa programera s određenim ključnim riječima, poput `inline` u tipu funkcije, mogu znatno ubrzati program (na uštrb njegove veličine). Za razliku od funkcija, makroi (u C-u definirani direktivom `#define`) se uvijek ugrađuju na mjesto s kojeg se pozivaju.

Ponekad se iz raznih razloga (npr. simulacija, kratka odgoda) u kôd ugrađuju petlje koje nemaju nikakav utjecaj na sustav, osim odgode zbog obavljanja tih instrukcija. Zato prevoditelj pri optimiranju može zaključiti da je taj dio kôda suvišan i izbaciti ga. Ako je taj dio kôda ipak potreban (a vjerojatno jest), onda se treba osigurati da ga prevoditelj ipak ostavi. Postoji nekoliko načina za to. Jedan od njih je ubacivanje spremničke barijere (engl. *memory barrier*) u dio kôda koji želimo zadržati (npr. u C-u sa `asm volatile ("::: : : "memory")`).

Primjerice, ako želimo simulirati rad procesora izvođenjem neke petlje određeni broj puta (prethodno proračunat), tada u C-u možemo umjesto obične prazne petlje:

```
for ( i = 0; i < loops; i++ )
;
```

koristiti petlju s barijerom:

```
for ( i = 0; i < loops; i++ )
    asm volatile("::: : : "memory");
```

U višeprocesorskim sustavima može se pojaviti potreba za sinkronizaciju radnim čekanjem (engl. *spinlock*). Primjerice, dio kôda s kojim to ostvarujemo može izgledati:

```
while ( rq_cnt == 0 )
;
```

Međutim, ako je varijabla `rq_cnt` u uvjetu obična, u postupku optimizacije prevoditelj može za taj dio kôda varijablu postaviti u registar i tamo je držati. Promjena nad varijablom koju radi neka druga dretva tako neće biti vidljiva. Zato postoji mogućnost da se varijabla dodatno označi ključnom riječi `volatile` kojom se prevoditelju naglašava da se dotična varijabla može promjeniti i izvan određene funkcije (čak i djelovanjem drugih elemenata sustava, ne samo procesora), te da ju ne zadržava u registru procesora, već se svaki puta ponovno dohvata iz spremnika.

8.3. Problemi s višedretvenošću

Načini raspoređivanja dretvi, sinkronizacije i komunikacije su već prikazani u prethodnim poglavljima. Vrlo je važno da prilikom korištenja višedretvenosti sve zajedničko treba zaštитiti prikladnim mehanizmima od istovremenog korištenja. U prvom redu tu spadaju globalne varijable. Međutim, osim njih treba paziti i na staticki deklarirane varijable, primjerice sa `static` u funkcijama. Problem se pojavljuje kada istu funkciju paralelno pozovu i izvode dvije ili više dretvi. Da bi riješili problem, dio koji koristi zajedničke podatke treba zaključati od istovremenog korištenja, ili svakoj dretvi dati svoje podatke, primjerice tako da ih šaljemo u funkciju.

Isječak kôda 8.2. Funkcija neprikladna za višedretveni rad

```
void neka_funkcija ( neka_struktura )
{
    static int brojac = 0; //statički alocirana varijabla
    brojac += nešto(); //ovu liniju koda treba zaštititi
}
```

Sličnih problema ima u mnogim funkcijama, primjerice funkcija `rand` te jedno njen “rješenje” u obliku `rand_r`. Prva nije sigurna (u nekim sustavima) za višedretvene sustave (engl. *thread*

unsafe, MT unsafe) jer koristi globalnu varijablu za čuvanje međurezultata (sjeme, engl. *seed*) potrebnog za izračun pseudo slučajnog broja. Druga funkcija, `rand_r`, očekuje da parametar koji joj se šalje bude taj međurezultat te je zato ona sigurna i za višedretvreno korištenje (engl. *thread safe, MT safe*), pa i za istovremene pozive od raznih dretvi (engl. *reentrant function*).

Isječak kôda 8.3. Korištenje zasebne varijable za svaku dretvu

```
void neka_funkcija ( neka_struktura, int *brojac )
{
    *brojac += nešto();
}
```

Isječak kôda 8.4. Zaštita od istovremenog korištenja sinkronizacijskim funkcijama

```
void neka_funkcija ( neka_struktura )
{
    static int brojac = 0; //statički alocirana varijabla
    static pthread_mutex_t KO = PTHREAD_MUTEX_INITIALIZER;
    int tmp;
    tmp = nešto();
    pthread_mutex_lock ( &KO );
    brojac += tmp;
    pthread_mutex_unlock ( &KO );
}
```

U arhitekturama koje dozvoljavaju nedjeljive operacije nad operandima u radnom spremniku, problem iz prethodnog primjera se može riješiti i efikasnije korištenjem takvih operacija. Primjer takvih instrukcija je “dohvati i zbroji” (engl. *fetch and add*) koja se pojavljuje u mnogim arhitekturama (npr. GCC-ov makro `__atomic_fetch_add`).

Isječak kôda 8.5. Korištenje atomarnih operacija

```
inline int dohvati_i_zbroji ( int *a, int b )
{
    int c;
    c = __atomic_fetch_add ( a, b, __ATOMIC_RELAXED );
    // atomarno: c = *a; *a = *a + b;
    return c;
}
```

Mnoga sučelja operacijskih sustava koriste i globalne varijable i statički rezervirane varijable u funkcijama. Zato ih se ne preporuča za višedretvene sustave. Primjerice, POSIX standard definira da sve funkcije, osim iznimaka [Thread-Safety], trebaju biti napisane da podržavaju višedretvene programe.

8.4. Preciznost

U sustavima gdje je preciznost vrlo bitna treba uzeti u obzir svojstva različitih tipova podataka. Primjerice, u C-u postoje dvije osnovne skupine brojeva: cjelobrojni i podaci s pomičnom točkom (realni brojevi). Obje skupine imaju nekoliko tipova podataka, prikazanih tablicama 8.1. i 8.2.

Tablice prikazuju raspone pojedinih tipova podataka. Pri odabiru tipa varijabli treba paziti da ne dođe do preljeva, tj. da se zbrajanjem ili oduzimanjem vrijednosti ne pređe preko najveće ili najmanje vrijednosti i time dobije krivi rezultat.

Kada se koriste konstante, tj. brojevi u formulama (a ne samo varijable), uz rjede korištene tipove ne zaboraviti da uz sam broj treba dodati oznaku tipa (npr. `long⇒L`, `long long⇒LL`, `unsigned long⇒UL`, `long double⇒L`), inače će se prepostaviti običan tip podataka i

Tablica 8.1. Uobičajene granice cjelobrojnih tipova podataka u C-u

oznaka tipa	broj bitova	raspon	raspon uz unsigned
char	8	$[-2^7; 2^7 - 1]$	$[0; 2^8 - 1]$
short int	16	$[-2^{15}; 2^{15} - 1]$	$[0; 2^{16} - 1]$
int	32	$[-2^{31}; 2^{31} - 1]$	$[0; 2^{32} - 1]$
long int	32	$[-2^{31}; 2^{31} - 1]$	$[0; 2^{32} - 1]$
long long int	64	$[-2^{63}; 2^{63} - 1]$	$[0; 2^{64} - 1]$

Tablica 8.2. Granice realnih tipova podataka u C-u (prema IEEE 754)

oznaka tipa	broj bitova	raspon (okvirno)	preciznost u znamenkama
float	32	$[10^{-38}; 10^{38}]$	7
double	64	$[10^{-308}; 10^{308}]$	16
long double	128	$[10^{-4932}; 10^{4932}]$	34

možda će se izgubiti željena preciznost (iako bi prevoditelj na to trebao upozoriti).

Npr. broj 12345678901234LL će se prepoznati kao long long tip, a broj 3.14e1000L kao long double.

Kad se koriste realni brojevi tada treba, kao prvo, obratiti pozornost na preciznost. Tablica 8.2. prikazuje preciznost pojedinih tipova podataka u broju decimala – do kuda je broj zapisan u varijabli ispravan (koliko decimalnih znamenki). Često je i najosnovniji tip float zadovoljavajući. Kada to nije tako može se koristiti dvostruko precizniji double, odnosno još precizniji long double.

Osim preciznosti, treba pripaziti da se u matematičkim operacijama ne pojavljuju brojevi bitno različih veličina. Primjerice ako se zbroje: $1 + 10^{-100}$ dobit će se 1, bez obzira na tip, jer preciznost svih tipova nije tolika da pamti 100 decimala. Prethodni rezultat i ne mora nužno biti problem, većinom to i nije. Međutim, kada bi iduća operacija bila oduzimanje s jedinicom, u stvarnosti bi očekivali rezultat 10^{-100} , dok bi zapravo u računalu dobili 0. Broj 10^{-100} je zaista jako mali i moguće je da se on u nekim primjenama može poistovjetiti s nulom. Što ako to nije tako? Što ako su očekivane vrijednosti zaista u tom rangu (10^{-100})? Onda treba prilagoditi algoritam izračunavanja. Primjer 8.1. prikazuje takav pristup.

U rijetkim slučajevima potrebna je značajno veća preciznost nego što ju nude osnovni tipovi podataka. Tada je potrebno ili samostalno izgraditi strukture i algoritme koji će raditi u većoj preciznosti ili za to koristiti gotove biblioteke. Jedna od takvih biblioteka jest *The GNU Multiple Precision Arithmetic Library* (<http://gmplib.org/>). Ipak, pri razmatranju korištenja takvih postupaka i biblioteka treba imati na umu da su operacije s podacima visoke preciznosti značajno sporije od uobičajenih!

U numeričkim postupcima integracije treba razmotriti stabilnost postupka, tj. provjeriti da li za dane prilike (npr. duljina koraka integracije) postupak konvergira.

U postupcima optimiranja nekim od heurističkih ili kombinatoričkih algoritama, potrebno je osigurati izračun u dozvoljenom vremenu, obzirom da ti postupci mogu duže potrajati. Ponekad je bolje uzeti i lošije rješenje, ali koje je izračunato u zadanom ograničenom vremenu, nego dobiti najbolje rješenje, ali prekasno.

Primjer 8.1. Rješavanje kvadratne jednadžbe

Rješenja kvadratne jednadžbe zadane formulom:

$$a x^2 + b x + c = 0 \quad (8.1.)$$

se računaju prema:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (8.2.)$$

Ako pretpostavimo da se zaista radi o kvadratnoj jednadžbi, tj. da je $a \neq 0$, te da su rješenja realna (korijen veći od nule), onda se svejedno može pojaviti problem s preciznošću. Ako je drugi dio ispod korijena značajno manji od prvog, tj. ako je:

$$b^2 \gg 4ac \quad (8.3.)$$

onda će se dobiti rješenja:

$$\begin{aligned} x_1 &\approx -b/a \\ x_2 &\approx 0 \end{aligned} \quad (8.4.)$$

Ako se želi precizniji broj za x_2 uz isti uvjet 8.3., jer se očekuje da je on jako mali, onda se formula (8.2.) može transformirati prema (8.5.):

$$\begin{aligned} x_2 &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \cdot \frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}} \\ x_2 &= \frac{b^2 - (b^2 - 4ac)}{2a \cdot (-b - \sqrt{b^2 - 4ac})} = \frac{4ac}{2a \cdot (-b - \sqrt{b^2 - 4ac})} \\ x_2 &= \frac{2c}{-b - \sqrt{b^2 - 4ac}} \approx \frac{2c}{-2b} \\ x_2 &\approx -\frac{c}{b} \end{aligned} \quad (8.5.)$$

Zadnja formula će dati precizniju vrijednost za x_2 kada vrijedi (8.3.).

8.5. Predvidivost trajanja – složenost postupaka

SRSV-i moraju biti deterministički sustavi. Da bi to mogli biti potrebno je poznavati i trajanje svih operacija, u najboljem slučaju, prosječno te u najgorem slučaju. Ocjena trajanja pojedinog postupka ili operacije može se napraviti:

- ispitnim pokretanjem sustava ili samo jedne njegove komponente (kad je sustav već izrađen)
- procjenom trajanja obzirom na poznavanje postupka (u fazi projektiranja sustava).

Procjena treba ustanoviti kakve instrukcije procesor pokreće i koliko one traju. U današnjim sustavima u kojima prevladavaju RISC procesori (engl. *reduced instruction set computing*), nema prevelikih razlika između trajanja pojedinih instrukcija (kao nekada kada su operacije množenja, dijeljenja i složenije značajno duže trajale od zbrajanja, oduzimanja i logičkih operacija). Međutim, velike razlike mogu nastati zbog neučinkovitog korištenja priručnog spremnika procesora (ako se podaci ne koriste uglavnom sljedno nego im se pristupa raštrkano). Ipak, pri procjeni se takvi problemi često razmatraju odvojeno, tj. zasebno se procjenjuje složenost postupka, primjerice u broju operacija ili broju instrukcija, te zasebno dodatna kašnjenja zbog

raznih razloga, kao što je problem promašaja u priručnom spremniku, ili u postupku straničenja.

Složenost postupka se često izražava u notaciji veliko O . Složenost se izražava u dimenzijama problema. Primjerice, $O(n^2)$ kaže da će broj operacija u postupku koji koristi skup od n podataka biti proporcionalan sa n^2 , odnosno preciznije, da je gornja ograda broja operacija proporcionalna sa n^2 . Ako n raste, broj operacija raste kvadratno (zato se i za notaciju O kaže da označava mjeru rasta složenosti postupka). Tablica 8.3. prikazuje uobičajene vrijednosti za složenost algoritama.

Tablica 8.3. Primjeri složenosti postupaka

notacija	naziv složenosti	primjeri postupaka takve složenosti
$O(1)$	konstantna	pristup podacima iz tablice; izračun vrijednosti bez iteracija ili uvijek s istim brojem iteracija
$O(\log n)$	logaritamska	binarno pretraživanje uređenog polja; pretraživanje binarnog stabla
$O(n)$	linearna	pretraživanje neuređenog polja
$O(n \log n)$	linearno log.	FFT, <i>quick-sort</i>
$O(n^2)$	kvadratna	množenje dva vektora, <i>buble-sort</i>
$O(n^c)$, $c > 1$	polinomska	množenje matrica (i višedimenzionalnih podataka)
$O(c^n)$	eksponencijalna	pretraga n -dimenzionalnog neuređenog prostora
$O(n!)$	faktorijelna	traženje optimalnog rješenja kombinatoričkog problema potpunom pretragom skupa mogućih rješenja, primjerice problem trgovačkog putnika

Što je složenost veća postupak će dulje potrajati, ali i razlika u trajanju istog postupka nad različitim podacima (u različitim trenucima) će biti veća – povećava se problem predvidivosti trajanja postupaka. Ukoliko složenost postaje problem, onda ju treba pokušati riješiti na razne načine, ili odabirom drugog postupka manje složenosti (makar i manje kvalitete rješenja), ili optimiranjem postojećeg.

Jedan od često primjenjivanih postupaka smanjenja složenosti jest korištenjem tablica. Umjesto da se vrijednosti izračunavaju tijekom rada korištenjem složenih postupaka, one se proračunaju prije pokretanja (engl. *offline*), kada vrijeme nije kritično i to za sve moguće vrijednosti ulaza. Pri radu za svaku traženu vrijednost uzima se već proračunata vrijednost iz tablice. Nažalost, takvo pojednostavljenje nije uvijek moguće, a i kada jest tada može znatno povećati zahtjeve na spremnički prostor računala za pohranu tablice, tj. složenost se iz vremenske domene rješava ili smanjuje povećanjem složenosti u prostornoj domeni.

Primjer 8.2. Složenost zbrajanja matrica

Složenost zbrajanja matrica (prema sljedećem algoritmu) proporcionalna je umnošku dimenzija matrica: $broj_redaka \times broj_stupaca$.

```
zbroji_matrice ( a, b, c ) // c = a + b
    za i = 0 do a.broj_redaka radi
        za j = 0 do a.broj_stupaca radi
            c[i][j] = a[i][j] + b[i][j];
```

Povećanjem samo jedne dimenzije matrica (npr. samo broj redaka) složenost se linearno povećava. Povećanjem obiju dimenzija složenost raste kvadratno. Primjerice, ukoliko su matrice kvadratne s n redaka i n stupaca, složenost zbrajanja jest $O(n^2)$.

Primjer 8.3. Složenost umetanja u uređenu listu

Umetanje u listu jest linearne složenosti $O(n)$.

```
ubaci ( lista, novi_objekt ) # uređenje je od najmanjeg prema najvećem
    ako je lista.prvi != null I lista.prvi > novi_objekt tada
        novi_objekt.iduci = lista.prvi
        lista.prvi = novi_objekt
    inače
        i = lista.prvi
        dok je i.iduci != null I i.iduci < novi_objekt radi
            i = i.iduci
        novi_objekt.iduci = i.iduci
        i.iduci = novi_objekt
```

Iako će se u prosjeku (kada se raspodjele elemenata koje stavljamo u listu podvrgavaju uniformnoj razdiobi obzirom na element uređenja liste) pretraživati pola elemenata liste, najgori slučaj će biti kada novi element mora na kraj liste. Stoga je složenost jednak broju elemenata u listi, tj. linearna.

8.6. Višeprocesorski i višejezgreni sustavi

Noviji sustavi često su opremljeni s jednim procesorom koji ima više procesnih jedinki (višestruki, višejezgreni procesor, engl. *multicore*). Operacijski sustavi uglavnom ne rade veliku razliku između jedne jezgre višejezgrenog procesora od zasebnog, običnog procesora u višeprocesorskem sustavu. Kod oba sustava, i višeprocesorskog i sustava s višejezgrenom procesorom ili više njih i dalje treba paziti na korištenje dijeljenih sredstava.

Međutim, obzirom da procesne jedinke višejezgrenog procesora dijele zajedničku sabirnicu prema glavnom spremniku i ostalim sklopovima, rad jedne dretve na jednoj procesorskoj jedinki (jezgri) može utjecati na druge. Nadalje, te dretve dijele i priručni spremnik tog procesora (označavanog s L2 i L3). Naime, iako se dvije dretve različita prioriteta mogu izvoditi paralelno na dvije različite procesorske jedinke, zbog intenzivnih promašaja (zahtjeva za podacima koji nisu u priručnom spremniku procesora) kojima uzrok može biti dretva manjeg prioriteta, i sama prioritetska dretva može biti manje ili više usporena u svom radu. Dretva manjeg prioriteta će koristiti (zbog promašaja) veći dio priručnog spremnika pa će manji dio ostati za prioritetsku koja će zbog toga i sama više promašivati, tj. usporit će se i njen rad. Rješenje ili ublažavanje tog problema zahtjeva nadogradnju raspoređivača dretvi koji treba uzeti u obzir prioritete te ponekad zaustaviti dretvu manjeg prioriteta na jednoj jezgri kada ustanovljava porast broj promašaja koje ona izaziva, ili takvu dretvu premjestiti na drugi procesor. Takvi raspoređivači su tek u razvojnoj i ispitnoj fazi, ali je za očekivati da će se u bliskoj budućnosti

pojaviti u operacijskim sustavima za rad u stvarnom vremenu (a možda i ostalima).

8.7. Korištenje ispitnih uzoraka

Ispitivanje ispravnosti je jedna od najvažnijih aktivnosti u procesu izgradnje programske potpore (a i sklopolovske). Ispitivanje se treba provoditi tijekom cijele faze izrade, ali je onaj najopsežniji dio ipak na kraju kad je programska potpora već izgrađena. Za taj zadnji dio potrebni su ispitni uzorci koji će se iskoristiti pri ispitivanju. Ispitni uzorci često predstavljaju scenarije uporabe i mogu biti pokretač izgradnje, nešto oko čega će se oblikovati arhitektura sustava i komponente. Opća je preporuka da se već u početku, paralelno s izradom specifikacije sustava definiraju ispitni uzorci. Oni mogu poslužiti i radi provjere specifikacije, je li definirano zaista ono što sustav treba raditi.

Pitanje je kako definirati ispitne uzorke? Više o toj problematici može se pronaći drugdje (u drugim predmetima ili materijalima [Bogunović, 2012]). Sažeto se može reći da ti ispitni uzorci trebaju pokrivati sve distinkтивne dijelove domene, da ih ne smije biti previše (jer će ispitivanje jako dugo trajati), treba ih provjeriti s naručiteljem (i pravim korisnicima tog sustava), treba ih snimiti u domeni korištenja (od strane razvojnog tima, ne samo od strane naručitelja).

Obzirom da su SRSV često kritični sustavi, ispitivanje i programske komponente mora biti temeljito, tj. značajno opsežnije od ispitivanja koje se provodi kod ostalih sustava.

8.8. Oporavak od pogreške

Složenost računalnih sustava uzrokuje greške i u vrlo opsežno ispitanim sustavima. Što je sustav bolje ispitani manja je vjerojatnost pojave greške tijekom rada. Međutim, što napraviti kada se greška ipak dogodi?

U poglavljju 6.1.2. prikazan je jedan postupak oporavka od pogreške koja se prikazuje kao zastoj u radu kritičnih operacija, kada se zbog toga aktivira nadzorni alarm. Naime, zbog neke greške u jednom dijelu programa, upravljački program ne dolazi do kritičnog dijela u kojem se nadzorni alarm resetira. Stoga je jedno od rješenja ponovno pokrenuti sustav s nadom da neće opet doći u takvo stanje.

Grešku je ponekad moguće otkriti dodavanjem kôda koji služi isključivo za provjeru ispravnog rada i stanja sustava.

Pri učenju programiranja (radi jednostavnijeg prikaza) se obično smatra da će uobičajene funkcije ispravno obaviti svoje operacije (npr. `printf`, `read`). Međutim, radi veće ispravnosti potrebno je provjeriti svaku povratnu vrijednost svih funkcija koje vraćaju vrijednost. U slučaju greške (uobičajeno je da tada funkcija vrati vrijednost `-1`) može se uvidom u oznaku greške (engl. `error number`, `errno`) doznati i više informacija o razlogu greške i dalje postupiti odgovarajuće.

Ponekad je kôd koji ispituje ispravno stanje sustava (npr. provjeru varijabli, argumenata funkcije, povratnih vrijednosti i slično) aktivan samo pri ispitivanju rada, ali ne i kasnije tijekom stvarnog korištenja sustava (nakon ispitivanja taj se kôd isključuje radi ostvarivanje veće učinkovitosti). Primjer ovakva ispitivanja jest korištenje makroa `assert`:

```
assert ( nešto_što_bi_trebalo_bitи_istina );
```

Ukoliko se uvjet naveden u `assert`-u ne izračuna u istinu, program se zaustavlja (poziva se `abort`).

Zaustavljanje uz ispis poruke o grešci može imati smisla u sustavima koja su nadzirana od strane čovjeka ili su pristupačna. Međutim, ukoliko su takvi sustavi nedostupni (udaljeni) navedeni

pristup će ih jednostavno zaustaviti. Možda i takav postupak ima smisla u neki sustavima. U drugim sustavima će biti bolje da je sustav i dalje aktivan, da se pokuša nekako izvući iz stanja kvara.

Operacije otkrivanje grešaka i akcija na njih trebaju biti ugrađeni u sam kôd. Primjerice, sljedeći kôd pokušava otkriti grešku u sustavu i otkloniti ju.

Isječak kôda 8.6. Ugrađeni postupak oporavka od pogreške

```

...
if ( temp < Tmin )
{
    log ( WARN, "Temperatura manja od očekivane!" );

    //očitaj ponovo, možda je bio problem u senzoru
    temp = TEMPERATURA ( ocitaj_senzor ( ulaz_55 ) );
    if ( temp < Tmin )
    {
        //je li grijač upaljen?
        if ( ocitaj_senzor ( ulaz_22 ) & GRIJAC )
        {
            //grijač upaljen, ali ne grijee?
            log ( ERROR, "Grijač 22 ne grijee, resetiram sustav!" );
            reset ();
        }
        else {
            postavi_senzor ( ulaz_22, GRIJAC );
            goto pricekaj_radno_stanje;
        }
    }
}
...

```

U prethodnom primjeru se sustav resetira tek ako ne postoji operacija koja bi iz tog stanja dovela sustav u normalno stanje (brže od reseta). Ako ne postoji mogućnost da se sustav izbavi iz uočenog problema (npr. ako resetom nema nade da će grijač ipak proraditi), onda je jedino logično zaustaviti sustav (primjerice, umjesto `reset` staviti `abort` ili slično).

Ponekad je problem u nedostatku sredstava sustava. Naime, ugradbeni sustavi često imaju ograničenu veličinu radnog spremnika koji može postati usko grlo. Dinamička dodjela spremnika koja s jedne strane omogućava fleksibilnost u njegovu korištenju (veću učinkovitost korištenja spremnika), s druge strane može biti i problem kada slobodnog spremnika ponestane. Primjer u nastavku prikazuje jedno moguće upravljanje u takvim slučajevima.

Isječak kôda 8.7. Postupanje u slučaju nedostatnih sredstava sustava

```

while ( posluzivanje_traje )
{
    zahtjev = dohvati_sljedeci_zahtjev (...);

pokusaj_ponovo:
    stanje_obrađe = malloc ( sizeof (stanje_obrađe) );

    if ( !stanje_obrađe )
    {
        /* nema memorije ! */

        #ifdef KAD_NEMA_MEMORIJE__ODBACI_ZAHTJEVE
        /* 1. pristup: odbaci zahtjeve */
        log ( WARN, "Nema memorije za nove zahtjeve!" );
        odbaci_zahtjev ( zahtjev );
        continue;
        #else
        /* 2. pristup: probati kratko pričekati pa ponoviti */

```

```
    nanosleep ( kratko );
    goto pokusaj_ponovo;
    #endif
}
//obrada zahtjeva
...
}
```

U SRSV-ima bi svakako trebalo uočiti moguće probleme i za njih ugraditi prikladan postupak oporavka od pogreške.

Pitanja za vježbu 8

1. Usporediti programske jezike C i Adu u kontekstu korištenja za SRSV.
2. Koja svojstva prevoditelja je moguće ili potrebno koristiti pri izgradnji programske potpore za SRSV-e?
3. Kada ugrađivati funkcije na mjesto poziva (definirati ih sa inline), a kada ne?
4. Čemu služi oznaka `volatile` pri definiranju varijabli? Koje probleme takva definicija rješava?
5. Za kakve funkcije kažemo da nisu prikladne za višedretveno okruženje? ◇
6. Koje su granice zapisa cijelih tipova podataka te koja je preciznost realnih brojeva? ◇
7. Što je potrebno sve paziti pri korištenju višeprocesorskih i višejezgrinih sustava (kod raspoređivanja dretvi) u SRSV okruženju?
8. Kako se ponekad može smanjiti vremenska složenost postupka?
9. Što je to prostorna složenost?
10. Kako ispitni uzorci mogu utjecati na razvoj programske potpore?
11. Za zadani odsječak koda navesti nedostatke (u kontekstu korištenja u SRSV-ima).

```
printf("Unesi parametar A (5-10):");
scanf("%d", &A);
a = obrada (A);
if ( a > 5 )
    exit(-1);
else
    posalji(A, a);
```

12. Za dodjelu jedinstvenih identifikacijskih brojeva koristi se funkcija:
`int id () { static int broj = 0; return broj++; }.`
 Koje sve nedostatke ima navedena funkcija?
13. Navesti nedostatke navedena koda (u raznim "okruženjima") i mogućnosti za njihovo rješavanje.

```
int op ( ... ) {
    static int brojilo = 0;
    ... /* neka operacija, bez korištenja varijable brojilo */
    brojilo = brojilo + 1;
    return brojilo;
}
```

14. Odrediti složenost idućeg algoritma u O (veliko O) notaciji (funkcija1 je linearne složenosti – $O(N)$).

```
for ( i = 0; i < N; i++ )
    for ( j = i + 1; j < N; j++ )
        A[i][j] = funkcija ( i, j, N );
```

15. Popraviti sljedeću funkciju tako da općenito bude maksimalno moguće precizna uz zadane tipove podataka (bez mjenjanja tipova).

```
double funkcija2 ( double a, double b )
{
    double c;
```

```
c = 1 - a / 1e50;
if ( b > 0 )
    c = c - 1;
return c;
}
```

16. Neka se x i y računaju sa:

```
x = a - b*c;
y = d - x;
```

Koji problemi preciznosti mogu nastati izvođenjem programa, ako su sve varijable:

- a) cjelobrojne
- b) realne?

17. Između očitanja neke ulazne naprave treba proći između 20 i 50 sabirničkih ciklusa.

Kako to ostvariti ako se može programirati u asembleru, te kako ako se mora koristiti C?

9. Operacijski sustavi

9.1. Uloga operacijskih sustava

Operacijski sustav je skup osnovnih programa koji se nalaze između korisnika računala, tj. primjenskih programa i sklopolja. Njegova je zadaća da korisniku (primjenskim programima) pruži prikladnu okolinu za učinkovito korištenje računalnog sklopolja, za suradnju na obavljanju zajedničkih zadataka i slično. U prethodnim poglavljima koristila se pretpostavka da operacijski sustav postoji i da ne unosi dodatne probleme u sustav. U ovom se poglavlju razmatraju neki detalji operacijskih sustava koji mogu utjecati na ukupna svojstva sustava, pogotovo ako ih se ne uzme u obzir pri projektiranju sustava.

Slika 9.1. prikazuje uobičajeno mjesto operacijskog sustava u računalnom sustavu.



Slika 9.1. Slojevi računalnog sustava

Operacijski sustav je najkritičniji dio programske okruženja nekog računalnog sustava te ga kao takvog smišljaju, izgrađuju i mijenjaju iskusni stručnjaci s ovog područja. Njegova korisnost mjerljiva je njegovim mogućnostima i brzini rada.

Različiti računalni sustavi postavljaju vlastite zahtjeve na programsko okruženje. Operacijski sustav osobnog računala, namijenjen mnoštvu aktivnosti, kao što su uredski programi, pregledavanje i stvaranje multimedijalnih sadržaja te računalne igre, bitno se razlikuje od onog koje upravlja nekim proizvodnim procesom.

Većina funkcija koje operacijski sustavi danas imaju potiče od prvih operacijskih sustava razvijenih na poslužiteljskim računalima. Ti su prvi sustavi postavili temelje gotovo svim sustavima koje danas koristimo, bilo na osobnom računalu u uredu ili kući, ili poslužiteljima. Ipak, u području SRSV-a i ugrađenih računala uporaba operacijskih sustava znatno je manja. Neki ugrađeni sustavi toliko su jednostavnii da nemaju potrebe korištenja operacijskog sustava već je dovoljan upravljački program. Kada se operacijski sustavi i koriste u ovim okruženjima tada su ti operacijski sustavi znatno drugačijih svojstava.

Načela rada operacijskih sustava, osnovni koncepti i slično opširnije su opisani u literaturi.

Osnovni dijelovi operacijskog sustava – podsustavi su:

1. upravljanje vanjskim jedinicama
2. upravljanje spremničkim prostorom

3. upravljanje vremenom
4. upravljanje dretvama/procesima
5. mehanizmi komunikacije i sinkronizacije
6. datotečni podsustav
7. mrežni podsustav.

Svojstva nekih od podsustava (3., 4., 5. i 7.) su već predstavljena u okviru problematike raspo-ređivanja, sinkronizacije, komunikacije i upravljanja vremenom (poglavlja 4., 6. i 6.1.).

Upravljanje vanjskim jedinicama i spremničkim prostorom je vrlo bitno za operacijske sustave za rad u stvarnom vremenu jer oni mogu znatno utjecati na način upravljanja i svojstva takvih sustava. Stoga su upravljanje vanjskim jedinicama i spremnikom dodatno opisani u nastavku ovog poglavlja.

9.1.1. Upravljanje vanjskim jedinicama

Upravljanje vanjskim jedinicama u sustavima koji koriste operacijski sustav obavlja se preko operacijskog sustava. U sustavima koji ga nemaju, upravljanje se može izvesti i izravno iz programa. Svaka vanjska jedinica ima svoje posebnosti. Da bi se olakšalo izgradnju operacijskih sustava, složenost upravljanja napravama se izdvaja u posebne dijelove koje nazivamo “upravljačkim programima naprava” ili kraće samo “upravljački program” (u kontekstu operacijskih sustava) (engl. *device driver*).

Upravljanje vanjskim jedinicama, tj. ulazno-izlaznim napravama se može obavljati na tri načina:

1. upravljanje programskom petljom
2. korištenjem prekida
3. korištenjem sklopova za izravni pristup spremniku.

Upravljanje programskom petljom izvodi se samo za najjednostavnije naprave koje nemaju upravljačke sklopove. Upravljački program mora sve sam napraviti – provjeravati stanje naprave (treba li nešto napraviti s napravom), slati naredbe i podatke u napravu, čitati podatke i stanja naprave. Možda najveći problem jest ustanoviti kada se nešto dogodilo, jer naprava neće sama zatražiti akciju. Upravljački program za takvu napravu najčešće periodički provjerava stanje naprave i pokreće akcije kada se dogode promjene. Ukoliko sustav ne može nastaviti s radom dok ne dobije podatke te naprave, mora se korisiti mehanizam radnog čekanja, tj. u petlji dohvaćati status naprave dok se on ne promijeni.

S druge strane, sklopovi s izravnim pristupom spremniku se koriste za prijenos veće količine podataka od vanjske naprave ili prema njoj, a ne izravno za upravljanje napravama i okolinom s kojima su naprave povezane te se stoga ovdje neće posebno razmatrati.

Prekidi su najčešće korišten mehanizam za upravljanje vanjskim jedinicama. Oni omogućavaju da naprave izazivaju prekidni signal u trenucima kada se dogodi promjena na samim napravama. Operacijski sustav na prekidne signale poziva odgovarajuće funkcije koje će obraditi te događaje. Reakcija će stoga biti vrlo brza, osim ako se u trenutku pojave zahtjeva za prekid već poslužuje neki prethodni zahtjev. Naime, pri prihvatu prekida obavljaju se neki kućanski poslovi koje se ne smije prekidati (spremanja trenutno prekinute dretve – njenog konteksta, otkrivanja uzroka prekida). Nadalje, sama obrada prekida može biti kritična operacija koju se ponekad ne smije prekidati te se za vrijeme njene obrade svi ostali zahtjevi za prekid ne prihvataju.

Obzirom da je za SRSV-e vrlo bitno promptno odgovoriti na zahtjev iz okoline, način prihvata i obrade prekida treba prilagoditi da zadovoljavaju vremenska ograničenja koja su postavljena prema reakcijama na vanjske događaje. Kada bi se neka obrada prekida izvodila s zabranjenim preidanjem i potrajala malo duže, to bi onemogućilo prihvat drugog prekida i reakciju na njega. Stoga se prekidni podsustav treba ostvariti da razlikuje takve obrade, odnosno, da

omogući da se najkritičnjim elementi upravljanog sustava posveti najviše pažnje, tj. da se vrlo brzo reagira na njihove zahtjeve.

Uobičajeni pristupi pri oblikovanju prekidnog podsustava su:

1. obrađivati prekide redom prispijeća
2. zahtjeve za obradama stavljati u prioritetnu listu te
3. obrade prekida podijeliti na dva dijela: hitni dio i manje hitni dio.

Obrada prekida redom prispijeća uz zadržavanje svih prekidnih zahtjeva dok se prethodni ne obrade je najjednostavniji i najčešće korišteni mehanizam. Da bi on bio svrshodan neophodno je da sve obrade prekida traju vrlo kratko. Ako je to za neku primjenu moguće napraviti onda je to pravo rješenje.

Obrađivanje prekida prema prioritetima na prvi bi pogled moglo izgledati kao najbolje rješenje. Međutim, treba uzeti u obzir da kućanski poslovi iako traju kratko (i ispod mikrosekunde na bržim sustavima) nisu zanemarivi ako se često izvode. Naime, u takvim sustavima se na svaki prekid poziva prekida procedura koja mora utvrditi uzrok prekida, oblikovati zahtjev za obradu te ga postaviti u red. Česti prekidi stoga mogu prekidati trenutnu obradu koja zbog toga možda neće stići obaviti sve operacije do zadanih vremena.

Mogućnost podjele obrade u dva dijela kombinira prethodna dva pristupa: prekidi čija obrada traje kratko se izvode odmah, dok se duži prekidi dijele na hitni dio koji se odmah izvede te manje hitan dio koji se može izvesti i s većim odmakom. Kada za obradu prekida treba više vremena tada se u hitnom dijelu prikupe osnovni podaci potrebni za obradu u jedan zahtjev (koja naprava, parametri, koju funkciju treba pozvati) koji se stavlja u listu zahtjeva. Sam se zahtjev obrađuje naknadno prema prioritetu (u odnosu na druge zahtjeve u listi) i to s dozvoljenim prekidanjem. Na taj način se može dozvoliti da obrada i dulje traje, a da ipak ne utječe na prekide koji traže hitnu obradu.

Operacijski sustavi za SRSV uglavnom omogućavaju samo osnovni način obrade prekida – prema redu prispijeća, jer se podrazumijeva da obrade prekida trebaju trajati vrlo kratko. Ipak, neki od takvih sustava omogućavaju da sama obrada ipak bude prekidiva drugim prioritetnijim zahtjevima, te na ovaj način omogućavaju obradu prekida prema prioritetu (koji je pridijeljen napravama).

Podjela obrade prekida na dva dijela podržana je i u operacijskim sustavima koji nisu predviđeni za rad u stvarnom vremenu. Primjerice, na sustavima zasnovanim na Win32 jezgri za hitan dio obrade prekida koristi se termin prekidna rutina (engl. *interrupt service routine*), a za manje hitan dio prekidna dretva (engl. *interrupt service thread*). Ekvivalentni termini za sustave temeljenje na Linux jezgri su: gornja polovica (engl. *top half*) za hitan dio obrade, te donja polovica (engl. *bottom half*) za manje hitan dio.

Operacijski sustavi nude mehanizme, ali je odluka arhitekta sustava kako će ih iskoristiti u određenim okolinama – hoće li sve obrade oblikovati da budu kratke, ili će obrade dijeliti na dva dijela, ili će uz sklopovsku potporu obrade prekidati prioritetnijim zahtjevima.

U funkcijama za obradu prekida treba paziti koje se funkcije koriste. Nije poželjno ili čak i moguće u njima koristiti funkcije koje mogu blokirati. Npr. poziv funkcije *ČekajSemafor* nema smisla u obradi prekida, dok poziv *PostaviSemafor* se smije koristiti – on neće blokirati izvođenje obrade.

9.1.2. Upravljanje spremničkim prostorom

Upravljanje spremničkim prostorom koristi algoritme koji odlučuju u koje dijelove spremnika učitati instrukcije, podatke, gdje smjestiti stog, gomilu, kako zaštитiti jednu dretvu od druge, kako zaštитiti operacijski sustav i slično. Najjednostavnije metode imaju najviše nedostataka, ali ne traže dodatne složene sklopove, dok složenije metode nude više, ali traže i sklopovsku

potporu. Upravljanje spremnikom će stoga ovisiti o namjeni sustava (jednostavni ili složeniji) i o dostupnoj sklopovskoj potpori.

Osnovni načini upravljanja spremnikom su:

1. statičko upravljanje
2. dinamičko upravljanje
3. straničenje.

Sva tri načina imaju mogućnosti korištenja pomoćnog spremnika za privremenu pohranu procesa koji trenutno ne stanu u radni spremnik. Međutim, u kontekstu SRSV-a, korištenje pomoćnih spremnika je problematično obzirom da to može uzrokovati značajne odgode u izvođenju tih programa, a zbog dohvata potrebnih podataka s pomoćnog spremnika (što se može mjeriti u desecima milisekundi). Ako se ipak koriste pomoćni spremnici treba biti vrlo oprezan što u njih postaviti. Najbolje bi bilo kada bi na njih spremali samo nekritične programe, one koji ne upravljuju bitnim dijelovima sustava.

Kod statičkog upravljanja spremnik se podijeli u nekoliko segmenata u koji se smještaju različiti programi pripremljeni za te segmente (adrese koje se koriste u programima su pripremljene za taj segment). Odlike statičkog upravljanja su u njegovoj primjenjivosti i na najjednostavnijim procesorima (ne traži nikakvu sklopovsku potporu za upravljanje spremnikom). Nedostaci su u fragmentaciji, nepostojanju zaštite, potrebe za prilagođavanjem programa za pojedine segmente, neučinkovitosti u korištenju pomoćnog spremnika.

Za ostvarenje dinamičkog upravljanja potrebna je sklopovska potpora u obliku jednog registra i jednog zbrajala. Korištenjem samo tih dviju povezanih komponenti omogućava se da programi ostaju u logičkim adresama, može ih se učitati bilo gdje u spremnik (spremnik ne treba unaprijed podijeliti na segmente). Dodatkom dva komparatora može se ostvariti i zaštita spremnika, tj. ograničiti pristup spremniku na dodijeljeni segment. Nedostaci uključuju fragmentaciju i neučinkovitost u korištenju pomoćnog spremnika (kao i kod statičke metode).

Straničenje dijeli adresni prostor procesa na stranice, spremnik na okvire dimenzija stranice te uz pomoć sklopovlja omogućava da se stranice procesa nalaze u bilo kojim okvirima, bilo kojim redoslijedom. Prednost u ovoj, na prvi pogled usitnjenoj slici spremnika jest u fleksibilnosti. Pojedini proces pri svom pokretanju može korisiti jedan skup okvira, a kasnije tražiti još, otpuštati one koje više ne koristi i slično (kod statičkog i dinamičkog upravljanja, sav se adresni prostor za proces trebao odmah zauzeti pri pokretanju procesa). Nadalje, kao i kod dinamičkog upravljanja, kod straničenja proces ostaje u logičkim adresama dok se pretvorba adresa obavlja sklopovljem. Straničenje zahtijeva značajno složenije sklopovlje i strukturu podataka – to je njegov najveći nedostatak za primjenu u ugrađenim sustavima. Obzirom da sklopovlje adrese prevodi preko tablica (koje treba pripremati operacijski sustav) zaštita između procesa i procesa, procesa i jezgre je lako ostvariva. Nadalje, tablice se mogu tako izgraditi da omogućuju dijeljenje adresnog prostora između dva procesa, omogućujući iznimno brzu komunikaciju ta dva procesa bez korištenja posrednika (operacijskog sustava).

Od tri navedena načina upravljanja spremnikom, straničenje je najbolje po svojstvima, ali zbog potrebe složenog sklopa nije primjenjivo na svim sustavima (na jednostavnijim upravljačkim računalima, osobito ugrađenim).

Radi rješavanja problema korištenja pomoćnog spremnika, kritični programi mogu zatražiti stalni smještaj svojih elemenata u radnom spremniku. Primjerice korištenjem poziva `mlock` i `mlockall`. Ukoliko programi dinamički traže dodatni spremnički prostor (primjerice s `malloc`) tada je dodatno potrebno osigurati da i taj naknadno traženi prostor ne bi bio dodijeljen na pomoćnom spremniku. Jedna od mogućnosti za to jest na početku programa rezervirati dio spremničkog prostora koji će biti dostatan i u najzahtjevnijim trenucima. Potom zaključati sve stranice procesa u radni spremnik te na kraju oslobođiti taj dio spremnika. Naime, većina ostvarenja dinamičkog upravljanja spremnikom ne vraća traženi spremnik operacijskom sus-

tavu sve do završetka procesa. Tako će početno zauzimanje spremnika osigurati da i naknadni zahtjevi koriste taj već dodijeljeni dio procesu. Ipak, potrebno je provjeriti je li takvo ponašanje i na sustavu koji se koristi u pojedinom slučaju.

9.2. Operacijski sustavi posebne namjene

Upravljanje različitih sustavima se međusobno znatno razlikuje. Negdje je dovoljan kratki upravljački program, dok je drugdje potreban složeni višezačni sustav posebnih svojstava.

Primjerice, osobno računalo bilo u uredu ili u kući, kao i radna stаница, nije osmišljeno za obavljanje samo jednog posla već mnoštvo njih. Međutim, zbog svoje opće namjene takvo računalo nema odgovarajuće mehanizme potrebne za uporabu u većini SRSV-a. Kao prvo, dimenzijama i cijenom bitno odudara od zahtjeva ugrađenih sustava. Nadalje, što je i mnogo važnije, nema ugrađene mehanizme vremenske određenosti, kako ni u sklopolju tako ni u programskoj okolini. Na primjer, kod osobnog računala gotovo je sasvim nebitno je li za prikaz nekog prozora na zaslonu potrebna sekunda, dvije, tri ili se neki proračun obavlja desetak ili više sekundi ili minuta, ali je ta vrsta neodređenosti nedozvoljena u ugrađenim sustavima. Uredski operacijski sustavi i sustavi na radnim stanicama imaju vrlo lošu podršku vremenski uređenim zadacima i odzivu na vanjske događaje, pa su u većini slučajeva neupotrebljivi za SRSV-e (osim eventualno za one s blagim vremenskim ograničenjima).

U novije vrijeme pojavljuju se novi operacijski sustavi koji pokrivaju segment takozvanih prijenosnih i ručnih računala, bilo samostojećih ili povezanih s mogućnostima mobilnih telefona ili drugih uređaja. Primjeri takvih sustava su "pametni" mobilni telefoni (engl. *smartphones*), multimedijalni uređaji, čitači električnih knjiga, ručna računala. Primjeri operacijskih sustava koje koriste navedeni uređaji su iOS, Android i Windows Phone 8. Ne može se reći da su prethodni sustavi tipični ugrađeni sustavi. Dapače, navedeni su po svojstvima bliži općim operacijskim sustavima za stolna računala. Međutim, i ti operacijski nastoje proširiti područja svoje primjene u ugrađenim sustavima, primjerice za uređaje u automobilima, za "pametne" televizore i slično. Zato se i razvijaju u smjeru postizanja bar minimalnih svojstava potrebnih za takve sustave, tj. nastoje ostvariti mogućnosti za primjenu u sustavima s blagim vremenskim ograničenjima.

U pogledu mogućnosti za primjenu u SRSV-ima treba najprije izdvojiti operacijske sustave posebno izgredjene za tu primjenu. Tu primjerice spadaju QNX [Neutrino], Windriver [VxWorks], Enea [OSE], [MicroC/OS-II], [FreeRTOS] i slični operacijski sustavi. Njihovo korištenje ima svoje prednosti, ali i nedostatke.

U nastavku je malo detaljnije opisan FreeRTOS. Posebnost FreeRTOS-a jest što je to sustav na nižoj razini od ostalih, programi i funkcije za obradu prekida su jače povezane sa samim operacijskim sustavom (zahtijevaju posebna znanja programera o načinima rada FreeRTOS-a). Ostali sustavi imaju uglavnom znatno odijeljene programe od jezgre.

9.2.1. Operacijski sustav FreeRTOS

FreeRTOS je namijenjen ugradbenim računalnim sustavima za rad u stvarnom vremenu i optimiran prema kriteriju malih zahtjeva prema sklopolju (spremnički prostor i procesorska moć) te se može koristiti i na mikroupravljačima. Za razliku od prethodno navedenih, izvorni kod FreeRTOS-a je slobodno dostupan (besplatan).

Korištenje FreeRTOS-a zahtijeva detaljnije poznavanje njegova sučelja i načina rada. U ovom odjeljku navedena su neka svojstva tog sustava kao i mogućnosti izgradnje programske potpore zasnovane na njemu.

Isključivanje nepotrebnih dijelova jezgre

Radi postizanja što manjeg zahtjeva na spremnički prostor, mnoge se operacije jezgre mogu izostaviti iz prevođenja. Primjerice, ako se sučelje, tj. jezgrina funkcija Neka_Jezgrina_Funkcija ne koristi onda u odgovarajućoj datoteci s postavkama treba postaviti 0 umjesto 1 u liniji (dodati liniju ako ne postoji):

```
#define INCLUDE_Neka_Jezgrina_Funkcija 0
```

Datoteka s postavkama je najčešće FreeRTOSConfig.h smještena u direktorij s programima koji koriste FreeRTOS. Koje se sve funkcije mogu isključiti iz prevođenja vidi se iz koda FreeRTOS-a jer su takve funkcije ograđene mehanizmom opcionalnog uključivanja:

```
#if ( INCLUDE_Neka_Jezgrina_Funkcija == 1 )
    tip Neka_Jezgrina_Funkcija ( parametri )
{
    ...
}
#endif /* INCLUDE_Neka_Jezgrina_Funkcija */
```

Imenovanje varijabli

Imena varijabli i funkcija sadrže prefikse koji označavaju tip varijabli, odnosno, povratne vrijednosti funkcije. Tako će cjelobrojna varijabla imati prefiks x, pozitivna cjelobrojna varijabla ux, kazaljka p, tj. pux za kazaljku na cjelobrojnu varijablu, v za funkciju tipa void i slično. Npr. prototip funkcije za stvaranje dretve jest:

```
BaseType_t xTaskCreate (
    TaskFunction_t pvTaskCode,
    const char * const pcName,
    uint16_t usStackDepth,
    void *pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t *pvCreatedTask
);
```

Zaštita pri korištenju dijeljenih sredstava sustava

Poprilično detaljno poznавanje rada sustava je neophodno kod FreeRTOS-a. Korištenje mnogih operacija mora biti ostvareno kao kritični odsječak, ali se to ostvarenje treba ostvariti u programu, nije uvijek ugrađeno u jezgrine funkcije. Primjerice, pri korištenju gomile (engl. *heap*) sučeljem malloc i free treba onemogućiti zamjenu s trenutne dretve na neku drugu radi očuvanja konzistentnosti strukture podataka kojom se upravlja gomilom. Primjerice, to se može napraviti sa:

```
vTaskSuspendAll ();
pvReturn = malloc ( xWantedSize );
xTaskResumeAll ();
```

Navedeni primjer prikazuje potrebu upravljanja sustavom na najnižoj razini u samom programu. Gornji primjer neće prekidati druge dretve, ali prekidi hoće. Zabrana prekida manjeg prioriteta od dretve može se napraviti sa taskENTER_CRITICAL() (ponovna dozvola sa taskEXIT_CRITICAL() ili svih prekida sa taskDISABLE_INTERRUPTS()).

Pokretanje raspoređivača dretvi

Pri pokretanju sustava višedretvenost nije omogućena, niti ona to mora biti. Pokretanje višedretvenosti postiže se pozivom vTaskStartScheduler(). Primjerice, funkcija koja (nakon

inicijalizacije) preuzima kontrolu nad sustavom, stvara potrebne dretve i slično, može izgledati kao sljedeći primjer.

Isječak kôda 9.1. Primjer višedretvenosti u FreeRTOS-u

```
void vFunkcija ( void )
{
    TaskHandle_t xHandle = NULL;

    /* Inicijalizacija, stvaranje ostalih potrebnih objekata */

    /* Stvaranje bar jedne dretve (prije pokretanja raspoređivača) */
    xTaskCreate (
        vDretva, "IME_DRETE", VEL_STOGA, NULL, prioritet, &xHandle
    );

    /* ... stvaranje ostalih dretvi i potrebnih objekata */

    /* Pokretanje prioritetskog raspoređivača */
    vTaskStartScheduler ();

    /* Kontrola neće doći ovdje dok se ne zastavi raspoređivač
     * od strane neke dretve sa: vTaskEndScheduler ();
     * tj. najčešće se ovdje nikad ne vraća! */

    /* primjer micanja dretve iz sustava */
    if ( xHandle != NULL )
        vTaskDelete ( xHandle );
}

void vDretva ( void * pvParameters )
{
    for (;;)
    {
        /* Posao dretve */
    }
}
```

Same dretve trebaju biti oblikovane kao funkcije koje nikada ne završavaju. Odnosno, ako trebaju završiti onda one (ili druge dretve) trebaju pozvati `vTaskDelete(pxTask)`. Struktura početne funkcije dretve treba dakle izgledati kao u primjeru.

Manje zahtjevne dretve – niti

Višedretvenost može za neke sustave biti suviše zahtjevna na spremnički prostor jer za svaku dretvu treba opisnik i zaseban stog. Za takve sustave (ali i druge), umjesto pravih dretvi (ili i paralelno sa njima) mogu se koristiti jednostavnije dretve, nazovimo ih *nitima* (u FreeRTOS-u su nazvane *co-routine*). Sve niti jedne aplikacije dijele isti stog, funkcije upravljanja su jednostavnije, ali ne mogu se sve jezgirne funkcije koristiti. Zbog zajednička stoga potrebno je posebno paziti i na oblikovanje aplikacije.

Međudretvena sinkronizacija i komunikacija

Od mogućnosti međudretvene komunikacije treba izdvojiti redove poruka, semafore: binarne, opće, binarne s nasljeđivanjem prioriteta (izvorno *mutex semaphore*) i alarme. Ukoliko dretva treba čekati na više događaja (poruku ili semafor) mogu se koristiti mehanizmi skupova redova (engl. *queue sets*), gdje se dretva može blokirati dok bar jedan od redova u skupu ne postane prolazan (ima poruku ili semafor postane prolazan).

Pozivi jezgrinih funkcija iz prekidnih funkcija

Većina funkcija za upravljanje dretvama i povezanim objektima ima i inačice pripremljene za poziv iz obrade prekida (engl. *from interrupt service routine*) sa sufiksom `FromISR`. Primjerice, funkcija za slanje poruke u red poruka `xQueueSend` ima alternativnu inačicu `xQueueSendFromISR` za korištenje iz obrade prekida.

```
 BaseType_t xQueueSend (
    QueueHandle_t xQueue,
    const void * pvItemToQueue,
    TickType_t xTicksToWait
);
BaseType_t xQueueSendFromISR (
    QueueHandle_t xQueue,
    const void *pvItemToQueue,
    BaseType_t *pxHigherPriorityTaskWoken
);
```

Preko trećeg parametra funkcije `xQueueSendFromISR` dobiva se informacija o tome je li se dodavanjem poruke u red oslobođila neka prioritetsnija dretva od trenutno aktivne (i prekinute ovim prekid), u kom slučaju bi trebalo pozvati raspoređivač. Skica takve prekidne funkcije dana je u nastavku.

```
void vPrekidnaFunkcija(void)
{
    QueueHandle_t xQueue = dohvati_red();
    const void *pvItemToQueue = dohvati_poruku();
    BaseType_t pxHigherPriorityTaskWoken;

    xQueueSendFromISR(xQueue, &pvItemToQueue, &pxHigherPriorityTaskWoken);
    if ( pxHigherPriorityTaskWoken )
    {
        portSAVE_CONTEXT();
        vTaskSwitchContext();
        portRESTORE_CONTEXT();
    }
    asm volatile ( "reti" );
}
```

U primjeru treba primijetiti da se spremanje konteksta i obnova konteksta radi ‘ručno’ zato jer se radi o funkciji koja se zove iz obrade prekida (sama funkcija `vTaskSwitchContext()` samo odabire najprioritetniju za aktivnu dretvu ali ne radi promjenu konteksta).

Ovisno o sklopolju na koji se FreeRTOS stavlja obrade prekida mogu se prekidati prioritetnijim prekidima (kada to sklopolje dozvoljava). U tom slučaju treba pažljivo odabrati prioritete prekida i dretvi i programske prekida i sklopoških prekida.

FreeRTOS je operacijski sustav na vrlo niskoj razini upravljanja, ali zato omogućuje ugradnju u sustave s minimalnim spremničkim prostorom (npr. u sustavu koji ima samo 4 KB radnog spremnika!).

9.3. Svojstva različitih tipova operacijskih sustava

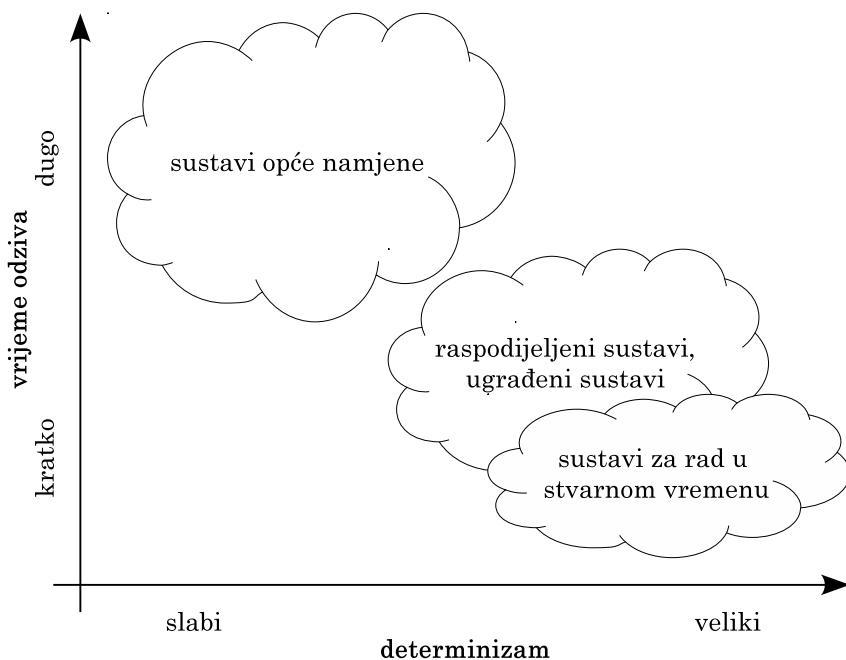
Odabir operacijskog sustava za pojedine primjene treba obaviti sukladno očekivanjima funkcionalnosti i usklađenosti s okolinom u koju se ugrađuje. Također treba planirati mogućnosti ažuriranja i nadogradnje kroz predviđeni životni vijek sustava. Cijena nabavke i podrška proizvođača pri održavanju trebaju također biti jedan od razloga pri odabiru (za “dugovječne sustave”).

Slika 9.2. prikazuje odnos svojstava različitih kategorija operacijskih sustava. Operacijski sus-

tavi opće namjene građeni su generički, kao i samo sklopolje za takva računala. Oni su napravljeni da mogu raditi mnoštvo stvari uz zadovoljavajuću kvalitetu i nisku cijenu. Promatrajući svojstva takvih sustava u kontekstu uporabe za upravljanje vremenski kritičnih procesa uočavaju se neki nedostaci koji su posljedica otvorenosti operacijskih sustava opće namjene raznim sklopolovskim i programskim rješenjima. Zato je odziv takvih sustava vrlo nepredvidiv, često neprihvatljivo dug, a kako bi se takvi sustavi mogli koristiti za upravljanje procesa sa strogim vremenskim ograničenjima. Za nekriticne elemente sustava, gdje su zadana blaža vremenska ograničenja, moguće je odabrat i operacijske sustave za opću uporabu. Pravilnim podešavanjem sustava može se znatno poboljšati ponašanje sustava u smislu pouzdanosti i predvidivosti ponašanja.

S druge strane, operacijski sustavi posebno osmišljeni za sustave sa strogim vremenskim ograničenjima, kao što su to operacijski sustavi za rad u stvarnom vremenu te operacijski sustavi za ugrađena računala, imaju, naravno, znatno bolja vremenska svojstva. Nedostatak takvih sustava je u manjoj podršci sklopolju i dobavljivosti gotovih programske rješenja.

Između pravih operacijskih sustava za rad u stvarnom vremenu i operacijskih sustava opće namjene postoji područje u kojemu se nalaze prilagođeni operacijski sustavi opće namjene, ali značajno boljih svojstava. Prednosti takvih sustava su u dostupnosti gotovo jednakih razvojnih alata i podržanih protokola kao i u sustavima opće namjene.



Slika 9.2. Odnos svojstava između općih sustava i onih posebne namjene

9.3.1. Operacijski sustavi za rad u stvarnom vremenu

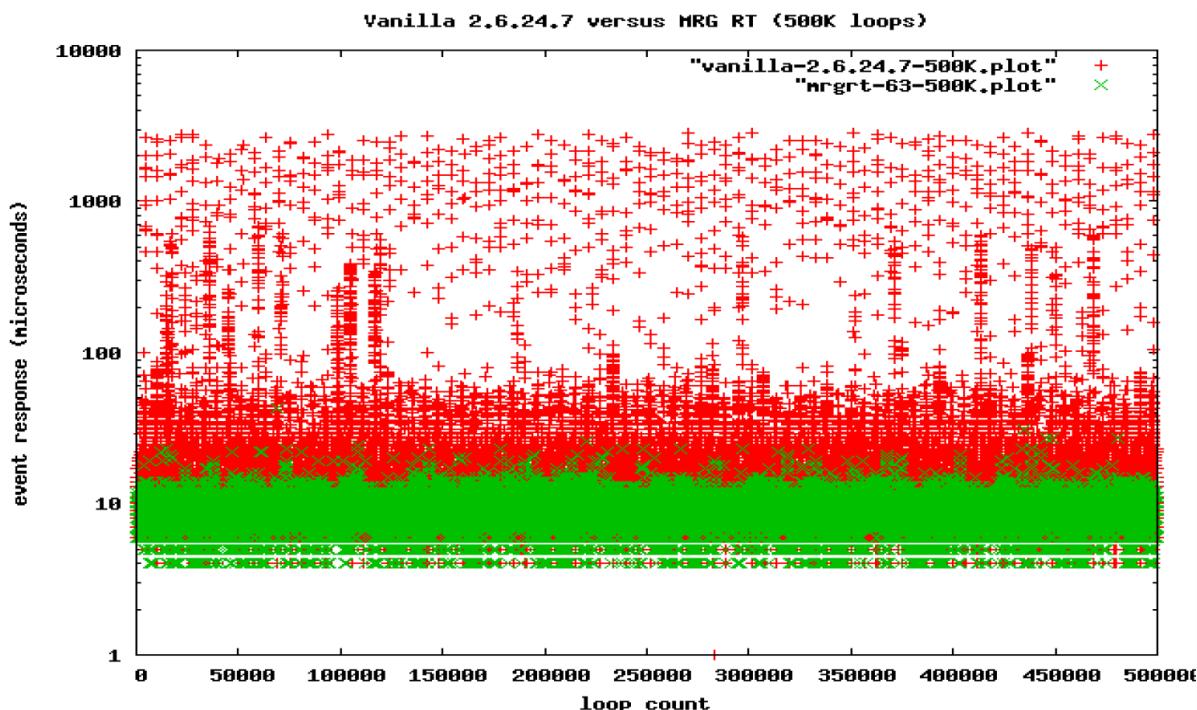
Primjere zašto neki operacijski sustavi opće namjene nisu pogodni za kriticne sustave mogu se potražiti iz raznih studija. Jedna od takvih uspoređuje eksperimentalne rezultate provedene na Linux sustavima s jezgrama 2.4 i 2.6 te s prilagođenim Linux sustavima za uporabu u SRSV-ima. U eksperimentu su mjerena kašnjenja od trenutka pojave događaja do početka njegove obrade (engl. *preemption latency*). Tablica 9.1. prikazuje rezultate jednog eksperimenta. Prosječna kašnjenje su samo malo veća kod običnih sustava, međutim, kada se pogledaju svi slučajevi i izvuku najgori slučajevi onda se jasno vidi da su operacijski sustavi opće namjene (koji nisu prilagođeni) značajno lošiji.

Slika 9.3. prikazuje vremena odziva na Linux-u s inačicom jezgre 2.6.24 i njegovoj SRSV inačici.

Tablica 9.1. Izmjerena kašnjenja do početka obrade prekida u mikrosekundama ([Laurich, 2004])

	2.4 Linux	2.6 Linux	2.4 Linux RTAI	2.4 Linux LXRT
Prosječna kašnjenja	12 – 14	12 – 14	8 – 10	10 – 12
Najveća kašnjenja	4446	578	42	50

Kao što se vidi iz slike 9.3., vrijeme odziva na događaje je znatno kraće za prilagođenu inačicu Linux operacijskog sustava i ne prelazi 50 [μs]. S druge strane, standardni Linux, iako u prosjeku događaje obrađuje s kašnjnjem od oko 50 [μs] ima značajan broj odziva čija su kašnjjenja za događajem veća i od nekoliko milisekundi!



Legenda: svaki znak "+" označava jedan događaj u običnom Linux sustavu dok znak "x" događaj u prilagođenom Linuxu (SRSV inačici); ordinata predstavlja vrijeme odziva, a apsisa redni broj događaja

Slika 9.3. Usporedba vremena odziva na Linux sustavima ([Clark, 2008])

Produljena kašnjenja u početku obrade, premda vrlo rijetka, za mnoge sustave nisu prihvatljiva. S druge strane, kod drugih sustava rijetka kašnjenja u obradi događaja ne moraju nužno označavati katastrofalnu grešku, već možda samo smanjenje kvalitete ili se njihov utjecaj može i zanemariti.

Ukoliko je zaista potrebno, neki elementi sustava moraju se izvoditi na nekom od operacijskih sustava za rad u stvarnom vremenu. Svojstva takvih sustava koji se danas mogu pronaći na tržištu su približno jednaka. Ono što se može razlikovati jest u podršci prema podržanom sklopoljju, alatima za razvoj programa, skupu podržanih protokola i standarda te podršci koju proizvođač pruža pri razvoju. Poželjno je odabrat sustave s podrškom za standardna POSIX sučelja za rad u stvarnom vremenu kako bi se isti programi u budućnosti mogli jednostavnije prenijeti na druge platforme.

9.3.2. Operacijski sustavi opće namjene

Operacijski sustavi opće namjene mogu biti dostatni za nekritične elemente sustava (npr. neki od Windows operacijskih sustava). Prednosti korištenja tih sustava su u dostupnosti svih teh-

nologija, protokola, standarda i alata za njih, kao i mogućnosti njihovog korištenja na svakom osobnom računalu. Odabir jednog od njih i njihova međusobna sukladnost i mogućnost suradnje pruža sigurnost u nastavak razvoja i održavanja (obzirom da su ti sustavi zastupljeni u velikoj većini današnjih računalnih sustava).

“Alternativno” rješenje može biti korištenje sustava zasnovanog na Linux operacijskom sustavu. Linux operacijske sustave u današnje vrijeme razvija dobrovoljna zajednica, ali često i uz pomoć većih tvrtki (primjerice Google). Iako su besplatni za korištenje, Linux sustavi su (barem) usporedivi s komercijalnim sustavima u pogledu učinkovitosti, dostupnosti programa i alata za razvoj. Kao i za Windows sustave i za Linux se može predvidjeti da će se u bližoj budućnosti nastaviti razvijati.

Iako značajno lošiji svojstava od operacijskih sustava za rad u stvarnom vremenu, operacijski sustavi opće namjene se ipak mogu primijeniti u nekim slučajevima. Naprotkom tehnologije današnja su računala vrlo brza, pa se nedostaci u nedeterminizmu ponekad mogu nadomjestiti brzinom rada (samo za sustave s blažim vremenskim ograničenjima).

9.3.3. Prilagođeni operacijski sustavi opće namjene

U području između operacijskih sustava za rad u stvarnom vremenu i onih opće namjene može se izdvojiti nekoliko značajnijih.

Real-Time Linux

Real-Time Linux je jezgra operacijskog sustava koja nastaje primjenom dodatka (engl. *patch*) CONFIG_PREEMPT_RT na izvorni kod Linux jezgre. Službeni naziv dodatak je dobio po svojoj osnovnoj funkciji – mogućnosti prekidanja potencijalno dugotrajnih jezgrinih funkcija (dugotrajnih s aspekta SRSV-a). Takve su dugotrajne funkcije osnovni razlog nemogućnosti korištenja obične Linux jezgre u SRSV-ima.

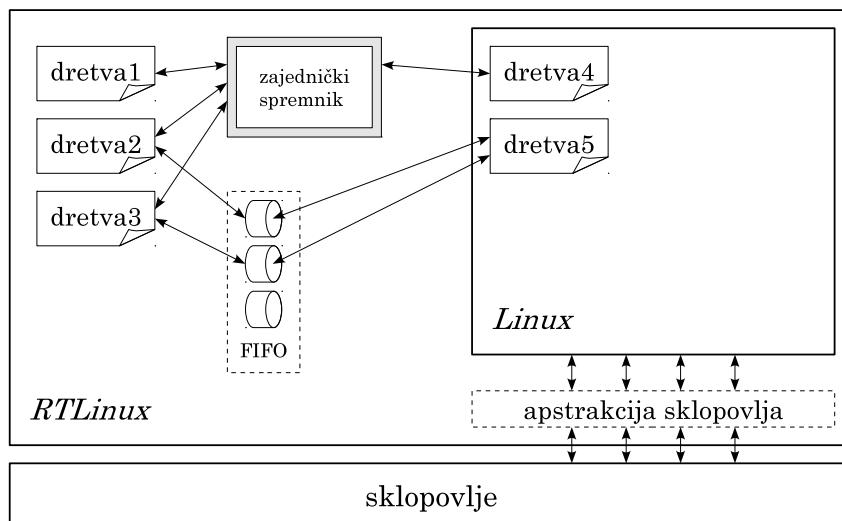
Na stranicama koje opisuju taj dodatak [Real-Time Linux] nalaze se i korisne preporuke kako koristiti Linux jezgru u SRSV okruženju (sa i bez tog dodatka) obzirom na svojstva nekih elemenata jezgre kao i Intelove arhitekture.

RTLinux

Sličnog imena ali drugčije ostvarenje koristi RTLinux. RTLinux ima svoju zasebnu jezgru koja izvodi kritične zadatke upravljanja (koji se pripremaju za tu jezgru). Linux je u tom sustavu samo jedan proces, i to proces s najmanjim prioritetom. Svi programi pripremljeni za jezgru imaju veći prioritet od Linux-a. Linux, i svi procesi u njemu vide RTLinux kao sklopovlje (RTLinux jezgra “maskira” sklopovlje pravom Linux-u). Komunikacija zadataka kojima upravlja RTLinux i zadataka kojima upravlja Linux može se odvijati preko zajedničkog spremnika te preko FIFO struktura. I jedni i drugi se u Linux-u vide kao uređaji, dok ih dretve u RTLinux-u koriste posebnim sučeljem. Navedeno prikazuje slika 9.4.

Osim navedena dva ostvarenja postoji i mnoštvo drugih, sličnih, primjerice: Lineo – Embedix Realtime, REDSonic – REDICE Linux, MonaVista Real Time Extensions, LynuxWorks – BlueCat RT, TimeSys – Linux/Real-Time+. Uobičajeno rješenje koje koristi većina navedenih sustava je korištenje vlastite jezgre u kojoj se sam Linux sustav izvodi kao zaseban proces.

Microsoft neke svoje opće operacijske sustave nudi i u posebnoj izvedbi u kojoj je moguće odrediti (pri postavljanju sustava) koji su elementi potrebni, a koji nisu. Na ovaj način se može napraviti znatno predvidljiviji sustav od standardnih operacijskih sustava izostavljajući u potpunosti nepotrebne komponente. Windows Embedded Standard 7 je jedan takav proizvod. On je potpuno sukladan s Win32 standardom, tj. svi programi koje rade na Windows platformama radit će i na ovom operacijskom sustavu. Standardni servisi i protokoli sadržani su i u ovim



Slika 9.4. RTLinux arhitektura

sustavima, kao i podrška za .NET tehnologiju.

9.3.4. Odabir operacijskih sustava

Analizom potreba za pojedine probleme treba odabrati prikladan operacijski sustav. Ukoliko su vremenska ograničenja vrlo stroga, za taj dio sustava potrebno je odabrati prikladan OS za SRSV. Za manje kritične komponente poželjno je ipak odabrati neki opći operacijski sustavi (Windows ili Linux) ili njihove modifikacije zbog veće dostupnosti razvojnih alata, podrške raznim tehnologijama i protokolima, kao i predviđenom životnom vijeku, njihovu daljem razvoju i podršci. Odabirom ovakvih operacijskih sustava s kontinuiranim razvojem i ogromnom korisničkom bazom maksimizira se rok podrške za operacijske sustave, pa tako i posredno i podršku za većinu novih tehnologija koje će se pojaviti u skoroj budućnosti, a koji bi mogli postati potrebni u procesu unaprjeđenja sustava.

Korištenjem standardnih tehnologija podržanih od strane više proizvođača omogućuje se modularna izgradnja sustava, koji po sastavu (sklopolski i programske) može biti i heterogen. Odluka o odabiru operacijskog sustava za pojedine elemente sustava u takvom slučaju ne mora biti konačna, već se on može promijeniti i naknadno bez značajnijeg dodatnog truda u prilagodbi aplikacija za taj sustav.

Konačno, ponekad ipak može biti potrebno izgraditi vlastiti sustav jer postojeći iz raznih razloga ne odgovaraju (cijenom, svojstvima, zahtjevima na sklopolje, ...).

Pitanja za vježbu 9

1. Koja je uloga operacijskog sustava u računalu? ◁
2. Koja svojstva ograničavaju uporabu operacijskog sustava u ugrađenim sustavima?
3. Opisati mehanizam prihvata prekida sa stanovišta operacija koje poduzima operacijski sustav te sa stanovišta njegova iskorištenja. Navesti prednosti i nedostatke različitih načina prihvata prekida. ◁
4. Koju podršku nude operacijski sustavi za upravljanje napravama mehanizmom prekida? Što ako bi obrada mogla duže potrajati?
5. Koja su svojstva upravljanja spremničkim prostorom bitna u kontekstu korištenja u SRSV-ima?
6. S obzirom na predviđenu uporabu opisati različite kategorije operacijskih sustava.
7. Po čemu se operacijski sustavi predviđeni za SRSV-e bitno razlikuju od ostalih operacijskih sustava? ◁
8. Navesti osnovna svojstva FreeRTOS-a u kontekstu ugrađenih sustava i SRSV-a.
9. Opisati operacijske sustave RTLinux te Real-Time Linux. Kako prvi i drugi koriste Linux?
10. Kada je potrebno koristiti posebne operacijske sustave za SRSV-e, a kada je moguće iskoristiti i obične operacijske sustave?
11. Pri korištenju običnih operacijskih sustava, što se sve može napraviti da se poboljšaju svojstva sustava (programa i operacijskog sustava) za primjenu u SRSV-ima?

Literatura

- [Budin, 2011] Leo Budin, Marin Golub, Domagoj Jakobović, Leonardo Jelenko-vić, *Operacijski sustavi*, Element, 2011.
- [Buttazzo, 2000] Giorgio C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*, Kluwer Academic Publishers, 2000.
- [Đonlagić, 1994] Dali Đonlagić, *Osnove projektiranja neizrazitih (fuzzy) regulacijskih sustava*, KoREMA, 1994.
- [Grantham, 1993] Walter J. Grantham, Thomas L. Vincent, *Modern control systems: analysis and design*, Wiley, 1993.
- [Jantsch, 2004] Axel Jantsch, *Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation*, Morgan Kaufmann Publishers, 2004.
- [Kovačić, 2006] Zdenko Kovačić, Stjepan Bogdan, *Fuzzy Controller Design: Theory and Application*, Taylor & Francis, 2006.
- [Krishna, 1997] C. M. Krishna, Kang G. Shin, *Real-time systems*, McGraw-Hill, 1997.
- [Labrosse, 2002] J. J. Labrosse, *MicroC/OS-II: The Real Time Kernel, 2nd edition*, CMP Books, 2002.
- [Laplante, 2012] Phillip A. Laplante, Seppo J. Ovaska, *Real-time systems design and analysis: Tools for the practitioner, 4th ed.*, IEEE Press, 2012.
- [Levi, 1990] Shem-Tov Levi, Ashok K. Agrawala, *Real-time system design*, McGraw-Hill, 1990.
- [Liu, 2000] Jane W. S. Liu, *Real-time systems*, Prentice Hall, 2000.
- [Nissanke, 1997] Nimal Nissanke, *Realtime systems*, Prentice Hall, 1997.
- [Rajkumar, 1991] R. Rajkumar, *Synchronization in real-time systems: A Priority Inheritance Approach*, Kluwer Academic Publishers, 1991.
- [Silberschatz, 2002] Abraham Silberschatz, Greg Gagne, Peter Baer Galvin, *Operating System Concepts, 6th edition*, Wiley, 2002.

-
- [Baruah, 2003] Sanjoy K. Baruah, Joel Goossens, *Rate-Monotonic Scheduling on Uniform Multiprocessors*, IEEE Trans. on Computers, Vol. 52, No. 7, July 2003.
- [Jain, 1994] Kamal Kumar Jain and V. Rajaraman, *Lower and Upper Bounds on Time for Multiprocessor Optimal Schedules*, IEEE Trans. on Parallel and Distributed Systems, Vol. 5, No. 8, August 1994.
- [Liu, 1973] C. L. Liu and James W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the ACM, Vol. 20, No. 1, January 1973.

- [Lopez, 2004] Jose M. Lopez, Jose L. Diaz, and Daniel F. Garcia, *Minimum and Maximum Utilization Bounds for Multiprocessor Rate Monotonic Scheduling*, IEEE Trans. on Parallel and Distributed Systems, Vol. 15, No. 7, July 2004.
- [Murphy, 2000] N. Murphy, *Watchdog Timers, Embedded systems Programming*, November 2000, p. 112.
- [Murphy, 2001] N. Murphy, M. Barr, *Watchdog Timers, Embedded systems Programming*, October 2001, pp. 79-80.
-

- [Budiselić, 2011] Ivan Budiselić, *Dinamičko programiranje i problemi raspoređivanja*, seminarски rad (doktorski studij), 2011,
<http://www.zemris.fer.hr/~leonardo/srsv/dodatno/Budiselic-Dinamicko-programiranje.pdf>.
- [Bogunović, 2012] Nikola Bogunović, *Oblikovanje programske potpore, materijali za predavanja*, FER2 preddiplomski studij,
<http://www.fer.unizg.hr/predmet/opp>.
- [Clark, 2008] Clark Williams, *An Overview of Realtime Linux*,
<http://people.redhat.com/bche/presentations/realtime-linux-summit08.pdf>.
- [Jelenković, 2010] Leonardo Jelenković, *Osnovni koncepti operacijskih sustava, prezentacija i primjeri*, 2010.,
http://www.zemris.fer.hr/~leonardo/os/Osnovni_koncepti_OSA.
- [Jones, 1997] M. Jones, *What really happened on Mars?*,
http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html.
- [Laurich, 2004] Peter Laurich, *A comparison of hard real-time Linux alternatives*, 2004,
<http://www2.hs-augsburg.de/informatik/vorlesungen/echtzeit/script/AT3479098230.html>.
-

- [Linux scheduling] *Linux Programmer's Manual: sched – overview of scheduling APIs*,
<http://man7.org/linux/man-pages/man7/sched.7.html>.
- [POSIX] *POSIX, 7. izdanje*, 2008,
<http://pubs.opengroup.org/onlinepubs/9699919799/>.
- [QNX, 6.3] *The QNX Neutrino Microkernel – Scheduling*, 2013.,
http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/sys_arch/kernel.html#SCHEDULING
- [Thread-Safety] *POSIX: Thread-Safety*, 2008,
http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_09_01.
-

- [Android] *Android (operacijski sustav)*,
<http://www.android.com>.
- [FreeRTOS] *FreeRTOS*,
<http://www.freertos.org/>.
- [iOS] *iOS (operacijski sustav)*,
<http://www.apple.com/ios/>.

- [Linux] *Linux operating system*,
<http://www.kernel.org>.
- [Neutrino] *QNX Neutrino Realtime Operating System*,
<http://www.qnx.com/products/neutrino-rtos/>.
- [OSE] *Enea OSE*,
<http://www.enea.com/solutions/rtos/ose/>.
- [RTLinux] Victor Yodaiken, *The RTLinux Manifesto*),
<http://www.yodaiken.com/papers/rtlmanifesto.pdf>.
- [Real-Time Linux] *Real-Time Linux (CONFIG_PREEMPT_RT)*,
<https://rt.wiki.kernel.org>.
- [VxWorks] *Windriver VxWorks real time system*,
<http://www.windriver.com/products/vxworks/>.
- [Windows EC] *Windows Embedded Compact 7 (Formerly CE)*,
<http://www.microsoft.com/windowsembedded/en-us/windows-embedded-compact-7.aspx>.
- [Windows ES] *Windows Embedded Standard*,
<http://www.microsoft.com/windowsembedded/en-us/windows-embedded-standard-7.aspx>.