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

Interni materijal za predavanja iz predmeta

Operacijski sustavi

Leonardo Jelenković

Zagreb, 2025.

Sadržaj

Prije uvoda **1**

1. UVOD **3**

1.1. Računalni sustav 3

1.2. Operacijski sustav 5

1.3. Podsustavi, slojevi (info) 5

1.4. Crtice iz povijesti operacijskih sustava (info) 6

1.5. O mjernim jedinicama 9

Pitanja za vježbu 9

2. MODEL JEDNOSTAVNOG RAČUNALA **11**

2.1. Sabirnički model računala 11

2.2. Kratki opis komponenata sabirničkog modela računala 15

2.3. Instrukcijska dretva 19

2.4. Višedretveni rad 21

Pitanja za vježbu 23

3. OBAVLJANJE ULAZNO-IZLAZNIH OPERACIJA, PREKIDNI RAD **25**

3.1. Spajanje naprava u računalo 25

3.2. Korištenje UI naprava radnim čekanjem 27

3.3. Prekidni rad 31

3.4. Korištenje sklopova s izravnim pristupom spremniku 42

3.5. Usporedba načina upravljanja UI napravama 44

3.6. Prekidi generirani unutar procesora, poziv jezgre 45

3.7. Višeprocesorski (sabirnički povezani) sustavi 45

3.8. Prekidi u “stvarnim sustavima”? (info) 46

3.9. “Upravljački programi” (engl. *device drivers*) (info) 47

Pitanja za vježbu 52

4. MEĐUSOBNO ISKLJUČIVANJE U VIŠEDRETVENIM SUSTAVIMA **53**

4.1. Osnovni pojmovi – program, proces, dretva 53

4.2. Višedretveno ostvarenje zadatka – zadatak i podzadaci 57

4.3. Model višedretvenosti, nezavisnost dretvi 58

4.4. Problem paralelnog korištenja zajedničkih varijabli (engl. *race condition*) 61
4.5. Međusobno isključivanje 62
4.6. Potraga za algoritmima međusobnog isključivanja 63
4.7. Sklopovska potpora međusobnom isključivanju 69
4.8. Problemi prikazanih mehanizama međusobnog isključivanja 72
Pitanja za vježbu 73

5. JEZGRA OPERACIJSKOG SUSTAVA 75
5.1. Strukture podataka jezgre 78
5.2. Jezgrine funkcije 79
5.3. Semafori 86
5.4. Izvedba jezgrinih funkcija za višeprocesorske sustave 93
5.5. Primjeri sinkronizacije binarnim semaforima 95
5.6. Operacije stvaranja dretvi i semafora (info) 96
5.7. Ostali mehanizmi jezgre (info) 97
5.8. Brži sinkronizacijski mehanizmi (info) 98
Pitanja za vježbu 101

6. MEĐUDRETVENA KOMUNIKACIJA I KONCEPCIJA MONITORA 103
6.1. Primjer sinkronizacije semaforima: proizvođač i potrošač 103
6.2. Problemi sa semaforima 107
6.3. Monitori 111
6.4. Primjeri sinkronizacije semaforima i monitorima 114
6.5. Dodatno o sinkronizaciji (info) 121
Pitanja za vježbu 122

7. ANALIZA VREMENSKIH SVOJSTAVA 123
7.1. Deterministički sustav 123
7.2. Nedeterministički sustav 128
7.3. Poissonova razdioba (izvod iz binomne) 133
7.4. Eksponencijalna razdioba 135
7.5. Posluživanje s Poissonovom i eksponencijalnom razdiobom 140
7.6. Osnovni načini dodjeljivanja procesora dretvama – raspoređivanje dretvi 144
7.7. Raspoređivanje dretvi u operacijskim sustavima 149
Pitanja za vježbu 157

8. UPRAVLJANJE SPREMNIČKIM PROSTOROM 159

8.1. Uvod 159
8.2. Statičko upravljanje spremnikom 164
8.3. Dinamičko upravljanje spremnikom 165
8.4. Straničenje 170
8.5. Detaljnije o dinamičkom upravljanju spremnikom (info) 190
Pitanja za vježbu 194

9. DATOTEČNI SUSTAV **195**
9.1. Diskovi 195
9.2. Datotečni sustav 205
9.3. Primjeri datotečnih sustava 208
9.4. Datotečni podsustav operacijskog sustava 216
9.5. Primjer slojevitog datotečnog podsustava (info) 218
9.6. Uloga međuspremnika u povećanju učinkovitosti (sažetak) 220
Pitanja za vježbu 221

10. VIRTUALIZACIJA **223**
10.1. Uvod 223
10.2. Oblici virtualizacije i slični mehanizmi 224
10.3. Procesi u operacijskom sustavu 224
10.4. Virtualizacija na razini aplikacije 225
10.5. Virtualno okruženje za skup procesa – spremnici 226
10.6. Tipovi (prave) virtualizacije 227
10.7. Korištenje različitih oblika virtualizacije istovremeno 229
10.8. Problemi ostvarenja virtualizacije 229
Pitanja za vježbu 231

Literatura **233**

Prije uvoda

Materijali ove skripte su dijelom sažeci (natuknice) sadržaja knjige *Operacijski sustavi* autora *Lea Budina i ostalih* te prate isti slijed izlaganja. Neke teme su prikazane malo drukčije, neke proširene, neke izostavljene, uz uglavnom druge primjere zadataka.

U okviru ovog predmeta razmatraju se mehanizmi operacijskog sustava – algoritmi, postupci (i slično) koji se koriste da bi se ostvarila neka operacija te kako (kojim sučeljem) se ona može iskoristiti iz programa. Uglavnom se ne razmatra korištenje operacijskog sustava od strane korisnika za neke (uobičajene) operacije (npr. kako pronaći neku datoteku, kako promijeniti neku postavku) – pretpostavlja se da slušači ovog predmeta to već znaju ili mogu samostalno savladati. Izuzetak je ovaj kratki uvod u kojem se u vrlo kratkim crticama opisuje funkcionalno ponašanje računala.

Podaci i programi pohranjeni su:

- na disku: HDD, SSD, USB ključić, CD/DVD i sl.
  - takozvani perzistentni spremnici (trajni, čuvaju podatke i kad se računalo ugasi)
  - na njih su pohranjeni: operacijski sustav, programi, podaci, multimedija, ...
- u radnom spremniku (memoriji, RAM)
  - dijelovi OS-a, pokrenuti program i njihovi podaci
  - koristi se samo kada računalo radi (nije ugašeno)
  - kad se računalo ugasi ti se podaci gube (ako nisu pohranjeni na disk)
- u ROM-u – memoriji vrlo mala kapaciteta, koja se samo koristi prilikom pokretanja računala
  - BIOS/UEFI za osobna računala, radne stanice i poslužitelje
    - BIOS – *Basic Input/Output System* (starija računala)
    - UEFI – *Unified Extensible Firmware Interface* (novija računala)
  - "program pokretač" (engl. *bootloader*) kod drugih sustava (ugrađenih)

Što se dogodi kad se pokrene računalo?

1. Najprije se pokreću programi iz ROM-a (BIOS/UEFI) koji započinju s inicijalizacijom sklopovlja (procesora, memorije, ...)
2. Nakon toga program iz BIOS/UEFI s diska učitava drugi program pokretač (engl. *bootloader*) koji preuzima upravljanje te učitava operacijski sustav (ili taj program učita drugi koji to napravi do kraja)
3. Operacijski sustav učitava i priprema svoju jezgru (upravljanje procesima, datotečnim podsustavom, mrežnim podsustavom, korisničko sučelje, ...).
4. Pokreću se dodatne usluge (servisi – procesi u "pozadini") koji su dio ili proširenje operacijskog sustava kao što su sigurnosna stijena (engl. *firewall*), antivirusna zaštita, ažuriranje sustava, upravljanje elementima sustava, ostale usluge (vođenje dnevnika o događajima u sustavu, praćenje promjena priključenih naprava, ...).
5. Pokreću se usluge koje su dio korisničkog proširenja sustava kao što su:

- programi za prilagodbu ponašanja dijelova sustava (zaslon, zvuk, posebne naprave, provjera ažuriranja upravljačkih programa, ...)
- pohrana podataka u oblaku (npr. Dropbox, OneDrive, Google Drive)
- komunikacijski alati (npr. Skype, Viber, WhatsApp, . . .)
- ostale usluge

Korištenje računala od strane korisnika

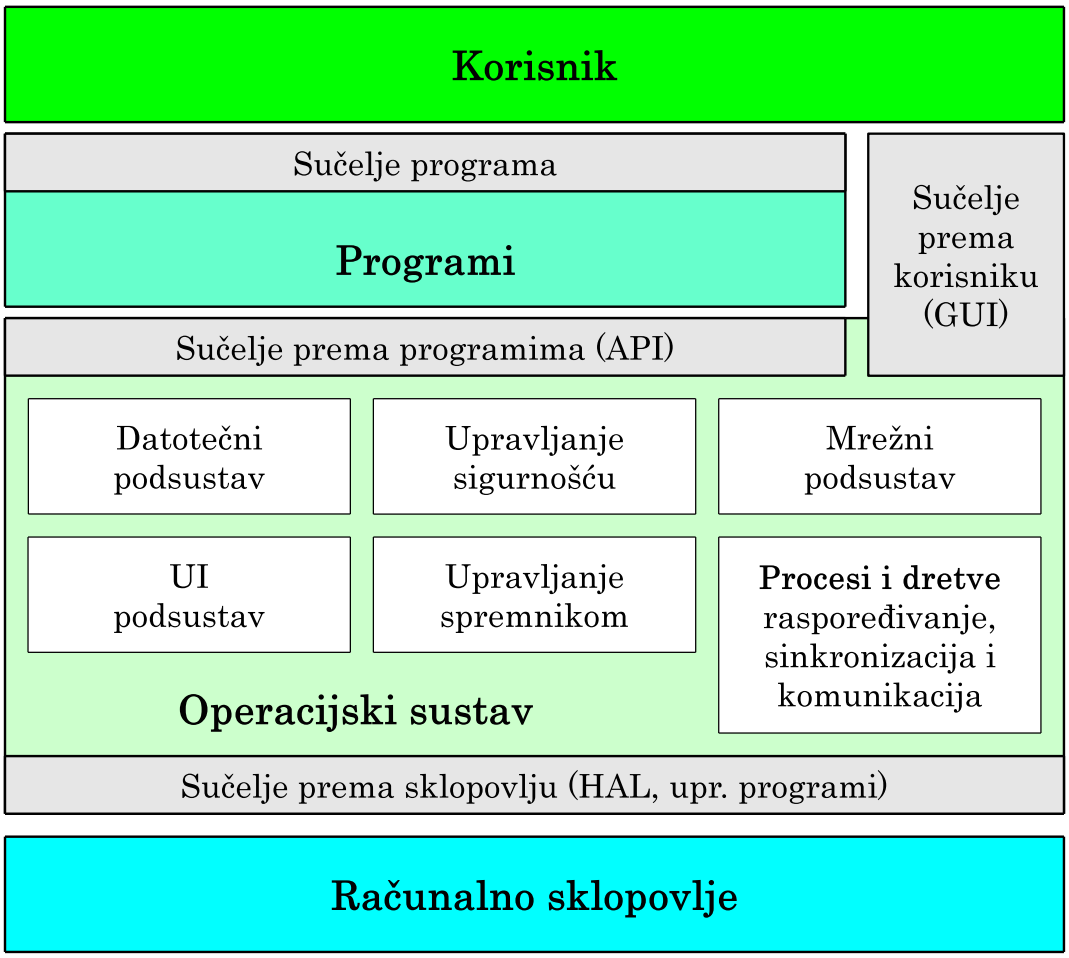
1. Korisnik preko sučelja operacijskog sustava (najčešće grafičkog):
   - pokreće programe (koji time postaju procesi)
   - zaustavlja procese
   - zadaje naredbe operacijskom sustavu (gašenje, stavljanje u stanje pripravnosti, promjena postavki izgleda i potrošnje i slično)
   - dodaje (*instalira*) nove programe, miče programe koji više nisu potrebni
   - ...
2. Svaki program preko svojeg sučelja omogućava korisniku rad s tim programom.
   - radi jednostavnosti korištenja uobičajeno program nude ista/slična sučelja za obavljanje istih operacija (npr. na jednaki način zadaju otvaranje datoteke)

1. UVOD

1.1. Računalni sustav

U današnje doba postoji mnoštvo različitih računala i sustava temeljenih na njima kao što su osobna i prijenosna računala, radne stanice, poslužitelji, pametni telefoni i slični uređaji, mikroupravljači. Operacijski sustavi (OS) koji se u njima koriste se također međusobno razlikuju. Međutim, temeljni elementi svih sustava se zasnivaju na istim osnovnim načelima. Iako je sustav koji se ovdje razmatra najbliži osobnom računalu, većina navedenog vrijedi i za ostale.

Računalni sustav se sastoji od sklopovlja (*hardvera* – računala), programske potpore (operacijski sustav i programi) te korisnika, kako je prikazano na slici 1.1.



Slika 1.1. Računalni sustav, komponente (podsustavi) OS-a

Korisnici koriste sustav radi svojih potreba, npr. obrada teksta, rad na Internetu, igre, multimedija i slično.

Programi su napisani da nude neke korisne operacije korisnicima. Obične operacije izvode se izravnim izvođenjem instrukcija programa na procesoru. Operacije koje zahtjevaju korištenje sklopovlja ili drugih zajedničkih sredstava izvode se tako da program pozove funkciju operacijskog sustava te ju OS obavi za program.

Operacijski sustav upravlja sustavom prema zahtjevima korisnika i programa. Omogućuje korisnicima i programima jednostavno korištenje sredstava sustava kroz odgovarajuća sučelja. Korisnici i programi stoga ne moraju poznavati (složene) detalje sklopovlja koje se koristi radi izvedbe njihovih naredbi.

Među **slojevima** se nalazi **sučelje**

- sučelje definira način korištenja/komunikacije među slojevima

Podsustavi OS-a imaju svoju ulogu, ali nisu posve razdvojeni kao na slici 1.1.

Korištenje podsustava mogli bi razmotriti i s povijesnog aspekta (info)

Prva računala su se programirala "izravno" preko sklopki. Kasnije su uvedene kartice gdje svaka kartica predstavalja jednu instrukciju i na taj se način učitavao program (preko čitača kartica). Poslije su uvedeni terminali preko kojih su se programi unosili korištenjem tipkovnice. Podsustavi za upravljanje ulazno-izlaznim napravama i spremnikom su bila prva dva podsustava koja su se vremenom nadograđivala.

Korištenje računala od strane samo jednog korisnika (kako je bilo u početku) je poprilično neučinkovito: dok on unosi program, i kasnije gleda rezultate sustav je neiskorišten. Stoga je jedno od idućih poboljšanja bilo omogućavanje višekorisničkog rada. Za to, bilo je potrebno dodati upravljanje procesima. Tada je više korisnika koristilo sustav, naizgled istovremeno, iako zapravo naizmjence – procesorsko vrijeme podijeljeno je u male kvante vremena koji su kružno dodijeljeni svim procesima.

Dodavanje diska ili trake ili sličnih mehanizama za pohranu podataka omogućilo je jednostavnije stvaranje programa, njihovo pokretanje, zapisivanje ulaznih podataka kao i rezultata.

Pojavom prvih mreža pojavila se potreba za podsustavom za upravljanje mrežom te sigurnošću.

Početno vrlo teško za korištenje i od strane vrlo stručnih ljudi/inženjera/programera, računalo je korištenjem operacijskog sustava postalo jednostavno za korištenje i programima i običnim korisnicima, koji pokreću programe i neke akcije operacijskog sustava. Jaz između programa, koji su u početku uključivali i upravljačke programe, i sklopovlja je premošten operacijskim sustavom – on je složenost preuzeo na sebe.

Npr. program za obradu i ispis teksta ne mora znati koji se pisač koristi. On će samo pripremiti podatke za ispis, a operacijski sustav će to preuzeti i po potrebi prilagoditi i prosljediti do pisača korištenjem i upravljačkih programa pisača, ali i kontrolom ispisa – da se ispis ne izmiješa s ispisom drugih programa.

1.2. Operacijski sustav

DEFINICIJA: *Operacijski sustav* je skup osnovnih programa koji:

- omogućuju izvođenje radnih zahvata na računalu
- omogućuju izvođenje operacija računala

Svrha/uloga OS-a

- olakšavanje korištenja računala (skriva detalje)
- omogućava učinkovito korištenje svih dijelova računala (ima upravljačke programe)
- omogućava višeprogramski rad – najbitnija uloga, povećava učinkovitost sustava

"**OS olakšava korištenje računala**"

- skriva nepotrebne detalje od korisnika i programa
  - korisnici koriste korisničko sučelje koje nude programi i OS
  - programi koriste API koje nudi OS
- upravljački programi ("drajveri") znaju kako sa sklopovljem
  - OS ih koristi
  - jedan upravljački program radi samo za pojedinu komponentu računala
  - na različitim računalima koriste se različiti upravljački programi
- “prenosivost” – sklopovski različita računala ali s istim OS-om se na jednaki način koriste!
  - isto korisničko sučelje
  - isto sučelje prema programima (API)
  - OS je skoro identičan, samo se koriste drugi upravljački programi

1.3. Podsustavi, slojevi (info)

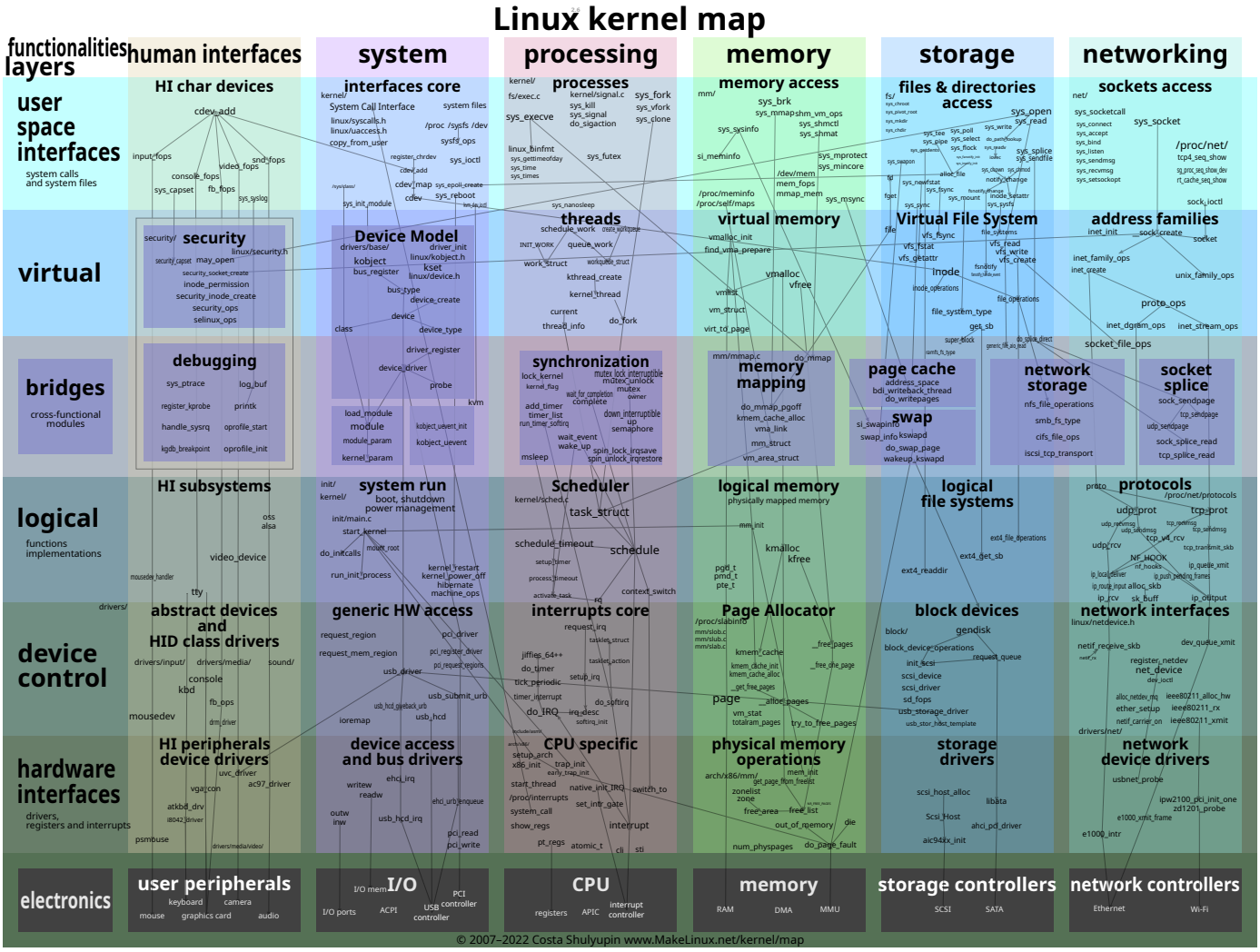
Stvarni operacijski sustavi su jako složeni sustavi. Zato se interno dijele na slojeve i podsustave.

Primjer podjele na slojeve:

- sustavske funkcije i sučelje prema programima (API)
- jezgra operacijskog sustava (s većinom podsustava)
- apstrakcija sklopovlja (HAL – hardware abstracion layer)

Arhitektura postojećih operacijskih sustava uglavnom prati slojevitost prikazanu na slici 1.1. Međutim, obzirom da su operacijski sustavi značajno složeniji, i sami slojevi su često podijeljeni na manje cjeline i podslojeve.

Složenost operacijskih sustava možda najbolje prikazuje nešto detaljnija slika arhitekture operacijska sustava Linux, prikazanog na slici 1.2. U okviru ova predmeta razmatraju se najosnovnije operacije te se neće koristiti podjela na slojeve.



Slika 1.2. Arhitektura jezgre Linuxa (izvor: https://makelinux.github.io/kernel/map/)

1.4. Crtice iz povijesti operacijskih sustava (info)

Povijest UNIX-a u crticama

- (1960-te) "najjače računalo" IBM 709
  - bušene kartice na početku (FORTRAN programi)
  - MIT razvija Compatible Time-Sharing System (CTSS)
- 1964. MIT+Bell Labs + GE:
  - MULTiplexed Information and Computing Service (MULTICS)
  - prezahtjevan u startu (>288 kB)
  - 1973. komercijalno dostupan (radio do 2000.)
- 1969. Bell Labs, Thompson, PDP-7:
  - UNiplexed Information and Computing Service (UNICS=>UNIX)
  - u asembleru!
- 1972. Bell Labs, Thompson, Ritchie, PDP-11:
  - UNIX je "prepisan" u C (na PDP-11)
  - 1975. UNIX v6 prodavan sveučilištima za 300 $

  - s obzirom na to da je PDP-11 bio popularan, UNIX se brzo proširio
- 1977. Berkeley Software Distribution (BSD)
  - temeljen na UNIX v6
  - AT&T je kasnije tužio BSD jer koristi dijelove njegova koda što je ograničilo razvoj BSD-a
  - dalji razvoj u okviru: FreeBSD, OpenBSD, NetBSD,
    Darwin (Darwin je podloga za Appleove OS-ove: OS X, iOS, watchOS i tvOS)
- 1977. John Lions, University of New South Wales, Sydney:
  - komentirao UNIX kod, liniju po liniju
  - mnoštvo fakulteta preuzelo te materijale i učilo svoje studente o UNIX-u i operacijskom sustavu po tome
- 1979. UNIX V7
  - AT&T (vlasnik Bell Labsa) dodao licencu koja zabranjuje korištenje koda u knjigama
  - od tada studenti uče "teoretski" o operacijskim sustavima (ne na primjeru koda)
- 1983. Richard Stallman, GNU projekt: ideja je napraviti besplatan OS sličan UNIX-u te ostale potrebne programe, sve besplatno s dostupnim izvornim kodom
  - napisao GNU General Public License (GPL)
  - rad na kernelu (Hurd) je slabo napredovao, sam OS nije bio dovoljno popularan da privuče više ljudi na projekt, ali su zato ostali GNU alati napredovali brže i proširili se na gotovo sve UNIX sustave
- 1988. POSIX
  - *The Portable Operating System Interface (for UNIX)* (opis na wikipediji)
  - službeni standard (najnoviji) POSIX.1-2024 == IEEE Std 1003.1-2024
  - standardizacija sučelja za UNIX operacijske sustave
  - problem: tada je postojalo (bar) nekoliko inačica UNIX-a, svaki sa svojim dodatnim sučeljima; ovim projektom se to htjelo zaustaviti i omogućiti da se isti program u C-u može prevesti na svim UNIX sustavima
  - sučelja (“man” stranice): https://pubs.opengroup.org/onlinepubs/9799919799/
- 1987., Tanenbaum, MINIX
  - ponukan nedostatkom koda nastao je i MINIX, ali i mnogi drugi UNIX klonovi i radi korištenja u nastavi
  - MINIX – 16-bitovni OS za PC; početku zatvoren, plaćao se, kasnije se otvorio
- 1991., Linus Torvalds
  - u početku koristi MINIX, ali kasnije započinje sa svojim OS-om
  - glavni motivi: nedostatak besplatnog UNIX-a s dostupnim kodom (koji se može mijenjati i nadograđivati)
  - inačica 0.01 izdana 1991.
  - od 0.99 (1992.) koristi GPL (kasnije GPLv2, ali ne i GPLv3)
  - dobro prihvaćen od strane entuzijasta i ostalih (industrije)

  - distribucije operacijskih sustava temeljene na jezgri Linuxa:
    * Ubuntu, Linux Mint, Debian, openSUSE, ...
  - Android koristi jezgru Linuxa, dijelom prilagođenu
  - zadnja inačica (19. 1. 2025.) jezgre Linuxa je 6.13.

Apple operacijski sustavi

- 1984. Mac System Software (puno kasnije nazvan Mac OS, danas macOS)
  - prvi operacijski sustav s grafičkim sučeljem kao osnovnim sučeljem
- nastavak razvoja sve do danas
- zadnja inačica (2024.) macOS 15 Sequoia

Microsoft operacijski sustavi

- 1981. MS-DOS
  - nastao na temelju CP/M (koji je radio na procesorima 8080/Z80)
  - razne inačice, zadnja "samostalna" MS-DOS 6.22
- Windows 1.0 (1985.) - Windows 3.11 (1993.) 16-bitovni OS
- daljnji razvoj je podijeljen na "kućne" i "poslovni" korisnike
- "kućni korisnici": Windows 95, 98, ME
- "poslovni korisnici": Windows NT (3.1, 3.51, 4.0), 2000
- Windows XP (2001.) objedinjuje obje kategorije (uz Home i Professional inačice)
- poslužitelji: Windows NT Server, 2000 Server, 2003, 2008, 2012
- novije verzije: Windows Vista (2007.), Windows 7 (2009.), Windows 8 (2012.), Windows 8.1 (2013.), Windows 10 (2015.), Windows 11 (2021. početna, 24H2 zadnja inačica)

Operacijski sustavi za mobilne uređaje

- Android: pojava 2008; zadnja inačica 15 (2024.)
- iOS: pojava iPhone OS 1 2007.; zadnja inačica iOS 18.3.1. (2025.)

Operacijski sustavi za kritične sustave

- VxWorks (od 1987.) (NASA ga koristi u nekim sustavima, npr. Mars Curiosity)
- QNX (od 1982.)
- FreeRTOS (od 2002.)

Jedna zanimljiva ilustracija udjela operacijskih sustava na osobnim računalima kroz godine može se pronaći u videu.

1.5. O mjernim jedinicama

S obzirom na to da računalo radi u binarnom sustavu, sklopovlje je podređeno takvom načinu rada. To uključuje i organizaciju spremnika kao i strukture podataka. Baza binarnog sustava jest dva te su uobičajene jedinice oktet/bajt B $= 2^3$ bitova, i veće KB $= 2^{10}$ B $= 1024$ B, MB $= 2^{20}$ B $= 2^{10}$ KB, GB $= 2^{30}$ B $= 2^{10}$ MB, itd.

Navedene jedinice su u suprotnosti sa SI sustavom koji koristi potencije broja 10 (k $= 10^3$, M $= 10^6$, G $= 10^9$ ...).

Stoga su smišljene nove oznake s dodatkom slova 'i': KiB $= 2^{10}$ B, MiB $= 2^{20}$ B, ...

Međutim, vrlo često i u operacijskim sustavima i u literaturi se i dalje koriste "stare" oznake (bez 'i'), ali podrazumijevajući potencije od 2 a ne 10. Isto je i s ovom skriptom. Izuzetak su brzine prijenosa, gdje je uobičajeno (i u računalnom okruženju) koristiti potencije broja 10.

Kada se te jedinice promatraju u nekom okruženju prvo treba ustanoviti koje jedinice se tamo koriste. Npr. na "Windows" operacijskim sustavima koriste se navedene jedinice u potencijama broja 2 (KB, MB, ...), dok na Linux operacijskim sustavima koriste SI jedinice s potencijama broja 10 (kB, MB, ...). Proizvođači diskova koriste SI sustav jer u tom sustavu se manja vrijednost "prikazuje" većom oznakom. Npr. ako proizvođač kaže da disk ima kapacitet od 1 TB, on misli na 10^{12} B, što je $10^{12}/2^{40}$ TiB $= 0,909$ TiB, odnosno, $10^{12}/2^{30}$ GiB $= 931,3$ GiB.

Primjeri (kako interpretirati brojke u ovoj skripti):

- 5 KB $= 5 \cdot 2^{10}$ B $= 5 \cdot 1024$ B
- 5 MB $= 5 \cdot 2^{20}$ B $= 5 \cdot 1024 \cdot 1024$ B
- 5 GB $= 5 \cdot 2^{30}$ B $= 5 \cdot 1024 \cdot 1024 \cdot 1024$ B
- 5 kbita/s $= 5 \cdot 10^3$ bita/s $= 5 \cdot 1000$ bita/s
- 5 Mbita/s $= 5 \cdot 10^6$ bita/s $= 5 \cdot 1000000$ bita/s

Pitanja za vježbu 1

1. Što je to "računalni sustav"? Od čega se sastoji?
2. Što je to "operacijski sustav"? Koja je njegova uloga u računalnom sustavu?
3. Što je to "sučelje"? Koja sučelja susrećemo u računalnom sustavu?
4. Navesti osnovne elemente (podsustave) operacijskog sustava.

2. MODEL JEDNOSTAVNOG RAČUNALA

- Dijelovi računala prema modelima:
  - funkcijski model: ulazni dio, izlazni dio, (radni/glavni) spremnik (memorija), aritmetičko-logička jedinka, upravljačka jedinka
  - sabirnički model: procesor, sabirnica, spremnik, UI pristupni sklopovi
- Teorijski modeli računala – korištenje spremnika
  - Von Neumannov model – instrukcije i podaci u istom spremniku (dohvaćaju se preko zajedničke sabirnice
    - radni spremnik današnjih računala uglavnom spada u ovaj model
  - Harvardska arhitektura – instrukcije odvojene od podataka i dohvaćaju se različitim sabirnicama
    - dio priručnog spremnika procesora (L1) je uglavnom izveden Harvardskom arhitekurom (podijeljen je na dio za instrukcije i dio za podatke).

2.1. Sabirnički model računala

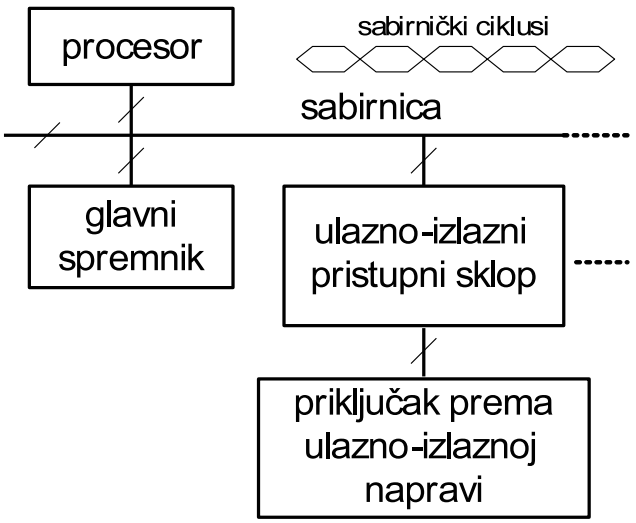
U sabirničkom modelu, računalo se (iznutra) sastoji od procesora, spremnika, ulazno-izlaznih naprava te sabirnice.

Procesor je osnovni element sustava, on izvodi instrukciju za instrukcijom i time omogućava provođenje korisniku potrebnih operacija.

Spremnik (radni spremnik, memorija) služi za privremenu pohranu instrukcija, podataka i rezultata. Procesor neprestano dohvaća i pohranjuje podatke iz spremnika pri svom radu. Stoga je spremnik, nakon procesora, najbitniji element sustava (i najbrži, nakon procesora).

Ulazno-izlazne naprave koriste međusklop (pristupni sklop, kontroler) za povezivanje u računalo. Više o napravama u idućem poglavlju.

Sabirnica omogućuje povezivanje elemenata računala – u sabirničkom modelu svi se elementi spajaju na sabirnicu i koriste ju za međusobnu komunikaciju.



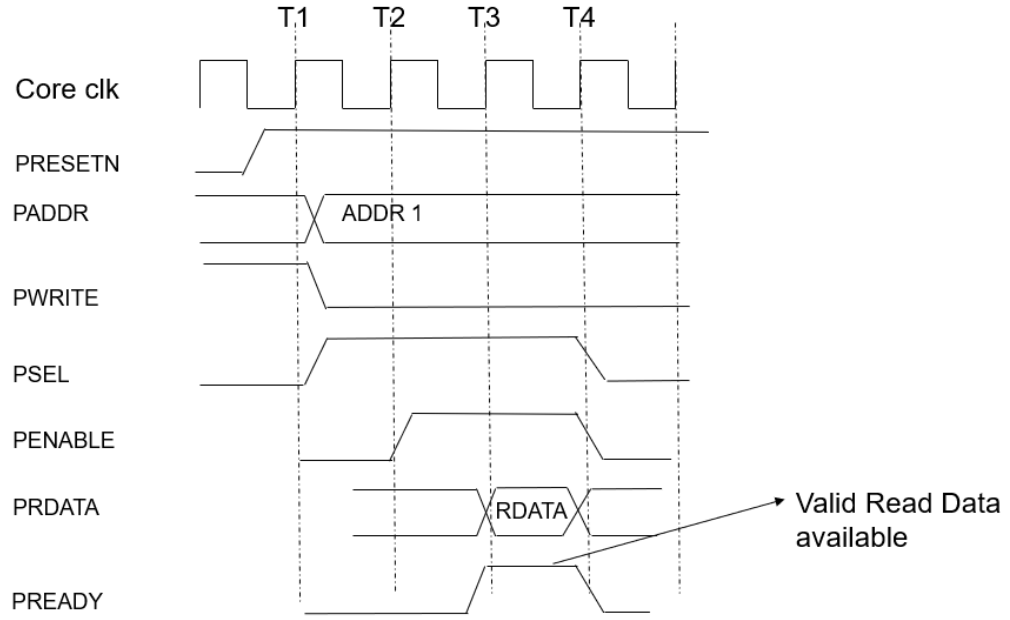
Slika 2.1. Sabirnički model računala

Sabirnički ciklus

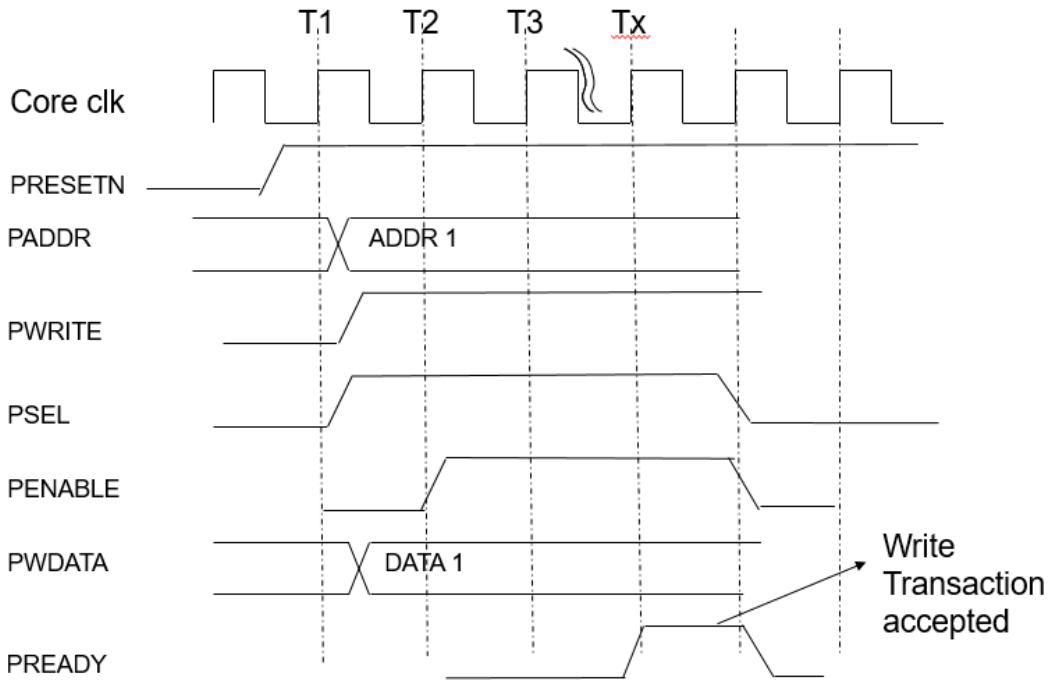
- prijenos jednog podatka između dva sklopa (procesor – spremnik, procesor – pristupni sklop, pristupni sklop – spremnik)

Sabirnicom upravlja procesor

1. postavlja adresu na adresni dio sabirnice
2. postavlja podatak (kada se on zapisuje u spremnik ili UI) na podatkovni dio
3. postavlja upravljačke signale



Slika 2.2. Primjer stanja sabirnice pri čitanju



Slika 2.3. Primjer stanja sabirnice pri pisanju

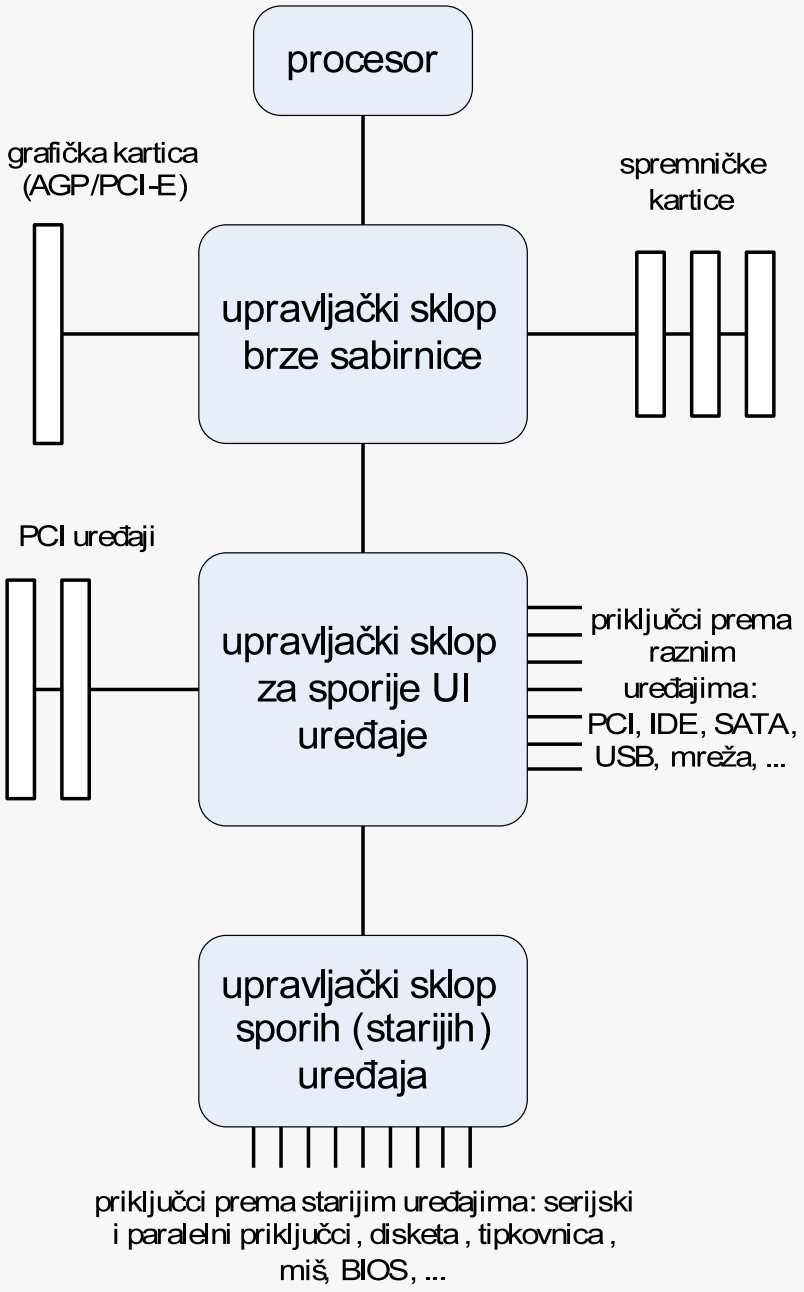
Primjer 2.1. Trajanje sabirničkog ciklusa

Neka je trajanje jednog sabirničkog ciklusa $T_B = 1$ ns, tada:

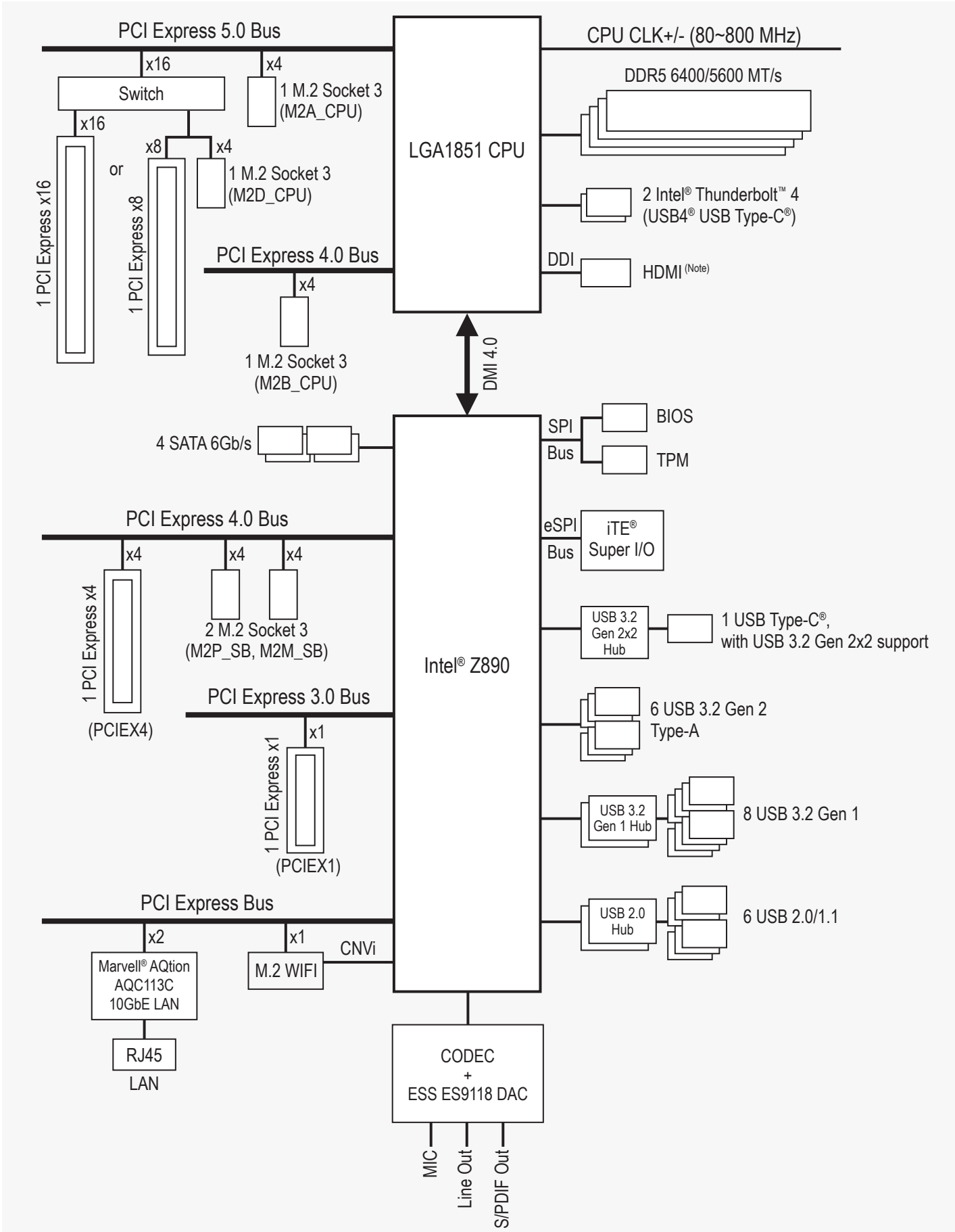
- frekvencija rada sabirnice jest $\frac{1}{T_B} = 1$ GHz
- ako je širina sabirnice 64 bita tada je propusnost sabirnice: $64 \cdot 1$ GHz $= 64$ Gbita/s (G je u ovom slučaju (za brzine) 10^9 a ne 2^{30})

Primjer 2.2. Hijerarhijsko povezivanje sabirnica (info)

Zbog različitih brzina komponenata računalnog sustava, stvarne arhitekture imaju više sabirnica kojima su one povezane.



Slika 2.4. Primjer modela računala

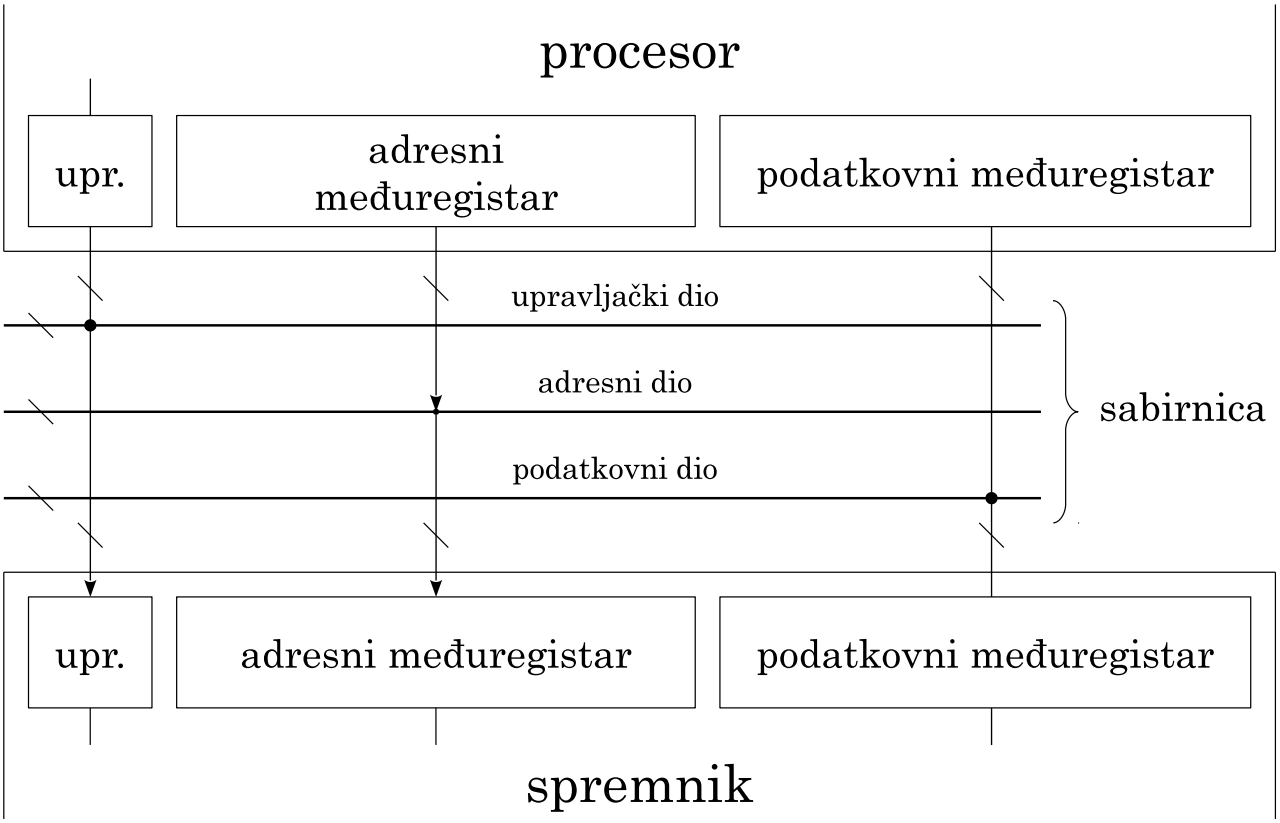


Slika 2.5. Primjer arhitekture matične ploče Z890 AORUS MASTER (izvor: GIGABYTE)

2.2. Kratki opis komponenata sabirničkog modela računala

U ovom poglavlju su razmotrene samo osnovne komponente: procesor, spremnik i sabirnicu.

2.2.1. Sabirnica

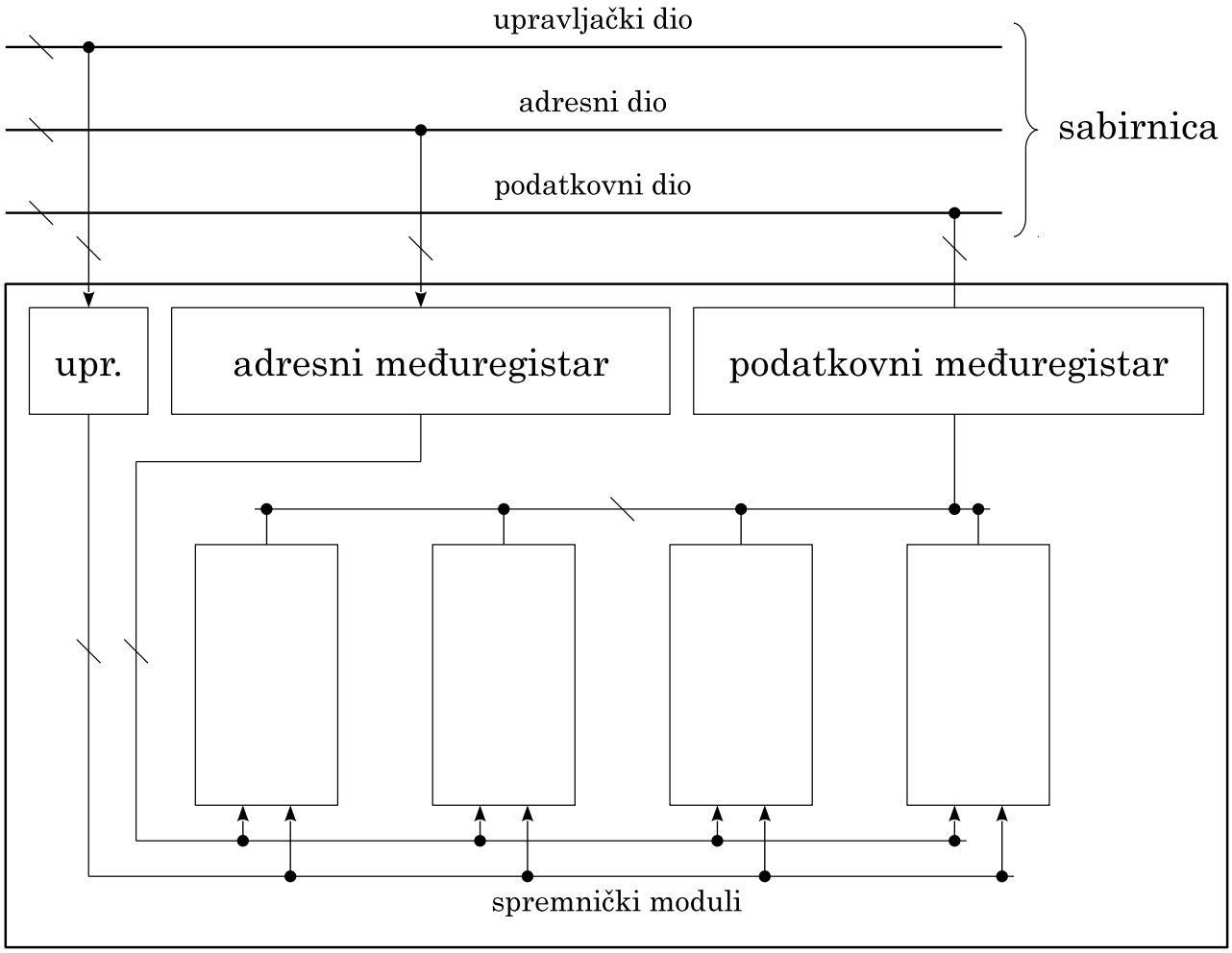


Slika 2.6. Model sabirnice

Sabirnica se sastoji od tri dijela:

- adresnog dijela
- podatkovnog dijela
- upravljačkog dijela (npr. signali piši ili čitaj, BREQ, BACK, prekidi, . . .)

2.2.2. Spremnik



Slika 2.7. Model spremnika

O brzinama rada spremnika (info)

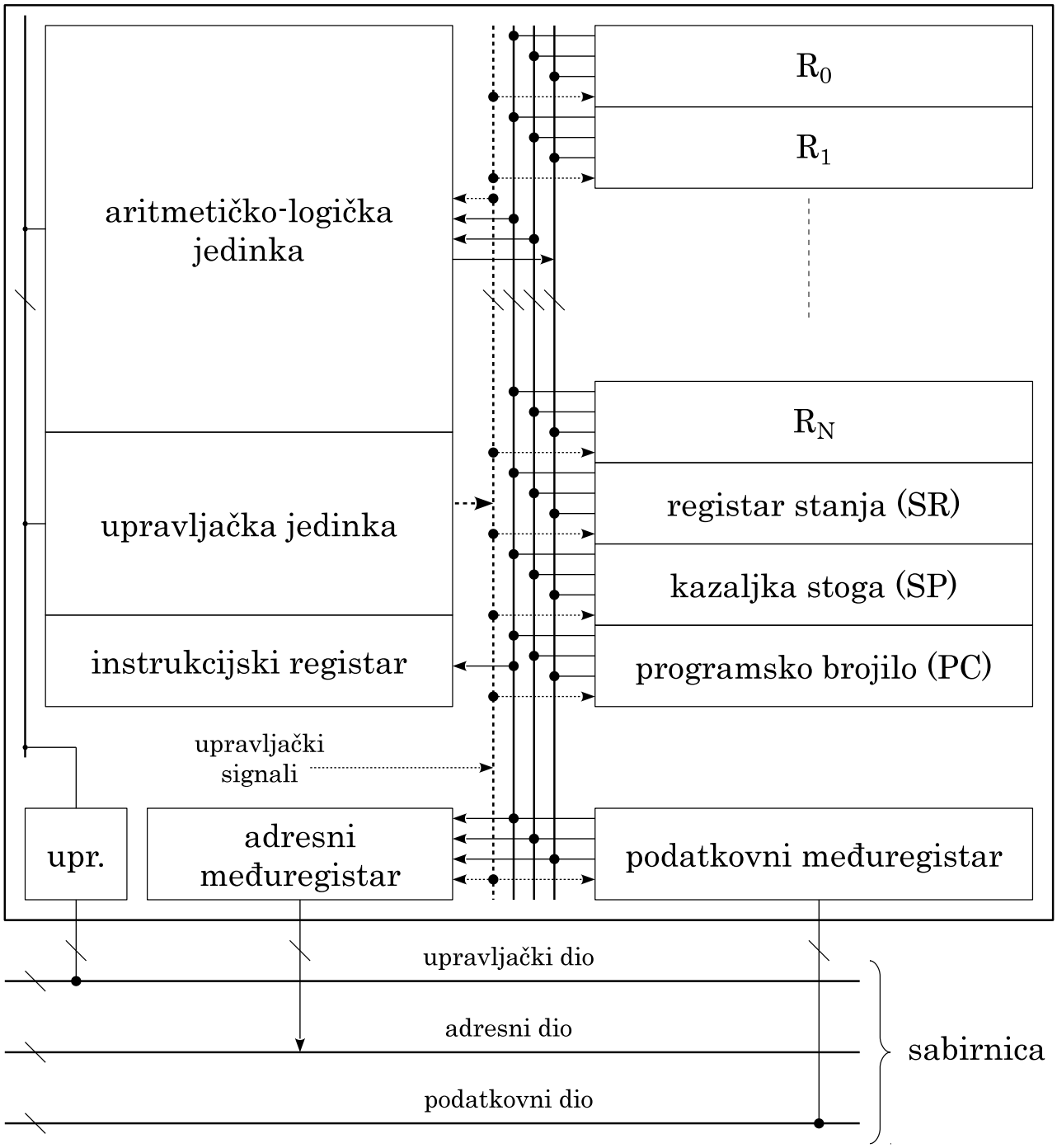
- brzina prema sabirnici (npr. 2133 MHz)
- interna brzina rada na samoj kartici (pojedini moduli) (npr. 200 MHz)
- današnji spremici su optimirani za prijenos veće količine podataka (bloka)
  - zato jer procesori imaju 'velike' interne spremnike – priručne spremnike
- tablica 2.1. prikazuje neka svojstva različitih tipova spremnika

Tablica 2.1. Usporedba propusnosti (info)

Names	Memory clock	I/O bus clock	Transfer Rate	Channel bandwidth
DDR3-1600	200 MHz	800 MHz	1600 MT/s	12.8 GB/s
DDR4-2400	300 MHz	1200 MHz	2400 MT/s	19.2 GB/s
DDR4-3200	400 MHz	1600 MHz	3200 MT/s	25.6 GB/s
DDR5-4800	300 MHz	2400 MHz	4800 MT/s	38.4 GB/s
DDR5-6400	400 MHz	3200 MHz	6400 MT/s	51.2 GB/s

GT/s: 10^9 prijenosa po sekundi: svaki put 10 bita, 8 korisnih; GBps: 10^9 okteta po sekundi
Izvor: https://en.wikipedia.org/wiki/Double_data_rate

2.2.3. Procesor



Slika 2.8. Model procesora

Procesor se sastoji od elemenata:

- aritmetičko-logička (AL) jedinka
- registri opće namjene (npr. R0-R7)
- programsko brojilo PC
- kazaljka stoga SP
- registar stanja SR
- *upravljačka jedinka*
  - *instrukcijski registar*
  - *upravljački signali*
- *adresni međuregistar*
- *podatkovni međuregistar*

Koso označeni elementi nisu izravno dostupni programeru. Ostali elementi jesu i čine "programerski model" procesora.

Procesor se može opisati kao automat koji ciklički obavlja sljedeće operacije:

Isječak kôda 2.1. Opis rada procesora

```
ponavljaj {
    dohvati instrukciju na koju pokazuje PC
    povećaj PC tako da pokazuje na iduću instrukciju
    dekodiraj instrukciju
    obavi operaciju zadanu instrukcijskim kodom
        ( ovisi o instrukciji, npr.\ za AL instrukciju može biti:
          dohvati operande, obavi AL, spremi rezultat )
}
dok je procesor uključen
```

Instrukcije se mogu podijeliti na instrukcije za:

- premještanje sadržaja
- obavljanje AL operacija
- programske skokove i grananja
- posebna upravljačka djelovanja (npr. zabrana prekida)

Instrukcija (u strojnom obliku – niz bitova) se sastoji od:

- operacijskog koda (“koja operacija”) i
- adresnog dijela (operandi, adrese)

Posebno zanimljive instrukcije (s aspekta upravljačkog djelovanja OSa)

- ostvarenje instrukcija skoka: `adresa skoka => PC`
- poziv potprograma: `PC => stog; adresa potprograma => PC`
- povratak iz potprograma: `stog => PC`

[dodatno]

ARM procesori koriste poseban registar (LR=R14, *link register*) za pohranu povratne adrese; međutim, radi ugnježđavanja programski je potrebno na početku funkcije pohraniti taj registar na stog (čime se postiže sličan učinak kao i gornji prikaz poziva i povratka iz potprograma koji je uobičajen za većinu arhitektura).

U nastavku se koristi asembler sličan ARM-ovom, uz neke razlike (primjerice poziv i povratak iz potprograma nije kao kod ARM-a već uobičajeni sa CALL i RET).

2.3. Instrukcijska dretva

Pojam "dretva" – postolarski konac; iz "Čudnovate Zgode Šegrta Hlapića" Ivane Brlić Mažuranić: [...] *U torbu metne jedan modar rubac, pa jedno šilo, malo dretve i nekoliko komadića kože. Hlapić je, naime, bio pravi mali majstor, a postolar ne može da bude bez šila i dretve, kao ni vojnik bez puške.* [...]

Razlikujemo niz instrukcija (program) i izvođenje niza instrukcija (dretva).

Primjer 2.3. Primjer programa s grananjem i pozivima potprograma

Program (učitan u memoriju):

```
; Računanje faktorijela
; zadano: N, pretpostavlja se N > 0 !
; rezultat spremiti u: REZ
100         LDR R1, N   ; učitaj N u R0
104         LDR R0, N   ; R0 akumulira rezultat (umnožak)
108 PETLJA: SUB R1, 1   ; R1 = R1 - 1
112         CMP R1, 1   ; usporedi R1 i 1
116         BLE KRAJ    ; ako je R1 <= 1 skoči na KRAJ
120         PUSH R1     ; stavi R1 na stog - parametar funkcije
124         CALL MNOZI
128         ADD SP, 4   ; makni R1 s vrha stoga
132         B PETLJA    ; skoči na PETLJA
136 KRAJ:   STR R0, REZ ; spremi R0 u REZ
            ...
; potprogram
200 MNOZI:  LDR R2, [SP+4]  ; dohvati parametar
204         MUL R0, R2      ; R0 = R0 * R2;
208         RET
```

Program ima 13 instrukcija. Međutim, pri izvođenju za različite `N`-ove dretva će obaviti različiti broj instrukcija.

Za `N=3` samo će se jednom pozvati potprogram `MNOZI`, tj. ukupno će obaviti: $2+(7+3)+4 = 16$ instrukcija.
Za `N=4`: $2 + 2 * (7 + 3) + 4 = 26$ instrukcija.
Za `N=100`: $2 + 98 * (7 + 3) + 4 = 986$ instrukcija.

Primjer izvođenja za `N=4` (dretva)

```
100         LDR R1, 4
104         LDR R0, 4
108 PETLJA: SUB R1, 1 ; R1 = 3 nakon ove instrukcije
112         CMP R1, 1 ;
116         BLE KRAJ  ; nije ispunjen uvjet skoka
120         PUSH R1
124         CALL MNOZI
200 MNOZI:  LDR R2, [SP+4] ; R2 = 3
204         MUL R0, R2     ; R0 = 4*3 = 12
208         RET
128         ADD SP, 4
132         B PETLJA
108 PETLJA: SUB R1, 1 ; R1 = 2
112         CMP R1, 1
116         BLE KRAJ  ; nije ispunjen uvjet skoka
120         PUSH R1
124         CALL MNOZI
```

```
200 MNOZI:  LDR R2, [SP+4] ; R2 = 2
204         MUL R0, R2     ; R0 = 12*2 = 24
208         RET
128         ADD SP, 4
132         B PETLJA
108 PETLJA: SUB R1, 1 ; R1 = 1
112         CMP R1, 1
116         BLE KRAJ  ; ispunjen je uvjet skoka
136 KRAJ:   STR R0, REZ
```

Program

- niz instrukcija (i podataka) koji opisuju kako nešto (korisno) napraviti
- program = slijed instrukcija u spremniku (“prostorno povezanih”)
- nalazi se u datoteci, na disku, ili u spremniku

Proces

- nastaje pokretanjem programa
- traži i zauzima sredstva sustava (spremnik, procesorsko vrijeme, naprave)

Izvođenjem programa (u procesu) instrukcije se ne izvode isključivo slijedno – zbog skokova i poziva potprograma

Slijed instrukcija kako ih procesor izvodi („vezane“ vremenom izvođenja) nazivamo *instrukcijska dretva* ili samo kraće *dretva* (engl. *thread*).

Dretva

- dretva = slijed instrukcija u izvođenju (“povezanih vremenom izvođenja”)
- dretva izvodi instrukcije programa, ona radi taj koristan posao

U nekom trenutku stanje dretve je određeno:

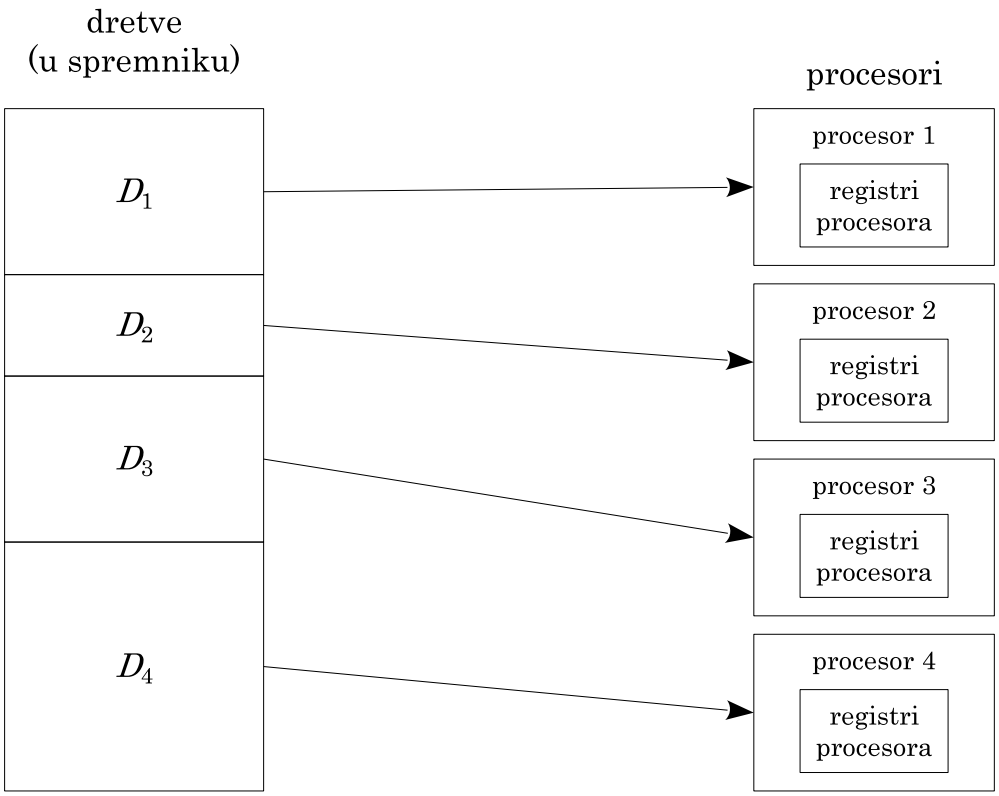
- instrukcijama, podacima koje koristi, sadržajem stoga → sve u spremniku
- sadržajima u registrima procesora → *kontekstom dretve*

Kako izvoditi više dretvi na jednom procesoru ili na manjem broju procesora od dretvi?

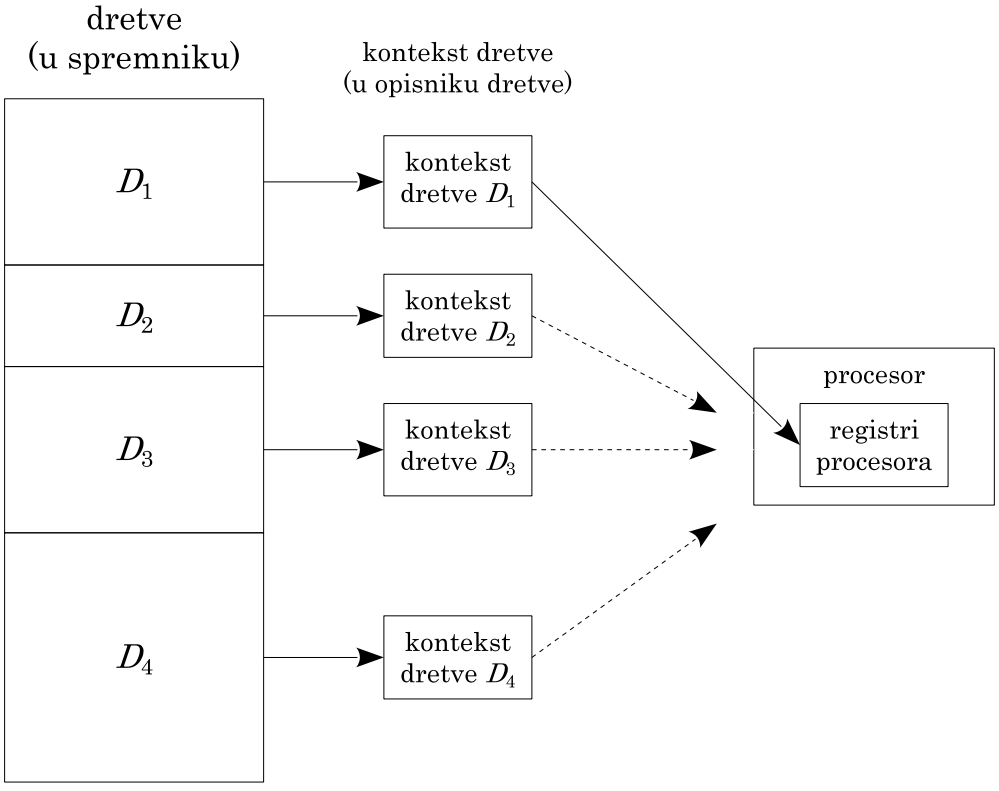
- ideja:
  1. “zamrznuti” dretvu koja se trenutno izvodi; maknuti ju u stranu;
  2. odabrati/uzeti drugu “zamrznutu” dretvu;
  3. “odmrznuti” tu dretvu i nastaviti s njenim radom
- kako to napraviti:
  1. “zamrznuti”: zaustaviti dretvu i pohraniti kontekst te dretve u spremnik
  2. “odmrznuti”: obnoviti kontekst dretve iz spremnika i nastaviti s njenim radom

2.4. Višedretveni rad

1. kada ima dovoljno procesora – svaka dretva se može izvoditi na zasebnom procesoru
2. kada nema dovoljno procesora – dretve dijele procesore naizmjeničnim radom



Slika 2.9. Višedretvenost na višeprocesorskom sustavu



Slika 2.10. Višedretvenost na jednoprocesorskom sustavu

Višedretvenost se (na jednoprocesorskom sustavu) ostvaruje tako da se u nekom odabranom trenutku jedna dretva “zamjeni” drugom.

Postupak zamjene jedne dretve drugom:

1. prekida se izvođenje trenutno aktivne dretve (prekidom, opisano u idućem poglavlju)
2. sprema se kontekst aktivne dretve (trenutni sadržaji registara) u za to predviđeno mjestu u spremniku (u opisnik te dretve)
3. odabire se nova aktivna dretva
4. obnavlja se kontekst novoodabrane dretve (nove aktivne) te
5. aktivna dretva nastavlja s radom

Primjer 2.4. Zamjena dviju dretvi (info)

```
dretva suma
1:    mov R0, 0 //R0 = 0
2:    mov R1, 1
3:    mov R2, X
4: petlja:
4:    add R0, R1 //R0 = R1 + R0
5:    add R1, 1
6:    cmp R1, R2
7:    ble petlja
```

```
dretva produkt
1:    mov R0, 1
2:    mov R1, 1
3:    mov R2, X
4: petlja:
4:    mul R0, R1
5:    add R1, 1
6:    cmp R1, R2
7:    ble petlja
```

U nekom trenutku procesor izvodi dretvu `suma`.

Neka je procesor trenutno upravo završio izvođenje instrukcije na liniji 6 te neka su vrijednosti u registrima: R1=4, R0=10, R2=30 te PC=7 (pretpostavimo da linija odgovara adresi instrukcije)

Za zamjenu dretve `suma` s dretvom `produkt`, potrebno je:

- zaustaviti dretvu `suma` (prekidom)
- pohraniti trenutne vrijednosti registara R0-R3 (i ostalih) te PC, SP, RS u "opisnik dretve" (strukture podataka OSa)
- učitati vrijednosti registara kakve je imala dretva `produkt` u trenutku njena prekida
- nastaviti s radom dretve `produkt` – učitati zadnju vrijednost PC koju je imala dretva produkt pri prekidu i zamjeni
- u nekom idućem trenutku, kad se dretva `produkt` treba zamijeniti dretvom `suma`, napravit će se isto te će novo stanje biti:
  - obnovljeni registri će imati pohranjene vrijednosti (R1=4, R0=10, R2=30)
  - obnovom registra PC=7 označava se povratak u tu dretvu jer će iduća instrukcija biti s adrese (linije) 7
  - dretva `suma` "neće primijetiti" da je bila prekinuta: nastavit će se dalje izvoditi s ispravnom logikom i vrijednostima (kao da i nije bila prekinuta)
- sadržaji registara R0-RX, uz PC, RS i SP u trenutku izvođenja neke dretve nazivaju se *kontekstom* te dretve

Primjer 2.5. Moguća operacija zamjene dretve izravnim pozivom raspoređivača (info)

Kod prve dretve

```
... ; niz instrukcija
CALL Rasporedi
... ; niz instrukcija
```

Funkcija sustava – Rasporedi

```
Rasporedi:
    SPREMI_SVE_REGISTRE_OSIM_PC na adresu kontekst_aktivne
    CALL Odaberi_novu_aktivnu_dretvu //možda složeno!
    OBNOVI_SVE_REGISTRE_OSIM_PC s adrese kontekst_aktivne //promjena stoga!
    RET
```

Programsko brojilo se u ovom primjeru ne pohranjuje jer nije potrebno – dretva ovdje staje te kasnije nastavlja.

Varijabla `kontekst_aktivne` mora pokazivati na spremnički prostor na kojem se nalazi spremljen kontekst dretve koju želimo aktivirati (te prije njenog micanja s procesora tamo spremamo njen kontekst)

Operacija `ODABERI_NOVU_AKTIVNU_DRETVU` mijenja varijablu `kontekst_aktivne` tako da pokazuje na kontekst nove aktivne dretve

U nastavku (idućim poglavljima) koristi se prikazani model jednostavnog računala, koji se postupno nadograđuje potrebnim sklopovljem i funkcionalnošću.

Pitanja za vježbu 2

1. Skicirati procesor – njegove osnovne dijelove, registre.
2. Kako se koristi sabirnica?
3. Što procesor trajno radi?
4. Kako se ostvaruju instrukcije "za skok", "za poziv potprograma", "za povratak iz potprograma"?
5. Što predstavljaju pojmovi: program, proces, dretva?
6. Kako se ostvaruje višedretveni rad? Što je to kontekst dretve?

3. OBAVLJANJE ULAZNO-IZLAZNIH OPERACIJA, PREKIDNI RAD

U ovom poglavlju se razmatra upravljanje UI napravama s aspekta OS-a. OS treba upravljati napravama i preko nekog sučelja omogućiti i programima da komunikaciju s napravama.

Kako iz programa koristiti naprave?

- pristupa im se kao datotekama, uz dodatne opcije i potrebne provjere povratnih vrijednosti
- u C-u: open, close, read, write, fcntl ...

3.1. Spajanje naprava u računalo

Naprave koje se spajaju u računalo imaju različita svojstva

- način rada
  - znak-po-znak ili u bloku stalne veličine (*character/block devices*)
  - pristup preko adrese ili korištenje posebnih instrukcija
- brzina prijenosa podataka
- primjeri naprava: zaslon, tipkovnica, miš, zvučnici, disk, mreža, grafička kartica, ...
- u idućim razmatranjima sve osim procesora, spremnika i sabirnice su naprave

Usporedba vremenskih svojstava procesorsa i raznih naprava (info)

Učinkovitost sustava znatno ovisi o smještaju podataka (i njihovu dohvatu). Tablica 3.1. prikazuje primjer trajanja dohvata podatka u nekom računalnom sustavu, ovisno o tome gdje se podatak nalazi, gdje je osnovni takt procesora 1/0,3ns = 3,33 GHz.

Tablica 3.1. Okvirna vremena dohvata jednog podatka (info)

operacija	trajanje	skalirano
ciklus procesora	0.3 ns	1 s
dohvat registra procesora	0.3 ns	1 s
priručni spremnik L1	0.9 ns	3 s
priručni spremnik L3	12.9 ns	43 s
radni spremnik	120 ns	6 min
SSD	50 μs	2 dana
HDD	1 ms	1 mjesec
ponovno pokretanje OS-a	3 min	20000 godina

Uz podatke iz tablice 3.1. treba dodati da su to najgora vremena potrebna za dohvat. Naime,

mnoge su naprave optimirane za dohvat skupa podataka, a ne samo jednog. Tome je prilagođeno i korištenje naprava kroz međuspremnike. Stoga su njihova svojstva znatno bolja kada se gleda u "prosječnim" vremenima. Naime, iako se možda koriste podaci s diska prosječno vrijeme pristupa tim podacima može biti u rangu L1 priručnog spremnika ili i brže kada se koriste registri i kada je obrada nešto dulja. Na tom se mogućnošću zasniva i upravljanje spremnikom metodom straničenja (o tome više u 8. poglavlju).

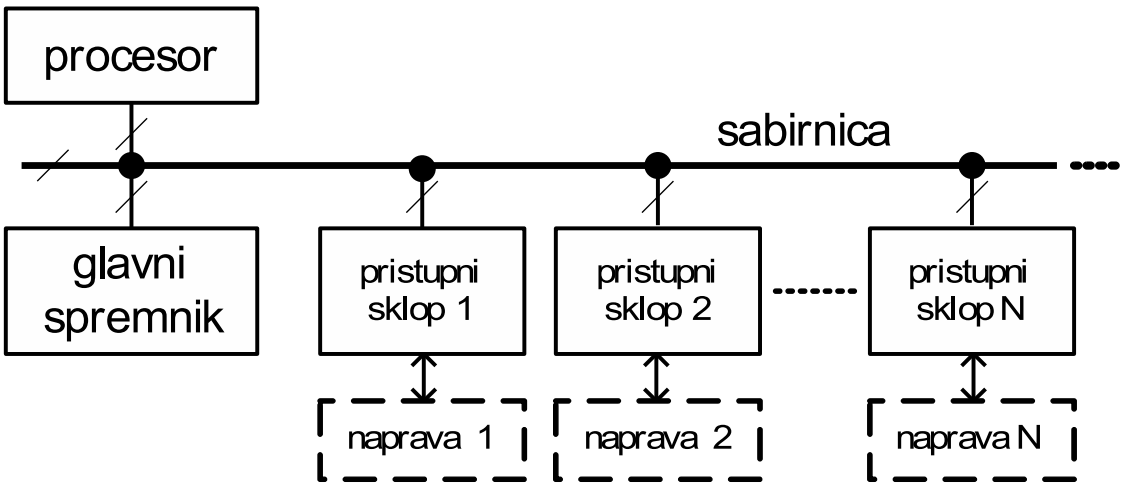
Zbog različitih potreba razne naprave rade različitim frekvencijama.

Primjerice, prema "van" potrebe su:

- tipkovnica i miš: 125-1000 Hz
- audio podsustav (izlaz): 44, 48, 96, ... kHz
- disk: čitanje jednog podatka $\sim$10 ms za HDD ($\sim$0.05 ms za SSD);
  prijenos kompaktno smještenih podataka $\sim$100 MB/s (i više)
- mrežna kartica: npr. 1 Gbit/s $\Rightarrow$ 125 MB/s; odziv mreže od $\sim$0.1 μs do $\sim$100 ms
- grafička kartica: npr. za 1920x1080 sa 60 Hz $\Rightarrow$ $\sim$124 MHz

Pristupni sklop naprava

Zbog različitih svojstava naprave se ne spajaju izravno na (glavnu) sabirnicu, već se spajaju preko međusklopa – *pristupnog sklopa*.



Slika 3.1. Spajanje UI naprava na sabirnicu

Pristupni sklop:

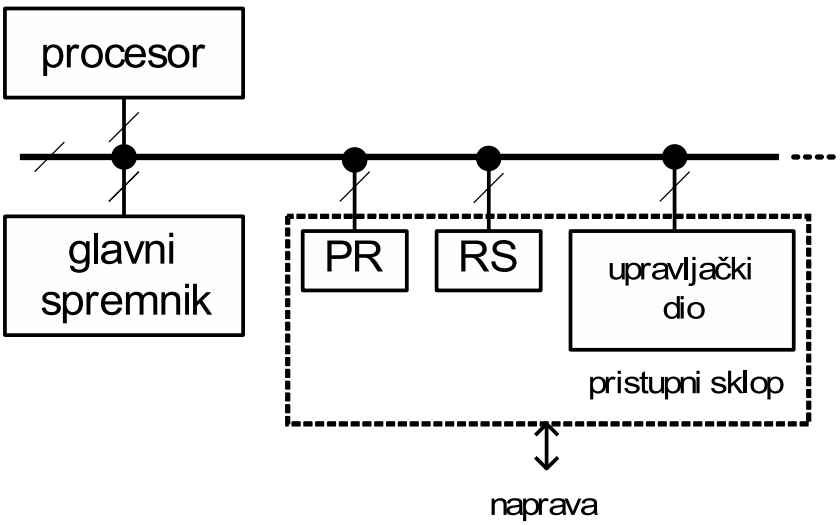
- "zna" komunicirati sa sabirnicom (njenim protokolima)
- "zna" komunicirati s napravom (njenim protokolima, primjerice USB, SATA, PCIe, Ethernet, WiFi, VGA, HDMI, DP, analogni audio izlaz, S/PDIF ...)

Naprave se mogu koristiti na nekoliko načina:

1. radnim čekanjem (engl. *busy-waiting, polling*)
2. prekidima (engl. *interrupts*)
3. izravnim pristupom spremniku (engl. *direct-memory-access – DMA*)

3.2. Korištenje UI naprava radnim čekanjem

Procesor “aktivno” (u programskoj petlji) čeka da naprava postane spremna za komunikaciju



Slika 3.2. Pristupni sklop UI naprave

Elementi pristupnog sklopa i njegovo ponašanje

- PR – podatkovni registar, služi za prijenos podataka
- RS – registar stanja, sadrži zastavicu ZASTAVICA koja pokazuje je li pristupni sklop spreman za komunikaciju s procesorom (ZASTAVICA==1) ili nije (ZASTAVICA==0)
- Upravljački dio s jedne strane osluškuje sabirnicu i radi odgovarajuću akciju kad detektira adresu PR-a ili RS-a, a s druge strane upravlja komunikacijom prema priključenoj napravi.
- Procesor u petlji čita RS dok ZASTAVICA ne postane jednaka 1.

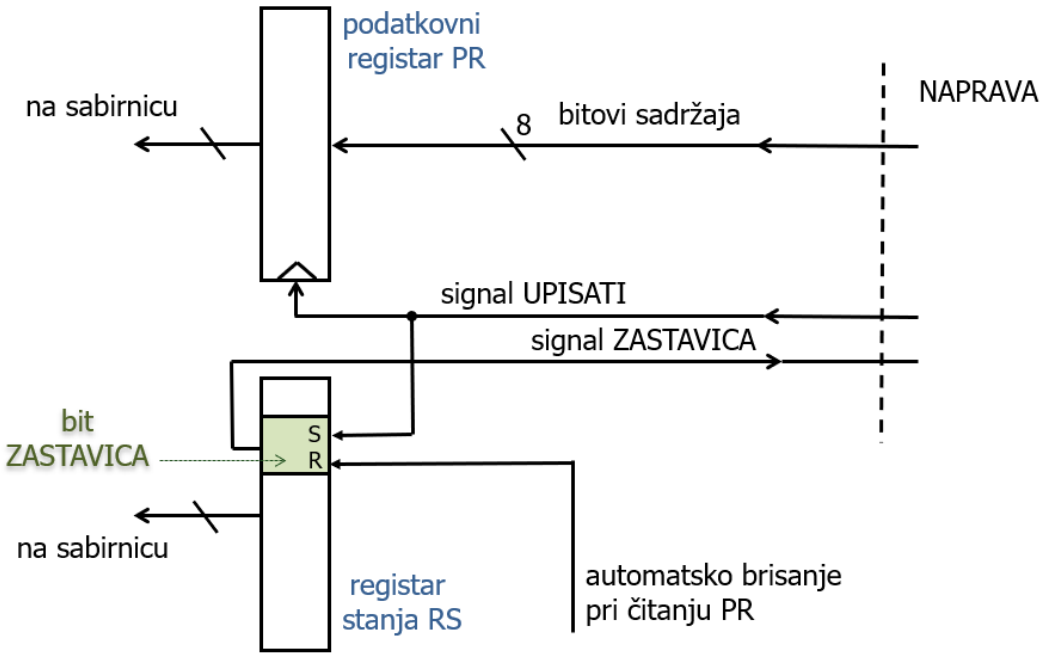
Isječak kôda 3.1. Primjer čitanja jednog podatka s naprave - u pseudokodu

```
    dok je ZASTAVICA == 0 radi  //radno čekanje
        ;                       //u petlji čita RS i ispituje bit ZASTAVICA
    pročitaj PR
```

Isječak kôda 3.2. Primjer čitanja jednog podatka s naprave - u asembleru

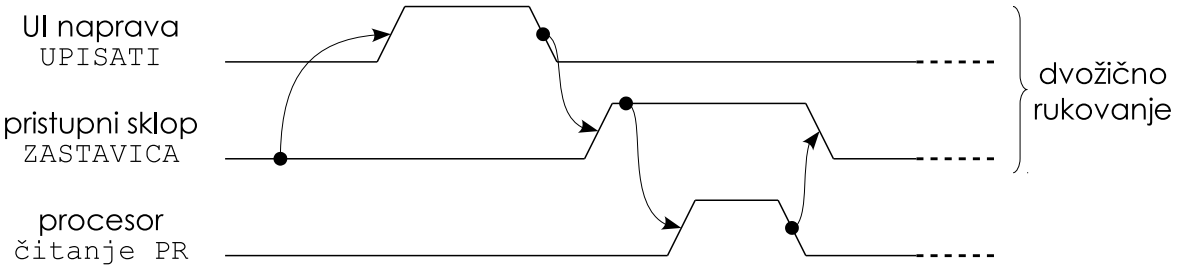
```
    ADR R0, RS      ; adresa registra stanja u R0
    ADR R1, PR      ; adresa podatkovnog registra u R1
petlja:                                                    +\
    LDR R2, [R0]    ; pročitaj registar stanja (sa zastavicom) +| radno
    CMP R2, 0                                                  +| čekanje
    BEQ petlja                                             +/
    LDR R2, [R1]    ; pročitaj znak iz pristupnog sklopa u registar R2
    ... ; obrada pročitanog podatka, npr. samo spremanje u spremnik
```

Brisanje ZASTAVICE po komunikaciji s procesorom može biti ostvareno u pristupnom sklopu (kao na slici 3.3.) ili to može procesor napraviti upisom vrijednosti u RS.



Slika 3.3. Primjer ostvarenja pristupnog sklopa

Signali UPISATI i ZASTAVICA u takvom slučaju koriste **dvožično rukovanje**.



Slika 3.4. Primjer sinkronizacije između naprave, pristupnog sklopa i procesora

Svojstva radnog čekanja kao načina upravljanja UI:

+ prednost: sklopovlje je vrlo jednostavno
− nedostatak: procesor ne radi produktivno, nema koristi od tisuća iteracija petlje
  – drukčijim programom se to ponekad može ublažiti, npr. pojedini sklop se provjerava periodički, provjerava se više sklopova (“prozivanje”), . . .

Radno čekanje se uglavnom koristi u jednostavnim sustavima (ugrađenim) pa ga nećemo više detaljnije razmatrati (iako takvih sustava ima najviše!).

U nastavku slijedi nekoliko primjera upraljvanja napravama korištenjem radnog čekanja u ugrađenim sustavima.

Primjer 3.1. Upravljanje temperaturom

Neko jednostavno računalo treba očitavati temperaturu od senzora i upravljati klimom. Senzoru treba neko vrijeme da generira podatak o temperaturi. U statusnom registru će biti nula dok se temperatura ne postavi u podatkovni registar senzora. Primjer upravljačkog programa koji radi s navedenim senzorom je u nastavku.

```
program() {
    ponavljaj {
        dok je dohvati_UI_registar(SENZOR_TEMPERATURE_STATUS) == 0 radi
            ; //ništa, ponavljaj petlju
        p = dohvati_UI_registar(SENZOR_TEMPERATURE_PODATAK)
        t = pretvori_u_temperaturu(p)
        ako je t < T1 onda {
            postavi_UI_vrijednost(HLAĐENJE, ISKLJUČI)
            postavi_UI_vrijednost(GRIJANJE, UKLJUČI)
        }
        inače ako je T > T2 onda {
            postavi_UI_vrijednost(GRIJANJE, ISKLJUČI)
            postavi_UI_vrijednost(HLAĐENJE, USKLJUČI)
        }
    }
}
```

Primjer 3.2. Periodička provjera

Upravljanje se može obavljati kao periodički posao, dodan u nešto drugo. Na taj način se istim postupkom može i upravljati klimom i raditi nešto drugo korisno.

```
funkcija upravljaj_klimom() {
    ako je dohvati_UI_registar(SENZOR_TEMPERATURE_STATUS) != 0 radi
        p = dohvati_UI_registar(SENZOR_TEMPERATURE_PODATAK)
        ... //ostalo isto kao i u prethodnom primjeru
    }
}
program() {
    ponavljaj {
        ... (upravljanje drugim elementima)
        upravljaj_klimom()
    }
}
```

Primjer 3.3. Arduino

Upravljanje mikrokontrolerima kroz Arduino sučelja najčešće koriste radno čekanje, tj. u petlji obavljaju sav posao. Skica koda takvih primjera je u nastavku.

```
#define PIN_ZAHTJEVA     4
#define PIN_ZA_ODGOVOR   2

//inicijalizacija, pokreće se jednom, pri pokretanju uređaja
void setup()
{
    //npr. postavi ulazni i izlazni pin
    pinMode(PIN_ZAHTJEVA, INPUT);
    pinMode(PIN_ZA_ODGOVOR, OUTPUT);
}

//kod koji se neprestano ponavlja
//npr. kao da na kraju setup() stoji: while(1) loop()
//ali nije baš tako, i druge se stvari pozivaju u međuvremenu
//neki kućanski poslovi, prekid, alarmi, ...
void loop()
{
    //kratki primjer korištenja ulaza/izlaza:
    if (digitalRead(PIN_ZAHTJEVA) == HIGH) { //pročitaj ulaz
        //napravi nešto, npr.
        digitalWrite(PIN_ZA_ODGOVOR, HIGH);
    }
    else {
        digitalWrite(PIN_ZA_ODGOVOR, LOW);
    }
}
```

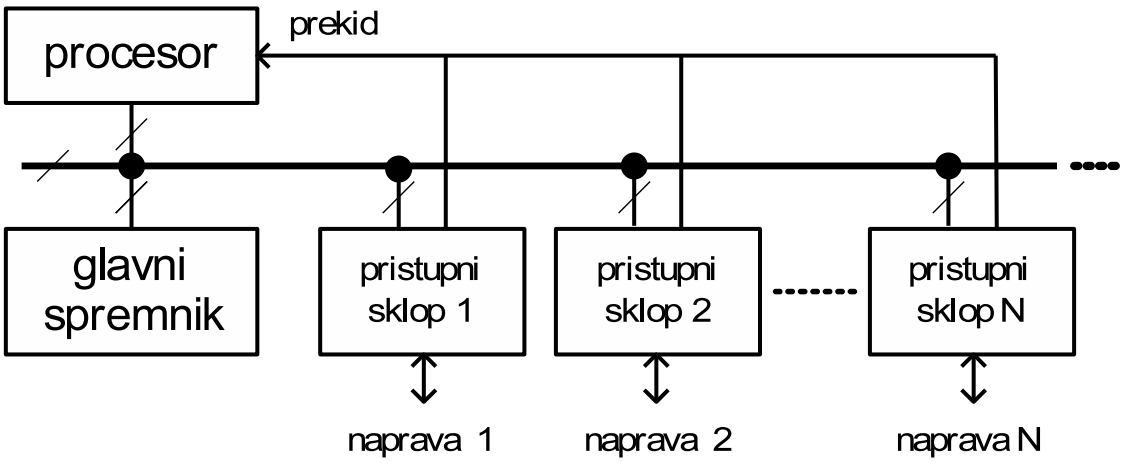
3.3. Prekidni rad

Ideja: kada nema podataka od naprave, procesor radi neki drugi manje bitan ali koristan posao. Kada dođe novi podatak od naprave, naprava sama traži obradu slanjem električnog signala do procesora (npr. bit ZASTAVICA iz registra stanja pristupnog sklopa postaje taj signal).
U nastavku se poistovjećuju pojmovi: prekidni signal == zahtjev za prekid == prekid

Prekidni rad pokazan je bez dodatnog sklopa za prihvat prekida ("s minimalnim sklopovljem"; poglavlja 3.3.1. i 3.3.3.) i s dodatnim sklopom (poglavlje 3.3.4.).

3.3.1. Prekidni rad bez sklopa za prihvat prekida ("bez prioriteta")

Svi zahtjevi za prekide izravno dolaze procesoru preko zajedničkog vodiča (npr. spojeni ILI).



Slika 3.5. Skica najjednostavnijeg sustava za prekidni rad

Kako se procesor treba ponašati kad dobije zahtjev za prekid?

Zahtjev za prekid može doći u bilo kojem trenutku:

1. dok se izvodi neka dretva (program)
2. dok se obrađuje prijašnji prekid

Za 1. procesor (najčešće) treba prihvatiti zahtjev i privremeno prekinuti dretvu.
Za 2. procesor (najčešće) treba privremeno ignorirati zahtjev.

Prihvat zahtjeva za prekid mora popratiti nekoliko akcija:

- mora se omogućiti kasniji nastavak rada prekinute dretve (spremiti njen kontekst)
- mora se spriječiti daljnje prekidanje, dok se prekid ne obradi (ili dok se drukčije ne definira)

Načini rada procesora

- Korisnički način rada
  - manje privilegirani, neke stvari nisu dostupne (instrukcije, registri, spremničke lokacije)
- Prekidni načina rada – jezgrin/sustavski način rada
  - privilegirani način rada – dozvoljene su sve operacije (instrukcije), dostupni su svi registri/podaci/resursi
  - jednostavniji procesori (mikrokontroleri) imaju samo ovaj način rada

Postupak prihvata prekida od strane procesora (ispitno pitanje)

Operacije koje procesor bez sklopa za prihvat prekida radi u slučaju zahtjeva za prekidom

a) početno stanje: procesor izvodi neku instrukciju

b) pojavljuje se zahtjev za prekid

c) procesor dovršava trenutnu instrukciju (regularno, ona se ne prekida)

d) po dovršetku instrukcije (na kraju, prije dohvata iduće):

- ako su prekidi omogućeni (u procesoru preko SR) i zahtjev za prekid je postavljen, procesor radi slijedeće ("postupak prihvata prekida" u užem smislu):

  1. zabrani daljnje prekidanje
  2. prebaci se u prekidni način rada (jezgrin, sustavski)
     - adresiraj sustavski dio spremnika, aktiviraj sustavku kazaljku stoga
  3. na stog pohrani programsko brojilo (PC) i registar stanja (SR)
  4. u PC stavi adresu “prekidnog potprograma”
     - ta je adresa najčešće ili ugrađena u procesor (bira se prema vrsti prekida), ili se koristi tablica za odabir adrese (i koristi se prekidni broj)

Navedeno je **ugrađeno ponašanje procesora**.

Opis rada procesora (kod sa 2.1.) treba proširiti prema 3.3..

Isječak kôda 3.3. Opis rada procesora s podrškom za prekide

```
ponavljaj {
   dohvati instrukciju na koju pokazuje PC          \ ovaj dio
   povećaj PC tako da pokazuje na iduću instrukciju \ je isti
   dekodiraj instrukciju                            \ kao i prije
   obavi operaciju zadanu instrukcijskim kodom      \

   ako su prekidi omogućeni i zahtjev za prekid je postavljen tada
      zabrani daljnje prekidanje
      prebaci se u prekidni način rada
      na stog pohrani programsko brojilo i registar stanja
      u programsko brojilo stavi adresu prekidnog potprograma
}
dok je procesor uključen
```

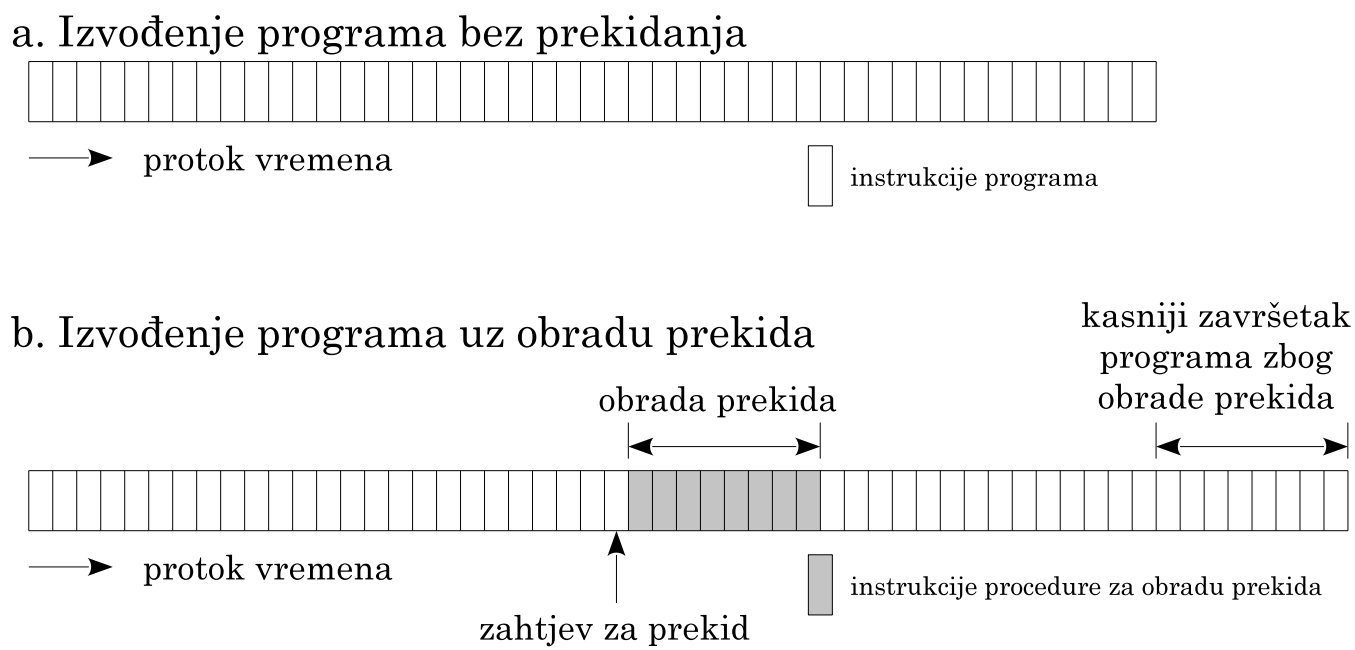
Prekidni potprogram može izgledati ovako (u najjednostavnijem slučaju):

```
Prekidni potprogram
{
   pohrani kontekst na sustavski stog (osim PC i SR koji su već tamo)
   ispitnim lancem odredi uzrok prekida (ispitujući RS pristupnih sklopova) => I
   signaliziraj napravi da je njen zahtjev za prekid prihvaćen

   obradi_prekid_naprave_I //npr. poziv funkcije upravljačkog programa

   obnovi kontekst sa sustavskog stoga (osim PC i SR)
   vrati_se_u_prekinutu_dretvu //jedna instrukcija opisana u nastavku
}
```

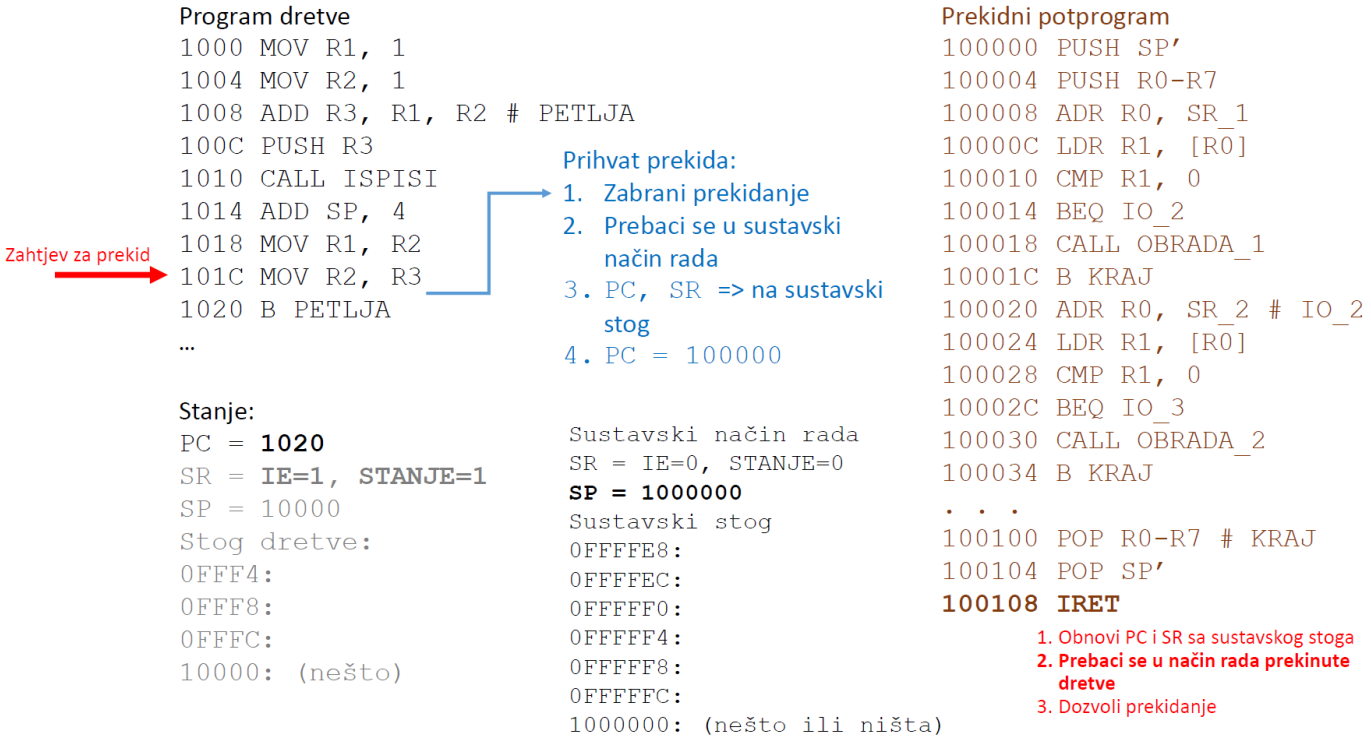
```
instrukcija: vrati_se_u_prekinutu_dretvu
{
   obnovi PC i SR sa sustavskog stoga
   prebaci (vrati) se u način rada prekinute dretve (definiran u SR)
   dozvoli prekidanje
}
```

Obnavljanje registra stanja vraća dretvu u prethodni način rada (korisnički, ako se vraćamo u dretvu ili sustavski ako se vraćamo u obradu nekog prekinutog prekida).

Obrada prekida naprave može uključivati razne operacije, od samog kopiranja podatka u radni spremnik do složenih operacija (npr. aktivaciju neke dretve).



Slika 3.6. Utjecaj prekida na odgodu programa



Slika 3.7. Primjer prihvata prekida

Zadatak 3.1. Primjer obrade niza zahtjeva za prekid

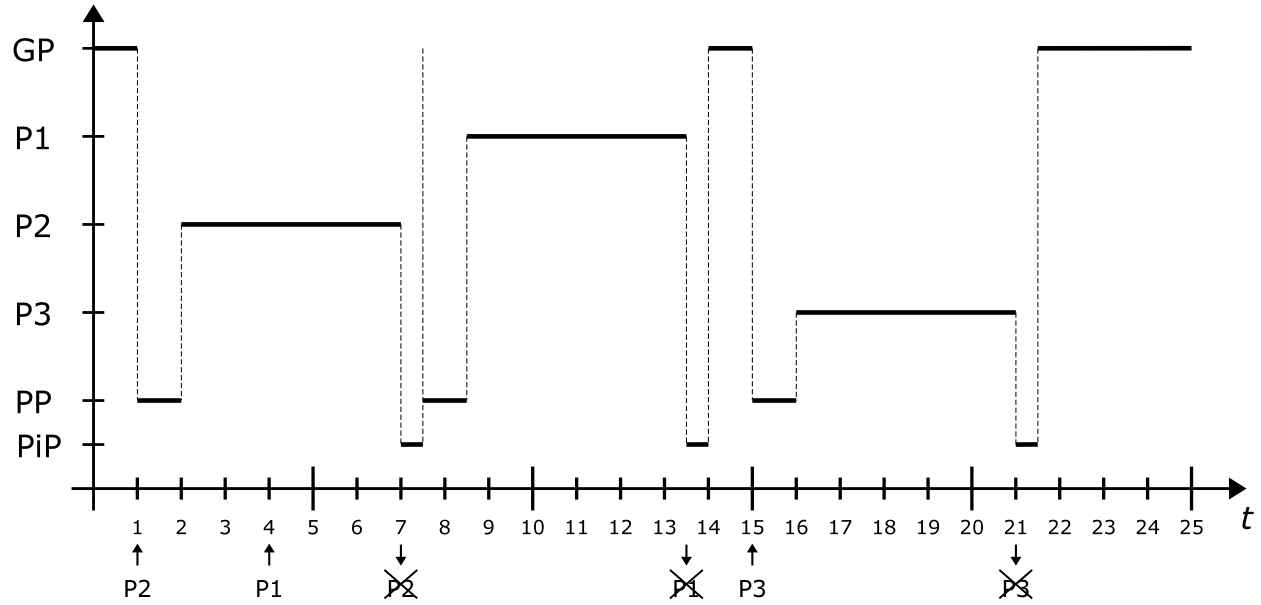
U nekom sustavu javljaju se prekidi: P1 u 4. ms, P2 u 1. ms te P3 u 15. ms. Obrada svakog prekida traje 5 ms (koristan dio obrade). Postupak prihvata prekida (PP, spremanje konteksta prekinute dretve, ispitni lanac) neka traje 1 ms. Povratak iz prekida (PIP, obnova konteksta i povratak u dretvu) neka traje 0,5 ms.

a) Grafički prikazati rad procesora u glavnom programu (GP), obradama P1, P2 i P3 te kućanskim poslovima PP i PIP.

b) Koliko se ukupno vremena potroši na kućanske poslove (PIP + PP)?

c) Koliko se odgađa obrada GP zbog ta tri prekida?

a)



b) 3 prekida puta (PP + PIP) $= 3 * 1,5 = 4,5$ ms

c) u intervalu $1 - 21,5$ GP je radio 1 ms; odgođen je $21,5 - 1 - 1 = 19,5$ ms

---

Svojstva upravljanja napravama prekidom, bez sklopa za prihvat prekida

+ radi, jednostavno sklopovlje i programska potpora

− potrebno je dodatno sklopovlje (procesor s potporom za prihvat prekida)

• problem: nakon početka obrade jednog prekida (tijekom njegove obrade) svi novi zahtjevi MORAJU pričekati kraj obrade prethodnog

  – u nekim se sustavima (za rad u stvarnom vremenu) podrazumijeva da obrade prekida traju vrlo kratko te je kod njih i ovakav način prihvata i obrade prekida dovoljno dobar

  – ako to nije prihvatljivo, mora se promijeniti način prihvata prekida (programski ili uz sklop, opisano u nastavku)

  – kada obrada jednog prekida završi, jedan od postojećih zahtjeva će se obraditi idući

    * koji zahtjev će se obraditi idući?

    * ovisi o redoslijedu ispitivanja naprava u ispitnom lancu u prekidnom potprogramu

    * taj redoslijed definira “prioritet” naprava

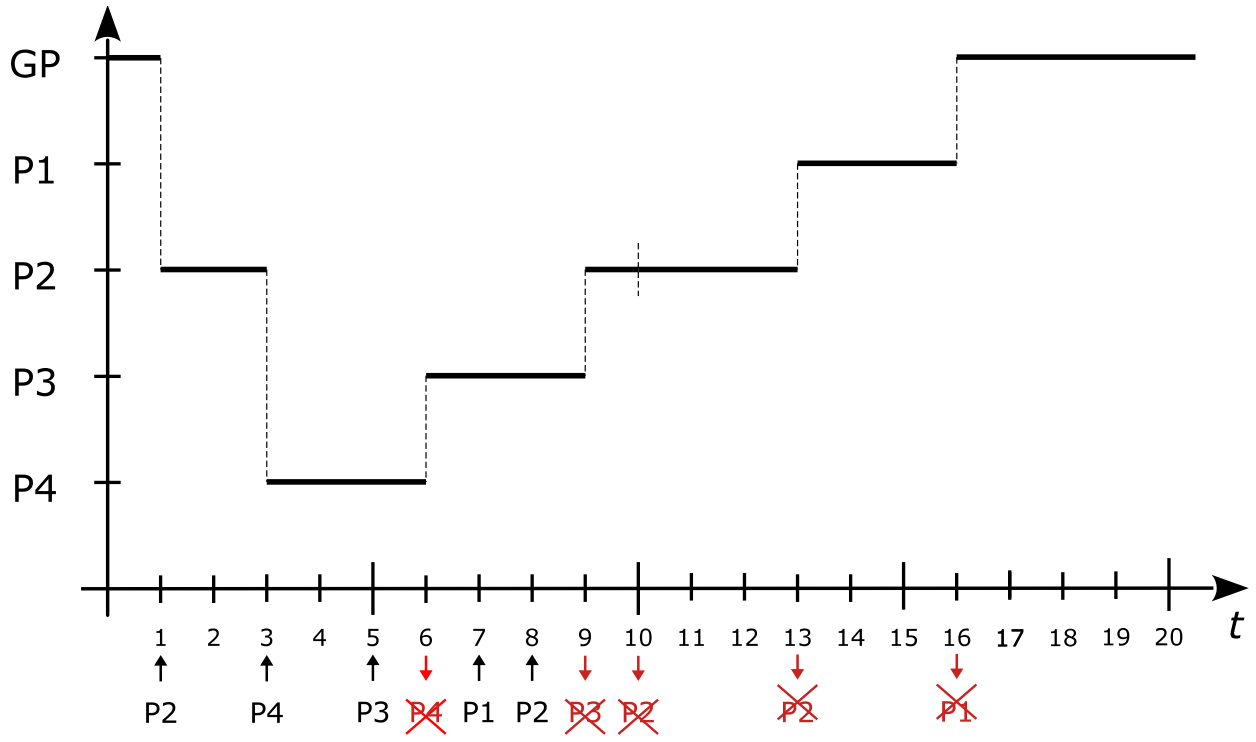
3.3.2. Poželjni (idealan) način prihvata prekida

Obrada prema prioritetu

- Želimo zahtjevima dodijeliti prioritet da se obrađuju prema njemu, prvo oni većeg prioriteta.
- Kada zahtjev većeg prioriteta dođe za vrijeme obrade zahtjeva manjeg prioriteta, on treba privremeno prekinuti tu obradu i započeti svoju
  - po završetku te obrade nastaviti prethodno prekinutu obradu ili
  - ako se u međuvremenu (za vrijeme ove obrade) pojavio novi zahtjev manjeg prioriteta od ovog u obradi ali većeg od prekinute obrade, započeti obradu takva zahtjeva (i odgoditi nastavak prekinute)
- Arhitekt sustava treba pridijeliti prioritete zahtjevima prema UI napravama koje ih izazivaju. U stvarnosti se to ostvaruje sklopovima, odnosno, dovođenjem zahtjeva za prekid do jednog od ulaza sklopa za prihvat prekida te programiranjem takvog sklopa da taj ulaz ima takav prioritet.
- U pojednostavljenom prikazu svakoj napravi ćemo dodijeliti identifikacijski broj (redni broj) koji će ujedno označavati i prioritet. Naprava X ima prioritet X. Neka veći broj označava bitnije naprave, tj. "veći prioritet"
- obradu prema prioritetima prekida možemo ostvariti: *programski* ili *sklopovski*

Kućanski poslovi

- u postupku prihvata prekida obavljaju se razne operacije: prihvat prekida od strane procesora te operacije u prekidnom potprogramu – sve su one neophodne u pojedinom načinu prihvata prekida
- ali samo "`obradi prekid naprave I`" radi koristan posao
- sve ostalo su "kućanski poslovi" te bi željeli da traju što kraće, tj. u idelnom scenariju kao da ih nema, slično kao na primjeru 3.8.



Slika 3.8. Primjer idealnog prihvata prekida

3.3.3. Obrada prekida prema prioritetima, bez sklopa za prihvat prekida

Ideja programskog rješenja:

- sama obrada prekida (korisna/čista obrada) neka se obavlja s dozvoljenim prekidanjem
- na svaki zahtjev za prekid pozvati proceduru koja će ustanoviti prioritet zahtjeva i usporediti ga s trenutnim poslom:
  - ako je novi zahtjev prioritetniji odmah započinje njegova obrada (prekinuti posao se nastavlja kasnije)
  - u protivnom, novi zahtjev se samo zabilježi te se nastavlja s prekinutom obradom (novi zahtjev će doći na red kasnije)
- prekidima (napravama) treba dodijeliti prioritete
  - u nastavku je (radi jednostavnosti algoritma) prioritet određen brojem naprave, veći broj $\Rightarrow$ veći prioritet

Podatkovna struktura rješenja:

- `TEKUĆI_PRIORITET` – prioritet tekućeg posla, 0 kad se izvodi obična dretva ("glavni program"), broj `I` za obradu prekida naprave rednog broja `I` (njena prioriteta)
- `OZNAKA_ČEKANJA[]` – polje od `N` elemenata (`N` je najveći prioritet)
  - `OZNAKA_ČEKANJA[I]` označava da li naprava `I` čeka na početak obrade ili ne
- `KON[N]` – rezervirano mjesto u spremniku za pohranu konteksta dretve pri obradi pojedinog prekida; uz kontekst dretve sprema se i tekući prioritet

Isječak kôda 3.4. Prihvat prekida prema prioritetima (bez sklopa)

```
Prekidni potprogram  //prihvat prekida
{
   spremi kontekst na sustavski stog (osim PC i SR)
   ispitnim lancem utvrdi uzrok prekida, tj. indeks I
   signaliziraj napravi da je njen zahtjev za prekid prihvaćen
   OZNAKA_ČEKANJA[I] = 1

   dok je (I > TEKUĆI_PRIORITET) //idemo u obradu prekida prioriteta "I"?
      OZNAKA_ČEKANJA[I] = 0
      pohrani kontekst sa sustavskog stoga i TEKUĆI_PRIORITET u KON[I]
      TEKUĆI_PRIORITET = I

      omogući prekidanje
      obradi_prekid_naprave_I //koristan posao (ostalo su "kućanski poslovi")
      zabrani prekidanje

      iz KON[I] obnovi TEKUĆI_PRIORITET i a kontekst stavi na sustavski stog

      I = max { J | za sve J za koje vrijedi: OZNAKA_ČEKANJA[J] != 0 }
      //od naprava koje čekaju na početak obrade, naprava I ima najveći prioritet
      //npr. I=0; za j=0 do N radi: ako je OZNAKA_ČEKANJA[J]!=0 onda I=J;
   }

   obnovi kontekst sa sustavskog stoga (osim PC i SR)
   vrati_se_u_prekinutu_dretvu
}
```

(info) Umjesto korištenja strukture KON[] mogli bi koristiti samo stog, na njega spremiti tekući prioritet (kontekst prekinutog posla je već tamo). Međutim, to ne mora biti svugdje moguće

– možda pri prihvatu prekida se u nekim procesorima automatski sustavska kazaljka stoga postavlja u neku vrijednost (npr. 0xF0000) – tada bi se na svaki prihvat prekida prepisao stog! Ipak, gotovo svi procesori ne rade tako te bi korištenje stoga bilo efikasnije (stog će se koristiti u idućem načinu prihvata prekida uz korištenje sklopa).

Zadatak 3.2. Primjer obrade niza zahtjeva za prekid, uz prioritete

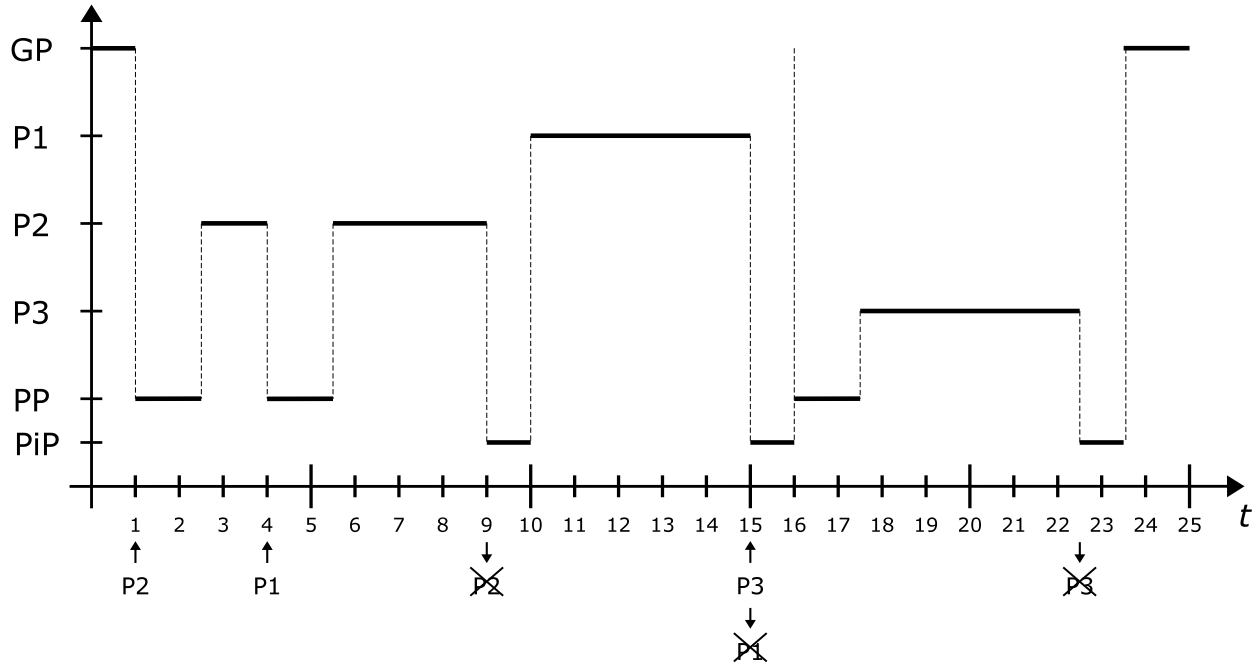
U nekom sustavu koji nema sklop za prihvat prekida, ali ima programski rješenu obradu prekida prema prioritetima, javljaju se prekidi: P1 u 4. ms, P2 u 1. ms te P3 u 15. ms. Prioriteti prekida zadani su brojem: P1 ima najmanji, a P3 najveći prioritet. Obrada svakog prekida traje 5 ms. Postupak prihvata prekida (PP) neka traje 1,5 ms. Povratak iz prekida (PIP) neka traje 1 ms.

a) Grafički prikazati rad procesora u glavnom programu (GP), obradama P1, P2 i P3 te kućanskim poslovima PP i PIP.

b) Koliko se ukupno vremena potroši na kućanske poslove?

c) Koliko se odgađa obrada GP zbog ta tri prekida?

a)



b) 3 prekida puta (PPP + PIP) $= 3 * (1,5 + 1) = 7,5$ ms

c) GP prekinut u $1.$ ms, nastavlja u $23,5 \Rightarrow 22,5$ ms je odgođen

---

Svojstva prihvata i obrade prekida prema prioritetima, bez posebnog sklopa

+ prekidi se obrađuju u skladu s prioritetima
+ nije potrebno posebno sklopovlje za to
− nedostatak – svaki prekid uzrokuje "kućanske poslove" tj. poziv prekidnog potprograma, čak i prekidi manjeg prioriteta
  – kako riješiti taj nedostatak? jedino korištenjem dodatnog sklopovlja

Primjer 3.4. (Info) "Optimiranija" inačica prihvat prekida prema prioritetima

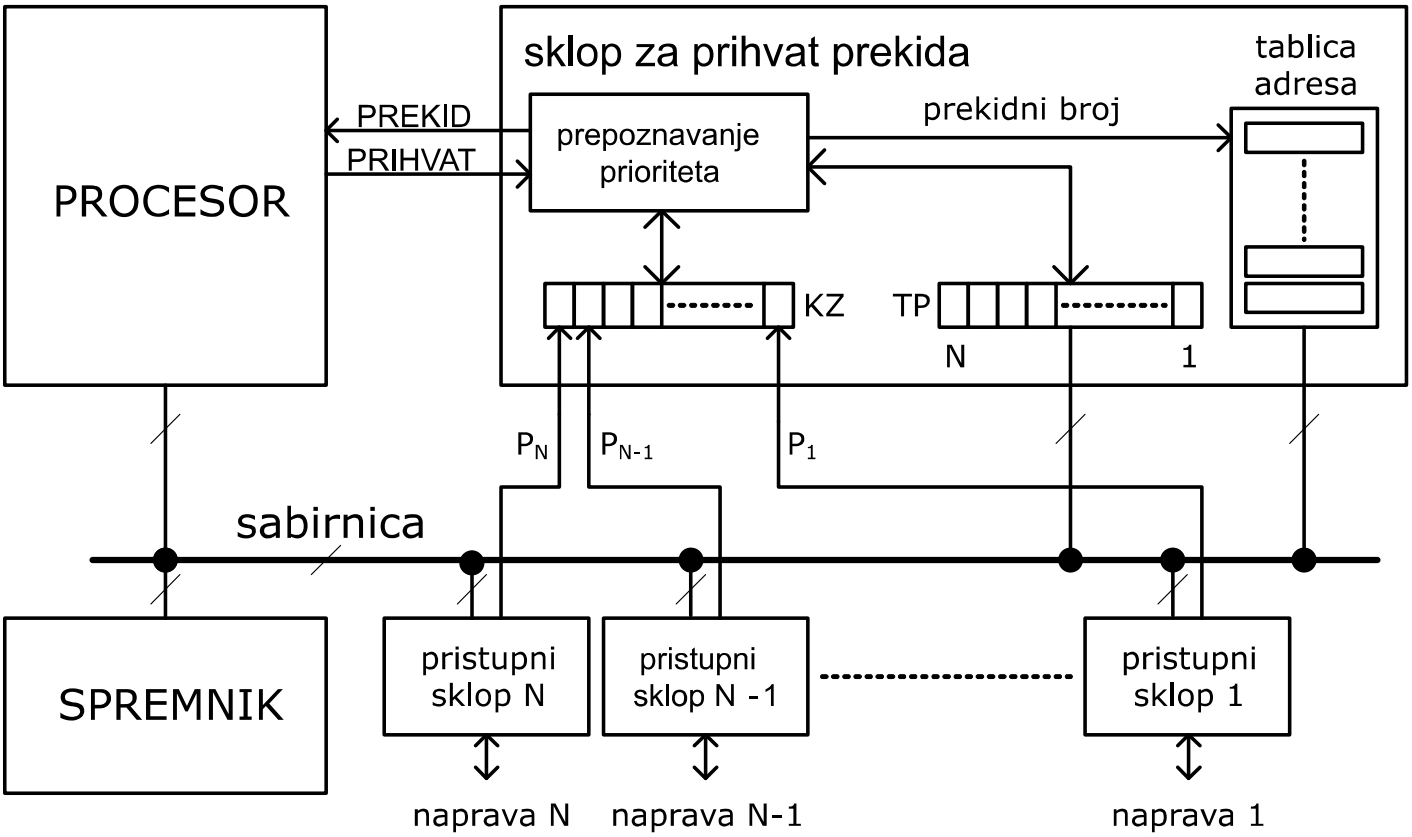
Idući primjer prikazuje moguće ostvarenje prihvata prekida koje ne koristi strukturu KON već kontekst prekinute dretve ostaje na stogu kamo se sprema i stara vrijednost varijable TEKUĆI_PRIORITET. Također, ovdje se provjeravaju sve naprave i ažuriraju sve zastavice, ne samo ona najvećeg prioriteta. Na taj način će se izbjeći neki nepotrebni prekidi, tj. kućanski poslovi kada se zahtjevi preklapaju (jave istovremeno).

```
Prekidni potprogram  //prihvat prekida
{
    spremi kontekst na sustavski stog (osim PC i SR)

    ponavljaj {
        I = 0
        za J = 1 do MAX_PRIO { //ispitni lanac
            ako (naprava J traži prekid) {
                signaliziraj napravi da je njen zahtjev za prekid prihvaćen
                OZNAKA_ČEKANJA[J] = 1
            }
            ako OZNAKA_ČEKANJA[J] == 1 onda
                I = J //najveći prioritet do sada
        }
        ako (I > TEKUĆI_PRIORITET) {
            OZNAKA_ČEKANJA[I] = 0
            pohrani TEKUĆI_PRIORITET na sustavski stog
            TEKUĆI_PRIORITET = I

            omogući prekidanje
            obradi_prekid_naprave_I
            zabrani prekidanje

            obnovi TEKUĆI_PRIORITET sa sustavskog stoga
        }
    }
    dok je (I > 0)

    obnovi kontekst sa sustavskog stoga (osim PC i SR)
    vrati_se_u_prekinutu_dretvu
}
```

S obzirom da se kontekst ne premješta sa sustavskog stoga u KON i natrag ovdje je manje kućanskog posla u usporedbi s prethodnim prekidnim potprogramom.

3.3.4. Obrada prekida korištenjem sklopa za prihvat prekida

Sustav sa sklopom za prihvat prekida povećava efikasnost sustava, smanjujući "kućanske poslove" zadržavanjem prekide manjeg prioriteta od onog koji je trenutno u obradi.



Slika 3.9. Sklop za prihvat prekida

Elementi sklopa:

1. registar KZ – kopije ZASTAVICA pristupnih sklopova
2. registar TP – tekući prioritet – prioritet posla koji procesor trenutno radi
3. tablica adresa – za svaki prioritet sadrži adresu odgovarajuće funkcije za obradu prekida
4. sklop za prepoznavanje prioriteta

Pristupni sklopovi svoj zahtjev za prekid prosljeđuju sklopu za prihvat prekida – postavlja se odgovarajuća zastavica u registru KZ koja odgovara prioritetu spojene naprave.

Sklop za prihvat prekida cijelo vrijeme uspoređuje registre KZ i TP:

- uspoređuju se indeksi najznačajnijeg postavljenog bita iz oba registra
  - "matematički": MSB(KZ) i MSB(TP), uz MSB$(x) = \lfloor \log_2(x) \rfloor$ (*the most significant bit*)
- kada se dogodi zahtjev većeg prioriteta (MSB(KZ) > MSB(TP)):
  1. sklop šalje signal PREKID prema procesoru
  2. procesor mu u nekom trenutku odgovara s PRIHVAT (ne mora odmah!)
  3. na signal PRIHVAT sklop radi slijedeće:
     a) definira prekidni broj I=MSB(KZ) (najveći prioritet zahtjeva pristupnih sklopova)
     b) iz tablice adresa uzima I-tu adresu i stavlja ju na sabirnicu
     c) briše bit KZ[I]=0 (zahtjev više ne čeka, ide u obradu)

Ponašanje procesora u postupku prihvata prekida treba promijeniti u odnosu na prihvat prekida bez sklopa, ali samo u zadnjem koraku, korak 4 treba zamijeniti koracima 4 i 5:

1. zabrani daljnje prekidanje
2. prebaci se u prekidni način rada (jezgrin, sustavski)
3. na stog pohrani programsko brojilo (PC) i registar stanja (SR)
4. **postavi signal PRIHVAT**
5. **sa sabirnice dohvati adresu funkcije za obradu prekida** te ju stavi u PC

Funkcije za obradu prekida trebaju izgledati prema kodu:

```
1  obrada_prekida_naprave_I {
2     pohrani kontekst na sustavski stog (osim PC i SR)
3     dohvati i pohrani registar TP na sustavski stog
4     TP[I] = 1
5
6     dozvoli prekidanje
7     obradi_prekid_naprave_I
8     zabrani prekidanje
9
10    obnovi TP sa sustavskog stoga
11    obnovi kontekst sa sustavskog stoga (osim PC i SR)
12    vrati_se_u_prekinutu_dretvu
13 }
```

(info) Moguća su dodatna poboljšanja sklopa:

- sklop može sam postavljati i brisati zastavice u TP
  - na signal PRIHVAT može uz sve već navedeno postaviti i TP[I] = 1
  - povratak iz ovakvog prekida (“vanjskog”) obavlja se posebnom instrukcijom sličnom opisanoj `vrati_se_u_prekinutu_dretvu` koja dodatno šalje signal KRAJ (engl. *EOI – end of interrupt*) sklopu na što on postavlja TP[I] = 0
  - uz ovakvo proširenje ne treba pohranjivati/postavljati/obnavljati TP u prekidnom potprogramu (bez linija 3, 4 i 10 u prethodnom pseudokodu)

Svojstva upravljanja prekidima sa sklopom za prihvat prekida:

\+ sa sklopom za prihvat prekida izbjegavaju se nepotrebna prekidanja $\Rightarrow$ propuštaju se samo prioritetniji zahtjevi

\+ manje “kućanskih poslova” (kraće traje prihvat prekida i povratak iz prekida)

− potreban je sklop

U idućim poglavljima će se podrazumijevati da sustav posjeduje sklop za prihvat prekida, koji će od sada biti dio procesora.

Zahtjevi za prekid iz zadataka 3.1. i 3.2. bi se korištenjem sklopa obradili redoslijedom kao u 3.1. zbog toga što se u tim primjerima ne pojavljuje prekid većeg prioriteta koji bi prekinuo obradu prekida manjeg prioriteta. Međutim, s obzirom na to da se kod sustava koji imaju sklop ne radi prozivanje naprava već se izravno poziva funkcija koja obrađuje prekid, sam postupak prihvata prekida je brži (samo se pohranjuje kontekst i omogućuje prekidanje).

Zadatak 3.3. (ispitni zadatak)

U nekom sustavu javljaju se zahtjevi za prekid: P1 u 3. ms, P2 u 1. ms te P3 u 4. ms. Prioritet prekida određen je brojem (P3 ima najveći prioritet). Obrada svakog prekida traje po 4 ms. Grafički prikazati aktivnosti procesora u glavnom programu (GP), procedurama za obradu prekida (Pi) te procedurama za prihvat prekida (PP) i povratak iz prekida (PiP) i to:

a) u idealnom slučaju *(prekidi se obrađuju prema prioritetu, trajanje kućanskih poslova se zanemaruje)*

b) bez sklopa za prihvat prekida *(u sustavu koji nema sklop za prihvat prekida u kojem se po prihvatu nekog prekida on obrađuje do kraja – nema ni programske ni sklopovske potpore za obradu prekida prema prioritetima)*, uz trajanje prihvata prekida (PP) od 1 ms *(uključuje potragu za izvorom prekida – zahtjevi većeg prioriteta se prvi prihvaćaju)* te trajenje povratka iz prekida (PiP) od 0,5 ms *(obnova konteksta prekinute dretve)*

c) bez sklopa ali s programskom potporom *(u sustavu koji nema sklop za prihvat prekida, ali se programski određuje prioritet prekida – obrada prekida se odvija s dozvoljenim prekidanjem)*, uz trajanje procedure za prihvat prekida i određivanje prioriteta prekida od 1,5 ms (PP), te 1 ms za povratak iz prekida (PiP) *(na dovršetku obrade prekida ponovno treba pogledati ima li još nešto za obraditi)*

d) sa sklopom za prihvat prekida uz vrijeme prihvata prekida *(pohrana konteksta prekinute dretve)* od 0,5 ms (PP) te vrijeme povratka iz prekida *(obnove konteksta)* od 0,5 ms (PiP)

U trenutku t_x prikazati stanje korištene podatkovne strukture (za c) i d)).

Dodatna pravila za rješavanje:

- ako u istom trenutku neka obrada završava i pojavljuje se novi zahtjev za prekid pretpostavljamo da je ipak najprije završila obrada, a potom se dogodio zahtjev za prekid
- ako u istom trenutku više naprava generira zahtjev za prekid najprije se prihvaća onaj najvećeg prioriteta – ostale i dalje imaju postavljen zahtjev za prekid (na prekidnom ulazu procesora ili sklopa za prihvat prekida)

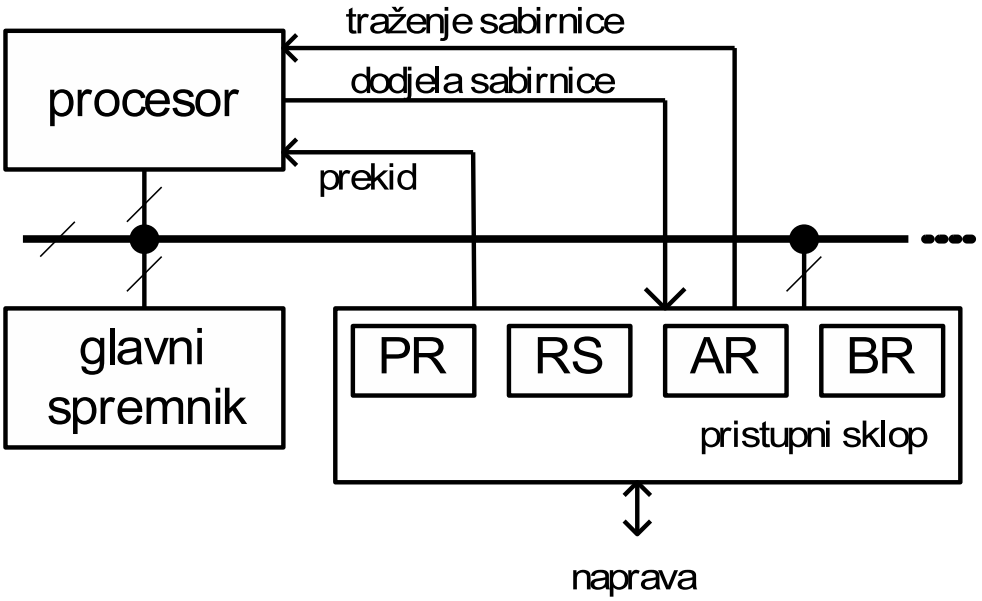
Primjer 3.5. Arduino + prekidi

Skica programa s korištenjem prekida u Arduino okruženju prikazana je u nastavku.

```
#define PIN_ZAHTJEVA  4
#define PIN_ZA_ODGOVOR 2
void obrada_prekida()
{
    digitalWrite(PIN_ZA_ODGOVOR, HIGH);
}
void setup()
{
    pinMode(PIN_ZAHTJEVA, INPUT);
    pinMode(PIN_ZA_ODGOVOR, OUTPUT);
    attachInterrupt(digitalPinToInterrupt(PIN_ZAHTJEVA),obrada_prekida,RISING);
}
void loop()
{
    digitalWrite(PIN_ZA_ODGOVOR, LOW);
    delay(1); //odgodi 1 ms
}
```

3.4. Korištenje sklopova s izravnim pristupom spremniku

- pri korištenju prekida za upravljanje UI napravama dosta je neproduktivnog rada - kućanski poslovi spremanja/obnove konteksta, prozivanja naprava, ...
- koristan posao je često samo prijenos jednog podatka u spremnik ili iz spremnika
- može li prenošenje u spremnik ili iz njega prema UI napravama biti učinkovitije?
- može, proširenjem sklopovlja dodatnim mogućnostima



Slika 3.10. Pristupni sklop s izravnim pristupom spremniku

Elementi i rad pristupnog sklopa s izravnim pristupom spremniku (*direct memory access – DMA*)

- PR, RS – podatkovni registar i registar stanja (kao i prije)
- AR – adresni registar – od kuda/kamo (iz/u memoriju) se učitavaju/pohranjuju podaci naprave
- BR – brojilo podataka – koliko podataka još za prenijeti
- kad UI pristupni sklop ima novi podatak ili može preuzeti novi, on:
  - od procesora traži upravljanje sabirnicom za jedan ciklus (*bus request* – BREQ)
  - kad dobije upravljanje (*bus acknowledge* – BACK), prenosi podatak iz PR u spremnik ili obratno
- procesor prati zahtjeve tek kada on upravlja sabirnicom (u ovom modelu)
- po zahtjevu, procesor prepušta **idući** sabirnički ciklus zahtjevu

Pristupni sklop po inicijalizaciji (nakon učitavanja AR i BR od strane procesora) radi sljedeće:

```
dok je (BR > 0) {
    čekaj na podatak VJ //ili na spremnost VJ za prihvat novog podatka
    zatraži sabirnicu i čekaj na dodjelu sabirnice //dvožično rukovanje!
    {AR, PR} na sabirnicu (+čitaj/piši signal)
    AR++
    BR--
}
postavi signal PREKID
```

Po jednom prijenosu procesor “izgubi” samo jedan ciklus na sabirnici

Ovakav sklop je pogodan kada treba prenijeti veću količinu podataka, inače je jednak običnome

Kada se prenesu svi podaci sklop izaziva prekid te ga procesor (eventualno) opet programira.

Primjer 3.6. Usporedba načina korištenja UI jedinice

a) neka procesor ima 10 MIPS-a te neka u sekundi prosječno treba prenijeti 1000 znakova

- radno čekanje => 100% procesorskog vremena, 10 korisnih instrukcija po prijenosu
  - 1000 (zn. u 1 s) * 10 (instr.) / 10 000 000 (instr./s) = 0,1% korisnog rada
  - (ili 10 / 10 000 = 0,1% korisnog rada)
- prekidi: ∼200 instr. po prekidu => 1000 * 200 / 10 000 000 = 2% => 98% ostane!
  - (ili 200 / 10 000 = 2%, 98% ostaje)
- DMA: 1000 / 10 000 000 = 0,01 % => 99,99 % ostane!
  - (ili 1 / 10 000 = 0.01%, 99.99% ostaje)
  - nakon prijenosa bloka znakova izaziva se prekid; npr. kada je blok 1000 znakova onda bi nakon 1000. znaka bio prekid i njegova obrada s 200 instrukcija bi neznatno povećala opterećenje s 0,01 na 0,012

b) isti procesor, maksimalna brzina prijenosa (jako brza UI)

- radno čekanje: 10 000 000 / 10 = 1000 000 znakova/s uz 100% opt. procesora
- prekidi: 10 000 000 / 200 = 50 000 znakova/s uz 100% opt. procesora
- DMA (svaki 2. sabirnički ciklus): 10 000 000 / 2 = 5 000 000 znakova/s uz 50% opt. procesora

---

Primjer 3.7. Usporedba načina korištenja UI jedinice (2)

Neka sabirnica radi na 1 GHz, a naprava generira podatke frekvencijom 1 MHz.

- radno čekanje: 100% procesorskog vremena, $10^6 \cdot 10/10^9 = 0,01 = 1\%$ korisnog rada
- prekidi: $10^6 \cdot 200/10^9 = 0,2 = 20\%$; 80% ostaje!
- DMA: $10^6/10^9 = 0,001 = 0,1\%$; 99,99% ostaje!

Najveća brzina posluživanja vrlo brze naprave (uz isti procesor):

- radno čekanje: $10^9/10 = 10^8$ podataka u sekundi uz 100% opt. procesora
- prekidi: $10^9/200 = 5 \cdot 10^6$ podataka u sekundi uz 100% opt. procesora
- DMA: $10^9/2 = 5 \cdot 10^8$ podataka u sekundi uz 50% opt. procesora

---

3.5. Usporedba načina upravljanja UI napravama

1. Radno čekanje
   - u petlji se preko zastavica provjerava mogućnost slanja/primanja podataka

   \+ jednostavno sklopovlje (jeftino)

   − neefikasno korištenje procesora (radno čekanje)
2. Prekidi
   - ideja – biti efikasniji od radnog čekanja omogućujući procesoru da izvodi neki koristan posao, ali da ipak brzo reagira na događaj naprave
   - pri prihvatu prekida potrebno je spremiti kontekst prekinute dretve, a pri povratku obnoviti kontekst neke dretve (prekinute ili neke druge), tj. postoji cijena, dodatni poslovi (*overhead*) ovog pristupa ("kućanski poslovi")
   - nekoliko načina prihvata prekida:

   2.1. bez prekidanja započete obrade (bez sklopa, bez prioriteta)
      - kad se ustanovi tko traži prekid (ispitnim lancem) krenuti u njegovu obradu i dok ona ne završi ne prihvaćati nove zahtjeve

      \+ ne treba sklop za prihvat prekida

      − ali bitni događaji (zahtjevi za prekid) mogu duže (predugo) čekati

   2.2. programski prihvat prekida prema prioritetu (bez sklopa)
      - uz dodatnu strukturu podataka pratiti što se događa i na novi zahtjev pravilno reagirati – samo zapisati novi zahtjev ako je manjeg prioriteta od onog što procesor radi, ili odmah i započeti obradu

      \+ ne treba sklop a ipak se prekidi obrađuju prema prioritetu

      − malo više kućanskog posla

   2.3. sa sklopom za prihvat prekida
      - sklop prati prioritet onog što procesor radi i kad dođe zahtjev veće prioriteta prosljeđuje ga procesoru; u protivnom takav zahtjev čeka (sklop ga zadržava)

      − treba sklop

      \+ prekidi se obrađuju prema prioritetu

      \+ puno manje kućanskih poslova (obzirom da dio njih obavi sklop)
3. Izravan pristup spremniku
   - pristupni sklop sam prenosi podatke u/iz memorije (bez prekida, troši po jedan sabirnički ciklus za prijenos jednog podatka)
   - sklop ima smisla kod naprava koje šalju/primaju više podataka, a tek kad su svi poslani/primljeni ide neka akcija (tek tada pristupni sklop izaziva prekid); npr. disk, mreža

   \+ efikasniji sustavi (uz gornju pretpostavku)

   − složeniji pristupni sklop i procesor

   − nisu za jednostavne naprave (npr. za tipkovnicu, miša, ...)

3.6. Prekidi generirani unutar procesora, poziv jezgre

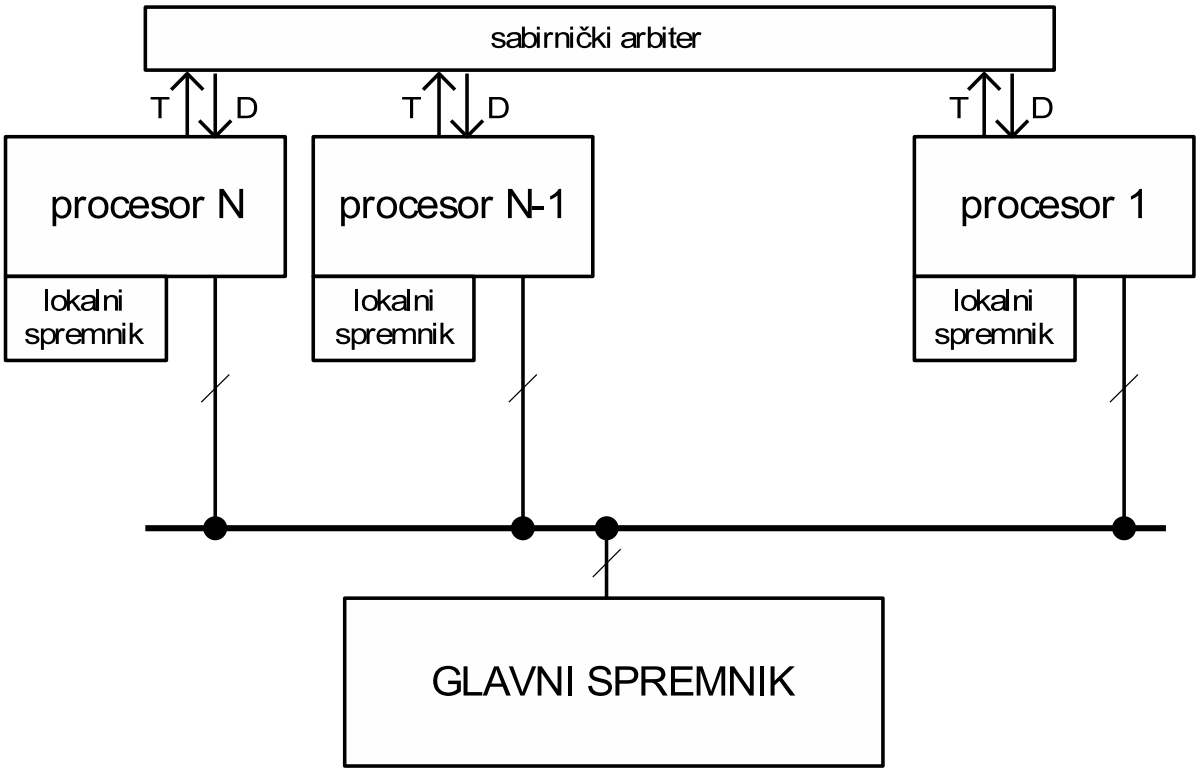
1. Pri radu, procesor može izazvati razne greške kao što su:
   - dijeljenje s nulom,
   - nepostojeća adresa (operanda, instrukcije),
   - nepostojeći operacijski kod,
   - instrukcija se ne može izvesti s trenutnim ovlastima
   - ...

Dretvu koja je izazvala takvu grešku treba zaustaviti (ne može se oporaviti od takve greške). Mehanizam prekida je prikladan za takvu operaciju: procesor sam izaziva prekid, a u obradi prekida operacijski sustav prekida dretvu i miče ju iz sustava.

2. Kako dretva obavlja operacije koje zahtijevaju veće privilegije (privilegirane instrukcije)?
   - ona namjerno izaziva prekid – *programski prekid* te se poziva *jezgrina funkcija* (5. poglavlje)
   - jezgrine funkcija = unaprijed pripremljena, dio sustava

3.7. Višeprocesorski (sabirnički povezani) sustavi

Ideja: proširiti DMA pristup, svaki DMA sklop zamijeniti procesorom, a "stari" procesor arbiterom (upravljačem) sabirnice



Slika 3.11. Višeprocesorski sustav

T – traženje sabirnice (BREQ – Bus Request)

D – dodjela sabirnice (BACK – Bus Acknowledge)

- sabirnicu dijele svi procesori
- lokalni (priručni) spremnici procesora se koriste da se smanji potreba za sabirnicom

- slika 3.11. prikazuje programski model višeprocesorskog sustava (koji vrijedi i za stvarne procesore iako su oni fizički drukčije ostvareni)

[dodatno]

Pojmovi:

- simetrični višeprocesorski sustavi – SMP (symmetric multiprocessing) – memorija je zajednička za sve procesore
- homogeni više procesorski sustavi – svu su procesori jednaki
- NUMA – Non-Uniform Memory Access – spremnik je raspodijeljen (po procesorskim karticama), pristup pojedinom spremniku nije jednako brz – ovisi o procesoru s kojeg zahtjev dolazi i memoriji koju ta instrukcija traži

Više o višeprocesorskim sustavima u okviru predmeta Napredni operacijski sustavi.

3.8. Prekidi u “stvarnim sustavima”? (info)

Primjer Intelove arhitektura (pojednostavljeno)

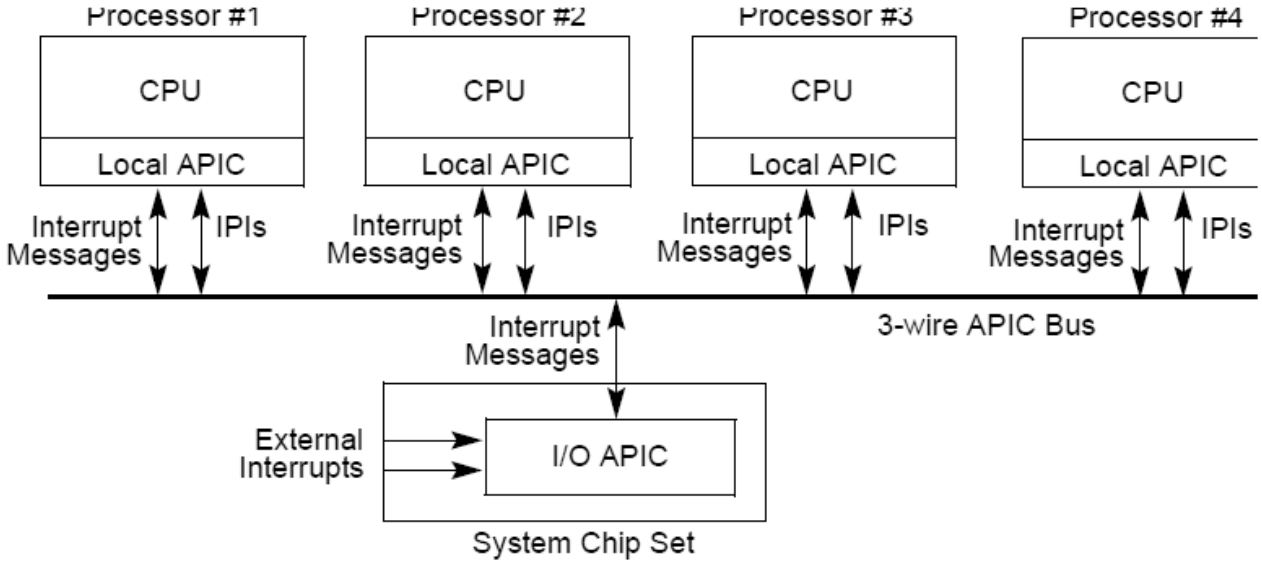
a) prihvat prekida od strane procesora

- obrada prekida se programira preko zasebne tablice – IDT (interrupt description table)
- u registar procesora IDTR se stavlja adresa te tablice (IDTR dostupan samo u prekidnom načinu rada procesora))
- kad se dogodi prekid, uz prekid dolazi i informacija o uzroku – broj prekida $\rightarrow N$
- broj prekida se koristi kao indeks za IDT, uzima se N-ti redak i tamo piše koju funkciju treba pozvati za obradu tog prekida

b) izvori prekida – “prosljeđivanje prekida do procesora” (novije APIC sučelje)

(APIC – advanced programmable interrupt controller)

- uz procesor nalazi se lokalni APIC sklop (Local APIC) koji:
  - prima i prosljeđuje “lokalne” prekide procesoru:
    - lokalno spojeni uređaji, prekidi lokalnog brojila (sata), greške, . . .
  - prima poruke o prekidima uređaja spojenih na I/O APIC
  - prima/šalje poruke od/prema lokalnih APIC-a drugih procesora (u višeproc. sustavima)
- u sustavu postoji I/O APIC sklop na koji su spojeni "vanjski" izvori prekida
  - s lokalnim APIC-ima komunicira preko sabirnice (šalje poruke o prekidima)
  - može se programirati: koje prekide prosljeđuje i kome



Slika 3.12. Sklopovlje za prihvat i prosljeđivanje prekida u Intelovim arhitekturama (izvor Intel)

Upravljanje prekidima naprava – zabrana prihvata prekida:

- na razini procesora
- na razini lokalnog APIC-a
- na razini I/O APIC-a
- na razini naprave (nju se isto može programirati da ne generira zahtjeve za prekid)

Prihvat prekida sa strane operacijskih sustava

- podjela obrade u dva dijela:
  - prvi, odmah po primitku prekida, kraći, uz zabranjeno prekidanje
  - drugi, duži, naknadno
- Windows: ISR (Interrupt Service Routine) + IST (Interrupt Service Thread)
- Linux: Top half, Bottom half

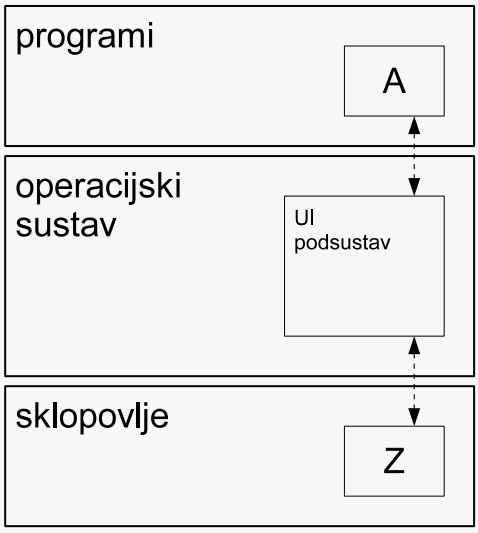
3.9. "Upravljački programi" (engl. *device drivers*) (info)

- predstavljaju skup funkcija koje omogućuju korištenje pojedinog sklopovlja
- primjerice za neku jednostavnu UI napravu upravljački program bi minimalno trebao imati funkcije:
  - `inicijaliziraj`
    - početna inicijalizacija naprave
  - `status`
    - dohvat stanja naprave (ima li novi podatak, može li se slati, ...)
  - `pošalji`
    - slanje podataka prema napravi
  - `pročitaj`
    - čitanje podataka od naprave

- OS pri inicijalizaciji upravljačkog programa za neku napravu treba registrirati prekid koji naprava generira te u obradi tog prekida pozvati funkcije upravljačkog programa
  - primjerice, OS treba imati funkciju `registriraj_prekid(id, obrada)`
  - u registriranoj funkciji (`obrada`) treba na osnovi dobivenih podataka (koje daje prekidni podsustav) pogledati što se dogodilo s napravom koja je izazvala prekid te pozvati odgovarajuću proceduru

Primjer 3.8. Primjer korištenja naprave kroz upravljačke programe (info)

kako?



"A" koristi sučelje OS-a:

```
pošalji ( id, podatak );
pročitaj ( id, podatak );
```

kako?

OS koristi upravljački program naprave "Z":

```
Z.pošalji ( podatak );
Z.pročitaj ( podatak );
Z.status ();
```

Slanje prema napravi (1)

(u programu)

```
pošalji ( id, podatak );
```

(u jezgrenoj funkciji)

```
Z = nađi_napravu (id);
ako je Z.status () == MOŽEŠ_SLATI
        Z.pošalji ( podatak );
```

Slanje prema napravi (2)

(u programu)

```
pošalji ( id, podatak );
```

(u jezgrenoj funkciji)

```
Z = nađi_napravu (id);
ako je Z.status () == NE_MOŽEŠ_SLATI
        stavi_u_IZ_MS ( Z, podatak );
```

Čitanje s naprave (1)

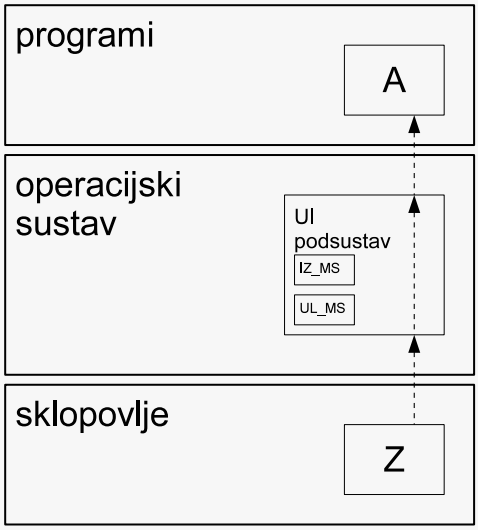
(u programu)

```
pročitaj ( id, podatak );
```

(u jezgrenoj funkciji)

```
Z = nađi_napravu (id);
ako je UL_MS(Z) neprazan
        pročitaj_UL_MS ( Z, podatak );
```

Čitanje s naprave (2)



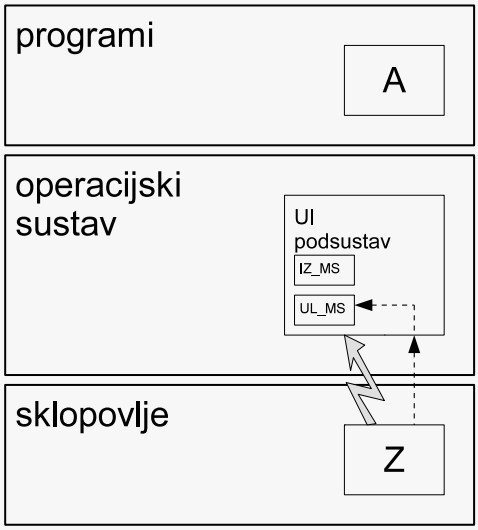
(u programu)

```
pročitaj ( id, podatak );
```

(u jezgrenoj funkciji)

```
...
inače ako je Z.status () == IMA_PODATAKA
        Z.pročitaj ( podatak );
inače
        vrati 0; //nema podataka
```

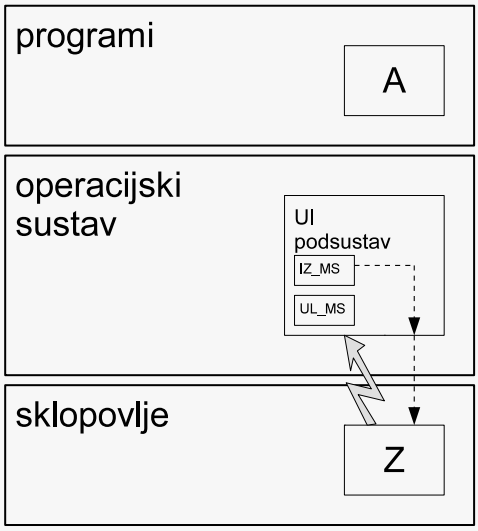
Prekid naprave Z (1)



(u obradi prekida naprave)

```
Z = nađi_napravu (id_prekida);
ako je Z.status () == IMA_PODATAKA
        Z.pročitaj ( UL_MS );
```

Prekid naprave Z (2)



(u obradi prekida naprave)

```
...
inače ako je Z.status () == SVE_POSLANO
        ako je IZ_MS neprazan
                Z.pošalji ( IZ_MS );
        //inače: nema ništa za slanje
//inače: prekid iz drugih razloga ...
```

Isječak kôda 3.5. Primjer prekidnog podsustava i korištenja upravljačkog programa (info)

```
/* prihvat prekida vanjskih naprava ("prekidni podsustav") */
Prekidni_potprogram() /* slično prethodno opisanim načinima prihvata */
{
    pohrani_kontekst
    idp = ustanovi_uzrok_prekida()
    za svaku napravu "upp" koja je registrirana za prekid "idp"
    {
        status = upp.status()
        ako je (status == PRISTIGLI_NOVI_PODACI) tada
        {
            upp.pročitaj(UL_MS)
            (pogledaj kome podaci trebaju ...)
        }
        inače ako je (status == MEĐUSPREMNIK_PRAZAN) tada
        {
            ako je (IZ_MS neprazan)
                upp.pošalji(IZ_MS)
        }
        inače ako je ...
    }
    povratak_iz_prekida
}
/* dodavanje naprave s upravljačkim programom */
os_dodaj_napravu(upravljački_program upp)
{
    upp.inicijaliziraj()
    registriraj_prekid(dohvati_id_prekida(upp), upp)
}
```

Prekidni podsustav je dio operacijskog sustava. Program može tražiti komunikaciju s nekom napravom (preko OS-a), ali i tražiti ‘da bude obaviješten’ kad se s napravom nešto dogodi. Kao reakciju na prekid naprave operacijski sustav može poslati poruku/signal nekom procesu. Primjerice, pritisak na tipku, pomak miša i sl. operacijski sustav može prosljediti programu (prozoru u fokusu) kao poruku.

Više o ulazno-izlaznim napravama u okviru predmeta Napredni operacijski sustavi.

Pitanja za vježbu 3

1. Navesti načine upravljanja ulazno-izlaznim napravama u računalnom sustavu (programska izvedba). Vrlo kratko opisati svaki od načina.
2. Čemu služi pristupni sklop? Od kojih se elemenata minimalno sastoji? Što je to “dvožično rukovanje”?
3. Što su to i čemu služe "prekidi"?
4. Zašto su potrebni različiti načini rada procesora (korisnički i sustavni/prekidni/nadgledni)?
5. Kako procesor prihvaća prekide (postupak prihvata)?
6. Što je to sklop za prihvat prekida? Koje prednosti donosi njegovo korištenje?
7. Na koje sve načine se mogu prihvaćati i obrađivati prekidi?
8. Što su to “upravljački programi” (engl. *drivers*)?
9. Koji se problemi javljaju pri/zbog obrade prekida? (zabrana prekidanja, prioriteti)
10. Koji su sve izvori prekida? (izvan procesora, prekidi izazvani u procesoru-koji?)
11. Skicirati ostvarenje višeprocesorskog sustava. Kako se upravlja zajedničkom sabirnicom u takvom sustavu? Čemu služe priručni spremnici uz procesor?
12. U nekom sustavu javljaju se prekidi P1 u 5. i 9. ms, P2 u 2. ms te P3 u 4. i 11. ms. Prioritet prekida određen je brojem (P3 ima najveći prioritet). Obrada svakog prekida traje po 2 ms. Grafički prikazati aktivnosti procesora u glavnom programu (GP), procedurama za obradu prekida (Pi) te procedurama za prihvat prekida (PP) i povratak iz prekida (PiP) i to:
    a) u idealnom slučaju
    b) bez sklopa za prihvat prekida, obrada uz zabranjeno prekidanje, uz trajanje prihvata prekida (PP) od 1 ms te 0,5 ms za povratak iz prekida (PiP)
    c) bez sklopa ali s programskom potporom, uz trajanje prihvata prekida (PP) od 1,5 ms te 1 ms za povratak iz prekida (PiP)
    d) sa sklopom za prihvat prekida, uz trajanje prihvata prekida (PP) od 0,5 ms te 0,5 ms za povratak iz prekida (PiP).

    Odrediti stanje sustava i vrijednosti korištenih struktura podataka u t=8.5 ms.
13. U nekom sustavu sa sklopom za prihvat prekidajavljaju se prekidi P1 u 0. ms, P3 u 4. ms, P2 se javlja u 6. ms. Prioritet prekida određen je brojem (P3 ima najveći prioritet). Obrada svakog prekida traje po 4 ms. Grafički prikazati aktivnosti procesora u glavnom programu (GP), procedurama za obradu prekida (Pi) te procedurama za prihvat prekida (PP) i povratak iz prekida (PiP) uz trajanje prihvata prekida od 0,5 ms (PP) te trajanje povratka iz prekida od 0,5 ms (PiP).
14. Usporediti svojstva sustava za upravljanje UI napravama korištenjem radnog čekanja, prekida te metode izravnog pristupa spremniku za sustav kod kojeg preko jedne UI naprave prosječno dolazi novi podatak svakih 0,1 ms, a sabirnica radi na 25 MHz.

4. MEĐUSOBNO ISKLJUČIVANJE U VIŠEDRETVENIM SUSTAVIMA

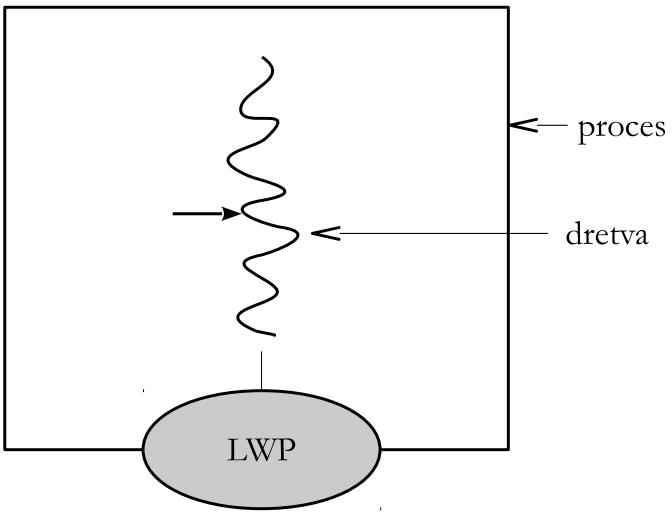
4.1. Osnovni pojmovi – program, proces, dretva

Program je statični niz instrukcija, nešto što je pohranjeno na papiru, disketi, memoriji

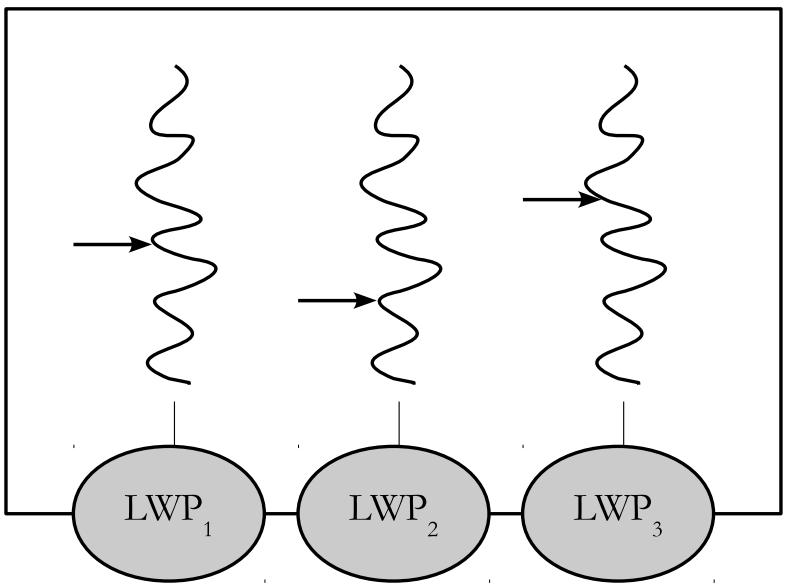
Proces je:

- skup računalnih resursa koji omogućuju izvođenje programa ili
- okolina u kojoj se program izvodi ili
- "sve što je potrebno" za izvođenje programa.

Dretva je niz instrukcija koji se izvodi. Proces se sastoji od *barem* jedne dretve



Slika 4.1. Tradicionalni proces



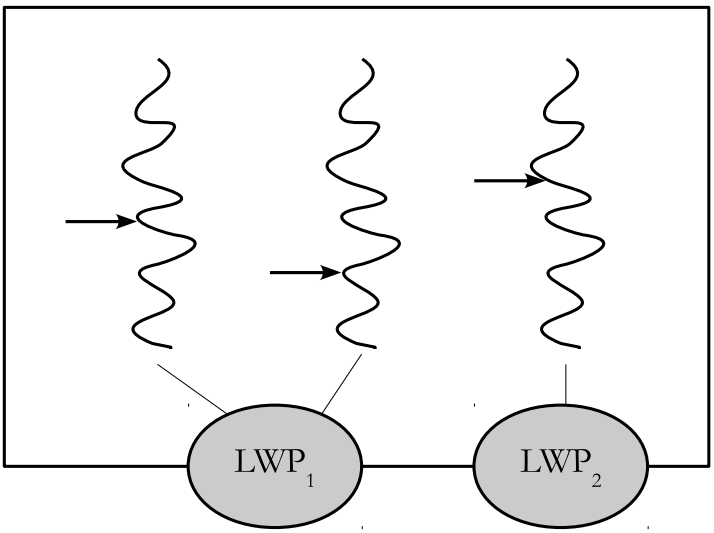
Slika 4.2. Moderan proces

LWP:

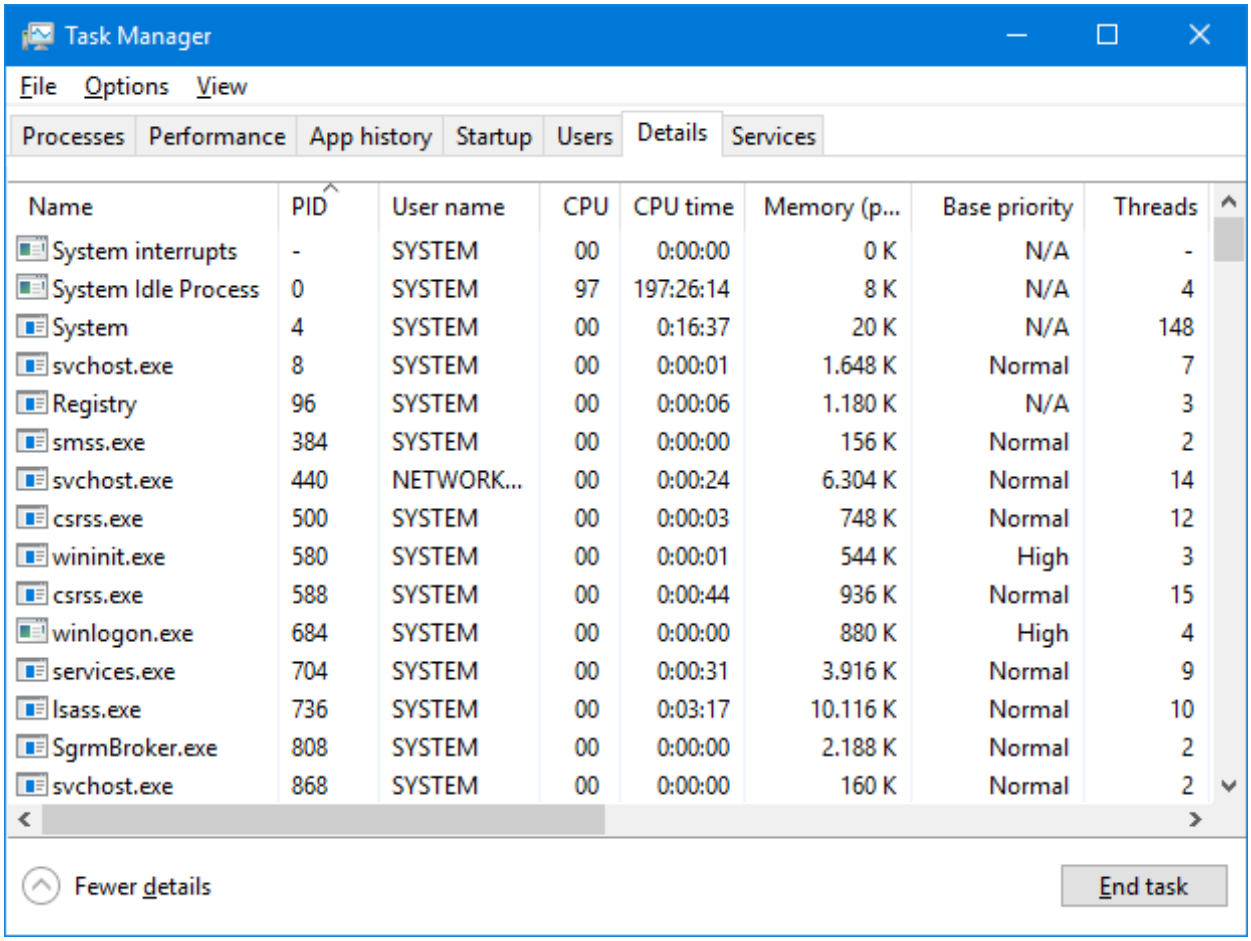
- laki (engl. *lightweight*) proces, virtualni procesor, ono što OS vidi kao dretvu procesa
- u većini slučajeva LWP je isto što i dretva procesa

Danas: proces se sastoji od *barem* jedne dretve

Uobičajeno je da OS vidi i upravlja svim dretvama. Međutim ima i iznimaka kada se nekim dretvama upravlja unutar procesa – OS ih ne vidi sve (npr. *fiber*).



Slika 4.3. Proces s vlastitim upravljanjem dretvi



Slika 4.4. Primjer popisa dijela procesa s raznim parametrima i brojem dretvi u zadnjem stupcu

Skup zauzetih sredstava je isti za sve dretve istog procesa.

- postoji zajednički spremnik (proces)
  - cijeli adresni prostor (proces) je “zajednički spremnik” (programski gledano, najčešće se to osjeti u korištenju globalnih varijabli koje su “globalne” za sve dretve)
- komunikacija među dretvama je znatno brža (koriste se globalne varijable)
- komunikacija među dretvama istog procesa može se odvijati i bez uplitanja OSa

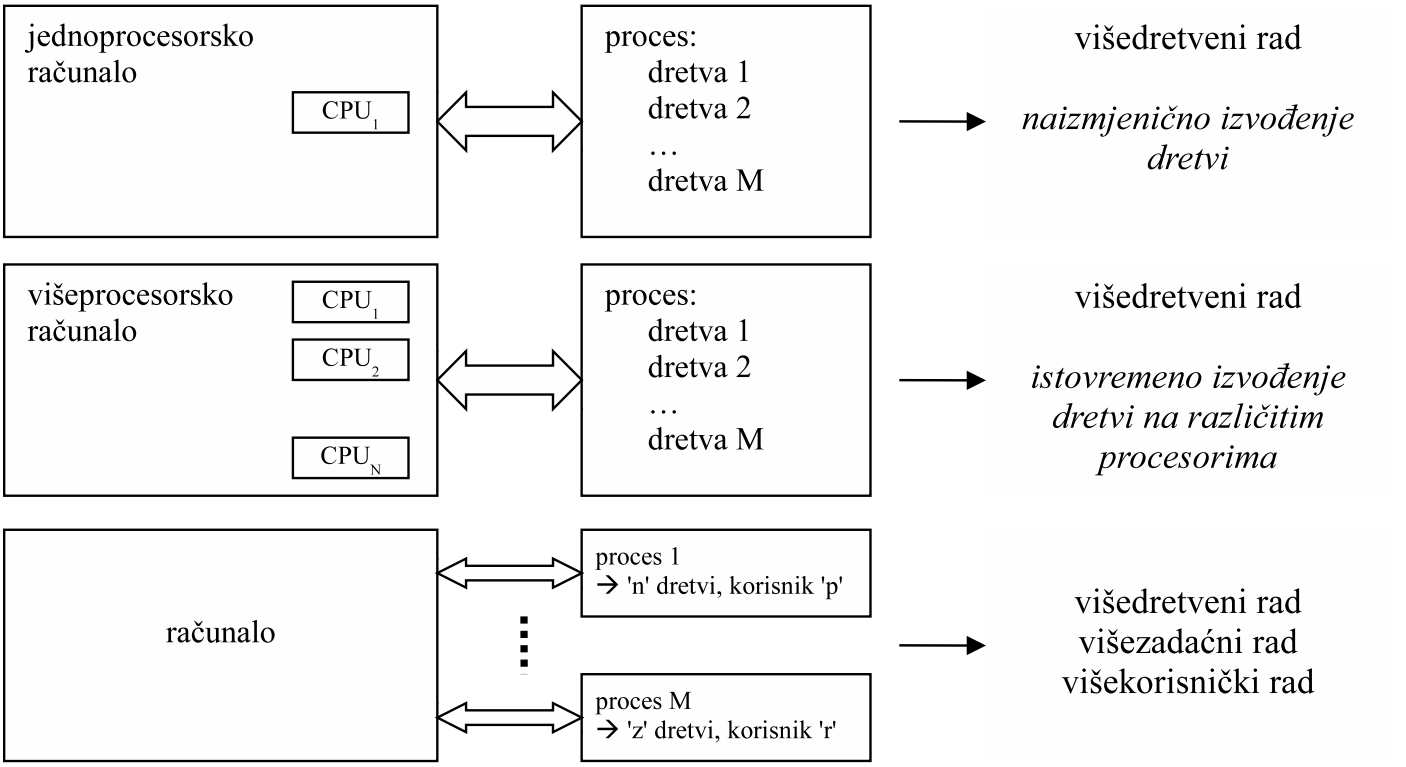
4.1.1. Višedretvenost

Višedretveni rad – izvođenje više dretvi u paraleli (ne nužno i istovremeno)

Višezadaćni rad – više zadataka (različitih problema/programa) odjednom. Svaki zadatak izvodi se u zasebnom procesu, svaki s barem jednom dretvom

Kod *višeprocesorskih računala* više dretvi se može paralelno izvoditi na različitim procesorima

Kada imamo više dretvi od procesora višedretveni rad ostvaruje se *naizmjeničnim radom dretvi* na dostupnim procesorima



Slika 4.5. Usporedba raznih višedretvenih sustava

Današnji (moderni) operacijski sustavi su i višedretveni i višezadaćni i višekorisnički (pogledati procese na UNIX/Win32 sustavima s *ps / Task Manager-om*).

4.1.2. Zašto koristiti višedretvenost?

Nedostaci:

- višedretvenost je skupa za OS, podrška je jako složena!!! (kao što će se i vidjeti u nastavku)
- višedretveno programiranje je složeno i podložno greškama
- potrebni su mehanizmi sinkronizacije i komunikacije (s kojima treba pažljivo)

Korist:

- višezadaćnosti – više poslova paralelno
- učinkovito korištenje računala – paralelno koristiti elemente računala
  - iskoristiti višeprocesorske sustave – intenzivni računalni problemi koji se daju rastaviti na bar djelomično neovisne dijelove
  - dok jedna dretva čeka dovršetak UI operacije (naprava radi za dretvu), procesor izvodi drugu dretvu (npr. kod poslužitelja jedna dretva čeka da se učitaju traženi podaci s diska, druga čeka da se dohvate svi podaci preko mreže, treća se izvodi na procesoru)
- složeni sustavi – načelom “podijeli i vladaj” smanjuje se složenost odvajanjem zasebnih aktivnosti u zasebne dretve (ali oprezno!)
- različite dretve upravljaju različitim elementima sustava – potrebni su različiti prioriteti i načini raspoređivanja

4.2. Višedretveno ostvarenje zadatka – zadatak i podzadaci

Svaki se zadatak, barem i "umjetno" može podijeliti na podzadatke, ako to nije očito iz strukture zadatka (dijelovi, paralelna obrada, ...)

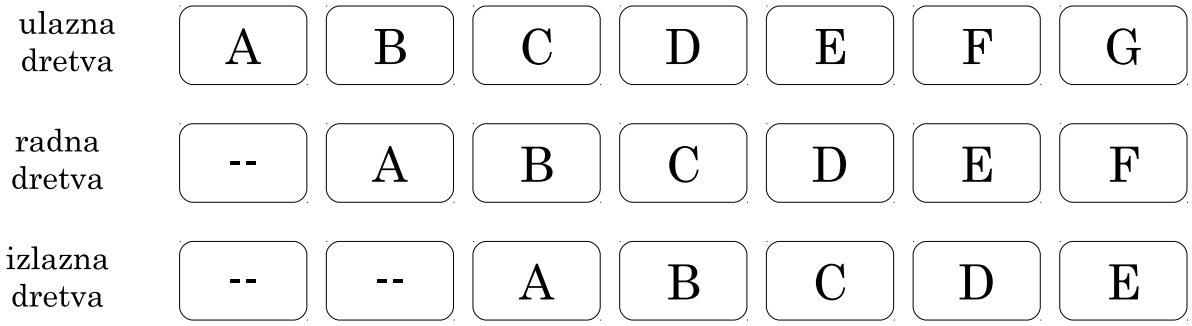
Jedna od mogućih podjela zadatka na podzadatke analogna protočnoj strukturi procesora je prikazana u nastavku.

Zadatak se može podijeliti na:

- podzadatak za čitanje ulaznih podataka – ulazna dretva
- podzadatak za obradu – radna dretva
- podzadatak za obavljanje izlaznih operacija – izlazna dretva

Navedena podjela može doprinijeti učinkovitosti sustava jer paralelno rade: ulazna jedinica, procesor i izlazna jedinica.

Korištenjem navedene podjele, zadatak se može izvoditi načelom cjevovodnog rada



Slika 4.6. Načelo cjevovodnog rada dretvi

Problem: razmjena podatka između dretvi kada dretvama treba različito vrijeme za odradu posla. Npr. radna dretva treba prije preuzimanja podataka od ulazne dretve pričekati da ulazna dovrši dohvat tih podataka. Isto tako ulazna dretva treba prije predaje podataka radnoj dretvi pričekati da radna dretva dovrši započetu obradu (rad na prethodno predanim podacima). I slično.

Dretve je potrebno uskladiti, tj. *sinkronizirati*!

Mnogi zadaci se mogu (smisleno) rastaviti na podzadatke, npr.: $Z_1 \rightarrow Z_2 \rightarrow \cdots \rightarrow Z_N$.

Ipak, i u takvim slučajevima potrebno je sinkronizirati zadatke – urediti slijed njihova izvođenja – tko prije a tko poslije.

Problemi sinkronizacije:

- Kako ostvariti mehanizme sinkronizacije?
- Koje je dretve potrebno sinkronizirati?

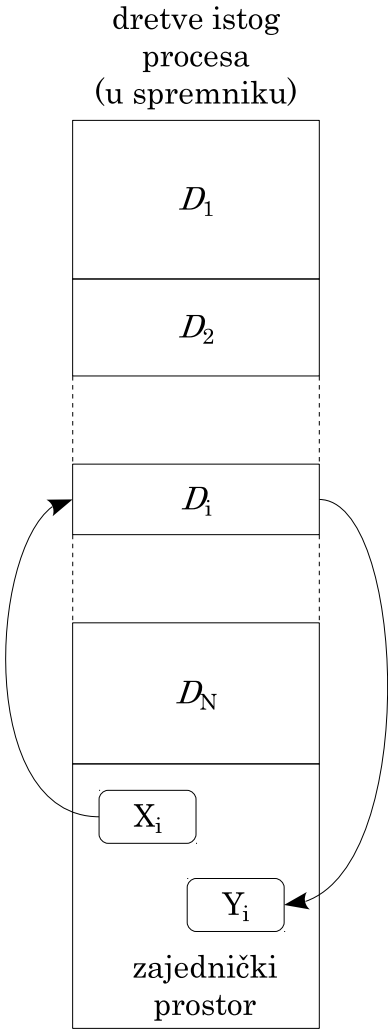
Postoje različiti oblici sinkronizacije. Za prethodne probleme potreban je mehanizam koji će jednu dretvu zaustaviti dok joj druga ne signalizira da može nastaviti. Jedan od sinkronizacijskih mehanizama za takvu sinkronizaciju jest semafor. Semafor se razmatra u idućem poglavlju, dok je ovdje prikazan jednostavniji sinkronizacijski mehanizam: *međusobno isključivanje*.

Svaki podzadatak se izvodi u svojoj dretvi. U nastavku se ponegdje umjesto podjele *zadatak⇒podzadaci* koristi *sustav zadataka⇒zadatak*, ali je načelo jednako.

4.3. Model višedretvenosti, nezavisnost dretvi

Pretpostavke:

- sve su dretve unutar istog procesa – dijele njegov spremnički prostor
- svaka dretva ima skup instrukcija, skup podataka te vlastiti stog
- postoji zajednički spremnički prostor koji dretve koriste pri rješavanju zadatka



Slika 4.7. Dretva, domena i kodomena

Svaki zadatak ima svoju:

- domenu X_i (iz koje samo čita) te
- kodomenu Y_i (koju mijenja)

tj. zadatak se može funkcijski opisati kao preslikavanje: $Y_i = f(X_i)$

Kada se dva zadatka (dretve koje ih izvode) mogu izvoditi paralelno, a kada ne?

Dva su podzadatka *nezavisna* ako nemaju nikakvih zajedničkih spremničkih lokacija ili imaju presjeka samo u domenama.

Uvjet nezavisnosti podzadataka, odnosno dretvi D_i i D_j:

$$(X_i \cap Y_j) \cup (X_j \cap Y_i) \cup (Y_i \cap Y_j) = \phi \tag{4.1.}$$

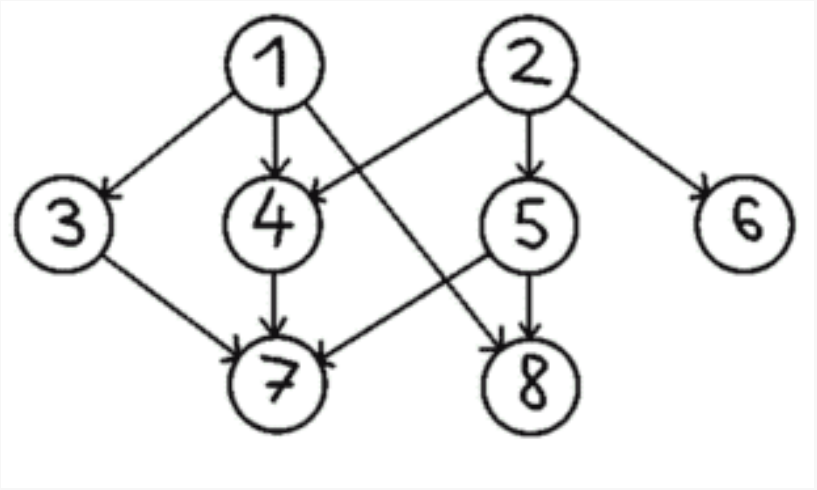
Ako su podzadaci *zavisni* tada se mora utvrditi redoslijed izvođenja njihovih dretvi.

Ako su podzadaci *nezavisni* tada se njihove dretve mogu se izvoditi *proizvoljnim* redoslijedom, pa i *paralelno*!

Primjer 4.1. Sustav zadataka zadan u obliku usmjerena grafa

Ako se neki posao može rastaviti na lanac zadataka $Z_1 \rightarrow Z_2 \rightarrow \cdots \rightarrow Z_N$, tada se razmatranjem njihovih domena i kodomena mogu ustanoviti zavisni i nezavisni zadaci te se sustav može prikazati usmjerenim grafom koji to uzima u obzir.

Npr. neki sustav zadataka se možda može prikazati kao na slici:



Slika 4.8. Primjer sustava zadataka

Na prikazanom primjeru, zadatak 1 se mora obaviti prije zadataka: 3, 4, 8 (i 7 tranzitivno).

Za svaki zadatak se može napraviti usporedba ovisnosti sa svim ostalim zadacima.

Zadaci na istim putovima su zavisni – mora se poštivati redoslijed izvođenja.

Zadaci na različitim putovima su nezavisni – mogu se izvoditi paralelno

---

Često kada dretve koriste zajednička sredstva nije potrebno utvrđivati redoslijed njihova izvođenja već je dovoljno da se osigura da ta sredstva ne koriste u istom trenutku – da se promjene nad njima ne obavljaju paralelno. Npr. ako dvije dretve žele povećati istu varijablu za jedan dovoljno ih je spriječiti da to ne rade istovremeno, redoslijed nije bitan jer je u oba slučaja ispravan.

Zadatak 4.1. Sustav zadataka (ispitni zadatak)

Sustav zadataka je zadan u obliku lanca (kada se zadaci izvode ovim redom rezultat će biti ispravan): $Z_1 \to Z_2 \to Z_3 \to Z_4 \to Z_5 \to Z_6 \to Z_7$.
Zadaci imaju domene (D) i kodomene (K) prema tablici:

Tablica 4.1. Domene i kodomene za zadatke

	Z_1	Z_2	Z_3	Z_4	Z_5	Z_6	Z_7
M_1	D	D	D			K	
M_2	K			K		D	D
M_3		K		D	K		K
M_4			K		D		
M_5							

Odrediti maksimalno paralelni sustav zadataka, uzimajući u obzir njihov međusobni odnos u lancu i domene i kodomene prema tablici 4.1.

[dodatno]

Svaki stupac/redak može imati više D-ova i K-ova, a može biti i prazan.
Svaka ćelija može biti prazna, imati D, imati K ili i oboje D i K. Ako je u ćeliji i D i K, onda ju se razmatra kao da je samo K unutra jer je on "problematičniji" pri usporedbi zadataka.

4.4. Problem paralelnog korištenja zajedničkih varijabli (engl. *race condition*)

Pristup zajedničkim sredstvima (npr. varijablama) treba zaštititi jer se paralelnim radom može doći do krivog rezultata.

Primjer 4.2. Primjer problema s paralelnim radom dretvi

Neka u sustavu postoji više dretvi koje koriste zajedničke strukture podataka. Jedna od tih struktura je varijabla `brojilo` koja označava broj poruka u međuspremniku. Ta se varijabla negdje povećava, a negdje smanjuje, kao u odsječku:

```
...
ako je (brojilo > 0) tada {
    uzmi poruku
    brojilo = brojilo - 1
}
...
```

Problem kod paralelnog korištenja varijable `brojilo` prema gornjem kodu jest da se od provjere vrijednosti (`ako je`) do akcije (kada je uvjet bio ispunjen) zbog stanja te vrijednosti u sustavu svašta može dogoditi.

Za gornji primjer se može dogoditi da dvije dretve paralelno provjere vrijednost brojila. U slučaju da je njegova vrijednost bila jedan, obje bi mogle ući u `ako je` dio koda te pokušati uzeti poruku. Kako brojilo broji poruke, samo je jedna poruka na raspolaganju te će jedna od dretvi (ona druga) napraviti grešku u dijelu `uzmi poruku` (ovisno o izvedbi reda, dohvatit će smeće ili staru poruku i pritom unijeti grešku u tu strukturu podataka).

Rješenje navedena problema ostvaruje se tako da se dio koda koji koristi zajedničke varijable zaštiti od istovremenog korištenja. Npr. prije `ako je` staviti ogradu koja će zabraniti ulazak više od jedne dretve u kod iza nje.

```
...
ulaz_u_kritični_dio_koda
ako je (brojilo > 0) tada {
    uzmi poruku
    brojilo = brojilo - 1
}
izlaz_iz_kritična_dijela_koda
...
```

Često nam se čini da je scenarij u kojem će se dogoditi ovakav problem vrlo malo vjerojatan – puno toga se mora poklopiti između različitih dretvi. Međutim, dretve mogu paralelno obavljati jako puno ponavljanja problematična koda (u petlji) te vjerojatnost pojave problematična scenarija značajno naraste – najčešće je vjerojatnije da će se on dogoditi nego da neće.

4.5. Međusobno isključivanje

Međusobno isključivanje (MI) je najjednostavniji mehanizam sinkronizacije.

Odsječke koda koji koriste zajednička sredstva nazivamo *kritičnim odsječcima* i njih treba zaštititi sinkronizacijskim mehanizmom međusobnog isključivanja.

Izvorni kod dretve se stoga može podijeliti na kritične odsječke i nekritične odsječke, primjerice:

```
dretva_x
{
    ...
    nekritični odsječak
    kritični odsječak
    nekritični odsječak
    kritični odsječak
    ...
}
```

Radi jednostavnosti u nastavku se razmatraju cikličke dretve koje imaju jedan kritičan odsječak

```
ciklička_dretva
{
    ponavljaj {
        kritični odsječak
        nekritični odsječak
    }
    do zauvijek
}
```

Primjer 4.3. Poslužiteljska dretva

U nekom poslužitelju dretve koje poslužuju zahtjeve mogu se modelirati sljedećim kodom:

```
dretva_poslužitelja //jedna od "radnih dretvi"
{
    ponavljaj {
        uzmi_idući_zahtjev_iz_reda // kritični odsječak
        obradi_zahtjev_i_vrati_rezultat //nekritični odsječak
    }
    do kraja_rada_poslužitelja
}
```

Kako ostvariti kritični odsječak?

- koristiti mehanizme međusobnog isključivanja – funkcije `uđi_u_KO` i `izađi_iz_KO`

```
ciklička_dretva
{
    ponavljaj {
        uđi_u_KO()
        kritični odsječak
        izađi_iz_KO()
        nekritični odsječak
    }
    do zauvijek
}
```

Kako ostvariti te funkcije (`uđi_u_KO` i `izađi_iz_KO`)? Koja svojstva moraju one imati?

Zahtjevi na algoritme međusobnog isključivanja (ispitno pitanje)

1. U kritičnom odsječku u svakom trenutku smije biti najviše jedna dretva.
2. Mehanizam međusobnog isključivanja mora djelovati i u uvjetima kada su brzine izvođenja dretvi proizvoljne.
3. Kada neka od dretvi zastane u svom nekritičnom dijelu ona ne smije spriječiti ulazak druge dretve u svoj kritični odsječak.
4. Izbor jedne od dretvi koja smije ući u kritični odsječak treba obaviti u konačnom vremenu.

Algoritam mora vrijediti i za jednoprocesorske i za više procesorske sustave (tj. za sve sustave u kojima se želi primijeniti).

4.6. Potraga za algoritmima međusobnog isključivanja

ZAŠTO se "traže" kad se zna koji valjaju?
ZATO da se usput pokažu problemi višedretvenih sustava!

Opće pretpostavke:

- višeprocesorski sustav (barem 2 procesora)
- dretve su u istom procesu (dijele adresni prostor)

Neka se prvo "pronađu" algoritmi koji rade za dvije dretve. Ako algoritam ne radi za dvije dretve neće ni za više!

Koristit će se radno čekanje jer trenutno nije prikazano bolje rješenje (preko OS-a).

4.6.1. Prvi pokušaj (`ZASTAVICA`)

Koristi se zajednička varijabla `ZASTAVICA`

- kada je `ZASTAVICA == 0`, nitko nije u KO
- kada je `ZASTAVICA == 1`, jedna dretva je u KO te druga neće ući već će radno čekati da se ta varijabla promijeni

```
uđi_u_KO ()
{
    ponavljaj
        pročitaj varijablu ZASTAVICA
    sve dok je (ZASTAVICA == 1)

    ZASTAVICA = 1
}
```

```
izađi_iz_KO ()
{
    ZASTAVICA = 0
}
```

U kodu (ili pseduokodu) svako korištenje varijable podrazumijeva da se ona mora učitati iz spremnika. Stoga će u nastavku biti izostavljen dio `pročitaj varijablu` i svako pojavljivanje varijable će podrazumijevati da se ta varijabla prvo mora dohvatiti.

```
uđi_u_KO ()
{
    dok je (ZASTAVICA == 1)
        ;

    ZASTAVICA = 1
}
```

```
izađi_iz_KO ()
{
    ZASTAVICA = 0
}
```

U asembleru `uđi_u_KO`:

```
1  uđi_u_KO:
2      ADR R0, ZASTAVICA
3  petlja:
4      LDR R1, [R0] //pročitaj varijablu ZASTAVICA
5      CMP R1, 1    //ZASTAVICA == 1 ?
6      BEQ petlja
7      STR 1, [R0]  //ZASTAVICA = 1
8      RET
```

Problemi:

- ako dretve rade paralelno, u uzastopnim sabirničkim ciklusima mogu izvesti instrukciju s linije 4 – obje mogu pročitati 0 i obje ući u KO
- iako na prvi pogled izleda malo vjerojatno da će dretve ovaj kod izvoditi baš paralelno, treba uzeti u obzir da takve petlje procesor može izvesti milijune (i više) puta u sekundi – i vrlo mala vjerojatnost jednog događaja se ovako znatno poveća – stoga je često vjerojatnost da će se ovo dogoditi veća nego da se neće dogoditi
- nije ispunjen osnovni uvjet (1) (ostali jesu, ali moraju biti svi)

4.6.2. Drugi pokušaj (PRAVO)

Koristiti varijablu `PRAVO` koja može imati vrijednost 0 ili 1 što je indeks dretve koja iduća može ući u KO (`I = 1 - J; J = 1 - I;`)

```
uđi_u_KO (I) // dretva I želi ući u KO
{
    dok je (PRAVO != I)
        ;
}
```

```
izađi_iz_KO (I)
{
    PRAVO = 1 - I
}
```

Problemi:

- ulazak u KO je strogo naizmjeničan
- uvjeti 2 i 3 nisu ispunjeni!

4.6.3. Treći pokušaj (`ZASTAVICA[2]++`)

Koristiti dvije varijable `ZASTAVICA[I]` i `ZASTAVICA[J]` koje mogu imati vrijednost 0 ili 1 što označava jesu li zadane dretve u KO ili nisu

```
uđi_u_KO (I)
{
    J = 1 - I //druga dretva

    dok je (ZASTAVICA[J] != 0)
        ;
    ZASTAVICA[I] = 1
}
```

```
izađi_iz_KO (I)
{
    ZASTAVICA[I] = 0
}
```

Problem: u paralelnom radu obje dretve mogu pročitati 0 u suprotnim zastavicama i ući u KO

4.6.4. Četvrti pokušaj (`++ZASTAVICA[2]`)

Zastavicu postaviti prije provjere suprotne zastavice

```
uđi_u_KO (I)
{
    J = 1 - I
    ZASTAVICA[I] = 1

    dok je (ZASTAVICA[J] != 0)
        ;
}
```

```
izađi_iz_KO (I)
{
    ZASTAVICA[I] = 0
}
```

Problem: u paralelnom radu obje dretve mogu prvo postaviti svoje zastavice i onda u idućoj petlji beskonačno radno čekati jer se zastavice neće spustiti – nikad neće ući u KO (4) – *potpuni zastoj*

4.6.5. Peti pokušaj (`++ZASTAVICA[2]--`)

Privremeno spustiti zastavicu dok je ona druga podignuta

```
uđi_u_KO (I)
{
    J = 1 - I
    ZASTAVICA[I] = 1

    dok je (ZASTAVICA[J] != 0) {
        ZASTAVICA[I] = 0
        dok je (ZASTAVICA[J] != 0)
            ;
        ZASTAVICA[I] = 1
    }
}
```

```
izađi_iz_KO (I)
{
    ZASTAVICA[I] = 0
}
```

Problem: u paralelnom radu obje dretve mogu sinkrono podizati i spuštati zastavice i nikad ne ući u KO (4). Jedan mali "poremećaj" sinkronog rada može biti dovoljan da jedna "prođe".

4.6.6. Dekkerov algoritam

- rješenje koje je ponudio nizozemski matematičar T. Dekker, a opisao E.W.Dijkstra 1959.
- kombinacija drugog i petog pokušaja

```
uđi_u_KO (I)
{
    J = 1 - I

    ZASTAVICA[I] = 1

    dok je (ZASTAVICA[J] != 0) {
        ako je (PRAVO == J) {
            ZASTAVICA[I] = 0
            dok je (PRAVO == J)
                ;
            ZASTAVICA[I] = 1
        }
    }
}
```

```
izađi_iz_KO (I)
{
    ZASTAVICA[I] = 0
    PRAVO = 1 - I
}
```

Osnovna ideja algoritma: u slučaju obostrana zahtjeva za ulazak, dretva koja "nije na redu" će spustiti svoju zastavicu i pričekati onu drugu (da završi sa svojim KO).

Varijabla `PRAVO` se koristi samo kada i ona druga dretva ima podignutu zastavicu.

Bitno je napomenuti da bez obzira na `PRAVO` dretva koja želi ući u KO neće izaći iz vanjske petlje dok ona druga dretva ne spusti svoju zastavicu.

Algoritam zadovoljava sva četiri osnovna uvjeta na algoritme za MI (ispravan je).

Pojednostavljenje: Petersonov algoritam

```
uđi_u_KO (I)
{
    J = 1 - I

    ZASTAVICA[I] = 1
    PRAVO = J

    dok je (ZASTAVICA[J] != 0 && PRAVO == J)
        ;
}
```

```
izađi_iz_KO (I)
{
    ZASTAVICA[I] = 0
}
```

Osnovna ideja algoritma: u slučaju obostrana zahtjeva za ulazak, dretva koja je došla kasnije daje prednost onoj drugoj i čeka da ona druga završi sa svojim KO.

Prednosti Petersonova algoritma prema Dekkerovu:

- kraći i brži (potrebno manje instrukcija)
- ne ovisi o početnoj vrijednosti varijable PRAVO

Dekkerov i Petersonov algoritam rade za sustave sa samo dvije dretve

Proširenje algoritma za više dretvi napravio je Lamport.

4.6.7. Lamportov algoritam međusobnog isključivanja

- drugo ime: *pekarski algoritam*
- svaka dretva prije ulaska u KO dobije svoj broj, koji je za 1 veći od najvećeg do sada dodijeljenog
- u KO ulazi dretva s najmanjim brojem!
- što ako dvije dretve dobiju isti broj? onda se gleda i indeks dretve
- postupak dobivanja broja je također neki oblik KO pa se i on ograđuje s `ULAZ[]`
- zajednički podaci:
  - `BROJ[]` – dodijeljeni brojevi (0 kada dretva ne traži ulaz u KO)
  - `ULAZ[]` – štiti se dodjela broja
  - početne vrijednosti varijabli su nule

```
uđi_u_KO (I)
{
    //uzimanje broja
    ULAZ[I] = 1
    BROJ[I] = max(BROJ[1], ..., BROJ[N]) + 1
    ULAZ[I] = 0

    //provjera i čekanje na dretve s manjim brojem
    za J=1 do N
    {
        dok je (ULAZ[J] == 1)
            ; //čeka se da dretva J dobije broj, ako je u postupku dobivanja

        dok je (BROJ[J] != 0 &&
                (BROJ[J] < BROJ[I] || (BROJ[J] == BROJ[I] && J < I)))
            ; //čekaj ako J ima prednost
    }
}

izađi_iz_KO (I)
{
    BROJ[I] = 0
}
```

Svojstva Lamportova algoritma

+ **radi** za proizvoljan broj dretvi na proizvoljnom broju procesora!
− radno čekanje (kao i Dekkerov i Petersonov)
− (manji nedostatak) potrebna poveća struktura podataka

4.6.8. Problemi ostvarenja Dekkerova, Petersonova i Lamportova algoritma (info)

Za ispravan rad navedenih algoritama pretpostavlja se da procesor izvodi navedene radnje (instrukcije) jednu za drugom navedenim redoslijedom.

Međutim, moderni procesori radi postizanja ubrzanja rada dozvoljavaju izvođenje instrukcija i "preko reda" (engl. *out-of-order*). Takvo ponašanje može uzrokovati greške u algoritmu međusobnog isključivanja.

Razmotrimo dio koda Lamportova algoritma:

```
1: ULAZ[i] = 1
2: BROJ[i] = max(BROJ[1], ..., BROJ[N]) + 1
3: ULAZ[i] = 0
```

Uz pretpostavku izvođenja linija navedenim redoslijedom, Lamportov algoritam će sigurno raditi ispravno. Međutim, moderni bi procesor nakon linije 1 usporedbom međuovisnosti linije 2 i 3 mogao ustanoviti da među njima nema veze te da je ispravno izvesti ih proizvoljnim redoslijedom, npr. liniju 3 prije nego 2 (razlog može biti da mu se element ULAZ[i] nalazi u priručnom spremniku dok neki BROJ[] ne). Tom promjenom redoslijeda može se narušiti ispravnost algoritma (to onda nije Lamportov algoritam).

U ovakvim situacijama potrebno je dodatnim instrukcijama prevoditelju navesti da se redoslijed ovih naredbi treba poštivati. Jedan od načina jest korištenje podrške programskog jezika (npr. C++11 atomarne varijable) koje se onda posebno tretiraju i prevoditelj generira kod za koji se garantira navedeni redoslijed izvođenja.

Primjerice kada bismo gornje varijable definirali sa:

```
#include <stdatomic.h>
atomic_int ULAZ[N], BROJ[N];
```

redoslijed navedenih operacija bi bio očuvan.

4.7. Sklopovska potpora međusobnom isključivanju

Zabrana prekidanja – samo za jednoprocesorske sustave

Dretvu (i u KO) može prekinuti samo prekid (i u prekidu dretva može biti zamijenjena nekom drugom). Ako se prekid zabrani – ako dretva u svom izvođenju zabrani prekidanje, onda će ona raditi dok ta ista dretva ne dozvoli prekidanje.

U jednoprocesorskim sustavima bi mogli koristiti zabranu i dovolu prekidanja za ostvarenje kritična odsječka:

- `uđi_u_KO()` == `onemogući_prekidanje`
- `izađi_iz_KO()` == `omogući_prekidanje`

Problemi:

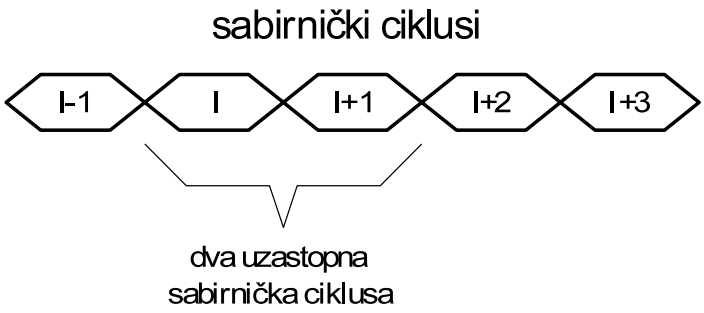
- radi samo u jednoprocesorskim sustavima
- traži privilegirani rad – zabrana i dozvola prekidanja su privilegirane operacije

Drugi načini?

Problem prvog pokušaja sa samo jednom zastavicom je bio u tome što se provjera zastavice i njeno postavljanje moglo prekinuti drugom dretvom (zajedničkim korištenjem sabirnice ili prekidom), koja je također, paralelno mogla provjeriti vrijednost zastavice.

Ako se to sklopovski onemogući, onda bi algoritam bio dobar! Kako?

Korištenje dva uzastopna sabirnička ciklusa



Slika 4.9. Korištenje dva uzastopna sabirnička ciklusa

U prvom ciklusu (I) se čita vrijednost zastavice, a u drugom (I+1) se zastavica postavlja u 1!

Sklopovske podrške u obliku instrukcija `TAS`, `SWP`, `INC`, `CAS`

4.7.1. Ostvarenje MI s instrukcijom *Ispitaj_i_postavi – TAS* (*test-and-set*)

Neka postoji **instrukcija** `TAS adresa` koja koristi dva **uzastopna** sabirnička ciklusa tako da:

1. u prvom sabirničkom ciklusu pročita vrijednost sa zadane adrese i postavlja zastavice registra stanja (npr. zastavicu Z (zero), kao da uspoređuje s nulom)
2. u sljedećem sabirničkom ciklusu na tu adresu sprema vrijednost 1

[dodatno]

Nekakav ekvivalent operacija te instrukcije bio bi:

```
TAS (adresa) = {
    //prvi dio instrukcije koristi 1. sabirnički ciklus
    LDR X, [adresa]
    CMP X, 0    //proširenje gornje instrukcije; samo pogleda je li 0

    //drugi dio instrukcije koristi 2. sabirnički ciklus (odmah iza 1.)
    STR 1, [adresa]
}
```

`X` je privremeni registar, a prva i treća naredba koriste dva uzastopna sabirnička ciklusa

Ali razlika je u tome što je TAS samo JEDNA instrukcija (ne tri)!

Rješenje MI s TAS u asembleru:

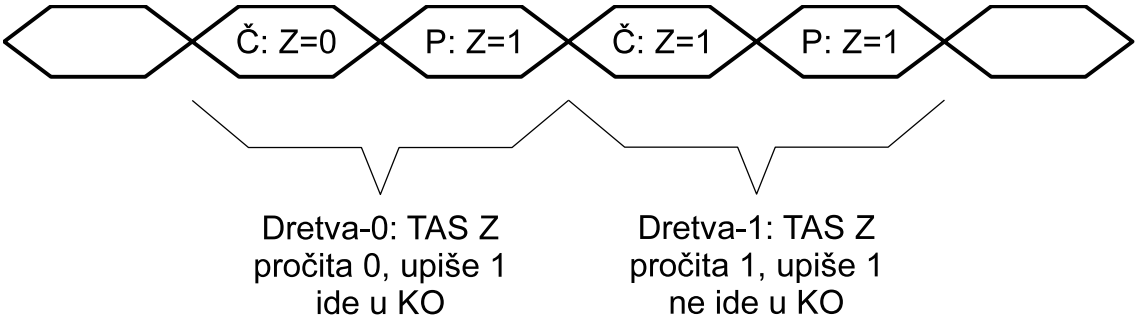
```
uđi_u_KO:
petlja:
    TAS zastavica
    BNE petlja //dok nije 0 ponovi
    RET
```

```
izađi_iz_KO:
    ADR R1, zastavica
    STR 0, [R1]
    RET
```

U pseudokodu neka `TAS(adresa)` označava izvođenje TAS instrukcije, ali neka i vraća pročitanu vrijednost (radi jednostavnosti i kraćeg zapisa)

```
uđi_u_KO ()
{
    dok je (TAS (zastavica) == 1)
        ;
}
```

```
izađi_iz_KO ()
{
    zastavica = 0
}
```



Slika 4.10. Primjer dvije dretve koje koriste TAS za MI

4.7.2. Ostvarenje MI sa *Zamijeni – SWP* (*swap* ili *XCHG – exchange*)

Neka postoji **instrukcija** `SWP R1,R2,[R3]` koja koristi dva uzastopna sabirnička ciklusa tako da:

- u prvom ciklusu dohvati u registar `R2` vrijednost s adrese zadane u registru `R3`
- u drugom ciklusu pohrani vrijednost iz registra `R1` na tu istu adresu (`R3`)

[dodatno]

Nekakav ekvivalent operacija te instrukcije bio bi:

```
SWP R1,R2,[R3] = {
    LDR R2, [R3] //1. sabirnički ciklus
    STR R1, [R3] //2. sabirnički ciklus
}
```

Ali razlika je u tome što je to JEDNA instrukcija!

Rješenje MI sa SWP u asembleru:

```
uđi_u_KO:
    ADR R3, zastavica
    LDR R1, 1
petlja:
    SWP R1, R2, [R3]
    CMP R2, 1
    BEQ petlja
    RET
```

```
izađi_iz_KO:
    ADR R1, zastavica
    STR 0, [R1]
    RET
```

Neka u psedudokodu funkcija `Zamijeni(adresa,var)` radi zamjenu vrijednosti varijabli `adresa` i registra `reg` u dva uzastopna sabirnička ciklusa.

```
uđi_u_KO ()
{
    reg = 1
    ponavljaj {
        Zamijeni(zastavica, reg)
    }
    dok je (reg == 1)
}
```

```
izađi_iz_KO ()
{
    zastavica = 0
}
```

4.7.3. Ostvarenje MI s *Usporedi_i_Zamijeni – CAS* (*compare-and-swap*)

U novijim procesorima postoji bolja inačica prethodnih instrukcija koja ne koristi drugi sabirnički ciklus ako nije potreban. Primjerice, ako smo već u prvom dijelu izvođenja instrukcije *TAS* ustanovili da je zastavica postavljena, drugi sabirnički ciklus ništa ne mijenja – zapisuje broj 1 preko broja 1. Stoga ima smisla taj drugi dio operacije izostaviti.

Naredba `CAS R1, R2, [R3]` u prvom sabirničkom ciklusu dohvaća vrijednost s adrese zadane u `R3` te uspoređuje tu vrijednost s onom u registru `R1`. Ako su vrijednosti jednake, tada koristi drugi sabirnički ciklus (uzastopni) i na istu adresu zapisuje vrijednost iz registra `R2`. Instrukcija pritom postavlja i odgovarajuće zastavice u statusnom registru (npr. zastavicu Z).

Asembler/pseudokod uz CAS sličan je kao i za SWP, ali bez instrukcije CMP.

Sučelje `pthread_spin_lock`, `pthread_spin_unlock`

POSIX definira sučelje za zaključavanje radnim čekanjem kroz navedene funkcije, a koje interno koriste neke od prethodnih operacija, ovisno o procesoru. Primjer korištenja tog sučelja (na webu je potpuni kod):

```
1  pthread_spinlock_t kljuc; //globalna varijabla
2
3  void *dretva (void *x) {
4          while (1) {
5                  pthread_spin_lock(&kljuc);
6                  ... //kritični odsječak
7                  pthread_spin_unlock(&kljuc);
8                  ... //nekritični odsječak
9          }
10 }
11 int main () {
12         ...
13         pthread_spin_init(&kljuc, PTHREAD_PROCESS_PRIVATE);
14         ...
15         pthread_create(...);
16         ...
17 }
```

4.8. Problemi prikazanih mehanizama međusobnog isključivanja

Osnovni problem svih prikazanih algoritama, sa i bez sklopovske potpore je radno čekanje – neefikasno korištenje procesora

Dodatno:

- Dekkerov i Petersonov algoritam rade samo za dvije dretve (Lamportov te ostali sa sklopovskom potporom rade za proizvoljan broj dretvi)
- manji problem algoritama ostvarenih sklopovskom potporom je nepoštivanje redoslijeda zahtijeva za ulaz u KO: prva dretva koja naleti u svom radnom čekanju na spuštenu zastavicu ulazi u KO – to može biti i zadnja dretva koja je došla do petlje, a ne ona koja je najduže čekala!

Da bi se riješili ovi problemi mehanizme sinkronizacije treba drukčije riješiti – korištenjem jezgrinih funkcija, gdje će se u kontroliranom okruženju dretva pustiti u KO ili neće, kada će se dretva maknuti s procesora (da ne troši procesorsko vrijeme na neproduktivnu petlju).

Pitanja za vježbu 4

1. Što je to proces, a što dretva?
2. Što je zajedničko dretvama istog procesa?
3. Zašto se koristi višedretvenost? Koje su prednosti (mogućnosti) sustava koji podržavaju višedretvenost?
4. Kada su dva zadatka međusobno zavisna, a kada nezavisna? Što sa zavisnim zadacima, kako izvoditi njihove dretve?
5. Odrediti maksimalno paralelan sustav zadataka uzimajući u obzir njihov međusobni odnos u lancu te domene i kodomene ... (+ opis zadataka)
6. Što je to *kritični odsječak* i *međusobno isključivanje*?
7. Kako se međusobno isključivanje može ostvariti u jednoprocesorskim a kako u višeprocesorskim sustavima?
8. Koji problem može nastati ako više dretvi obavlja operaciju: $A = A + 1$ nad zajedničkom varijablom A?
9. Navesti zahtjeve (4) na algoritme međusobnog isključivanja.
10. Čemu služi Dekkerov algoritam? Opisati njegov rad.
11. Čemu služi Lamportov algoritam? Opisati njegov rad.
12. Kako iskoristiti sklopovsku potporu međusobnom isključivanju korištenjem instrukcija TAS, SWP i sličnih?
13. Koji su problemi različitih načina ostvarenja međusobnog isključivanja?
14. Za zadani algoritam međusobnog isljučivanja provjeriti je li zadovoljava sve zahtjeve prema takvim algoritmima ... (+ pseudokod algoritama)

5. JEZGRA OPERACIJSKOG SUSTAVA

Što je jezgra OS-a?

- osnovni, najbitniji dijelovi bez kojih OS ne bi radio
- OS ima i druge dijelove (pomoćne programe, usluge) koji koriste jezgru

Potreba za jezgrom?

- Upravljanje dretvama – ostvarivanje višedretvenosti
- Upravljanje UI napravama
  - treba biti kontrolirano – ne prepušteno dretvama
  - prekidi su dobar mehanizam za prelazak u "kontrolirani način rada"
  - operacije koje se izvode u prekidu su posebne => dio jezgre OSa!
  - programski prekid: dretva preko njega poziva te "posebne funkcije" – jezgrine funkcije
- Sinkronizacija
  - nedostaci sinkronizacijskih mehanizama prikazanih u 4. poglavlju: radno čekanje, nepoštivanje redoslijeda ulaska u KO
  - treba izbijeći radno čekanje, ali i poštovati redoslijed zahtjeva
  - ipak, treba uzeti u obzir da ništa "nije besplatno", sve ima svoju cijenu (složenosti, dodatne kućanske poslove, strukture podataka, ...)

Jezgrine funkcije je potrebno pozivati mehanizmom *prekida* jer oni omogućuju:

- kritični odsječak na jednoprocesorskim sustavima (o proširenju za višeprocesorske kasnije)
- privilegirani način rada potreban za
  - korištenje zaštićene strukture podataka u sustavskom dijelu spremnika
  - upravljanje UI napravama
- upravljanje dretvama (manipulaciju opisnicima dretvi)

Jezgra OS-a se sastoji od:

- strukture podataka jezgre (opisnici, liste, međuspremnici, ...)
- jezgrinih funkcija

Tipovi prekida koji se koriste za pozive jezgrinih funkcija:

- sklopovski prekid (zahtjev UI naprava i satnog mehanizma)
- programski prekid (zahtjev iz programa)

Pretpostavke idućih razmatranja za ostvarenje jezgre:

- jednoprocesorski sustav (kasnije je dano i proširenje za višeprocesorske sustave)
- korisnički i jezgrin način rada podržani od strane procesora
- sve je u spremniku – neće se (za sada) razmatrati učitavanje i pokretanje programa
- jezgrine funkcije pozivaju se sklopovskim i programskim prekidom

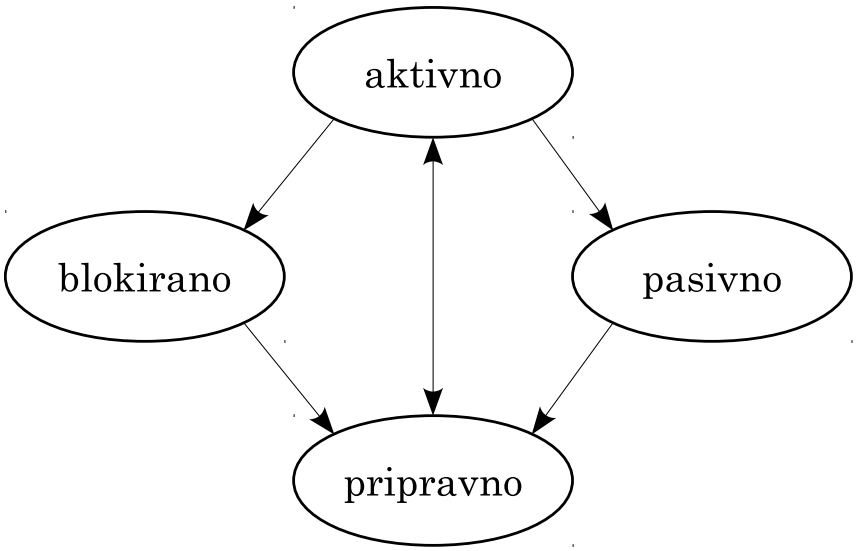
- postoji satni mehanizam koji periodički izaziva sklopovski prekid sata

Za sada (do 8. poglavlja) se neće razmatrati procesi, neka su sve dretve sustava u jednom zajedničkom procesu.

Pozivi jezgre ilustrirani su slikom 5.1.

Slika 5.1. Mehanizam poziva jezginih funkcija

Dretve mogu biti u različitim stanjima prema slici 5.2.



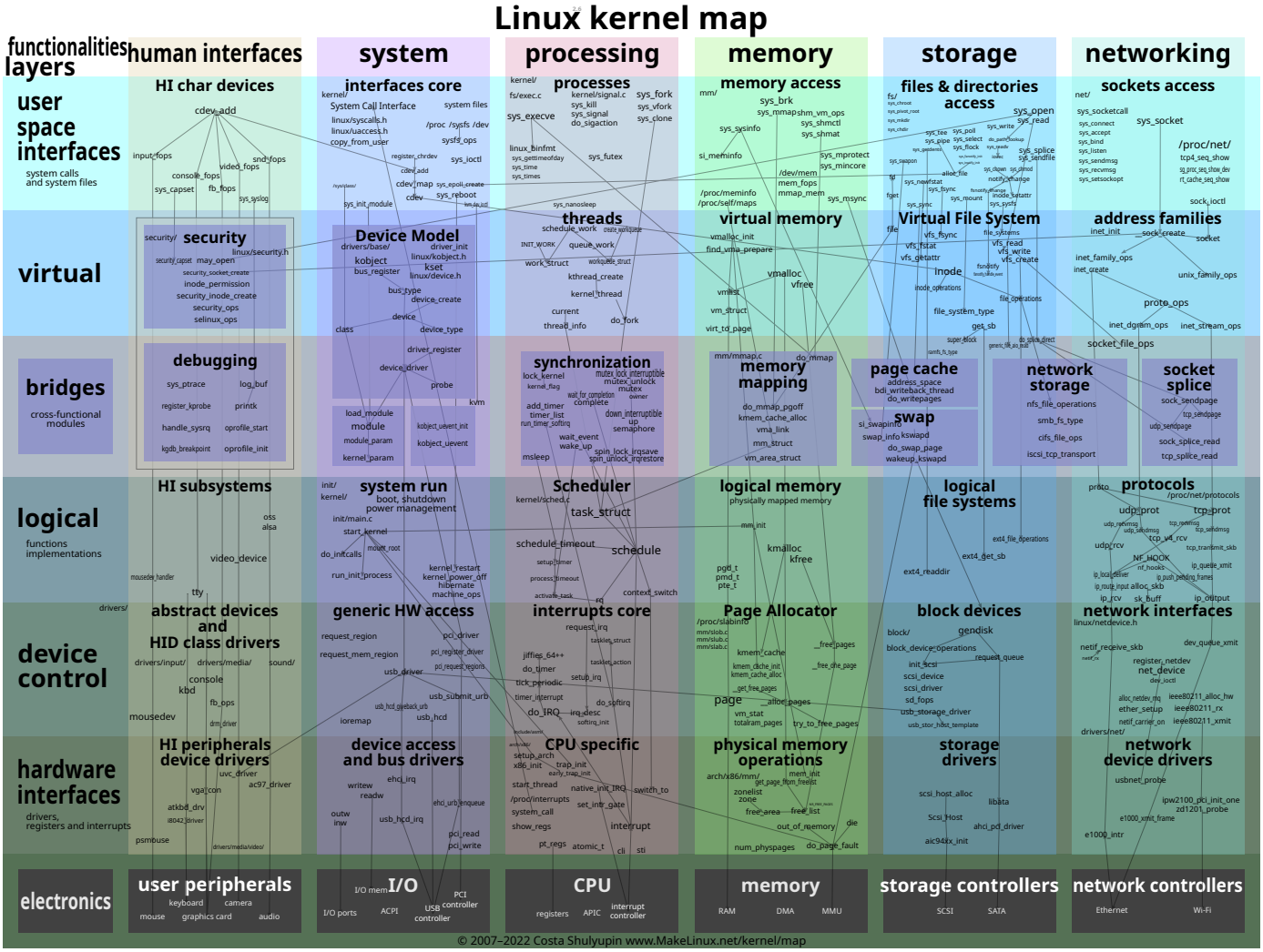
Slika 5.2. Moguća stanja dretve

Tablica 5.1. Model jezgre prikazan u ovom predmetu

podsustav	**sučelja, mogućnosti, . . .**
naprave	`Započni_UI(K,p)`, `Prekid_UI(K,p)`
dretve – raspoređivanje	(implicitno ugrađeno u jezgrine funkcije) po redu prispijeća, prema prioritetu, podjelom vremena (7. pogl.)

dretve
– sinkronizacija
	– binarni semafori: `ČekajBSEM(S)`, `PostaviBSEM(S)`
	– opći semafori: `ČekajOSEM(S)`, `PostaviOSEM(S)`
	– monitori (6. poglavlje):
	`Uđi_u_monitor(m)`
	`Izađi_iz_monitora(m)`
	`Čekaj_u_redu_uvjeta(red,m)`
	`Oslobodi_iz_reda_uvjeta(red)`
	`Oslobodi_sve_iz_reda_uvjeta(red)`

dretve – odgoda	`Zakasni(M)`, `Prekid_sata()`
procesi	opisano bez implementacije: statičko, dinamičko, straničenje
datotečni podsustav	opisano bez implementacije



Slika 5.3. Ponovljena slika 1.2. radi usporedbe (izvor: https://makelinux.github.io/kernel/map/)

5.1. Strukture podataka jezgre

Struktura podataka jezgre se sastoji od sljedećih elemenata:

- *opisnici dretvi*:
  - id (identifikacijski broj dretve)
  - podaci za raspoređivanje: način raspoređivanja, prioritet, ...
  - stanje dretve (aktivno, pripravno, ...)
  - opis spremničkog prostora dretve (gdje su instrukcije, podaci, stog)
  - `zadano_kašnjenje` – za ostvarivanje kašnjenja
  - kontekst – mjesto za pohranu konteksta dretve
  - kazaljka za liste prema stanjima (`Aktivna_D`, `Pripravne_D`, ...)
  - kazaljka za listu `Postojeće_D`
- *opisnici UI naprava* (info)
  - kazaljke na funkcije upravljačkog programa
  - međuspremnici i druge strukture podataka
  - lista za dretve koje čekaju na dovršetak svoje operacije nad napravom
- *liste stanja dretvi* – liste za opisnike dretvi
  - `Aktivna_D` – dretva koja se trenutno izvodi na procesoru
  - `Pripravne_D` – dretve koje se mogu izvoditi (aktivna se bira među njima)
  - blokirane dretve:
    - `UI[]` – liste dretvi koje čekaju na neku napravu
    - `Odgođene_D` – dretve koje su tražile odgodu
    - `BSEM[]` – binarni semafori
    - `OSEM[]` – opći semafori
  - `Postojeće_D` – lista u kojoj se nalaze sve dretve
    - ako se dretva nalazi samo u ovoj listi, dretva je *pasivna*

Liste dretvi mogu biti uređene (složene):

- prema redu prispijeća u listu
- prema prioritetu dretvi
- prema posebnim kriterijima (npr. vremenu odgode)

Za svaku listu je potrebno:

- kazaljka na prvu dretvu u listi (npr. `prva`)
- opcionalno/poželjno: kazaljka na zadnju dretvu u listi (npr. `zadnja`) – složenost umetanja na kraj je tada O(1)!

Jezgra upravlja sustavom – izvodi dretvu za dretvom

- kada se trenutno aktivna dretva blokira, uzima se prva dretva iz reda pripravnih

- u obradama prekida se neke blokirane dretve mogu odblokirati i staviti među pripravne

Latentna dretva (engl. *idle thread*)

- Dodatna, pomoćna dretva koja se izvodi kad nema niti jedne druge dretve u sustavu.
- Njen je zadatak dati procesoru nešto da radi (on mora nešto raditi).
- Takva dretva ima najmanji prioritet, tj. izvodi se jedino kada nema niti jedne druge dretve!
- U nastavku se podrazumijeva da takva dretva postoji, ali se ne spominje niti prikazuje.

5.2. Jezgrine funkcije

5.2.1. Kako se pozivaju j-funkcije iz programa? (info)

U kodu programa:

```
    ...
    J_Funkcija(parametri) //npr. write(fd, buf, size);
    ...
```

Sama funkcija ("omotač") koja poziva j-funkciju:

```
J_Funkcija(parametri)
{
    dodatni poslovi prije poziva jezgre

    pohrani za poziv jezgre(J_FUNKCIJA_ID, parametri)

    izazovi programski prekid
    //prekidom se ulazi u jezgru; povratkom iz prekida se vraća ovdje

    vrati = dohvati_povratnu_vrijednost_j_funkcije()

    dodatni poslovi nakon poziva jezgre
}
```

U obradi prekida:

```
    ...
    ako je (uzrok prekida programski prekid, tj. poziv j-funkcije) {
        {id, parametri} = dohvati ID tražene jezgrine funkcije te parametre
        JF[id](parametri) //J_FUNKCIJA(parametri)
    }
    inače ...
```

Opći izgled jezgrinih funkcija

```
j-funkcija J_FUNKCIJA (p)
{
    pohrani kontekst u opisnik Aktivna_D

    operacija jezgrine funkcije //može uzrokovati i promjenu aktivne dretve

    obnovi kontekst iz opisnika Aktivna_D
}
```

U nastavku je ime jezgrine funkcije u kôdu implementacije pisano `VELIKIM_SLOVIMA`, dok je poziv iz programa označen `Malim_Slovima` (npr. `ČEKAJ_OSEM` – `Čekaj_OSEM`)

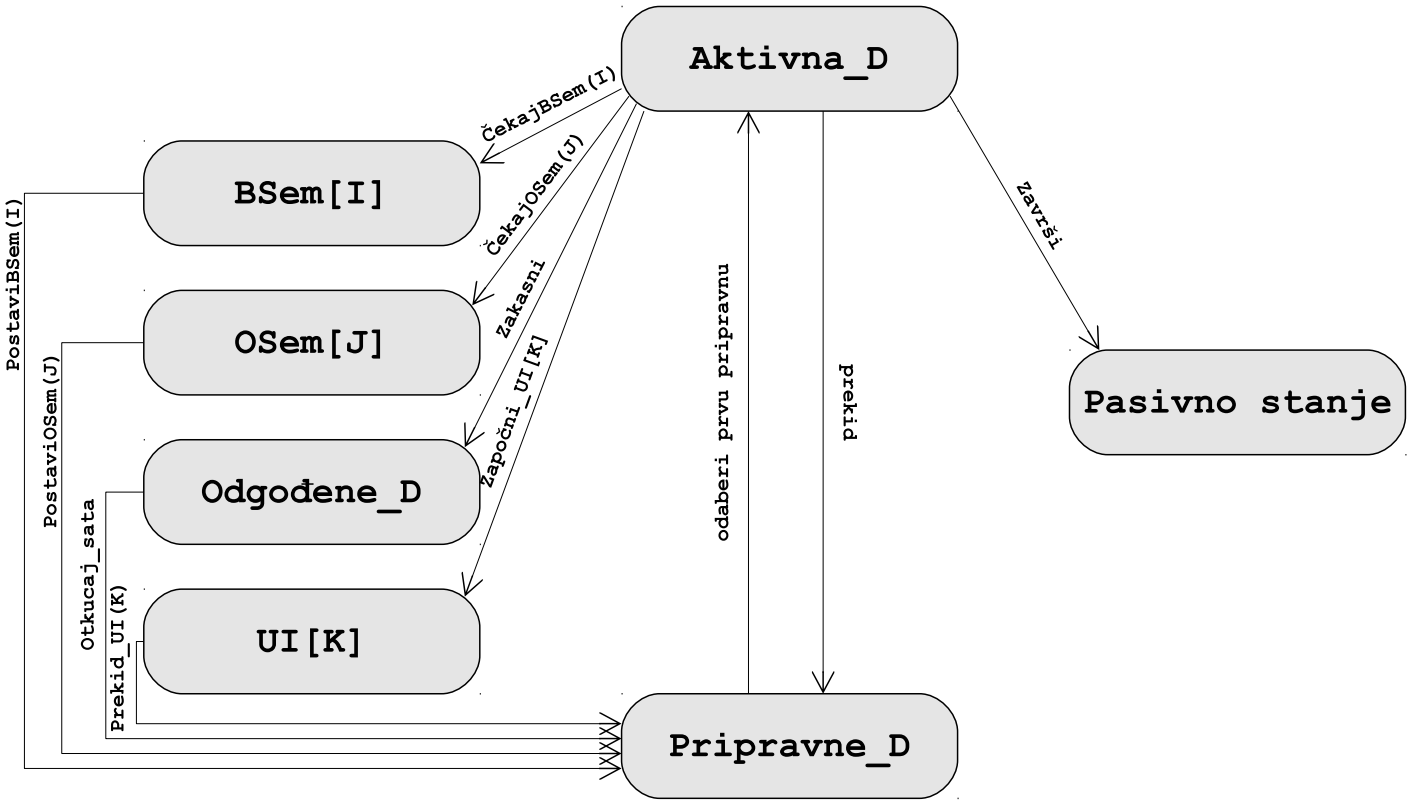
Operacija `pohrani kontekst u opisnik Aktivna_D` sugerira da je na stogu već pohranjen kontekst prekinute dretve prije poziva jezgrine funkcije i da se ovom operacijom on premješta u opisnik dretve. U stvarnim sustavima se najčešće pri pojavi prekida kontekst prekinute dretve automatski pohranjuje u opisnik te dretve tako da se izbjegne dvostruko kopiranje.

Izlazak iz jezgre pretpostavlja vraćanje u dretvu čiji je opisnik u redu `Aktivna_D`

Pretpostavlja se da `obnovi kontekst iz opisnika Aktivna_D`, osim obnove svih registara procesora izvodi i instrukciju: `vratiti_se_u_prekinutu_dretvu` (obnovi PC i SR sa stoga, prebaci se u način rada prekinute dretve, dozvoli prekidanje)

5.2.2. Moguća stanja dretvi u jednostavnom modelu sustava (ispitno pitanje)

Slika 5.5. prikazuje jednostavni model jezgre prikazan u ovom poglavlju sa svim mogućim stanjima i jezgrinim funkcijama koje uzrokuju promjene stanja.

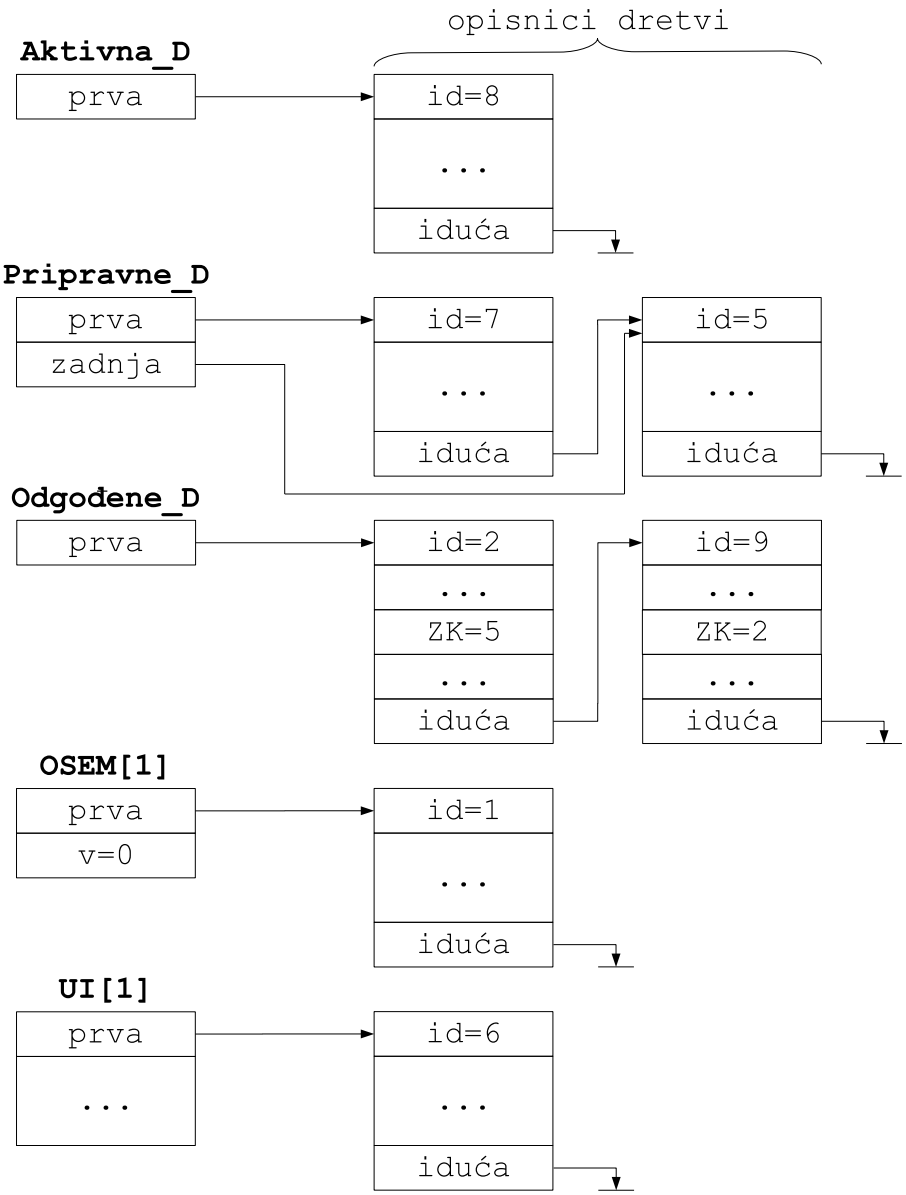


Slika 5.4. Moguća stanja dretve, jezgrine funkcije

Pasivno stanje je pseudo stanje – nema zasebne liste za to. Dretva je u tom stanju ako nije ni u jednoj drugoj listi, tj. nalazi se samo u listi `Postojeće_D` (pomoćnoj listi gdje se nalaze sve dretve).

Osim j-funkcija `Otkucaj_sata` i `Prekid_UI` (koje se pozivaju sklopovskim prekidima) sve ostale poziva aktivna dretva (`Aktivna_D`) !

Primjer strukture podataka jezgre



Slika 5.5. Primjer strukture podataka jezgre vizualizacijom objekata

Tablica 5.2. Primjer strukture podataka jezgre sažeto preko tablice

Red (liste opisnika)	Popis dretvi (id)
`Aktivna_D`	8
`Pripravne_D`	7 5
`Odgođene_D`	2^5 9^2
`OSEM[1]`	1
`UI[1]`	6

5.2.3. Razlike u ovdje prikazanom ostvarenju jezgre prema prikazanome u knjizi

Neke operacije su ovdje ostvarene na malo drukčiji način nego u knjizi. Međutim, oba pristupa pri rješavanju zadataka su ispravna u ispitima iako mogu ponekad dati različita rješenja. U nastavku su detaljnije prikazane razlike.

Iskazivanje operacija nad strukturama jezgre (opisnicima, listama)

- u knjizi – tekstom: `Premjesti opisnik iz reda Aktivna_D u red Bsem[I]`
- ovdje – funkcijama: `stavi_u_red(makni_prvu_iz_reda(Aktivna_D), BSEM[I])`

U knjizi – jezgrine funkcije u kojima se propuštaju/odblokiraju dretve

- u svim jezgrinim funkcijama koje potencijalno mogu odblokirati neku dretvu:
  1. najprije se aktivna prebacuje u red pripravnih
  2. potom se gleda stanje sustava i možda odblokira neka dretva – prebaci u red pripravnih
  3. na kraju se prva iz reda pripravnih uzme za aktivnu
- ovisno o načinu raspoređivanja (red prispijeća, prioritet), aktivna dretva nakon jezgrine funkcije može biti:
  - dretva koja je pozvala jezgrinu funkciju (i prije bila aktivna)
  - odblokirana dretva
  - dretva koja je bila prva u redu pripravnih prije poziva "ove" jezgrine funkcije

Ovdje – jezgrine funkcije u kojima se propuštaju/odblokiraju dretve

- ovdje se malo "optimiralo" rad takvih jezgrinih funkcija te se promjena aktivne događa samo po potrebi, tj. *raspoređivač* se poziva samo ako se neka dretva blokirala ili odblokirala
- ako se u obavljanju jezgrine funkcije aktivna blokira ili neka druga dretva odblokira sa:
  - `stavi_u_red(makni_prvu_iz_reda(Aktivna_D), neki_red)`
  - `stavi_u_red(makni_prvu_iz_reda(neki_red), Pripravne_D)`
- tada prije kraja jezgrine funkcije mora se pozvati:
  - `odaberi_aktivnu_dretvu()`
- operacija `odaberi_aktivnu_dretvu()` mogla bi se opisati sljedećim kodom:

```
odaberi_aktivnu_dretvu() //schedule()
{
    ako je (Aktivna_D.stanje != AKTIVNO) { //dretva je maknuta u neki drugi red
        stavi_u_red(makni_prvu_iz_reda(Pripravne_D), Aktivna_D)
    }
    inače ako je (Raspoređivanje == PREMA_PRIORITETU) {
        prva_pripravna = dohvati_prvu_iz_reda(Pripravne_D)
        ako je (Aktivna_D.prioritet < prva_pripravna.prioritet) {
            stavi_u_red(makni_prvu_iz_reda(Aktivna_D), Pripravne_D)
            stavi_u_red(makni_prvu_iz_reda(Pripravne_D), Aktivna_D)
        }
    }
}
```

- pretpostavlja se da funkcija `stavi_u_red` koristi zadano uređenje za dotični red; npr. za red pripravnih to može biti prema prispijeću ili prioritetu

Problematične funkcije

- funkcije koje mogu generirati različita stanja zbog navedenih promjena su `PREKID_UI()`, `OTKUCAJ_SATA()`, `POSTAVI_BSEM()` i `POSTAVI_OSEM()`

Razlike u stanju nakon jezgrinih funkcija

- kada je isto stanje nakon funkcija?
  - ako se dretve raspoređuju prema prioritetu i sve dretve imaju različit prioritet
    - nakon jezgrine funkcije aktivna je ona s najvećim prioritetom, a red pripravnih složen prema prioritetu
  - ili je red pripravnih bio prazan i a) raspoređivanje je po redu prispijeća, ili b) odblokirana dretva ima isti ili manji prioritet od aktivne
    - u knjizi je tada aktivna najprije stavljena u red pripravnih (prva i jedina u tom redu) potom je odblokirana stavljena u red pripravnih iza nje te konačno se uzima prva iz reda pripravnih – ona koja je i prije bila aktivna
- kada stanje nakon funkcija ne mora biti isto?
  - red pripravnih nije bio prazan prije poziva jezgrine funkcije i
    - dretve se raspoređuju po redu prispijeća, ili
    - dretve se raspoređuju prema prioritetu i prva u redu pripravnih ima isti prioritet kao i aktivna a odblokirana nema veći prioritet
  - razlika u ponašanju:
    - u knjizi: nakon poziva aktivna će biti ona koja je prije poziva bila prva u redu pripravnih
    - ovdje: aktivna će ostati ona koja je pozvala jezgrinu funkciju, a odblokirana dretva će biti zadnja u redu pripravnih (iza onih s istim prioritetom, ispred onih s manjim)
  - primjer: neka su svi redovi složeni po redu prispijeća
    - početno stanje: {Aktivna=D_5, Pripravne=$\{D_3, D_6, D_2\}$, red BSEM[S]=$\{D_4, D_1\}$};
    - aktivna poziva `PostaviBSEM(S)` (koja oslobađa prvu iz reda semafora)
    - stanje nakon prema knjizi:

      {Aktivna=D_3, Pripravne=$\{D_6, D_2, D_5, D_4\}$, red BSEM[S]=$\{D_1\}$};
    - stanje nakon prema ovdje prikazanom ostvarenju:

      {Aktivna=D_5, Pripravne=$\{D_3, D_6, D_2, D_4\}$, red BSEM[S]=$\{D_1\}$};
- VAŽNO: oba pristupa pri rješavanju zadataka (prikazana ovdje i prikazana u knjizi) su ispravna u ispitima, iako mogu dati različita rješenja!

5.2.4. Obavljanje ulazno-izlaznih operacija

Pretpostavke jednostavnog modela jezgre:

- UI operacije se obavljaju radi zahtjeva dretvi – one pokreću operaciju (preko jezgre)
- dovršetak UI operacije naprava javlja mehanizmom prekida
- UI naprave se koriste na "sinkroni način" – dretva čeka kraj operacije prije nastavka rada
- zahtjeve same jezgre prema UI napravama u ovom prikazu zanemarujemo

UI operacije obavljaju se preko jezgrinih funkcija:

- `ZAPOČNI_UI(K, parametri)` – poziva ju dretva
  - UI operacija traje (nije trenutna) pa će dretva čekati na njen kraj
- `PREKID_UI(K)` – poziva se na prekid naprave K
  - naprava K obavlja operacije redom kojim su joj zadane
  - kad je gotova s operacijom ona izaziva prekid

```
j-funkcija ZAPOČNI_UI(K, parametri)
{
    pohrani kontekst u opisnik Aktivna_D

    stavi_u_red(makni_prvu_iz_reda(Aktivna_D), UI[K])
    pokreni UI operaciju na napravi K
    odaberi_aktivnu_dretvu() //schedule()

    obnovi kontekst iz opisnika Aktivna_D
}
```

```
j-funkcija PREKID_UI(K)
{
    pohrani kontekst u opisnik Aktivna_D

    dovrši UI operaciju na napravi K
    stavi_u_red(makni_prvu_iz_reda(UI[K]), Pripravne_D)
    odaberi_aktivnu_dretvu()

    obnovi kontekst iz opisnika Aktivna_D
}
```

[dodatno]

Ukoliko naprava ne mora dovršavati zahtjeve prema redu prispijeća onda bi gornje funkcije mogli nadopuniti/izmijeniti:

```
u ZAPOČNI_UI(K, parametri) nakon stavi_u_red ubaciti:
    dodaj_zahtjev_u_listu(K, param, Aktivna_D);

u PREKID_UI(K) promijeniti kod:
    dretva = dovrši UI operaciju na napravi K
    ako je dretva != NULL tada
        stavi_u_red(makni_dretvu_iz_reda(dretva, UI[K]), Pripravne_D)
    odaberi_aktivnu_dretvu()
```

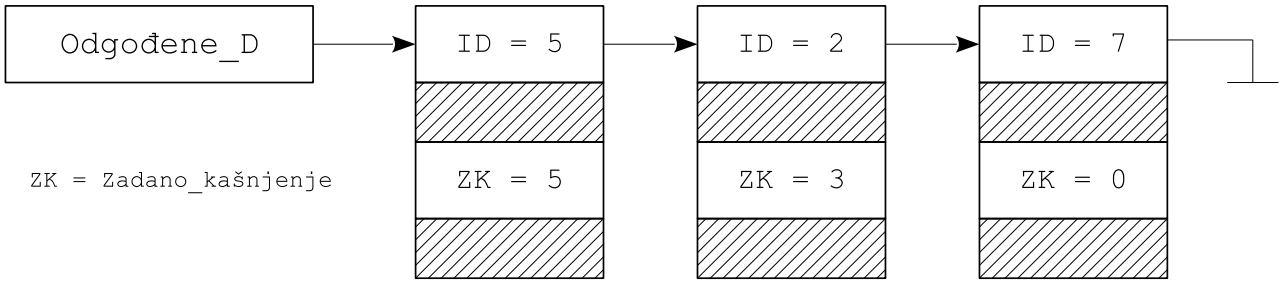
Druga mogućnost ostvarenja UI operacija jest asinkroni rad: dretva ne čeka na dovršetak UI operacije već naknadno provjerava status (OS ažurira status u prekidima naprave).

5.2.5. Ostvarivanje kašnjenja

Za ostvarenje kašnjenja koristi se prekid sata koji u pravilnim intervalima izaziva prekid.

Postoji jedan red (lista opisnika) za odgođene dretve: `Odgođene_D`

- lista je složena prema vremenima odgode dretvi
- prva dretva u listi ima broj otkucaja sata koje još mora čekati
- svaka iduća dretva ima relativan broj u odnosu na prethodnu
- koristi se element `Zadano_kašnjenje` u opisniku dretve



Slika 5.6. Primjer liste `Odgođene_dretve`

Opis primjera sa slike 5.6.

- dretva D5 treba čekati još 5 otkucaja sata
- dretva D2 treba čekati da se prethodna dretva makne iz reda (5 otkucaja) te još 3 otkucaja nakon toga (ukupno još 8)
- dretva D7 treba čekati da se D5 i D2 maknu, pa se i ona oslobađa iz reda (kada i D2)

Jazgrine funkcije za odgodu su: `ZAKASNI(M)` koju poziva dretve te `OTKUCAJ_SATA()` koja se poziva na prekid satnog mehanizma (brojila).

```
j-funkcija ZAKASNI(M)
{
  pohrani kontekst u opisnik Aktivna_D

  stavi_u_red_odgođenih(makni_prvu_iz_reda(Aktivna_D), M, Odgođene_D)
  odaberi_aktivnu_dretvu()

  obnovi kontekst iz opisnika Aktivna_D
}
```

```
j-funkcija OTKUCAJ_SATA()
{
  pohrani kontekst u opisnik Aktivna_D

  ako (Odgođene_D->prva != prazno) {
    Odgođene_D->prva.Zadano_kašnjenje--
    ako je (Odgođene_D->prva.Zadano_kašnjenje == 0) {
      ponavljaj
        stavi_u_red(makni_prvu_iz_reda(Odgođene_D), Pripravne_D)
      dok je (Odgođene_D->prva != prazno && Odgođene_D->prva.Zadano_kašnjenje == 0)

      odaberi_aktivnu_dretvu()
    }
  }
  obnovi kontekst iz opisnika Aktivna_D
}
```

5.3. Semafori

Semafori služe za sinkronizaciju dretvi.

Za razliku od semafora na raskršću koji mijenjaju stanje protokom vremena, sinkronizacijski semafori se mijenjaju preko poziva jezgrinih funkcija – koje pozivaju dretve.

Postoji nekoliko inačica semafora koji se koriste za razne potrebe. U nastavku su prikazani oni najvažniji, koji se najčešće koriste.

5.3.1. Binarni semafor

Binarni semafor ima dva stanja: jedno prolazno i jedno neprolazno.

Zato je prikladan za međusobno isključivanje, ali i za neke druge oblike sinkronizacije.

Struktura podataka potrebna za semafor:

- `.v` – vrijednost semafora
  - kada je `.v == 0` semafor je *neprolazan*
  - kada je `.v == 1` semafor je *prolazan*
- `.red` – blokirane dretve nad semaforom (lista opisnika tih dretvi)

Jezgrine funkcije:

- `ČEKAJ_BSEM(I)` (ili `ISPITATI_BSEM`)
- `POSTAVI_BSEM(I)` (ili `POSTAVITI_BSEM`)

```
j-funkcija ČEKAJ_BSEM(S)
{
    pohrani kontekst u opisnik Aktivna_D

    ako je (BSEM[S].v == 1) {
        BSEM[S].v = 0
    }
    inače {
        stavi_u_red(makni_prvu_iz_reda(Aktivna_D), BSEM[S])
        odaberi_aktivnu_dretvu()
    }

    obnovi kontekst iz opisnika Aktivna_D
}
```

```
j-funkcija POSTAVI_BSEM(S)
{
    pohrani kontekst u opisnik Aktivna_D

    ako je (red BSEM[S] nije prazan) {
        stavi_u_red(makni_prvu_iz_reda(BSEM[S]), Pripravne_D)
        odaberi_aktivnu_dretvu()
    }
    inače {
        BSEM[S].v = 1
    }
    obnovi kontekst iz opisnika Aktivna_D
}
```

Pri oslobađanju dretve može se dogoditi da oslobođena dretva ima veći prioritet od dretve koja ju je oslobodila (trenutno aktivna dretva) – zato treba pozvati `odaberi_aktivnu_dretvu()`

i nakon tog događaja. Npr. ako je prva u redu `BSEM[1]` dretva D4 prioriteta 4, a pozivajuća dretva (trenutno aktivna, koja je pozvala `PostaviBSEM(1)`) ima prioritet 3 onda po pozivu te jezgrine funkcije aktivna dretva je dretva D4 (a D3 je u redu pripravnih dretvi).

Primjer 5.1. Primjer ostvarenja kritičnog odsječka binarnim semaforom

```
dretva() {
    ponavljaj {
        Čekaj_BSEM(1)
        kritični odsječak
        Postavi_BSEM(1)

        nekritični odsječak
    }
    do zauvijek
}
```

Početna vrijednost semafora `BSEM[1]` (prije pokretanja ovakvih dretvi) treba biti 1.

Primjer 5.2.

Primjer: sinkronizirati ulaznu, radnu i izlaznu dretvu sa četiri binarna semafora.

Ulazna dretva:

```
while(1)
{
   U = dohvati podatke

   Čekaj_BSEM(1)
   UR = U
   Postavi_BSEM(2)
}
```

Radna dretva:

```
while(1)
{
   Čekaj_BSEM(2)
   R1 = UR
   Postavi_BSEM(1)

   R2 = obradi(R1)

   Čekaj_BSEM(3)
   RI = R2
   Postavi_BSEM(4)
}
```

Izlazna dretva:

```
while(1)
{
   Čekaj_BSEM(4)
   I = RI
   Postavi_BSEM(3)

   pohrani(I)

}
```

Početno su `BSEM[1]` i `BSEM[3]` postavljeni u 1, ostali u 0.

Međuspremnici `UR` i `RI` koriste se za primopredaju podataka među dretvama (Ulazna-Radna, Radna-Izlazna).

Masno označeni dijelovi koda dretve mogu izvoditi i paralelno (na višeprocesorskom sustavu).

5.3.2. Opći semafor

Opći semafor ima više stanja: jedno neprolazno i više prolaznih.

Zato je prikladan za brojanje događaja i sredstava te slične sinkronizacije.

Struktura za semafor:

- `.v` – vrijednost semafora
  - kada je `.v == 0` semafor je *neprolazan*
  - kada je `.v > 0` semafor je *prolazan*
- `.red` – blokirane dretve nad semaforom (lista opisnika tih dretvi)

Opći semafor se može ostvariti na razne načine. Mi ćemo koristiti ostvarenje koje je nabliže onome koji se koristi u stvarnim sustavima. Označimo takav semafor s OSEM.

Opći semafor – OSEM

- druga imena: brojački semafor, brojilo događaja
- po ostvarenju jako slično BSEM-u, uz `++` umjesto `=1` i `--` umjesto `=0`

```
j-funkcija ČEKAJ_OSEM(S)
{
    pohrani kontekst u opisnik Aktivna_D

    ako je (OSEM[S].v > 0) {
        OSEM[S].v--
    }
    inače {
        stavi_u_red(makni_prvu_iz_reda(Aktivna_D), OSEM[S])
        odaberi_aktivnu_dretvu()
    }

    obnovi kontekst iz opisnika Aktivna_D
}

j-funkcija POSTAVI_OSEM(S)
{
    pohrani kontekst u opisnik Aktivna_D

    ako je (red OSEM[S] nije prazan) {
        stavi_u_red(makni_prvu_iz_reda(OSEM[S]), Pripravne_D)
        odaberi_aktivnu_dretvu()
    }
    inače {
        OSEM[S].v++
    }

    obnovi kontekst iz opisnika Aktivna_D
}
```

Vrijednost `OSEM[I].v` nikako ne može postati negativna.

Vrijednost `OSEM[I].v` (kao i vrijednost `.v` ostalih semafora) može se mijenjati isključivo preko poziva jezginih funkcija. U programu dretve se `OSEM[I].v` ne može izravno koristiti (nije dohvatljiv izravno)!

Primjer 5.3. Dijkstrin semafor (info)

Osim općeg semafora OSEM postoje i druge zamisli za ostvarenje semafora.

Dijsktrin semafor (označimo ga s OS) jest "jednokratni" semafor pridijeljen dretvi, na koji samo jedna dretva može čekati. Svojstvo ovog semafora jest da može poprimiti i negativne vrijednosti. Ostvarenje takvog semafor prikazano je u nastavku.

```
j-funkcija ČEKAJ_OS(S)
{
    pohrani kontekst u opisnik Aktivna_D

    OS[S].v--
    ako je (OS[S].v < 0) {
        stavi_u_red(makni_prvu_iz_reda(Aktivna_D), OS[S])
        odaberi_aktivnu_dretvu()
    }

    obnovi kontekst iz opisnika Aktivna_D
}

j-funkcija POSTAVI_OS(S)
{
    pohrani kontekst u opisnik Aktivna_D

    OS[S].v++
    ako (OS[S].v >= 0 && red OS[S] nije prazan) {
        stavi_u_red(makni_prvu_iz_reda(OS[S]), Pripravne_D)
        odaberi_aktivnu_dretvu()
    }

    obnovi kontekst iz opisnika Aktivna_D
}
```

Binarni semafor i opći semafor BITNO se razlikuju po načinu rada. Pozivom `Postavi_OSEM(I)`, ako nema blokiranih dretvi u redu tog semafora, vrijednost semafora se povećava za 1. Ako je trenutna vrijednost binarnog semafora već jednaka 1, pozivom `Postavi_BSEM(I)` se ništa neće promijeniti – vrijednost će i nakon poziva biti jednaka 1.

Primjer 5.4. Primjer ostvarenja kritičnog odsječka semaforom

```
dretva () {
    ponavljaj {
        Čekaj_OSEM(1)
        kritični odsječak
        Postavi_OSEM(1)
        nekritični odsječak
    }
    do zauvijek
}
```

Početna vrijednost semafora `OSEM[1]` (prije pokretanja ovakvih dretvi) treba biti 1.

Primjer 5.5.

Primjer: sinkronizirati ulaznu, radnu i izlaznu dretvu sa četiri opća semafora.

Ulazna dretva:

```
while(1)
{
   U = dohvati podatke

   Čekaj_OSEM(1)
   UR = U
   Postavi_OSEM(2)
}
```

Radna dretva:

```
while(1)
{
   Čekaj_OSEM(2)
   R1 = UR
   Postavi_OSEM(1)

   R2 = obradi(R1)

   Čekaj_OSEM(3)
   RI = R2
   Postavi_OSEM(4)
}
```

Izlazna dretva:

```
while(1)
{
   Čekaj_OSEM(4)
   I = RI
   Postavi_OSEM(3)

   pohrani(I)

}
```

Početno su `OSEM[1]` i `OSEM[3]` postavljeni u 1, ostali u 0.

Primjer 5.6. Web poslužitelj s glavnom dretvom i više radnih dretvi (info)

```
glavna_dretva
{
   ponavljaj {
      C = čekaj_spoj_klijenta()

      Čekaj_BSEM(KO)
      stavi_zahtjev_u_red(C)
      Postavi_BSEM(KO)
      Postavi_OSEM(Z)
   }
   do kraja_rada
}
```

```
radna_dretva
{
   ponavljaj {
      Čekaj_OSEM(Z)
      Čekaj_BSEM(KO)
      uzmi_idući_zahtjev_iz_reda
      Postavi_BSEM(KO)

      obradi_zahtjev_i_vrati_rezultat
   }
   do kraja_rada
}
```

Početne vrijednosti: `BSEM[KO].v = 1`, `OSEM[Z].v = 0`

Primjer 5.7. Utjecaj jezgrinih funkcija na stanje dretvi

Pokažimo kako će se mijenjati stanje sustava pozivom jezgrinih funkcija u određenim trenucima (zadano ispod). Programske prekide uvijek izaziva trenutno aktivna dretva.

Pozivi jezgrinih funkcija (jedna za drugom, nakon nekog kratkog vremena):

1. `Prekid_UI(1)`
2. `Započni_UI(2)`
3. `Otkucaj_sata`
4. `Zakasni(3)`
5. `Otkucaj_sata`
6. `PostaviBSEM(1)`
7. `ČekajOSEM(1)`

Neka se dretve raspoređuju prema prioritetu – samo je red pripravnih dretvi složen prema prioritetu, dok su svi ostali po redu prispijeća (osim reda odgođenih).

Tablica 5.3. Promjena stanja u jezgri gornjim pozivima

Red / stanje	0	1	2	3	4	5	6	7
`Aktivna_D`	8	8	7	7	6	6	6	5
`Pripravne_D`	7 5	7 6 5	6 5	6 5	5	5 2	5 4 2	4 2
`BSEM[1]`	4 11	4 11	4 11	4 11	4 11	4 11	11	11
`OSEM[1]`	1	1	1	1	1	1	1	1 6
`UI[1]`	6 12	12	12	12	12	12	12	12
`UI[2]`	3	3	3 8	3 8	3 8	3 8	3 8	3 8
`Odgođene_D`	2^2 9^5	2^2 9^5	2^2 9^5	2^1 9^5	2^1 7^2 9^3	7^2 9^3	7^2 9^3	7^2 9^3

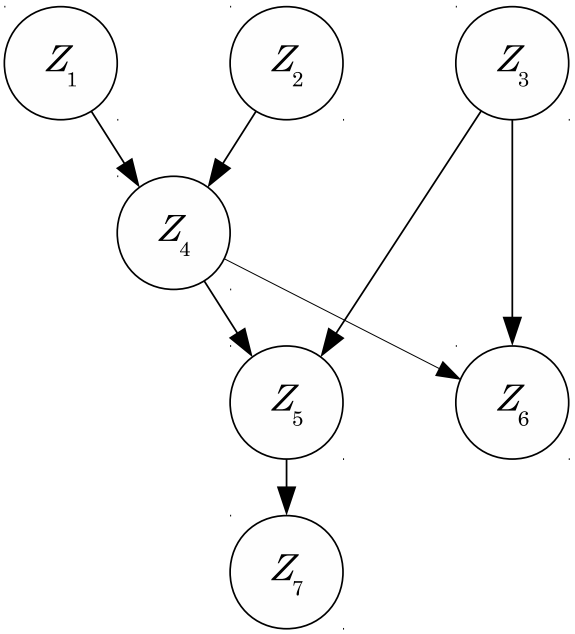
Zadatak 5.1. (ispitni zadatak)

Za ostvarenje sustava dretvi prema slici (rješenje zadatka 4.1) koriste se binarni/opći semafori.

a) Koliko semafora je potrebno za sinkronizaciju?

b) Koje su početne vrijednosti semafora?

c) Neka je T_i tekst zadatka i. Proširiti svaki zadatak T_i u T_i' s minimalnim brojem potrebnih procedura `Čekaj_OSEM(j)` i `Postavi_OSEM(j)`.



5.4. Izvedba jezgrinih funkcija za višeprocesorske sustave

- Zabrana prekida nije dovoljna za višeprocesorske sustave kao mehanizam međusobnog isključivanja jezgrinih funkcija jer je zabrana lokalna za pojedini procesor
- Kako ostvariti MI u takvim sustavima?
  - mora se dodati radno čekanje
  - neka postoji instrukcija `TAS` kako je prikazano u 4. poglavlju
  - neka se za svaki procesor definira po jedna struktura, tj. ukupno imamo:
    * lista aktivnih dretvi: `Aktivna_D[N]`
    * liste pripravnih dretvi: `Pripravne_D[N]` (opisano kasnije zašto i ovo)
  - neka se definira varijabla `OGRADA_JEZGRE` (zastavica za radno čekanje)

5.4.1. Opći semafor

```
j-funkcija ČEKAJ_OSEM(S)
{
  //neka je M indeks procesora, npr. iz opisnika dreve ili nekog registra procesora
  pohrani kontekst u opisnik Aktivna_D[M]
  //Aktivna_D[M] su lokalni podaci, nije ih potrebno štititi

  dok je (TAS (OGRADA_JEZGRE) == 1)   // radno čekanje
    ;                                 // za ostvarenje KO u jezgri

  ako je (OSEM[S].v > 0) {
    OSEM[S].v--
  }
  inače {
    stavi_u_red(makni_prvu_iz_reda(Aktivna_D[M]), OSEM[S])
    odaberi_aktivnu_dretvu_na_procesoru(M)
  }

  OGRADA_JEZGRE = 0

  obnovi kontekst iz opisnika Aktivna_D[M]
}
j-funkcija POSTAVI_OSEM(S)
{
  pohrani kontekst u opisnik Aktivna_D[M]

  dok je (TAS (OGRADA_JEZGRE) == 1)
    ;

  ako je (red OSEM[S] nije prazan) {
    stavi_u_red(makni_prvu_iz_reda(OSEM[I]), Pripravne_D[L])
    //L - iz opisnika dretve
    odaberi_aktivnu_dretvu_na_procesoru(M)
  }
  inače {
    OSEM[S].v++
  }

  OGRADA_JEZGRE = 0

  obnovi kontekst iz opisnika Aktivna_D[M]
}
```

[dodatno]

Kada nekom drugom procesoru stavljamo dretvu u red pripravnih on bi trebao pogledati je li sad to dretva najvećeg prioriteta kod njega (je li prioritet te dretve veći od aktivne). Stoga bi tada tom procesoru trebalo poslati i prekidni signal (trenutni procesor šalje signal tom drugom – mehanizam koji postoji u višeprocesorskim sustavima).

Koristi se *radno čekanje*:

- ali samo do ulaska u jezgrinu funkciju (onaj bitan dio)
- s obzirom na to da su jezgrine funkcije kratke to nije dugo čekanje

Zašto više redova pripravnih, po jedan za svaki procesor?

- osnovni razlog jest učinkovitost korištenja priručnog spremnika procesora
  - učinkovitost sustava poprilično ovisi o priručnim spremnicima procesora jer s njima procesor značajno brže komunicira nego s memorijom
  - dretve koje se izmjenjuju na istom procesoru (npr. podjelom vremena) pri povratku na procesor će vjerojatno u priručnom spremniku još uvijek naći neke svoje podatke koje stoga neće trebati dohvaćati iz memorije – ušteda vremena, tj. ubrzanje rada
- dodatni razlog jest u smanjenom zaključavanju podataka jezgre
  - u stvarnim operacijskim sustavima se ne koristi zaključavanje pri ulasku u jezgru (*big kernel lock*) kao što je to prikazano prethodnim primjerom sa semaforima, već se zasebno zaključavaju samo one strukture podataka koje se koriste u dijelovima jezgrinih funkcija
    - na ovaj način se jezgrine funkcije koje rade različite stvari mogu paralelno izvoditi
    - npr. u prethodnom primjeru bi prvo zaključali samo ključ povezan sa semaforom S; tek ako je potrebno pristupiti i nekom redu pripravnih, onda zaključati i taj red
  - ako svaki procesor ima zaseban red pripravnih, pri zamjeni jedne dretve drugom dovoljno je zaključati samo taj red pripravnih
- više o višeprocesorskim sustavima moguće je pronaći u okviru predmeta Napredni operacijski sustavi

5.5. Primjeri sinkronizacije binarnim semaforima

Zadatak 5.2. Problem pušača cigareta

(Ograda – ne reklamiramo cigarete (naprotiv), ali to je ime algoritma – Patil, 1971.)

Zamislimo sustav s tri dretve pušača i jednom dretvom trgovcem. Svaki pušač neprestano savija cigarete i puši. Kako bi se savila i popušila cigareta potrebno je imati tri sastojka: duhan, papir i šibice. Jedan pušač ima u neograničenim količinama samo papir, drugi samo duhan, a treći samo šibice. Trgovac ima sva tri sastojka u neograničenim količinama. Trgovac nasumice stavlja na stol dva različita sastojka. Pušač koji ima treći sastojak uzima sastojke sa stola, signalizira trgovcu, savija cigaretu i puši. Trgovac stavlja nova dva sastojka na stol i ciklus se ponavlja. Na početku je stol prazan. Napisati dretve pušača i trgovca tako da se one međusobno ispravno sinkroniziraju s pomoću dva semafora. Napisati početne vrijednosti semafora.

Zadatak 5.3. Prioritetni redovi i semafori (kraći)

U jednoprocesorskom računalu pokrenut je sustav dretvi D_1, D_2 i D_3 s prioritetima 1, 2 i 3 respektivno. Najviši prioritet je 3. Svi zadaci koje obavljaju dretve su istog oblika D_x. Red pripravnih dretvi i red semafora su prioritetni. Aktivna je dretva koja je prva u redu pripravnih (nema posebnog reda aktivnih dretvi). Prije pokretanja sustava dretvi semafor `S` je bio zatvoren. Nakon nekog vremena (kad se navedene dretve nađu u redu semafora S), pokrene se dretva G. Što će se ispisati na zaslonu?

```
Dretva Dx{
    dok je(1){
        Čekaj_BSEM(S)
        piši(Px)         a
        Postavi_BSEM(S)-------------
        piši(Zx)         b
    }
}
dretva G {
    Postavi_BSEM(S)
}
```

5.6. Operacije stvaranja dretvi i semafora (info)

Stvaranje novih dretvi kao i drugih jezgrinih objekata (npr. semafora) traži podsustav za upravljanje spremnikom. Stoga te operacije nisu uključene u prikaz jezgrinih funkcija. Ipak, u dosta općenitom obliku u ovom su potpoglavlju navedene skice i takvih operacija.

```
j-funkcija STVORI_DRETVU(početna_funkcija, parametar)
{
    pohrani kontekst u opisnik Aktivna_D

    stog = rezerviraj dio spremnika za stog
    opisnik = stvori novi opisnik dretve(početna_funkcija, parametar, stog)
    stavi_u_red2(opisnik, Postojeće_D)

    stavi_u_red(opisnik, Pripravne_D)
    odaberi_aktivnu_dretvu()

    obnovi kontekst iz opisnika Aktivna_D
}
j-funkcija ZAVRŠI_DRETVU ()
{
    oslobodi stog kojeg je Aktivna_D koristila

    makni_iz_reda(Aktivna_D, Postojeće_D)
    oslobodi opisnik Aktivna_D
    odaberi_aktivnu_dretvu()

    obnovi kontekst iz opisnika Aktivna_D
}

j-funkcija STVORI_BSEM(početna_vrijednost)
{
    pohrani kontekst u opisnik Aktivna_D

    opisnik = stvori novi opisnik za binarni semafor
    I = dodaj "opisnik" u polje binarnih semafora
    BSEM[I].v = početna_vrijednost
    postavi "I" kao povratnu vrijednost ove jezgrine funkcije

    obnovi kontekst iz opisnika Aktivna_D
}
j-funkcija UNIŠTI_BSEM(I)
{
    pohrani kontekst u opisnik Aktivna_D

    //ako ima blokiranih dretvi trebalo bi ih odblokirati i javiti im grešku

    opisnik = BSEM[I]
    makni opisnik "I" iz polja binarnih semafora
    oslobodi "opisnik"

    obnovi kontekst iz opisnika Aktivna_D
}
```

5.7. Ostali mehanizmi jezgre (info)

Operacijski sustav posjeduje mnoge mehanizme kojima omogućuje programima razne operacije. Neke od njih su predstavljeni u ovim materijalima, neki u sklopu priprema za laboratorijske vježbe, ali mnogi nisu. U ovom su potpoglavlju vrlo kratko opisani mehanizmi signala, odgode dretvi, alarmi i rad sa satovima sustava.

5.7.1. 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 pretpostavljenom 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).

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 treba biti oprezan zato jer mnoge funkcionalnosti koje definira POSIX ne moraju biti do kraja ostvarene. Nadalje, signali prekidaju neke jezgrine funkcije, tj. ako 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.

5.7.2. Korištenje satnih mehanizama u operacijskim sustavima

U ovom poglavlju pokazano je kako ostvariti odgodu izvođenja dretve za određeni broj intervala "otkucaja sata" (poziv `Zakasni(M)`). Osim te operacije koja koristi vrijeme postoje i druge. Dvije najpoznatije su dohvat trenutna sata (`Dohvati_sat(oznaka_sata)`) te programiranje akcije za budući trenutak (`Postavi_alarm(kada, akcija)`).

Pri korištenju sata treba biti svjestan mogućih problema zbog preciznosti sata, zbog drugih poslova (dretvi, obrade prekida) koji mogu dodatno odgoditi neke vremenski povezane akcije, zbog mogućnosti prekida odgoda signalima i drugim mehanizmima.

Osim sata koji odbrojava u stvarnom vremenu, operacijski sustavi često nude i druge satove koji mogu ponekad biti korisni (npr. u POSIX-u `CLOCK_REALTIME`, `CLOCK_MONOTONIC`, `CLOCK_PROCESS_CPUTIME_ID`, `CLOCK_THREAD_CPUTIME_ID`).

5.8. Brži sinkronizacijski mehanizmi (info)

Iako nema radnog čekanja i poštuje se redoslijed zahtjeva, postoji jedan problem: poziv jezgrine funkcije "košta". S obzirom na to da se koristi mehanizam prekida imamo kućne poslove (pohrana konteksta, obnova konteksta, a možda i više toga!). U usporedbi s primjerice međusobnim isključivanjem korištenjem instrukcije `ispitaj_i_postavi` (TAS), gdje je praktički potrebno svega par instrukcija za ulazi i izlaz iz KO u situacijama kada nitko nije u KO, ovo je veliki "gubitak vremena" ("overhead").

Zato sinkronizacijske funkcije u današnjim operacijskim sustavima kombiniraju rad izvan jezgre i pozivaju jezgru samo kada su ti pozivi neophodni – samo kada treba blokirati dretvu ili odblokiratni neku drugu.

Ideja je sljedeća:

- koristiti neku zastavicu za oznaku je li KO slobodan ili nije
- koristiti atomarne operacije kao što je `usporedi_i_zamijeni(var,test,set)`
- kada je KO slobodan s ovakvom operacijom postaviti zastavicu i ne ulaziti u jezgru
- kada je KO zauzet onda treba: 1. označiti da ima dretvi koje čekaju; 2. otići u jezgru i blokirati dretvu
- pri izlasku iz KO, ako nitko ne čeka dovoljno je označiti KO slobodnim – u protivnom treba otići u jezgru i odblokirati dretvu.

Primjer ostvarenja jednostavnog međusobnog isključivanja prikazan je u nastavku.

Ideje/kod: https://eli.thegreenplace.net/2018/basics-of-futexes/

Fast user-space locking: http://man7.org/linux/man-pages/man2/futex.2.html

Primjer algoritma međusobnog isključivanja

Dodatne j-funkcije: `blokiraj_uz_uvjet` i `odblokiraj`

1. u jezgrinim funkcijama se koristi varijabla `ključ` koja **nije dio jezgre već procesa**!
2. jezgra ima posebne redove (stvara takve redove po potrebi) za svaku novu varijablu `ključ` (npr. fizička adresa te varijable je "indeks" takvog reda)

```
j-funkcija blokiraj_uz_uvjet(ključ, vrijednost)          // futex_wait
{
    pohrani kontekst u opisnik Aktivna_D
    ako je (ključ == vrijednost) {
        stavi_u_red(Aktivna_D, posebni_red(ključ))
        stavi_u_red(makni_prvu_iz_reda(Pripravne_D), Aktivna_D)
    }
    obnovi kontekst iz opisnika Aktivna_D
}
j-funkcija odblokiraj(ključ)                             // futex_wake
{
    pohrani kontekst u opisnik Aktivna_D
    ako je (red posebni_red(ključ) neprazan) {
        stavi_u_red(Aktivna_D, Pripravne_D)
        stavi_u_red(makni_prvu_iz_reda(posebni_red(ključ)), Pripravne_D)
        stavi_u_red(makni_prvu_iz_reda(Pripravne_D), Aktivna_D)
    }
    obnovi kontekst iz opisnika Aktivna_D
}
```

Varijabla `ključ` ima vrijednosti:

- 0 => niti jedna dretva nije u KO
- 1 => jedna dretva je u KO, niti jedna druga ne čeka
- 2 => jedna dretva je u KO, možda ima drugih koje čekaju

Sljedeće funkcije `zaključaj` i `otključaj` se izvode izvan jezgre

Rješenje (s puno komentara, bez njih je iza)

```
zaključaj(ključ)
{
    pročitano = atomarno_usporedi_i_zamijeni(ključ, 0, 1)
    //atomarna operacija koja radi sljedeće:
    //1. dohvaća trenutnu vrijednost od "ključ" (u "pročitano")
    //2. uspoređuje dohvaćenu vrijednost s 0 (2. parametar)
    //3. ako je jednaka 0 onda (i samo onda) u "ključ" stavlja 1 (3. parametar)

    //ako je "pročitano" == 0 => ova dretva je zaključala (stavila 1) i može dalje
    //preskače se idući dio koda

    dok je (pročitano != 0) {
        //već je zaključano, ova dretva mora čekati

        //ako ima dretvi koje čekaju, ključ je 2
        //inače (ovo je prva koja čeka) tada treba postaviti ključ u 2
        //i to atomarno, samo ako je ključ == 1
        ako je (pročitano == 2 || atomarno_usporedi_i_zamijeni(ključ, 1, 2) != 0) {
            blokiraj_uz_uvjet(ključ, 2)
        }
        //provjeri je li sada slobodno; ako ne i dalje čekaj
        pročitano = atomarno_usporedi_i_zamijeni(ključ, 0, 2)
        //moguće je da više nema dretvi koje čekaju, a ključ sada ima vrijednost 2
        //ali to ne smeta algoritmu
    }
}
otključaj(ključ)
{
    pročitano = atomarno_dohvati_i_smanji(ključ, 1)
    //"pročitano" je prethodna vrijednost ključa, neposredno prije smanjivanja
    ako je (pročitano != 1) { //ako je pročitano == 2
        ključ = 0
        odblokiraj(ključ)
    }
}
```

Masno označene linije predstavljaju "brzi" način zaključavanja i otključavanja kad nema "sukoba" (više od jedne dretve želi ući, ili jedna dretva želi ući ali druga je već unutra).

Uz "očekivani" način izvođenja, zbog "usporedi" dijela atomarnih operacija, ovo radi i kad nešto i nije očekivanim redom. Npr. dretva koja je u KO izađe iz KO baš nakon što je prva već provjerila nešto. Međutim, i u jezgrinim funkcijama se opet provjerava uvjet prije blokiranja (najprije se u korisničkom načinu postavlja u 2 a onda tek poziva j.f. koja neće blokirati ako u varijabli nije 2).

Rješenje bez komentara

```
zaključaj(ključ)
{
    pročitano = atomarno_usporedi_i_zamijeni(ključ, 0, 1)
    dok je (pročitano != 0) {
        ako je (pročitano == 2 || atomarno_usporedi_i_zamijeni(ključ, 1, 2) != 0) {
            blokiraj_uz_uvjet(ključ, 2)
        }
        pročitano = atomarno_usporedi_i_zamijeni(ključ, 0, 2)
    }
}
otključaj(ključ)
{
    pročitano = atomarno_dohvati_i_smanji(ključ, 1)
    ako je (pročitano != 1) {
        ključ = 0
        odblokiraj(ključ)
    }
}
```

Složenije operacije (npr. semafori, monitori) koriste više bitova u "ključu" i druge atomarne operacije (npr. `atomarno_ispitaj_i_[postavi|obriši]_bit`, `atomarno_ispitaj_i_[povećaj|smanji]`, ...).

Pojednostavljeno rješenje za jednostavni mutex iz nptl/lowlevellock.h

```
ključ: 31. bit označava je li zaključan; ostali bitovi koliko ih čeka na ulaz

void lock (int *mutex)
{
    int v; //hmm ovdje je bio unsigned! ali to nema smisla

    if (atomic_bit_test_set(mutex, 31) == 0)
        return;
    atomic_increment (mutex); //označi da čekaš
    while (1) {
        if (atomic_bit_test_set (mutex, 31) == 0) {
            atomic_decrement (mutex);
            return;
        }
        v = *mutex;
        if (v >= 0) //broj je pozitivan, bit 31 u 0 => nema nikoga unutra!
            continue; //probaj opet ući, bez ulaska u jezgru
        //čekaj, ako se nešto nije promijenilo u međuvremenu
        futex_wait (mutex, v, ...);
    }
}
void unlock (int *mutex)
{
    if (atomic_add_zero(mutex, 0x80000000))
        return;
    //Nakon dodavanja 0x80000000 će mutex biti nula samo ako nitko nije
    //čekao. U protivnom će se samo obrisati taj bit (uz preljev).
    //atomic_add_zero vraća usporedbu prijašnje vrijednosti s poslanom:
    //mutex_stara_vrijednost == -0x80000000
    //a obzirom da je -0x80000000 == 0x80000000 vraća istinu kada nema blokiranih

    //ima dretvi koje čekaju, odblokiraj prvu dretvu
    futex_wake (mutex, 1, ...);
}
```

Pitanja za vježbu 5

1. Što je to *jezgra operacijskog sustava*?
2. Kako se *ulazi* u jezgru?
3. Od čega se jezgra sastoji?
4. U kojim se stanjima može naći dretva?
5. Navesti strukturu podataka jezgre za *jednostavni model jezgre* prikazan na predavanjima.
6. Što su to jezgrine funkcije? Navesti nekoliko njih (rad s UI jedinicama, semafori).
7. Što je to semafor? Koja struktura podataka jezgre je potrebna za njegovo ostvarenje? Skicirati funkciju `Čekaj_OSEM(I)/Postavi_OSEM(I)`.
8. Navesti primjer korištenja jezgrinih funkcija za ostvarenje međusobnog isključivanja (kritičnog odsječka).
9. Kako treba proširiti jezgrine funkcije za primjenu u višeprocesorskim sustavima?
10. Neki sustav se u promatranom trenutku sastoji od 5 dretvi u stanjima: D1 je aktivna, D2 u redu pripravnih, D3 u redu semafor OSEM[1], D4 i D5 u redu odgođenih (D4 treba čekati još jedan kvant vremena, a D5 ukupno još 5). Grafički prikazati stanje sustava (liste s opisnicima). Prikazati stanje sustava:
    a) nakon što dretva D1 pozove `Postavi_OSEM(1)`
    b) nakon što se obradi prekid sata.
11. Sinkronizirati sustav zadataka prikazan grafom (...) binarnim/općim semaforima. Naznačiti početne vrijednosti svih semafora.
12. Proizvodnja nekog elementa odvija se na traci na kojoj postoje tri robota koji svaki rade svoj dio posla, naprije R1, potom R2 te na kraju R3 (nakon tog proizvod se miče na drugu traku i odlazi iz sustava). Pomak trake obavlja se tek kad su sva tri robota završila sa svojim poslom nad zasebnim elementima. Rad pojedinih robota upravljan dretvama `r1()`, `r2()` i `r3()` (`r1` upravlja s R1, `r2` s R2 te `r3` s R3). Ako s `posao_r1()` označimo sam rad na objektom koji radi R1 (za R2 i R3 slično) te s `pomak()` aktivaciju trake i pomak za jedno mjesto, napisati pseudokod za dretve `r1`, `r2` i `r3`. Za sinkronizaciju koristiti binarne/opće semafore te po potrebi i dodatne varijable. Neka je i u početnom stanju ispred svakog robota dio nad kojim on treba obaviti svoje operacije.
13. Za sustav dretvi različita prioriteta koje izvode zadani kod (...), prikazati rad do idućih 15 ispisa.

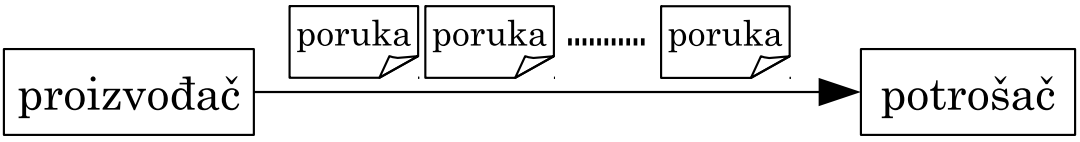
6. MEĐUDRETVENA KOMUNIKACIJA I KONCEPCIJA MONITORA

U ovom se poglavlju prikazuju mnogi primjeri sinkronizacije dretvi korištenjem semafora i monitora. U nastavku je najprije prikazan primjer komunikacije između dretve proizvođača i dretve potrošača. Potom su prikazani problemi koji mogu nastati korištenjem semafora te kako se neki od njih mogu izbjeći korištenjem monitora.

6.1. Primjer sinkronizacije semaforima: proizvođač i potrošač

Zadatak: Ostvariti komunikaciju između jednog proizvođača i jednog potrošača

Spremnik veličine jedne poruke, kakav smo već susretali kod ulazne, radne i izlazne dretve poprilično ograničava rad dretvi. U mnogim slučajevima u kojima dretve komuniciraju, poslovi koje takve dretve rade u prosjeku jednako traju. Međutim, za pojedinu poruku će ponekad jednoj strani trebati više vremena, a ponekad drugoj. Stoga ima smisla dodati međuspremnike većeg kapaciteta od jedne poruke koji će ublažiti razlike u obradi pojedinačnih poruka i omogućiti veću dinamičnost u izvođenju dretvi. U nastavku je stoga krenuto od pretpostavke da u međuspremnik stane više poruka.



Slika 6.1. Proizvođač i potrošač

6.1.1. Načelni pseudokod rješenja

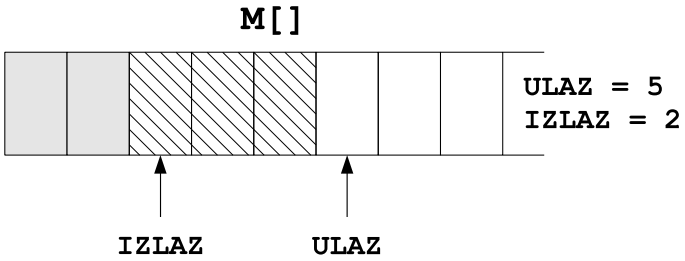
```
proizvođač
{
    ponavljaj {
        P = stvori poruku ()
        pošalji poruku(P)
    }
    do zauvijek
}
```

```
potrošač
{
    ponavljaj {
        čekaj na poruku
        R = uzmi poruku()
        obradi poruku (R)
    }
    do zauvijek
}
```

Kako ostvariti `pošalji poruku(P)`, `čekaj na poruku` i `uzmi poruku()`?

Potrebno je koristiti neke sinkronizacijske mehanizme i zajednički spremnik. Prikažimo to na primjerima u nastavku.

6.1.2. Korištenje neograničenog međuspremnika i radno čekanje



Slika 6.2. Neograničeni međuspremnik

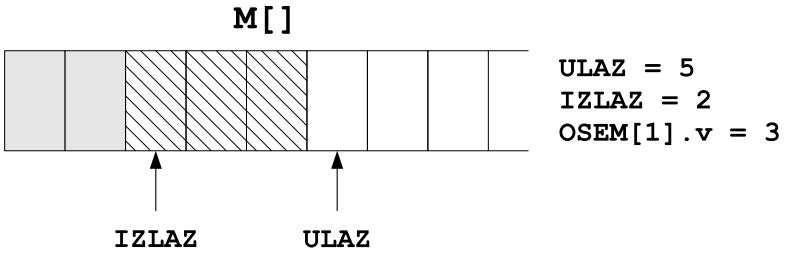
- `M[]` – međuspremnik neograničenje veličine
- `ULAZ` – indeks prvog praznog mjesta u međuspremniku, početno 0
- `IZLAZ` – indeks prvog nepročitanog mjesta u međuspremniku, početno 0

```
proizvođač
{
    ponavljaj {
        P = stvori poruku()
        M[ULAZ] = P
        ULAZ++
    }
    do zauvijek
}
```

```
potrošač
{
    ponavljaj {
        dok je (IZLAZ >= ULAZ) \\radno
            ;                  \\čekanje
        R = M[IZLAZ]
        IZLAZ++
        obradi poruku(R)
    }
    do zauvijek
}
```

Nedostaci: radno čekanje, neograničeni međuspremnik

6.1.3. Korištenje neograničenog međuspremnika i jednog semafora



Slika 6.3. Neograničeni međuspremnik sa semaforima

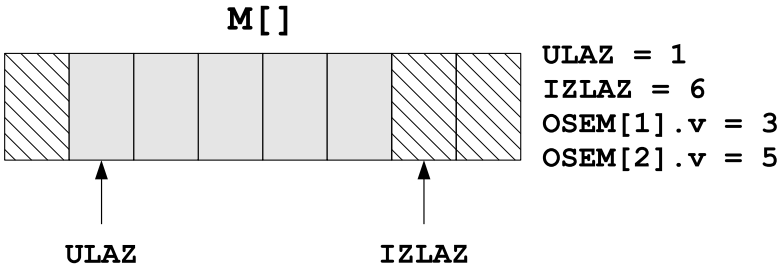
- `OSEM[1]` – broji poruke u međuspremniku, početno 0

```
proizvođač
{
    ponavljaj {
        P = stvori poruku()
        M[ULAZ] = P
        ULAZ++
        Postavi_OSEM(1)
    }
    do zauvijek
}
```

```
potrošač
{
    ponavljaj {
        Čekaj_OSEM(1)
        R = M[IZLAZ]
        IZLAZ++
        obradi poruku(R)
    }
    do zauvijek
}
```

Nedostatak: neograničeni međuspremnik

6.1.4. Korištenje ograničenog međuspremnika i dva semafora



Slika 6.4. Ograničeni međuspremnik

- `M[N]` – međuspremnik s `N` mjesta za poruke
- `ULAZ` i `IZLAZ` se povećavaju po modulu `N`
- `OSEM[1]` – broji poruke u međuspremniku, početno 0
- `OSEM[2]` – broji prazna mjesta u međuspremniku, početno `N` (veličina međuspremnika)

```
proizvođač
{
    ponavljaj {
        P = stvori poruku()
        Čekaj_OSEM(2)
        M[ULAZ] = P
        ULAZ = (ULAZ + 1) MOD N
        Postavi_OSEM(1)
    }
    do zauvijek
}
```

```
potrošač
{
    ponavljaj {
        Čekaj_OSEM(1)
        R = M[IZLAZ]
        IZLAZ = (IZLAZ + 1) MOD N
        Postavi_OSEM(2)
        obradi poruku(R)
    }
    do zauvijek
}
```

Varijablu `ULAZ` koristi samo proizvođač, dok varijablu `IZLAZ` samo potrošač. Međutim, obje dretve koriste međuspremnik `M`. Ipak, vrijednosti u varijablama `ULAZ` i `IZLAZ` te semafori `OSEM[1]` i `OSEM[2]` će spriječiti da obje dretve istovremeno pokušaju koristiti isti element međuspremnika.

Kada bi imali više proizvođača ili više potrošača, onda bi trebali dodati binarne semafore za zaštitu od istovremenog korištenja tih varijabli i istih dijelova međuspremnika.

6.1.5. Više proizvođača i jedan potrošač

Potrebno je dodati jedan binarni semafor za proizvođače da se spriječi istovremeni pokušaj stavljanja u međuspremnik na isto mjesto te promjenu varijable `ULAZ`.

```
proizvođač
{
    ponavljaj {
        P = stvori poruku()
        Čekaj_OSEM(2)
        Čekaj_BSEM(1)
        M[ULAZ] = P
        ULAZ = (ULAZ + 1) MOD N
        Postavi_BSEM(1)
        Postavi_OSEM(1)
    }
    do zauvijek
}
```

```
potrošač
{
    ponavljaj {
        Čekaj_OSEM(1)
        P = M[IZLAZ]
        IZLAZ = (IZLAZ + 1) MOD N
        Postavi_OSEM(2)
        obradi poruku(P)
    }
    do zauvijek
}
```

6.1.6. Više proizvođača i više potrošača

Potrebno je dodati dva binarna semafora: jedan za proizvođače da se spriječi istovremeni pokušaj stavljanja u međuspremnik i promjena varijable `ULAZ`, te jedan za potrošače da se spriječi pokušaj istovremenog uzimanja poruke iz međuspremnika i mijenjanja varijable `IZLAZ`.

```
proizvođač
{
    ponavljaj {
        P = stvori poruku()
        Čekaj_OSEM(2)
        Čekaj_BSEM(1)
        M[ULAZ] = P
        ULAZ = (ULAZ + 1) MOD N
        Postavi_BSEM(1)
        Postavi_OSEM(1)
    }
    do zauvijek
}
```

```
potrošač
{
    ponavljaj {
        Čekaj_OSEM(1)
        Čekaj_BSEM(2)
        P = M[IZLAZ]
        IZLAZ = (IZLAZ + 1) MOD N
        Postavi_BSEM(2)
        Postavi_OSEM(2)
        obradi poruku(P)
    }
    do zauvijek
}
```

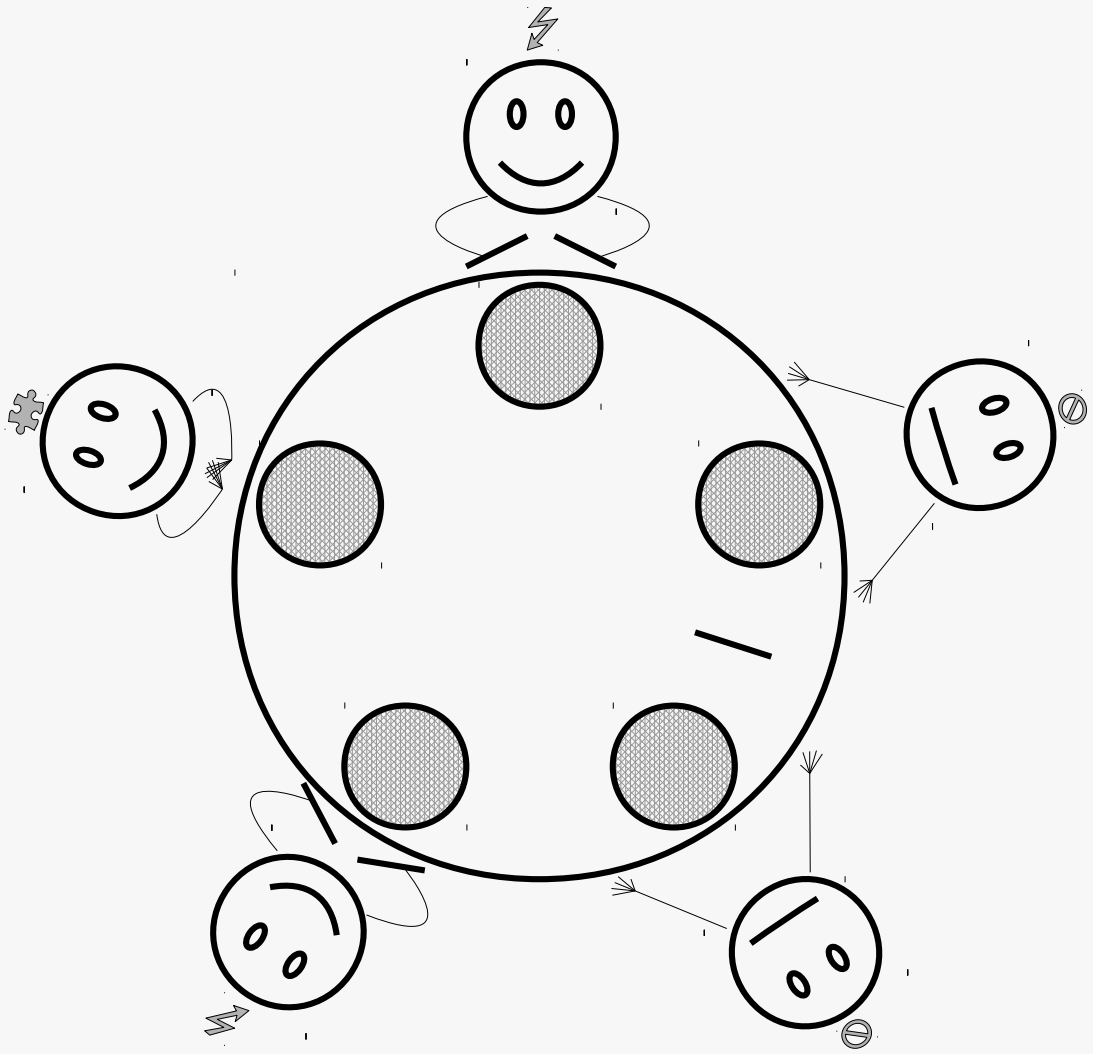
6.2. Problemi sa semaforima

Primjer 6.1. Problem pet filozofa

Pet filozofa sjedi za jednim okruglim stolom. Ispred svakog filozofa je zdjela s hranom (koju konobari nadopunjavaju čim se isprazni). Na stolu se nalazi pet štapića, po jedan između svaka dvije zdjele. Svaki filozof cijelo vrijeme ponavlja sljedeća korake:

1. razmišlja
2. uzima lijevi i desni štapić ako su slobodni, inače čeka da se oslobode
3. jede
4. opere štapiće i vraća ih na stol, lijevo i desno od svoje zdjele

S obzirom da nema dovoljno štapića, neće svi filozofi moći istovremeno jesti.



Slika 6.5. Problem pet filozofa

Na slici 6.5. dva filozofa jedu (na vrhu i dole lijevo), jedan razmišlja (lijevo u sredini) te dva su gladna i čekaju da se oslobode štapići (desno u sredini i desno dole).

U simulaciji filozofa dretvama, sinkronizacija semaforima kod koje bi svaki štapić simulirali jednim semaforom mogla bi izgledati kao u nastavku.

```
dretva Filozof(I)
{
    L = I; D = (I + 1) % 5
    ponavljaj {
        misli
        ČekajBSEM(L)       //uzmi_štapić(L)
        ČekajBSEM(D)       //uzmi_štapić(D)
        jedi
        PostaviBSEM(L)     //vrati_štapić(L)
        PostaviBSEM(D)     //vrati_štapić(D)
    }
    do ZAUVIJEK
}
```

Problem s navedenim rješenjem jest u slučaju da sve dretve paralelno rade te paralelno pozovu operaciju `ČekajBSEM(L)` (`L` je različit za svaku dretvu). Obzirom da bi u tom trenutku svi štapići još bili na stolu, sve bi dretve prošle kroz prvi poziv "čekaj" sa semaforima. Međutim, tada bi svi semafori (svih pet) bilo neprolazno i sve bi dretve zapele na drugom "čekaj" pozivu – nastao bi *potpuni zastoj*.

Na prvi pogled navedeni scenarij izgleda malo vjerojatan. Međutim, ako dretve petlju obavljaju jako brzo, jako puno puta, paralelno, vjerojatnost postaje jako velika.

Problem potpuna zastoja može nastati kada imamo i samo dvije dretve i dva semafora.

Primjer 6.2. Dvije dretve i dva semafora

Neka se dvije dretve zauzimanje sredstava A i B štite binarnim semaforima I i J čije su početne vrijednosti jednake 1.

```
dretva prva()                                dretva druga()
{                                            {
    ponavljaj {                                  ponavljaj {
        nešto              -----------               nešto
        Čekaj_BSEM(I)      -----1-----               nešto
        nešto              -----2-----               Čekaj_BSEM(J)
        Čekaj_BSEM(J)      -----3-----               nešto
        nešto                   4-----               Čekaj_BSEM(I)
        Postavi_BSEM(J)                              nešto
        nešto                                        Postavi_BSEM(I)
        Postavi_BSEM(I)                              nešto
        nešto                                        Postavi_BSEM(J)
        nešto                                        nešto
    }                                            }
    do ZAUVIJEK                                  do ZAUVIJEK
}                                            }
```

U nekom trenutku moguće je da dretve paralelno izvode linije koda koje su u istom retku. Pozivi `Čekaj_BSEM()` u linijama 1 i 2 će propustiti dretve obzirom da su u tim trenucima ti semafori bili prolazni. Međutim, pozivi u linijama 3 i 4 blokirati će dretve jer su sada ti semafori neprolazni – obje su dretve blokirane – dogodio se potpuni zastoj.

6.2.1. Potpuni zastoj

Nužni uvjeti za mogućnost nastanka potpunog zastoja su:

1. bar dvije dretve i bar dva sredstva koje te dretve obje koriste
2. u svakom trenutku sredstvo smije koristiti samo jedna dretva (međusobno isključivo)
3. dretvi se sredstvo ne može oduzeti, ona ga sama otpušta kada joj više ne treba
4. dretva drži dodijeljeno sredstvo dok traži dodatno sredstvo

Može li se neki uvjet maknuti?

Prva tri nema smisla – mijenja se logika programa.

Četvrti uvjet se može promijeniti drukčijim programiranjem:

- neka dretva zauzima potrebna sredstva odjednom, ne pojedinačno
- zauzeti oba semafora odjednom – preko jedne jezgrine funkcije kao atomarne operacije

Problem: do sada prikazane jezgrine funkcije za rad sa semaforima nemaju takvu operaciju. Mogli bi ih proširiti ili uvesti druge mehanizme.

Današnji operacijski sustavi imaju sučelja za rad atomarnih operacija nad skupom semafora, primjerice:

- `semop` na *UNIX*-u
- `WaitForMultipleObjects` na Win*

Mnogo sinkronizacijskih problema ne može se riješiti oslanjanjem samo na vrijednost semafora. Stoga se u nastavku neće uvoditi jezgrine funkcije za atomarne operacije nad skupom semafora već će se prikazati drugi mehanizam sinkronizacije koji se ne oslanja na vrijednosti semafora.

Prije toga, prikažimo algoritam koji može otkriti mogućnost za nastanak potpuna zastoja kada više sudionika treba više različitih sredstava.

6.2.2. Bankarev algoritam (info)

Pretpostavke:

- razmatra se sustav s N procesa $\{P_1, P_2, \dots P_N\}$ i M različitih sredstava $\{S_1, \dots S_M\}$
- u nekom trenutku:
  - broj slobodnih sredstava neka je: $S = \{s_1, \dots s_M\},\ s_k \geq 0\ \forall k \in \{1, \dots M\}$
  - broj zauzetih sredstava od procesa P_i je: $Z_i = \{z_{i,1}, \dots z_{i,M}\},\ z_{i,k} \geq 0\ \forall k \in \{1, \dots M\}$
  - broj sredstava koji će još trebati procesu P_i: $T_i = \{t_{i,1}, \dots t_{i,M}\},\ t_{i,k} \geq 0\ \forall k \in \{1, \dots M\}$
- svaki proces P_i ne otpušta već zauzeta sredstva sve do kraja svog rada
- na kraju rada proces P_i vraća sva sredstva (S se povećava)

Bankarev algoritam

- provjerava je li sustav u *stabilnom stanju* – postoji li redoslijed izvođenja procesa u kojem će svi procesi moći završiti s radom, tj. dobiti sva potrebna sredstva
- ako sustav nije u stabilnom stanju *potpuni zastoj* je neizbježan (nema dovoljno sredstava da bi završili procesi jedan po jedan)

- algoritam radi tako da među procesima koji još nisu završili traži proces P_j čiji su zahtjevi T_j manji ili jednaki raspoloživim sredstvima, tj. za takav proces treba vrijediti (u tom trenutku):

$$T_j \leq S, \quad \text{odnosno:} \quad t_{j,k} \leq s_k, \quad \forall k \in \{1, 2 \ldots M\} \tag{6.1.}$$

- ako se takav proces ne može pronaći sustav nije u stabilnom stanju
- kada se takav proces P_j pronađe, onda se on prebaci u skup završenih procesa a slobodnim sredstvima se vrate sva zauzeta sredstva tog procesa
  - pojednostavljeno: ako vrijedi (6.1.) tada makni P_j te povećaj S: $S = S + Z_j$

Bankarev algoritam se može koristiti pri posluživanju zahtjeva

- simulira se prelazak sustava u novo stanje odobravanjem zahtjeva
- provjerava se je li novo stanje stabilno
- ako novo stanje ne bi bilo stabilno zahtjev treba odbiti (npr. dretvu treba blokirati dok se njen zahtjev ne bude mogao odobriti uz ostanak u stabilnom stanju)

Primjer 6.3. Primjer rada bankareva algoritma

Neka je stanje sustava u nekom zadano sa:

```
         Z:zauzeto  T:trebaju   S:slobodno
procesi  A  B  C    A  B  C     A  B  C
   P1    1  0  2    5  2  0     4  7  2
   P2    3  0  3    1  5  3
   P3    0  2  2    4  5  1
   P4    2  1  1    2  4  2
```

U provjeri stabilnosti treba tražiti procese koji mogu dobiti tražena sredstva da bi završili s radom, tj. za koje procese se u tom trenutku mogu zadovoljiti zahtjevi. Za proces `P1` nema dovoljno sredstava tipa `A`, za proces `P2` nema dovoljno sredstava tipa `C`. Procesi `P3` i `P4` mogu završiti (jedan po jedan). Primjer stanja sustava nakon svakog završetka jest:

```
                           S:slobodno
                           A  B  C
zatečeno stanje:           4  7  2  -  mogu P3 i P4, kreće P3
nakon završetka P3 +(0 2 2): 4  9  4  -  mogu P2 i P4, kreće P2
nakon završetka P2 +(3 0 3): 7  9  7  -  mogu P1 i P4, kreće P1
nakon završetka P1 +(1 0 2): 8  9  9  -  može P4, kreće P4
nakon završetka P4 +(2 1 1): 10 10 10 -  svi gotovi
```

S obzirom na to da su svi procesi mogli završiti (navedenim redoslijedom) sustav se nalazio u stabilnom stanju. Kada se u simulaciji više procesa može odabrati, redoslijed odabira među tim procesima je nebitan (može prvo jedan pa drugi ili obratno).

Potpuni zastoj bi nastao ako bi u početnom stanju (uz `slobodno=(4 7 2)`) udovoljili zahtjevu procesa `P1` za tri sredstva tipa `A`. Tada bi sustav otišao u nestabilno stanje (uz `slobodno=(1 7 2)` te niti jedan proces više ne bi mogao završiti).

6.3. Monitori

Problemi sa semaforima (npr. potpuni zastoj) nastaju u sustavima s više dretvi i više sredstava ili složenijim problemima sinkronizacije gdje semafori nisu dovoljni.

C.A.R. Hoare je 70'tih predložio drukčiji mehanizam sinkronizacije:

- sve kritične radnje koje uključuju zajednička sredstva obavljati u kontroliranom okruženju – monitorskim funkcijama – u monitoru
- monitorske funkcije su vrlo slične kritičnim odsječcima (zapravo to i jesu) i jezgrinim funkcijama, ali NISU jezgrine funkcije već korisničke
- monitor ima proširenu funkcionalnost (dodatne operacije osim 'uđi' i 'izađi')
- u monitoru se mijenjaju i ispituju varijable koje opisuju stanje sustava
- ako stanje sustava nije povoljno za dretvu da ona nastavi s radom (npr. nema lijevog i desnog štapića) dretva se blokira – stavlja u *red uvjeta* i privremeno napušta monitor – jednom jezgrinom funkcijom!
- blokiranu dretvu odblokira druga jezgrinim funkcijama
- u monitorskim funkcijama je uvijek najviše jedna dretva aktivna (blokirane ne brojimo)
- mogli bi reći da je monitor proširenje binarnog semafora

Problem pet filozofa bi mogli IDEJNO riješiti monitorom:

```
dretva Filozof(I)
{
    ponavljaj {
        misli
        uzmi_štapiće(I)  // monitorska funkcija
        jedi
        vrati_štapiće(I) // monitorska funkcija
    }
    do zauvijek
}

m-funkcija uzmi_štapiće(I)
{
    dok je (oba štapića oko filozofa I nisu slobodna)
        blokiraj dretvu

    uzmi oba štapića oko filozofa I
}

m-funkcija vrati_štapiće(I)
{
    vrati oba štapića oko filozofa I
    propusti desnog filozofa ako čeka
    propusti lijevog filozofa ako čeka
}
```

6.3.1. Podrška jezgre (jezgrine funkcije) za ostvarenje monitora

U nastavku su navedene jezgrine funkcije za ostvarenje monitora (u prva dva izdanja knjige je to "suvremeni monitor" – ali mi samo njega koristimo!).

(Ne traži se reprodukcija koda, već "samo" razumijevanje što koja funkcija radi.)

```
j-funkcija Uđi_u_monitor(M) //mutex_lock
{
    pohrani kontekst u opisnik Aktivna_D

    ako je (Monitor[M].v == 1) {
        Monitor[M].v = 0
    }
    inače {
        stavi_u_red(makni_prvu_iz_reda(Aktivna_D), Monitor[M])
        odaberi_aktivnu_dretvu()
    }

    obnovi kontekst iz opisnika Aktivna_D
}
```

```
j-funkcija Izađi_iz_monitora(M) //mutex_unlock
{
    pohrani kontekst u opisnik Aktivna_D

    ako je (red Monitor[M] nije prazan)
    {
        stavi_u_red(makni_prvu_iz_reda(Monitor[M]), Pripravne_D)
        odaberi_aktivnu_dretvu()
    }
    inače {
        Monitor[M].v = 1
    }

    obnovi kontekst iz opisnika Aktivna_D
}
```

[dodatno]

Funkcije `Uđi_u_monitor` i `Izađi_iz_monitora` su identične funkcijama za rad s binarnim semaforima `Čekaj_BSEM` i `Postavi_BSEM` te se kao i on mogu koristiti za ostvarenje kritična odsječka. Međutim, nad monitorom se obavljaju i druge operacije; uz njega su povezane i druge strukture podataka.

Također, ulazak u monitor vezuje dretvu s monitorom. Ovdje to nismo radili radi jednostavnosti, ali u "stvarnim sustavima" se to provjerava. Stoga te funkcije nisu predviđenje za sinkronizaciju ulazne, radne i izlazne dretve na način kao što je korišten binarni semafor.

```
funkcija Čekaj_u_redu_uvjeta(R, M)  //cond_wait - nije jezgrina funkcija
{
    Uvrsti_u_red_uvjeta(R, M)
    Uđi_u_monitor(M)
}
```

```
j-funkcija Uvrsti_u_red_uvjeta(R, M) //nema ekv. u sustavima
{
    pohrani kontekst u opisnik Aktivna_D

    stavi_u_red(makni_prvu_iz_reda(Aktivna_D), Red_uvjeta[R])
    ako je (red Monitor[M] nije prazan) {
        stavi_u_red(makni_prvu_iz_reda(Monitor[M]), Pripravne_D)
    }
    inače {
        Monitor[M].v = 1
    }
    odaberi_aktivnu_dretvu()

    obnovi kontekst iz opisnika Aktivna_D
}
```

```
j-funkcija Oslobodi_iz_reda_uvjeta(R)       //cond_signal
{
    pohrani kontekst u opisnik Aktivna_D

    ako je (red Red_uvjeta[R] nije prazan) {
        stavi_u_red(makni_prvu_iz_reda(Red_uvjeta[R]), Pripravne_D)
        odaberi_aktivnu_dretvu()
    }

    obnovi kontekst iz opisnika Aktivna_D
}
```

```
j-funkcija Oslobodi_sve_iz_reda_uvjeta(R)   //cond_broadcast
{
    pohrani kontekst u opisnik Aktivna_D

    ako je (red Red_uvjeta[R] nije prazan) {
        dok je (red Red_uvjeta[R] neprazan)
            stavi_u_red(makni_prvu_iz_reda(Red_uvjeta[R]), Pripravne_D)
        odaberi_aktivnu_dretvu()
    }

    obnovi kontekst iz opisnika Aktivna_D
}
```

Tablica 6.1. Ekvivalencija poziva s pozivima stvarnih sustava (info)

Funkcija	POSIX	Win32 (Vista+)
Uđi_u_monitor	pthread_mutex_lock	EnterCriticalSection
Izađi_iz_monitora	pthread_mutex_unlock	LeaveCritivalSection
Čekaj_u_redu_uvjeta	pthread_cond_wait	SleepConditionVariableCS
Oslobodi_iz_reda_uvjeta	pthread_cond_signal	WakeConditionVariable
Oslobodi_sve_iz_reda_uvjeta	pthread_cond_broadcast	WakeAllConditionVariable

6.4. Primjeri sinkronizacije semaforima i monitorima

Osmišljavanje višedretvenog programa je znatno složenije i može dovesti do raznih grešaka i problema (neki već spomenuti). Što se očekuje od ispravnog rješenja problema?

Što znači "ispravno sinkronizirati dretve/procese/zadatke"?

- Redoslijed izvođenja mora biti istovjetan opisu u tekstu zadatka.
- Međusobno isključivo treba koristiti sredstava za koje se to zahtijeva ili je iz zadatka očito da treba.
- Nema mogućnosti za nastanak potpunog zastoja.
- Nema radnog čekanja.
- Uz algoritam potrebno je navesti početne vrijednosti svih varijabli i semafora.

Zadatak 6.1. Problem pet filozofa

```
dretva Filozof(I)
{
   ponavljaj {
      misli
      uzmi_štapiće(I)
      jedi
      vrati_štapiće(I)
   }
   do zauvijek
}
```

`štapić[i] = 1` – i-ti štapić slobodan (početna vrijednost)
`štapić[i] = 0` – i-ti štapić zauzet

```
m-funkcija uzmi_štapić(I)
{
   L = I; D = (I + 1) MOD 5

   Uđi_u_monitor(m)
   dok je (štapić[L] + štapić[D] < 2)
      Čekaj_u_redu_uvjeta(red[I], m)
   štapić[L] = 0
   štapić[D] = 0
   Izađi_iz_monitora(m)
}
```

```
m-funkcija vrati_štapiće (I)
{
   L = I; D = (I + 1) MOD 5
   LF = (L - 1) MOD 5; DF = D
   Uđi_u_monitor (m)
   štapić[L] = 1
   štapić[D] = 1
   Oslobodi_iz_reda_uvjeta(red[LF], m)
   Oslobodi_iz_reda_uvjeta(red[DF], m)
   Izađi_iz_monitora(m)
}
```

Dretva filozofa će ili uzeti oba štapića ili neće niti jedan. Na taj način je izbjegnut potpuni zastoj. Iako je uvjet koji se ovdje ispituje jednostavan, ista logika, tj. struktura koda bi vrijedila i za složene uvjete. To će se vidjeti kroz primjere u nastavku.

Iako navedeno rješenje radi ispravno, detaljnijom analizom mogu se pronaći neki nedostaci. Primjerice, u prikazanom rješenju može se dogoditi stanje u kojem filozofi oko jednog (koji čeka) naizmjence jedu, ali tako da se mali dio vremena preklapaju u toj aktivnosti (kad oba jedu). Tada ovaj između njih koji čeka nikada neće imati oba štapića slobodna, dok će njegovi susjedi moći jesti puno puta.

Za primjer, razmotrimo filozofe s indeksima 1, 2 i 3 te ih označimo s F1, F2 i F3. Neka prvi u monitor uđe filozof F1. Nakon što je F1 uzeo štapiće i izašao iz monitora, u monitor ulazi filozof F2. S obzirom na to da njemu nedostaje lijevi štapić, on će se blokirati u redu uvjeta

(i privremeno izaći iz monitora). Neka tada u monitor uđe F3, koji ima oba štapića slobodna (pretpostavimo da filozof F4 jako dugo misli). F3 uzima štapiće i izlazi iz monitora. Idući događaj neka bude vraćanje štapića od strane F1. On će u tom postupku poslati i signal prema F2, ali će F2 nakon provjere ponovno biti blokiran (sada mu nedostaje onaj drugi štapić koji ima F3). Ako F1 ponovno dođe i želi uzeti štapiće prije nego li je F3 vratio svoje, opet će ih naći slobodne te ih uzeti. Ako sada F3 vrati štapiće, F2 opet neće moći uzeti oba jer mu je opet F1 uzeo lijevi. F1 i F3 se tako mogu u nedogled izmjenjivati i onemogućiti filozofu F2 da dođe do oba štapića.

Navedeni problema nazivamo problemom *izgladnjivanja*.

U nastavku je prikazano rješenje koje se zasniva na tome da filozof koji ima oba štapića on ih ipak neće uzeti ako bilo koji od njegovih susjeda (lijevi i desni) čekaju puno više od njega. Međutim, ovo limitira paralelnost u korištenju sredstava – nije optimalno, iako ublažava izgladnjivanje.

```
T = 0 – referentno "vrijeme", koje se povećava događajima
t[5] – trenutak kada filozof hoće nešto; MAX kada ima ili mu ne treba
N – dozvoljena razlika u broju obroka među susjednim filozofima

m-funkcija uzmi_štapiće(I)
{
    L = I; D = (I + 1) MOD 5
    LF = (L – 1) MOD 5; DF = D
    Uđi_u_monitor(m)
    T++
    t[I] = T
    dok je (štapić[L] + štapić[D] < 2 ILI t[LF] < t[I] – N ILI t[DF] < t[I] – N)
        Čekaj_u_redu(red[I], m)
    štapić[L] = 0
    štapić[D] = 0
    t[I] = MAX
    Izađi_iz_monitora(m)
}
```

Razlika na izvornu funkciju je označena masno. Funkciju za vraćanje štapića nije potrebno mijenjati.

Zadatak 6.2. Problem starog mosta

Stari most je uski most i stoga postavlja ograničenja na promet. Na njemu istovremeno smiju biti najviše tri automobila koja voze u istom smjeru. Simulirati automobile dretvom `Auto` koja obavlja niže navedene radnje. Napisati pseudokod monitorskih funkcija `Popni_se_na_most(smjer)` i `Siđi_s_mosta()`.

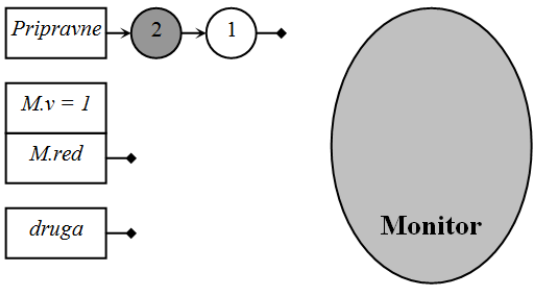
```
Dretva Auto(smjerA) // smjerA = 0 ili 1
{
    Popni_se_na_most(smjerA)
    prijeđi_most
    Siđi_s_mosta(smjerA)
}
```

Zadatak 6.3. Stanje strukture podataka na jednom primjeru

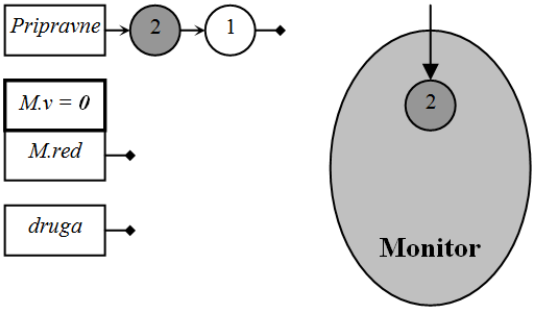
U sustavu se nalaze dvije dretve koje obavljaju kod prema pseudokodu ispod. Nakon stvaranja dretvi one se nalaze u redu pripravnih prema [1]. Redovi su složeni po redu prispijeća. Stanja sustava za vrijeme rada tih dretvi prikazana su u nastavku. Koriste se monitori s labosa.

```
Dretva1 {                       Dretva2 {
   mutex_lock(M);                  mutex_lock(M);
   ispiši("Prva");                 dok je (prva == 0)
   prva = 1;                          cond_wait(druga, M);
   cond_signal(druga, M);          ispiši("Druga");
   mutex_unlock(M);                mutex_unlock(M);
}                               }
```

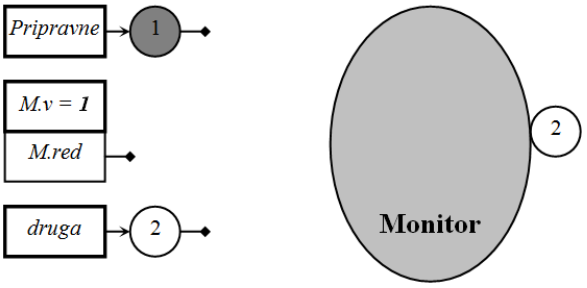
[1] Početno stanje



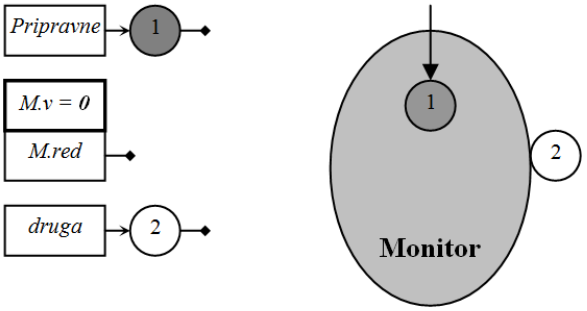
[2] (dretva 2) `mutex_lock(M);`



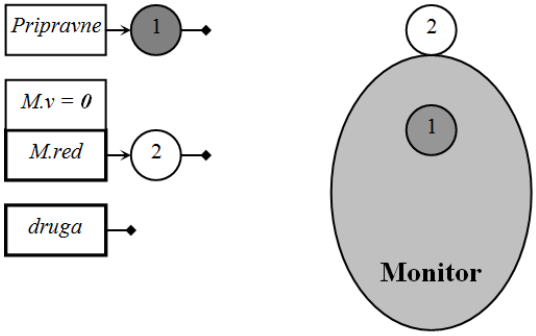
[3] (dretva 2) `dok je (prva == 0) cond_wait(druga, M);`



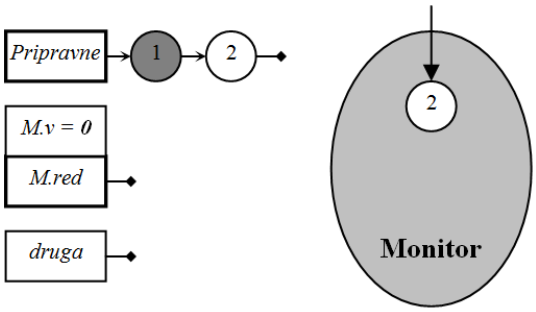
[4] (dretva 1) `mutex_lock(M);`



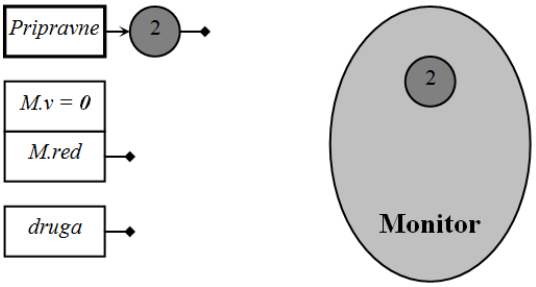
[5] (dretva 1) `ispiši("Prva"); prva = 1; cond_signal(druga, M);`



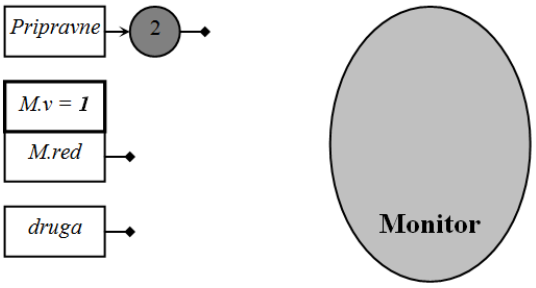
[6] (dretva 1) `mutex_unlock(M);`



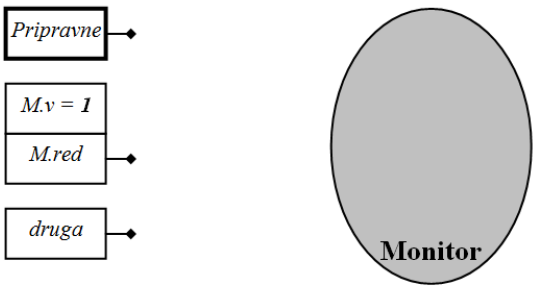
[7] Dretva 1 `završava`



[8] (dretva 2): `ispiši("Druga"); mutex_unlock(M);`



[9] Dretva 2 završava



Zadatak 6.4. Problem pušača

Neka se za rješenje koristi jedan monitor s dva reda uvjeta. U prvom će redu čekati trgovac kada čeka da se stol isprazni, a u drugom pušači koji čekaju da se na stolu pojave sastojci koji njima trebaju.

```
dretva Trgovac()
{
  ponavljaj {
    (s1, s2) = nasumice odaberi ...

    Uđi_u_monitor(m)

    dok je (stol_prazan != 1)
      Čekaj_u_redu_uvjeta(m, red[0])

    stavi_sastojke_na_stol(s1, s2)
    Oslobodi_sve_iz_reda_uvjeta(red[1])
    Izađi_iz_monitora(m)
  }
  do ZAUVIJEK
}
```

```
dretva Pušač(p)
{
  (r1,r2)=sastojci_koje_pušač_nema(p)

  ponavljaj {
    Uđi_u_monitor (m)

    dok(na_stolu() != (r1, r2))
      Čekaj_u_redu_uvjeta (m, red[1])

    uzmi_sastojke(r1, r2)
    stol_prazan = 1
    Oslobodi_iz_reda_uvjeta(red[0])
    Izađi_iz_monitora(m)

    napravi cigaretu ...
  }
  do ZAUVIJEK
}
```

Početne vrijednosti: `stol_prazan = 1`

Zadatak 6.5. Ping-pong dretve

Simulirati rad dretvi *ping* i dretvi *pong*. Dretve se nasumično pojavljuju u sustavu (stvaraju ih neke druge dretve) i u svom radu samo ispisuju poruku: dretve *ping* ispisuju `ping` dok dretve *pong* ispisuju `pong`. Sinkronizirati rad dretvi tako da:

a) ispis bude pojedinačan (unutar kritičnog odsječka)

b) dretve *ping* i *pong* naizmjence ispisuju poruke (ispis: `ping pong ping pong ...`)

c) dretve *ping* i *pong* ispisuju poruke tako da se uvijek pojavljuju dva `ping`-a prije svakog `pong`-a (ispis: `ping ping pong ping ping pong ...`)

d) dretve *ping* i *pong* ispisuju poruke tako da se uvijek pojavljuju barem dva `ping`-a prije svakog `pong`-a (ispis: `ping ping ping pong ping ping pong ping ...`)

Zadatak 6.6. Parking

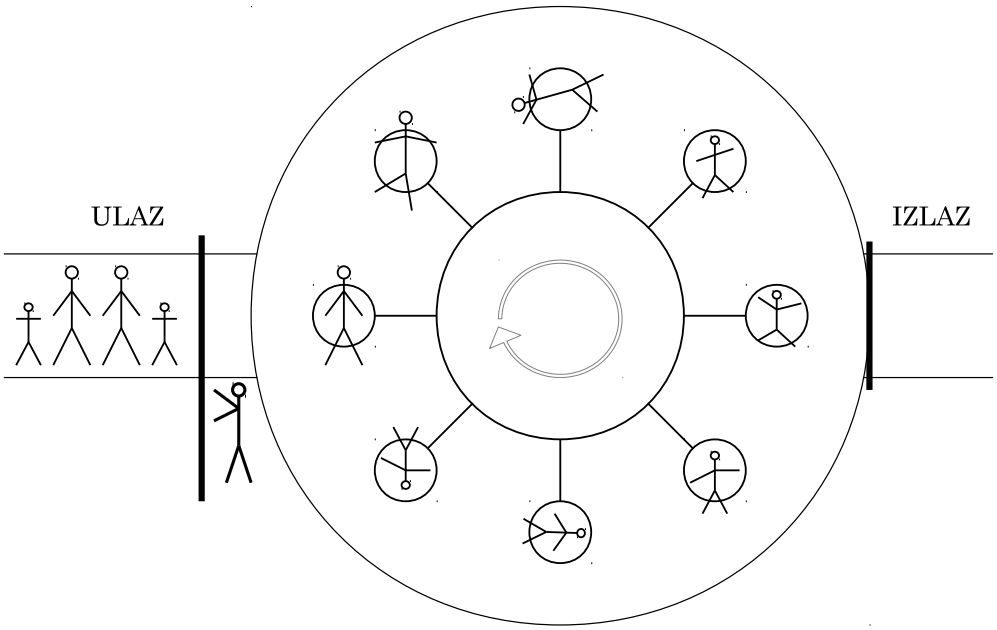
Neki parking (npr. FER-ov) ima dvije rampe: jednu za ulaz i drugu za izlaz. Automobili ulaze na parking uz pomoć daljinskog uređaja. Svaki uređaj ima jedinstveni broj za koji u bazi sustava postoji definirano stanje: 0 – auto nije na parkingu i 1 – auto je na parkingu. Ulazak je automobilima moguć samo u stanju 0, a izlazak samo u stanju 1. Automobil ne može ući ako je na parkingu već MAX automobila. Simulirati sustav dretvama, tj. napisati funkcije `ulaz(id)` i `izlaz(id)` koje pozivaju auti s brojem uređaja `id`.

Ponekad se greškom ne evidentira ulazak ili izlazak automobila te se ne promijeni stanje uređaja i broj mjesta. Ako je potrebno, portir ima pristup bazi i može promijeniti stanje automobila i

broj mjesta. Dodati monitorske funkcije `portir_postavi(id)` i `portir_obriši(id)` koje poziva portir, a koje postavljaju odgovarajuće stanje uređaja i broj mjesta.

Zadatak 6.7. Vrtuljak

Modelirati vrtuljak s dva tip dretvi: dretvama *posjetitelj* (koje predstavljaju posjetitelje koji žele na vožnju) te dretvom *vrtuljak* koja predstavlja sam vrtuljak (upravljanje vrtuljkom). Dretvama *posjetitelj* se ne smije dozvoliti ukrcati na vrtuljak prije nego li prethodna grupa ode te kada više nema praznih mjesta (`N`), a pokretanje vrtuljka napraviti tek kada je pun.



Zadatak 6.8. Restoran (Zadatak 6.2. Microsoft i Linux programeri)

Na nekom katu jedne zgrade nalaze se uredi tvrtke A (npr. s lijeve strane), tvrtke B (npr. s desne strane) te restoran (na kraju hodnika). Restoran koriste zaposlenici obiju tvrtki, ali zbog osjetljivosti njihova posla ne smije se dopustiti da se u njemu istovremeno nalaze zaposlenici jedne i druge tvrtke (u radno vrijeme; u hodniku, koji je pod video nadzorom mogu biti istovremeno i jedni i drugi). Simulirati zaposlenike dretvom `Zaposlenik(tvrtka)`.

```
Zaposlenik(tvrtka) //tvrtka je 0 ili 1
{
    ponavljaj {
        radi nešto u uredu
        udi_u_restoran(tvrtka)
        popij/pojedi nešto, pročitaj novine, ...
        izadi_iz_restorana(tvrtka)
    }
    do kraja radnog vremena
}
```

Ostvariti funkcije označene masno korištenjem monitora. Riješiti i problem izgladnjivanja.

Zadatak 6.9. Čitači i pisači

Riješiti problem sinkronizacije dretvi čitača i pisača (napisati pseudokod funkcija `pisač()` i `čitač()`) korištenjem semafora i/ili monitora te dodatnih varijabli (po potrebi). Pretpostaviti da pisači u kritičnom odsječku pišu funkcijom `piši(X)`, a čitači čitaju funkcijom `čitaj(Y)`.

Kada neki čitač čita niti jedan pisač ne smije pisati, dok istovremeno drugi čitači mogu čitati. Kada neki od pisača piše svi ostali moraju čekati (i čitači i pisači).

Zadatak 6.10. Normalni i kritični poslovi

Neki poslužitelj obrađuje dvije vrste poslova: normalne i kritične. Prihvat novog posla obavlja dretva prihvat (po programu desno). Obradu poslova radi više dretvi radna – čiji tekst programa je zadatak napisati. Kada neka radna dretva odabire koji će posao uzeti uvijek će najprije uzeti kritični posao, ako takav postoji. Iznimno, ako u nekom trenutku N ili više dretvi obrađuje kritične poslove, radna dretva koja bi mogla uzeti novi posao (bilo koji) to ne smije napraviti, već treba čekati. Stoga, svaki put kada dretva završi obradu kritičnog posla, ona mora osloboditi sve blokirane dretve. Za sinkronizaciju koristiti monitore. Uzimanje poslova obaviti funkcijama `uzmi_kritičan()` i `uzmi_normalan()` (koje postoje – nije ih potrebno ostvarivati), a obradu funkcijom `obavi_posao(posao)`. Pozivanje funkcija za preuzimanje poslova mora biti međusobno isključivo (funkcije nemaju ugrađenu sinkronizaciju).

```
dretva prihvat {
   ponavljaj {
      posao = čekaj_novi_posao()
      Uđi_u_monitor(m)
      ako je posao.tip == KRITICAN {
         dodaj_kritičan(posao)
         br_k++  (broj krit. poslova)
      } inače {
         dodaj_normalan(posao)
         br_n++  (broj norm. poslova)
      }
      Oslobodi_iz_reda_uvjeta(red)
      Izađi_iz_monitora(m)
   } do zauvijek
}
```

```
Varijable:
- m - monitor
- red - red uvjeta
- posao.tip => NORMALAN, KRITICAN
- br_k - broj kritičnih poslova u redu
- br_n - broj normalnih poslova u redu

Za rješenje treba još:
- kr_dr - broj dretvi koje rade kritične
     poslove
```

Zadatak 6.11. Kratki i dugi poslovi

U nekom sustavu postoje dva tipa dretvi: jedna dretva `prihvati` i `N` dretvi `obradi`. Dretva `prihvati` čeka i zaprima nove poslove sa: `posao = dohvati_idući()`. Posao može biti kratki ili dugi (`posao.tip == KRATKI/DUGI`). Kratke poslove dretva `prihvati` stavlja u njihov red sa `stavi_kratki(posao)`, a duge u njihov sa `stavi_dugi(posao)`. Dretve `obradi` preko `posao=uzmi_kratki()/uzmi_dugi()` uzimaju posao iz jednog ili drugog reda te ga obrađuju s `obradi(posao)`. Dretva `obradi` treba odabrati dugi posao osim ako:

- nema dugih poslova u njihovu redu (a kratkih ima)
- ima više od 10 kratkih poslova u njihovu redu
- ima kratkih poslova u njihovu redu i trenutno barem pet dretvi već obrađuje duge poslove.

Napisati pseudokod za oba tipa dretvi korištenjem monitora za sinkronizaciju. Korištenje redova (operacije `stavi*` i `uzmi*`) zaštititi. Obrade različitih poslova od strane različitih dretvi moraju se moći obaviti i paralelno (izvan monitora).

Zadatak 6.12. H_2O

Sinkronizirati dretve vodika i kisika koje stvaraju molekulu vode H_2O. Dretve se stvaraju nasumično. Za stvaranje molekule koristi se "kalup". Kad sva tri atoma budu u kalupu molekula

vode nastaje. Dretva završava s radom kad uđe u kalup, osim ako to nije ona koja treba "sastaviti molekulu" i izbaciti ju iz kalupa.

Zadatak 6.13. Predavanja

U nekom hipotetskom sustavu na predavanja dolazi N studenata. Svaki student pri ulasku u dvoranu treba se prijaviti preko zasebnog uređaja (prijava ide slijedno, student po student). Nakon što uđe N studenata u dvoranu može ući predavač. Po završetku sata, studenti odlaze i odjavljuju se (preko istog uređaja). Tek kad zadnji student izađe, izlazi i predavač. Napisati dretve `student()` i `predavač()` koji koriste semafore/monitore za sinkronizaciju na gore opisani način.

Zadatak 6.14. Barijera

Korištenjem monitora ostvariti mehanizam barijere, tj. funkcije `b_init(b,N)` i `b_wait(b)`. Mehanizam barijere radi na načelu da zastavlja rad dretvi na barijeri (dretve pozivaju `b_wait(b)`) dok sve dretve (njih N) ne dođu do barijere. Kad zadnja dretva dođe do barijere onda sve mogu nastaviti dalje s radom.

Zadatak 6.15. Igra na poteze

U nekoj igri na poteze (npr. šah, kartanje i sl.) niz akcija koje sudionici poduzimaju se mogu opisati sa: inicijalizacijom, te igrom koja započinje potezom prvog igrača, pa tek onda idućeg itd. Ako sudionike modeliramo dretvama `igrač` (svakog igrača svojom dretvom) te početnu inicijalizaciju i proglašenje rezultata dretvom `sustav` napisati pseudokod tih dretvi. Koristiti monitore.

Zadatak 6.16. Poslovi i potpuni zastoj

U nekom sustavu nasumično se stvaraju dretve A i dretve B. Dretve A za prvi dio posla `posaoAx()` trebaju samo sredstvo X, a za drugi dio `posaoAxy()` i sredstvo Y. Dretva A drži sredstvo X dok ne obavi oba dijela posla (ne otpušta X nakon `posaoAx()`). Dretve B za svoj rad (`posaoBy()`) trebaju samo sredstvo Y. Sinkronizirati dretve korištenjem binarnih semafora (po jedan za svako sredstvo). Može li se u takvom sustavu pojaviti potpuni zastoj? Obrazložiti.

Zadatak 6.17. Besplatno piće na sajmu

Sustav kojeg treba simulirati dretvama sastoji se od više *konobara* i više *posjetitelja*. Svaki konobar obavlja ciklički posao: uzima čistu praznu čašu; puni je ili čajem ili kuhanim vinom ili pivom (svaki konobar uvijek puni isto); stavlja punu čašu na šank te ponavlja posao. Postoje tri aparata za punjenje pića koji mogu raditi paralelno: jedan za čaj, drugi za kuhano vino a treći za pivo. Svaki aparat se koristi pojedinačno (npr. dok se jedan čaj ne natoči ostali konobari koji žele natočiti čaj čekaju). Stavljanje pića na zajednički stol također treba obaviti pojedinačno. Posjetitelji se kao i konobari dijele po tipu na one koji hoće čaj, one koji hoće vino te one koji hoće pivo. Više posjetitelja može paralelno uzimati piće sa šanka. Posjetitelj koji želi piće kojeg trenutno nema na stolu čeka (ali ne sprječava druge da uzmu svoje piće). Po uzimanju pića posjetitelj se miče od stola (nestaje iz simuliranog sustava).

6.5. Dodatno o sinkronizaciji (info)

6.5.1. Problemi sa sinkronizacijom

Sinkronizacijski mehanizmi su neophodni. Međutim, treba poznavati njihova svojstva i način korištenja. Osim već spomenutih problema potpuna zastoja i izgladnjivanja, uz sinkronizacijske mehanizme može se navesti još nekoliko problema.

Rekurzivno zaključavanje je problem kada dretva koja je već zaključala neki objekt (npr. pozvala i prošla kroz `pthread_mutex_lock(&lock)`) to pokuša opet, prije nego li ga je otključala. Drugi pokušaj može biti programska greška ili željeno ponašanje (npr. pri ulasku u funkciju `a()` objekt se zaključa prvi put, iz `a()` se pozove `b()` na čijem ulazu se koristi ista sinkronizacijska funkcija s istim objektom). Neki sinkronizacijski mehanizmi dopuštaju takva zaključavanja (uz prikladnu inicijalizaciju).

Inverzija prioriteta je problem u kojem dretva manjeg prioriteta blokira kritičnu dretvu većeg prioriteta tako što je zauzela neko sredstvo (npr. semafor) i nije ga otpustila u trenutku kad prioritetnija dretva treba to sredstvo. Obično dretve zauzmu sredstvo za vrlo kratko vrijeme, pa ovo i ne bi bio problem da ne postoji mogućnost da se tada aktiviraju i druge dretve manjeg prioriteta od kritične, ali većeg od početne, koje mogu nedefinirano dugo odgađati izvođenje dretve manjeg prioriteta, a time i nastavak rada kritične dretve. Ovaj problem je posebno opasan u sustavima za rad u stvarnom vremenu u kojem je neophodno da kritična dretva što prije dobije sredstvo i nastavi s radom. Algoritmi koji se u takvim slučajevima mogu koristiti su protokol nasljeđivanja prioriteta te protokoli stropnog prioriteta (detaljnije opisani u okviru predmeta Sustavi za rad u stvarnom vremenu).

6.5.2. Dodatne mogućnosti sinkronizacijskih operacija

Ponekad je potrebno zauzeti sredstvo, ali ne pod cijenu blokiranja dretve. Naime, u slučaju da je sredstvo već zauzeto (npr. neka druga dretva je u kritičnom odsječku), može biti potrebno da dretva nešto drugo napravi, a ne da se zaustavi na tom pozivu. Stoga postoje pozivi tipa `Probaj_Čekati` (npr. `sem_trywait`, `pthread_mutex_trylock`) koji će se ponašati identično kao obična funkcija kad je objekt slobodan, ali će u protivnom vratiti grešku i neće blokirati dretvu.

Ponekad se blokiranje i može podnijeti, ako je ono vremenski ograničeno, tj. ako će trajati manje od neke zadane vrijednosti. U protivnom, nakon isteka tog vremena dretvu treba odblokirati i pustiti je da nastavi s radom bez traženog sredstva i ta će operacija vratiti grešku nakon isteka zadanog vremena ako se u međuvremenu toj dretvi sredstvo nije dodijelilo. Takva sučelja su tipa `Ograničeno_Čekaj` (npr. `sem_timedwait`, `pthread_mutex_timedlock`).

Pitanja za vježbu 6

1. Sinkronizirati više proizvođača i više potrošača koji komuniciraju preko ograničenog međuspremnika korištenjem semafora (i dodatno potrebnih varijabli). Ako se u međuspremniku kapaciteta N poruka u promatranom trenutku nađe M poruka, koje su vrijednosti korištenih općih semafora?
2. Sinkronizirati dvije dretve tako da one neizmjence obavljaju svoje kritične odsječke.
3. Što je to “potpuni zastoj”?
4. Prikazati primjer nastanka potpunog zastoja u sustavu koji koristi semafore za sinkornizaciju.
5. Navesti nužne uvjete za nastanak potpunog zastoja.
6. Opisati koncept monitora.
7. Navesti jezgrine funkcije potrebne za ostvarenje monitora.
8. Koje su tipične operacije koje se izvode unutar monitorske funkcije?
9. Monitorima sinkronizirati dretve koje simuliraju rad “pet filozofa”.
10. Koje sve aspekte uključuje “ispravna sinkronizacija”?
11. Što je to “izgladnjivanje” u kontekstu sinkronizacije i korištenja zajedničkih sredstava?

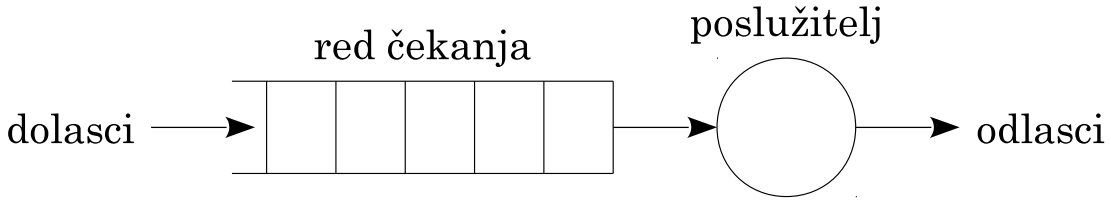
7. ANALIZA VREMENSKIH SVOJSTAVA

Ili: kako raspoređivati dretve?

Da bi to mogli odrediti trebamo analizirati dinamičko ponašanje računalnog sustava. Kako?

- simulacijom
- praćenjem stvarnog sustava
- korištenjem modela $\Longleftarrow$ ovako ćemo mi

7.1. Deterministički sustav



Slika 7.1. Model poslužitelja

- svi događaji su poznati ili predvidljivi
- neki posao se u sustavu pojavljuje u trenutku t_d a iz njega odlazi u t_n
- vrijeme zadržavanje posla u sustavu je:

$$T = t_n - t_d \tag{7.1.}$$

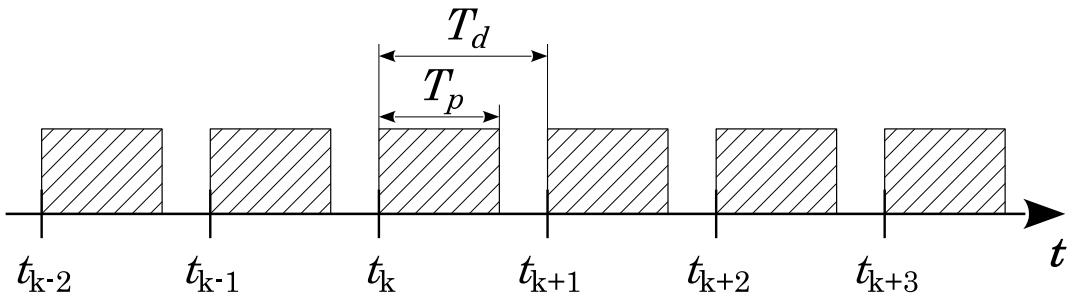
- kad dođe u sustav, poslužitelj može biti slobodan ili zauzet
  - ako je poslužitelj zauzet novi posao čeka u redu
- vrijeme čekanja u redu označavamo sa: T_r
- vrijeme posluživanja posla označimo sa: T_p
- vrijeme zadržavanja posla u sustavu označimo sa: T

$$T = T_r + T_p \tag{7.2.}$$

- ako je red uvijek prazan:

$$T_p = T \tag{7.3.}$$

Ako su svi poslovi isti i periodički dolaze s periodom T_d (mogli bi reći i da se jedan posao ponavlja s tom periodom) tada takve poslove požemo prikazati slikom 7.2.



Slika 7.2. Periodički poslovi

Da se poslovi ne bi gomilali mora biti:

$$T_p \leq T_d \tag{7.4.}$$

Iskoristivost procesora:

$$\rho = \frac{T_p}{T_d} \tag{7.5.}$$

ili u postocima:

$$\eta = \frac{T_p}{T_d} \times 100\% \tag{7.6.}$$

Recipročna vrijednost periode dolaska novih poslova T_d je:

$$\alpha = \frac{1}{T_d} \tag{7.7.}$$

- α – broj dolazaka novih poslova u jedinici vremena
- $\frac{1}{\alpha}$ – vrijeme (interval) između dva dolaska

Slično, recipročna vrijednost od T_p je:

$$\beta = \frac{1}{T_p} \tag{7.8.}$$

- β – broj poslova koje bi poslužitelj mogao obraditi u jedinici vremena
- $\frac{1}{\beta}$ – vrijeme obrade (posluživanja)

Očito je:

$$\rho = \frac{\alpha}{\beta} \tag{7.9.}$$

ρ – iskoristivost poslužitelja (opterećenje)

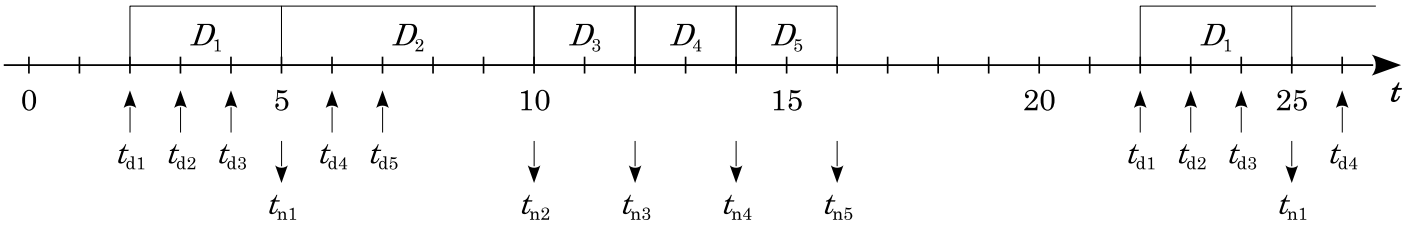
Kada bi T_p bilo jednako T_d poslužitelj bi imao 100% iskoristivost. To je dozvoljeno samo u deterministčkim slučajevima!!!

Zadatak 7.1. (ispitni zadatak)

Pretpostavimo da sustav obrađuje pet poslova koji u njega dolaze periodno s periodom od 20 jedinica vremena. Trenuci dolazaka u prvoj periodi koja započinje s t = 0 i trajanje poslova navedeni su u tablici.

Posao	t_d	T_p
D_1	2	3
D_2	3	5
D_3	4	2
D_4	6	2
D_5	7	2

Ta će se skupina poslova ponavljati s periodom od 20 jedinica vremena.



Vremensko ponašanje može se opisati tablicom:

Posao	t_d	T_p	t_n	T	T_r
D_1	2	3	5	3	0
D_2	3	5	10	7	2
D_3	4	2	12	8	6
D_4	6	2	14	8	6
D_5	7	2	16	9	7

Oznake:

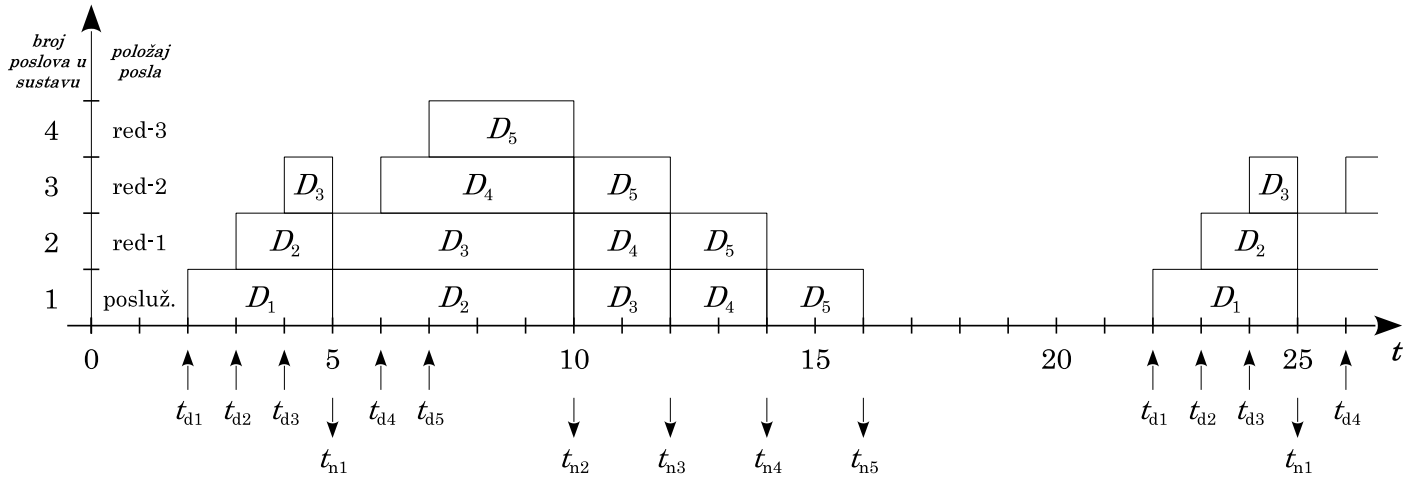
- $\overline{T}$ – prosječno zadržavanje poslova u sustavu
- $\overline{T}_r$ – prosječno čekanje u redu
- $\overline{T}_p$ – prosječno trajanje posluživanja

$$\overline{T} = \frac{3+7+8+8+9}{5} = 7 \tag{7.10.}$$

$$\overline{T}_p = \frac{3+5+2+2+2}{5} = 2.8 \tag{7.11.}$$

$$\overline{T}_r = \overline{T} - \overline{T}_r = 4.2 \tag{7.12.}$$

Prosječan broj poslova u sustavu: $\overline{n} = ?$



Slika 7.3. Broj poslova u sustavu

$\overline{n} =?$ se može izračunati kao integral površine podijeljen vremenom periode (20):

$$\overline{n} = \frac{35}{20} = 1,75 \tag{7.13.}$$

površina = suma zadržavanja svih poslova u sustavu

Iz slike se $\overline{n}$ može izračunati prema:

$$\overline{n} = \frac{T_1 + T_2 + T_3 + T_4 + T_5}{20} = \frac{5 \times \overline{T}}{20} = \frac{5}{20} \times \overline{T} = 1,75 \tag{7.14.}$$

S obzirom na to da je broj poslova u jedinici vremena $\alpha = \frac{5}{20}$ slijedi:

$$\overline{n} = \alpha \times \overline{T} \Longrightarrow \text{Littleovo pravilo} \tag{7.15.}$$

Littleovo pravilo vrijedi općenito, ne samo za determinističke sustave!

Intuitivni dokaz (statistički gledano):

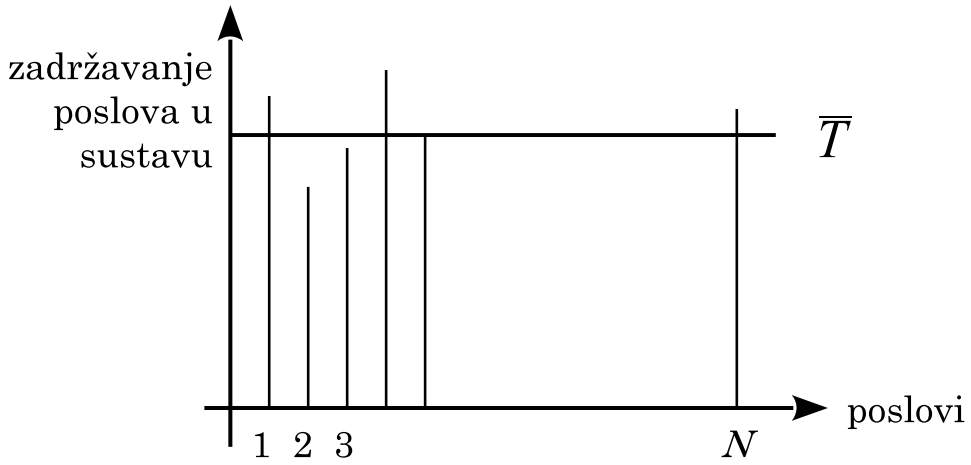
Promatrajmo vremenski interval $\overline{T}$. U njemu dođe $\alpha \cdot \overline{T}$ novih poslova (α u jedinici vremena). Na početku tog intervala u sustavu su samo poslovi koji su prije došli, a na kraju samo oni koji su došli za vrijeme tog intervala (jer statistički se poslovi zadržavaju $\overline{T}$ pa su svi koji su došli prije i izašli tijekom $\overline{T}$). Dakle, na kraju intervala u sustavu imamo $\alpha \cdot \overline{T}$ poslova. Ako u tom trenutku imamo toliko poslova zašto bi u nekom drugom trenutku to bilo različito (statistički gledano)?

Pokušaj dokaza Littleova pravila (informativno)

Promatrajmo veći vremenski interval $T \gg \overline{T}$

U tom intervalu poslužitelj obradi N poslova. Ako je T dovoljno velik, onda se rubni slučajevi mogu zanemariti (poslovi koji su počeli prije intervala i dovršili izvođenje u intervalu te poslovi koji su započeli u intervalu a dovršili iza njega). Razmotrimo detaljnije tih N poslova.

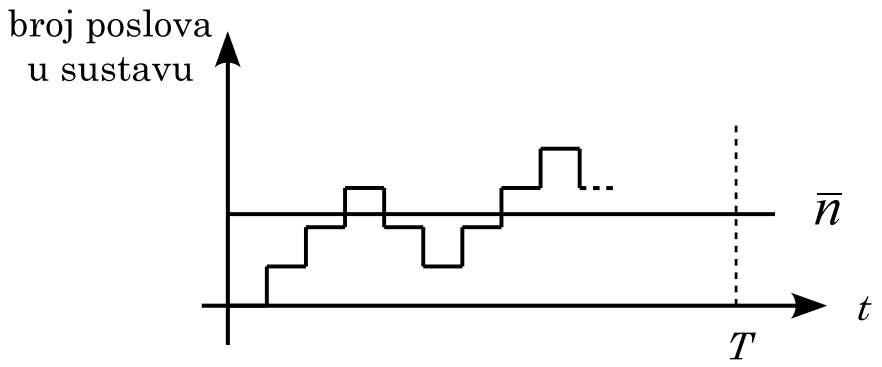
Graf zadržavanja poslova u sustavu prema pojedinom poslu (poslovi su numerirani od 1 do N) neka izgleda:



Iz slike slijedi:

$$\overline{T} = \frac{\sum T_i}{N}, \quad \alpha = \frac{N}{T} \tag{7.16.}$$

Isti se poslovi s njihovim trajanjima mogu pokazati i u grafu koji pokazuje broj poslova u sustavu u nekom trenutku. Karikirano, takva slika izgleda kao u nastavku



To su isti poslovi kao i na prijašnjem grafu samo "polegnuti" od trenutka pojave do napuštanja sustava.

Srednji broj poslova se može izračunati kao:

$$\overline{n} = \frac{\text{površina}}{\text{period}} = \frac{\sum T_i}{T} \tag{7.17.}$$

Ako se to raspiše (pomnoži s "1" i iskoriste prijašnje formule):

$$\overline{n} = \frac{\sum T_i}{T} = \frac{\sum T_i}{T} \times \frac{N}{N} = \frac{\sum T_i}{N} \times \frac{N}{T} = \overline{T} \times \alpha = \alpha \times \overline{T} \Longrightarrow \text{Littleovo pravilo} \tag{7.18.}$$

7.2. Nedeterministički sustav

Pretpostavke:

- dolasci se podvrgavaju Poissonovoj razdiobi s parametrom (očekivanjem) α
  - α je *prosječan* broj dolazaka novih poslova u jedinici vremena
  - $\frac{1}{\alpha}$ je *prosječno* vrijeme između dolaska dva posla
- trajanje obrade podvrgava se eksponencijalnoj razdiobi s parametrom (očekivanjem) $\frac{1}{\beta}$
  - $\frac{1}{\beta}$ je *prosječno* trajanje obrade jednog posla
  - β je *prosječan* broj poslova poslova koje poslužitelj može obraditi u jedinici vremena

Zašto te razdiobe?

- zato jer modeliraju stvarne sustave
- zato jer se inače koriste

7.2.1. O razdiobama (info)

Malo "matematike" ...

U nekom stohastičkom sustavu *slučajna varijabla* x može poprimiti neku vrijednost Y s vjerojatnošću $p(x = Y) = p(Y) =$ "vrijednost u intervalu [0;1]".

Slučajna varijabla može biti *diskretna* (poprimati samo neke vrijednosti) ili *kontinuirana* (poprimati sve vrijednosti iz nekog intervala). Stoga se i razdiobe dijele na:

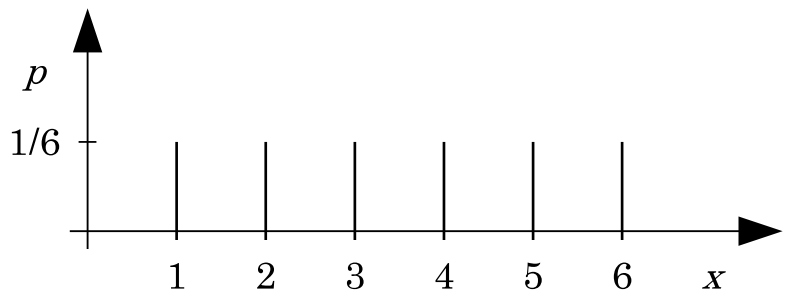
- diskretne
- kontinuirane

7.2.1.1. Diskretne razdiobe

U diskretnim sustavima slučajna varijabla može poprimiti diskretne vrijednosti (npr. samo prirodne brojeve).

Primjer: uniformna razdioba – bacanje kockice

- "Koja je vjerojatnost da dobijemo neki broj?"
- slučajna varijabla x može poprimiti vrijednosti: 1, 2, 3, 4, 5 i 6

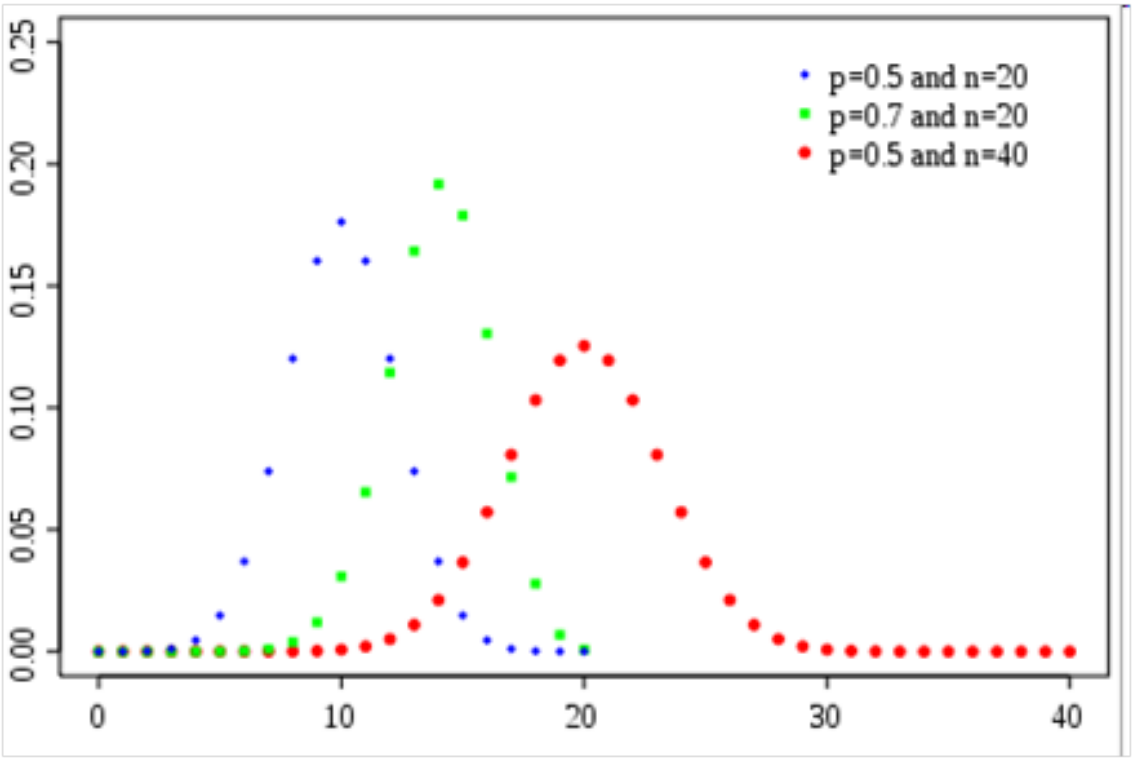


Slika 7.4. Uniformna diskretna razdioba (bacanje jedne kockice)

- vjerojatnost da x poprimi neku od vrijednosti je: $p(x) = 1/6$
- obzirom da je vjerojatnost bilo kojeg događaja jednaka, kažemo da se x podvrgava *uniformnoj razdiobi*
- suma vjerojatnosti po svim mogućim vrijednostima od x: $\sum_{i=1}^{6} p(i) = 1$

Primjer: binomna razdioba – višestruko bacanje kockica

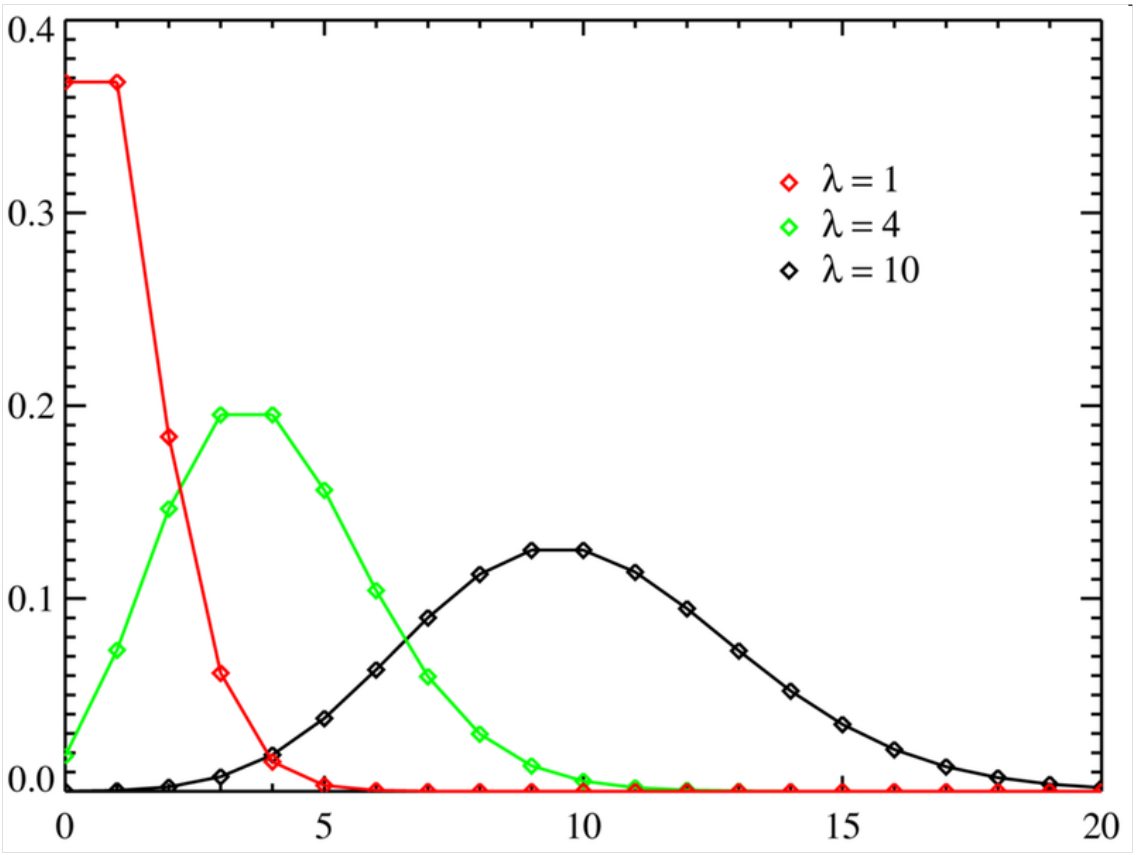
- "Koja je vjerojatnost da u n bacanja dobijemo x šestica?"
- bitna pretpostavka: svako bacanje je nezavisno – ne utječe na ostala
- slučajna varijabla x može poprimiti vrijednosti: 0, 1, 2, ..., n
- vjerojatnost da x poprimi neku od vrijednosti je:
  $p\,(x, n, p) = \binom{n}{x} p^x (1-p)^{n-x}$ (p – vjerojatnost dobivanja šestice u jednom bacanju)
- suma vjerojatnosti po svim mogućim vrijednostima od x: $\sum_{x=0}^{n} p(x) = 1$



Slika 7.5. Primjer binomne razdiobe

Primjer: Poissonova razdioba – broj zahtjeva u jedinici vremena (web, pošta, ...)

- "Koja je vjerojatnost da u nekom intervalu T dođe x novih zahtjeva?"
- slučajna varijabla x može poprimiti vrijednosti: 0, 1, 2, ..., ∞
- vjerojatnost da x poprimi neku od vrijednosti je: $p(x, T, \lambda) = \frac{(\lambda T)^x}{x!} e^{-\lambda T}$
  (uz T – vremenski interval, λ – prosječan broj dolazaka novih zahtjeva u jedinici vremena)
- suma vjerojatnosti po svim mogućim vrijednostima od x: $\sum_{x=0}^{\infty} p(x) = 1$

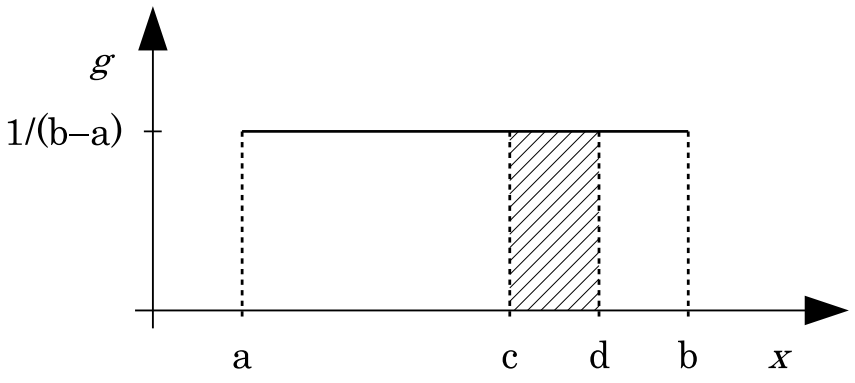


Slika 7.6. Primjer Poissonove razdiobe

7.2.1.2. Kontinuirane razdiobe

Primjer: uniformna razdioba – slučajan odabir jedne točke na pravcu između točke a i b

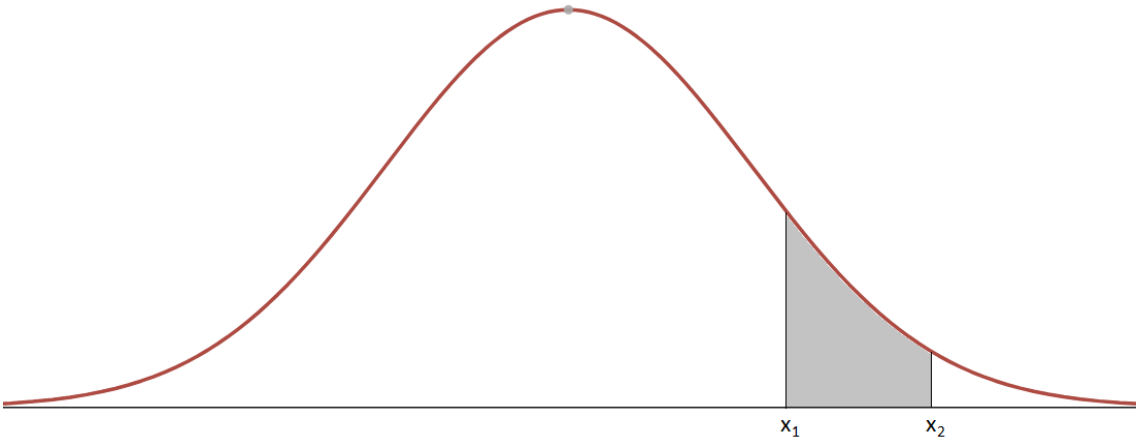
- vjerojatnost odabira bilo koje točke je jednaka
- problem: između a i b ima beskonačno točaka!
- vjerojatnost da je odabrana jedna točka je nevjerojatno mala (nula)
- stoga se ne računa vjerojatnost odabira neke pojedine točke već vjerojatnost da je točka iz intervala $[c; d]$
- za takve sustave zadaje se funkcija gustoće vjerojatnosti g
- vjerojatnost da slučajna varijabla x poprimi vrijednost iz intervala $[c; d]$ računa se kao integral funkcije gustoće vjerojatnosti od c do d



Slika 7.7. Primjer uniformne razdiobe

Primjer: Gausova razdioba – visina slučajno odabranog studenta

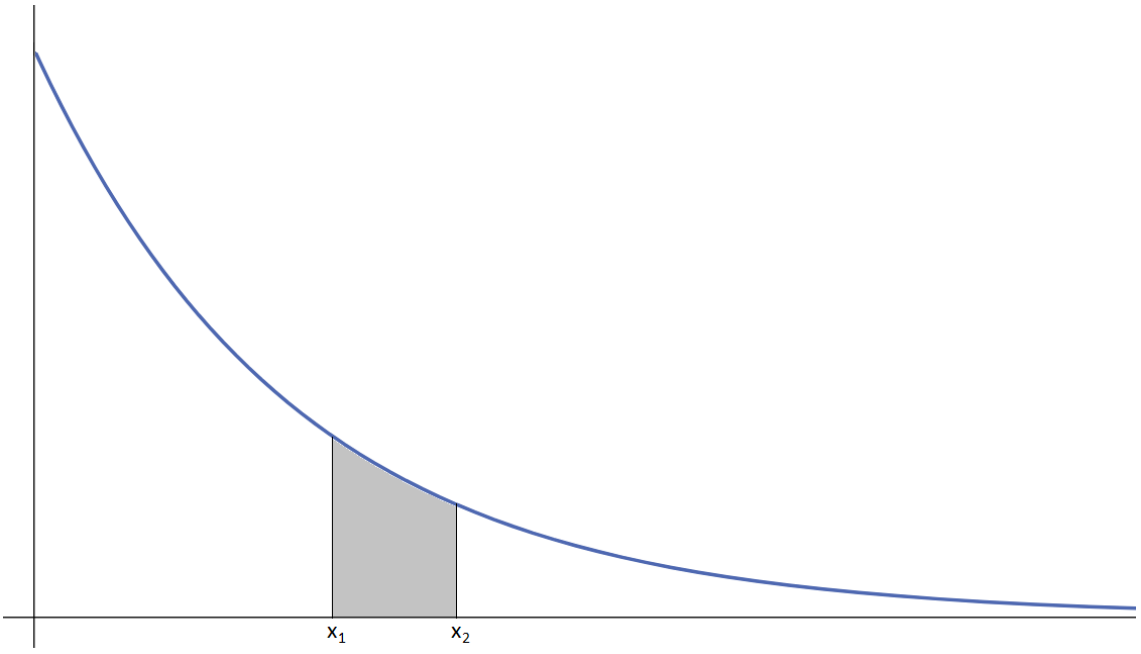
- vjerojatnost da je odabrani student "prosječne visine" veći će nego da je jako visok ili jako nizak
- vjerojatnost da je visina odabranog studenta točno x je vrlo mala (nula, i ovdje se radi o kontinuiranoj razdiobi, x može poprimiti bilo koju vrijednost, npr. $x = 179,271863...$)
- funkcija gustoće vjerojatnosti za Gaussovu razdiobu ima poznati "zvonolik" oblik, odnosno, računa se prema formuli: $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$
- vjerojatnost da slučajna varijabla x poprimi vrijednost iz intervala $[x_1; x_2]$ računa se kao integral funkcije gustoće vjerojatnosti od x_1 do x_2



Slika 7.8. primjer Gaussove razdiobe

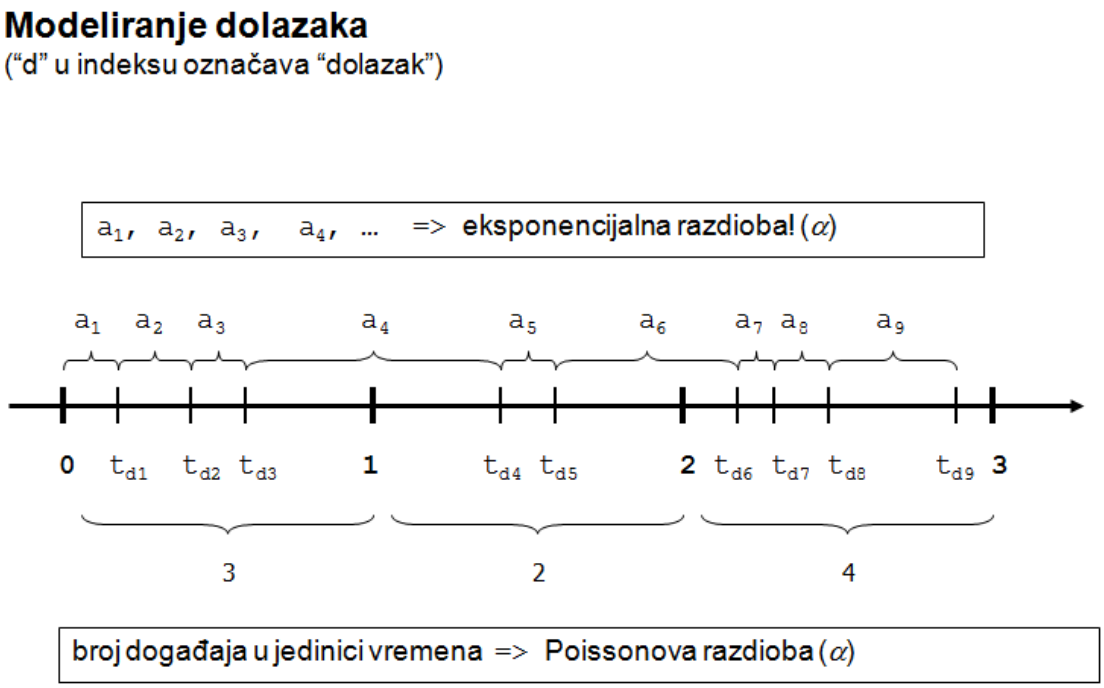
Primjer: eksponencijalna razdioba – vrijeme između dva događaja

- npr. vrijeme između dva zahtjeva, trajanje obrade i slično
- funkcija gustoće vjerojatnosti za eksponencijalnu razdiobu se računa prema formuli: $f(x) = \lambda e^{-\lambda x}$ (za $x \geq 0$)



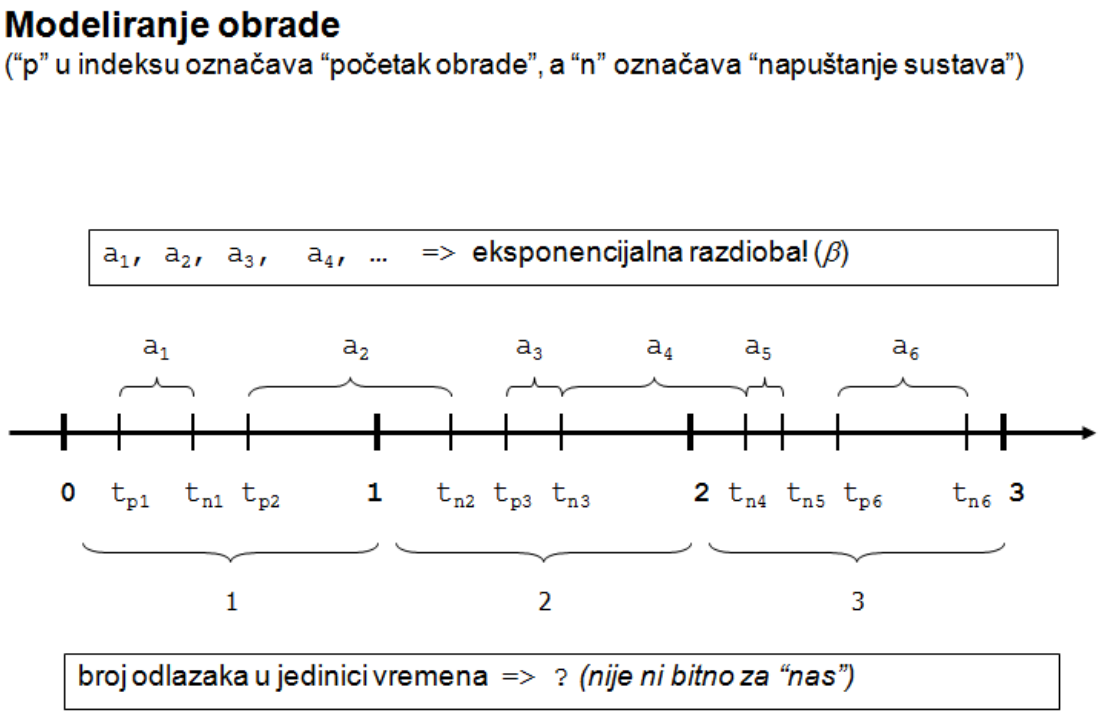
Slika 7.9. Primjer eksponencijalne razdiobe

7.2.2. Modeliranje dolazaka poslova



Slika 7.10. Modeliranje broja dolazaka poslova Poissonovom razdiobom

7.2.3. Modeliranje obrade poslova

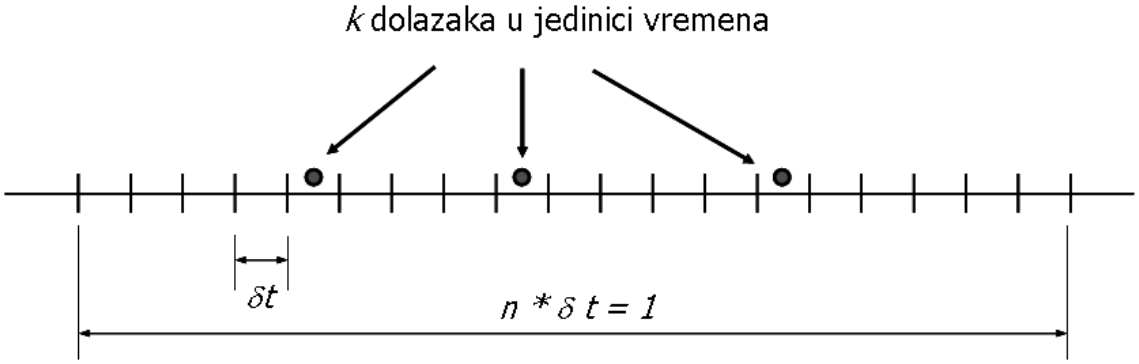


Slika 7.11. Modeliranje trajanja obrade eksponencijalnom razdiobom

7.3. Poissonova razdioba (izvod iz binomne)

Razmatramo jedinični vremenski interval (jedna sekunda).

Podijelimo ga na n odsječaka.



Neka je vjerojatnost događaja u jednom odsječku: p

Suprotna vjerojatnost (odsustvo događaja) je: $1 - p = q$

Vjerojatnost da se dogodilo k događaja u jedinici vremena je:

$$\boxed{b\left(k,n,p\right) = \binom{n}{k} p^k q^{n-k}} \tag{7.19.}$$

- n, p => parametri razdiobe
- k => slučajna varijabla

Primjer 7.1. Dobivanje 2 šestice u 10 bacanja kocke

$$b\left(2,10,\frac{1}{6}\right) = \binom{10}{2}\left(\frac{1}{6}\right)^2\left(\frac{5}{6}\right)^{10-2} = 0,29 \tag{7.20.}$$

Vjerojatnost da se nije dogodio niti jedan događaj (u jedinici vremena):

$$b\left(0,n,p\right) = q^n = \left(1-p\right)^n \tag{7.21.}$$

Vjerojatnost da se dogodio barem jedan događaj (u jedinici vremena):

$$b\left(k>0,n,p\right) = 1 - q^n \tag{7.22.}$$

Poissonova razdioba dobiva se kada $n \longrightarrow \infty$ i $p \longrightarrow 0$, ali tako da umnožak $n \cdot p$ bude konačna vrijednost: $\lambda = n \cdot p$:

$$b\left(0,n,p\right) = q^n = \left(1-p\right)^n = \left(1 - \frac{\lambda}{n}\right)^n = \left[\left(1 + \frac{1}{-\frac{n}{\lambda}}\right)^{-\frac{n}{\lambda}}\right]^{-\lambda} \tag{7.23.}$$

Kada $n \longrightarrow \infty$ tada:

$$\begin{aligned} p\left(0,\lambda\right) &= \lim_{n\to\infty}\left[\left(1+\frac{1}{-\frac{n}{\lambda}}\right)^{-\frac{n}{\lambda}}\right]^{-\lambda} = \left[\lim_{n\to\infty}\left(1-\frac{1}{n}\right)^{-n}\right]^{-\lambda} \\ &= \left[\lim_{n\to\infty}\left(\frac{n-1}{n}\right)^{-n}\right]^{-\lambda} = \left[\lim_{n\to\infty}\left(\frac{n}{n-1}\right)^{n}\right]^{-\lambda} \\ &= \left[\lim_{n\to\infty}\left(1+\frac{1}{n-1}\right)^{n}\right]^{-\lambda} = \left[\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^{n}\right]^{-\lambda} = e^{-\lambda} \end{aligned} \tag{7.24.}$$

Kako izračunati $p\left(k,\lambda\right)$ – vjerojatnost k događaja u jedinici vremena?

Opet krenemo od binomne razdiobe i omjera:

$$\frac{b(k,n,p)}{b(k-1,n,p)} = \frac{\binom{n}{k}p^k q^{n-k}}{\binom{n}{k-1}p^{k-1}q^{n-(k-1)}} = \frac{n-(k-1)}{k}\frac{p}{q} = \frac{n}{k}\frac{p}{q} - \frac{k-1}{k}\frac{p}{q} \tag{7.25.}$$

Prelaskom na limese: $n \longrightarrow \infty$, $p \longrightarrow 0$, $q \longrightarrow 1$, $n \cdot p \longrightarrow \lambda$:

$$\frac{p(k,\lambda)}{p(k-1,\lambda)} = \frac{\lambda}{k\cdot 1} - 0 = \frac{\lambda}{k} \quad \longrightarrow \quad p(k,\lambda) = \frac{\lambda}{k}\cdot p(k-1,\lambda) \tag{7.26.}$$

te indukcijom

$$\begin{aligned} p(1,\lambda) &= \frac{\lambda}{1}\cdot p(0,\lambda) = \frac{\lambda}{1}e^{-\lambda} \\ p(2,\lambda) &= \frac{\lambda^2}{2}e^{-\lambda} \\ p(3,\lambda) &= \frac{\lambda^3}{3!}e^{-\lambda} \\ &\vdots \end{aligned} \tag{7.27.}$$

$$\boxed{p(k,\lambda) = \frac{\lambda^k}{k!}e^{-\lambda}} \tag{7.28.}$$

Što je λ? Očito $n\cdot p$, ali ovisi kako dijelimo jedinični interval! A n i p koristimo samo pri izvodu, ne i kasnije pri korištenju.

Izračunajmo očekivanje slučajne varijable k (očekivanje je slično prosječnoj vrijednosti):

$$\begin{aligned} E(k) &= \sum_{k=0}^{\infty} k\cdot p(k,\lambda) = \sum_{k=1}^{\infty} k\cdot\frac{\lambda^k}{k!}\cdot e^{-\lambda} = e^{-\lambda}\sum_{k=1}^{\infty}\frac{\lambda\cdot\lambda^{k-1}}{(k-1)!} \\ &= e^{-\lambda}\cdot\lambda\sum_{k=1}^{\infty}\frac{\lambda^{k-1}}{(k-1)!} = e^{-\lambda}\cdot\lambda\cdot e^{\lambda} = \lambda \end{aligned} \tag{7.29.}$$

Vrijednost λ je prosječna vrijednost slučajne varijable k. Npr. prosječni broj dolazaka novih poslova u jedinici vremena.

Kako izračunati vjerojatnost za proizvoljni vremenski period (ne samo jedinični)?

Isti izvod, samo umjesto n staviti $t \cdot n$ te se dobiva:

$$p(k(t), \lambda) = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \tag{7.30.}$$

Koja je vjerojatnost da u intervalu t nema ni jednog događaja?

$$p(k(t) = 0, \lambda) = e^{-\lambda t} \tag{7.31.}$$

Da ima bar jedan?

$$p(k(t) > 0, \lambda) = 1 - e^{-\lambda t} \tag{7.32.}$$

Napomena: Poissonova razdioba je diskretna, k je cijeli broj!

Primjer 7.2.

U jednoj minuti prosječno padne 100 kapi kiše na površinu stola. Koja je vjerojatnost da u sljedećoj sekundi na stol padnu dvije kapi?

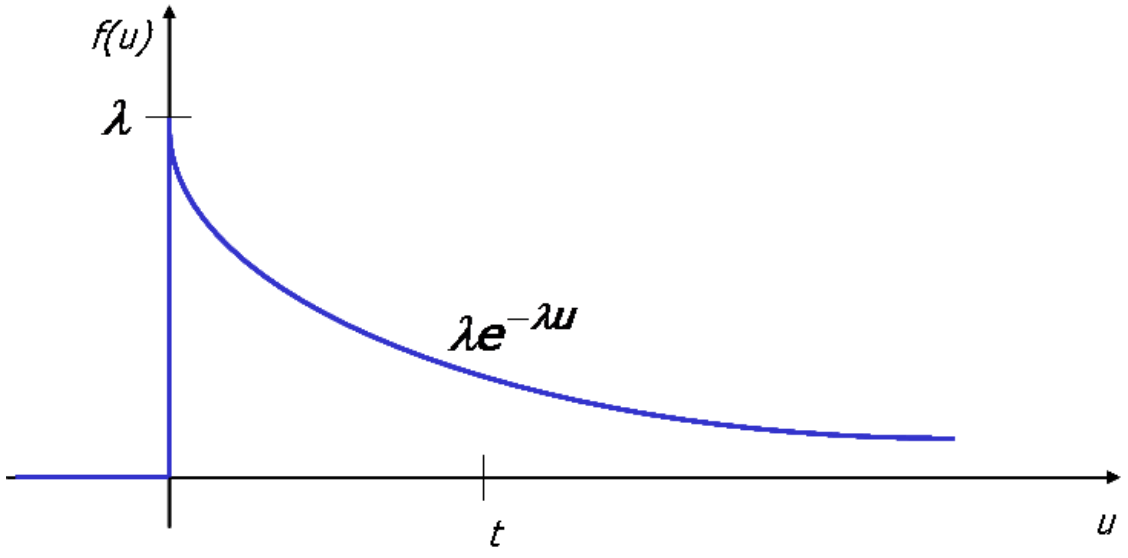
$$p\left(k(1) = 2, \lambda = \frac{100}{60}\right) = \frac{\lambda^2}{2!} \cdot e^{-\lambda} = 26,2\% \tag{7.33.}$$

Vjerojatnosti idu redom (u %) (k(1), za k iz {0-5}) p= {19; 31; 26; 15; 6; 2}

7.4. Eksponencijalna razdioba

Razdioba je zadana funkcijom gustoće vjerojatnosti:

$$f(u) = \begin{cases} 0, & u < 0 \\ \lambda e^{-\lambda u}, & u \geq 0 \end{cases} \tag{7.34.}$$



(opet λ, ali samo u modelu, kasnije ćemo uz tu razdiobu vezivati β)

Eksponencijalna razdioba je kontinuirana razdioba! Parametar razdiobe može poprimiti realnu vrijednost.

Vjerojatnost da slučajna varijabla T poprimi jednu diskretnu vrijednost je 0.

Računa se vjerojatnost da slučajna varijabla poprimi vrijednost iz intervala $[a, b]$ kao integral gustoće vjerojatnosti od a do b.

$$p(a < T < b) = \int_a^b f(u)\, du \tag{7.35.}$$

Eksponencijalnom razdiobom modeliramo razmake između događa kao i trajanje obrade.

Povezanost eksponencijalne i Poissonove razdiobe!

Vjerojatnost da je vrijeme između dva događaja veće od t:

$$p(t < T < \infty, \lambda) = \int_t^\infty \lambda e^{-\lambda u} du = \lambda \int_t^\infty e^{-\lambda u} du = \lambda(-\frac{1}{\lambda}) e^{-\lambda u}\Big|_t^\infty = e^{-\lambda t} \tag{7.36.}$$

To smo već imali, to je vjerojatnost da se u intervalu t nije dogodio ni jedan događaj!

I obratno, vjerojatnost da se dogodio barem jedan događaj jest vjerojatnost da je razmak između dva događaja manje od t:

$$p(T < t, \lambda) = \int_0^t \lambda e^{-\lambda u} du = -e^{-\lambda u}\Big|_o^t = 1 - e^{-\lambda t} \tag{7.37.}$$

Koje je značenje parametra λ? Koristimo isti princip, računamo očekivanje sl.var. T:

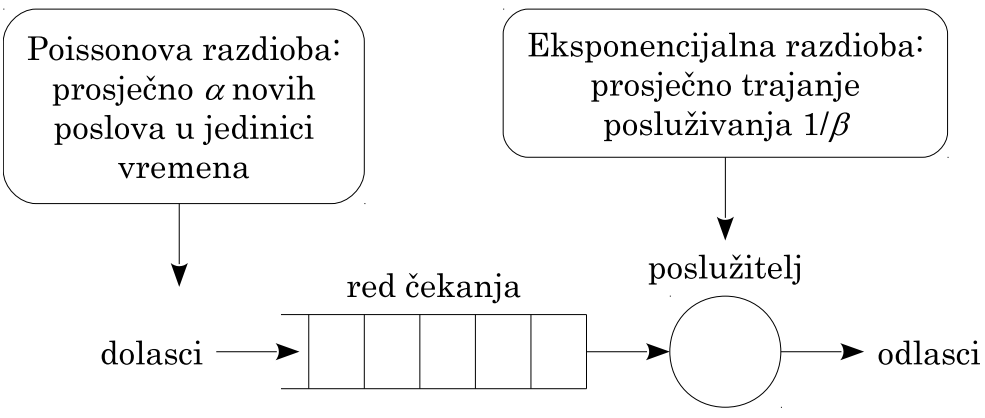
$$E(T) = \int_0^\infty u \cdot \lambda \cdot e^{-\lambda u} du = (\text{supst.} v = \lambda u) = \frac{1}{\lambda} \int_0^\infty v \cdot e^{-v} dv = \frac{1}{\lambda} \tag{7.38.}$$

$\frac{1}{\lambda}$ – prosječno vrijeme između dva događaja; ili prosječno trajanje obrade

λ – prosječan broj događaja u jedinici vremena; ili mogući broj obrada u jedinici vremena

7.4.1. Modeliranje poslužitelja s Poissonovom razdiobom dolaska novih poslova i eksponencijalnom razdiobom trajanja obrade

Kod korištene eksponencijalne razdiobe $\frac{1}{\lambda}$ predstavlja prosječno trajanje obrade, a λ je prosječan broj poslova koje poslužitelj može obaviti u jedinici vremena. U nastavku će se umjesto λ koristiti oznaka β.



Imamo razdiobe, α koji modelira dolaske i $1/\beta$ koji modelira obradu. Znamo izračunati opterećenje poslužitelja $\rho = \alpha/\beta$, vjerojatnosti dolaska i odlaska u nekom intervalu.

Nepoznate veličine:

- $\overline{T}$ – koliko poslovi prosječno čekaju
- $\overline{n}$ – koliki je prosječan broj poslova u sustavu
- tj. koja je kvaliteta usluge (koliko se čeka na uslugu) i koliki mora biti red?

Littleova formula $\overline{n} = \alpha\, T$ povezuje ova dva parametra pa je dovoljno izračunati jedan.

7.4.2. Izračun (izvod) prosječnog broja poslova u sustavu

Ako s i označimo slučajnu varijablu koja označava *broj poslova u sustavu* u nekom trenutku, tada očekivanje te varijable $E(i)$ je prosječan broj poslova u sustavu, tj. $\overline{n}$.

$$E(i) = \sum_{i=0}^{\infty} i \cdot p_i \tag{7.39.}$$

Označimo s $p_i(t)$ vjerojatnost da se u nekom trenutku u sustavu nalazi i poslova. Tu vjerojatnost još ne znamo! (Ne miješati ovo s $p(k, \lambda)$!)

Pretpostavimo da je sustav u stohastičkoj ravnoteži:

- sve vjerojatnosti konstantne (vjerojatnost događaja)
- ako dovoljno dugo promatramo sustav to je ispunjeno

Za sustav u stohastičkoj ravnoteži vrijedi $p_i(t) =$ konstantno!

Promatrajmo sustav u nekom stanju i. Kolike su vjerojatnosti da će se u intervalu Δt nešto dogoditi ili da se neće ništa dogoditi (doći novi poslovi ili otići/završiti neki)?

Vjerojatnost da neće doći novi poslovi (prema Poissonovoj razdiobi) je:

$$p(k(\Delta t) = 0, \alpha) = e^{-\alpha \Delta t} \tag{7.40.}$$

Ako se to razvije u Taylorov red:

$$p(k(\Delta t) = 0, \alpha) = e^{-\alpha \Delta t} = 1 - \alpha\Delta t + \frac{(\alpha\Delta t)^2}{2!} - \frac{(\alpha\Delta t)^3}{3!} + \cdots \approx 1 - \alpha\Delta t \tag{7.41.}$$

Suprotna vjerojatnost, da će se dogoditi barem jedan događaj može se aproksimirati sa:

$$p(k(\Delta t) > 0, \alpha) = 1 - e^{-\alpha \Delta t} \approx \alpha\Delta t \tag{7.42.}$$

Pretpostavimo da je Δt tako mali da je to vjerojatnost upravo jednog događaja.

Vjerojatnost da nema niti jednog odlaska u intervalu Δt (vjerojatnost da je trajanje obrade veće od Δt, vrijedi samo za stanja $i > 0$, tj. kada imamo bar 1 posao u sustavu) = vjerojatnost da je vrijeme između dva događaja veća od t (po eksponencijalnoj razdiobi):

$$p(t < T < \infty, \beta) = e^{-\beta \Delta t} = 1 - \beta\Delta t + \frac{(\beta\Delta t)^2}{2!} - \frac{(\beta\Delta t)^3}{3!} + \cdots \approx 1 - \beta\Delta t \tag{7.43.}$$

Suprotna vjerojatnost, da će se barem jedan posao napustiti (barem jedan odlazak) sustav je prema tome:

$$p(T < t, \beta) = 1 - e^{-\beta \Delta t} \approx \beta\Delta t \tag{7.44.}$$

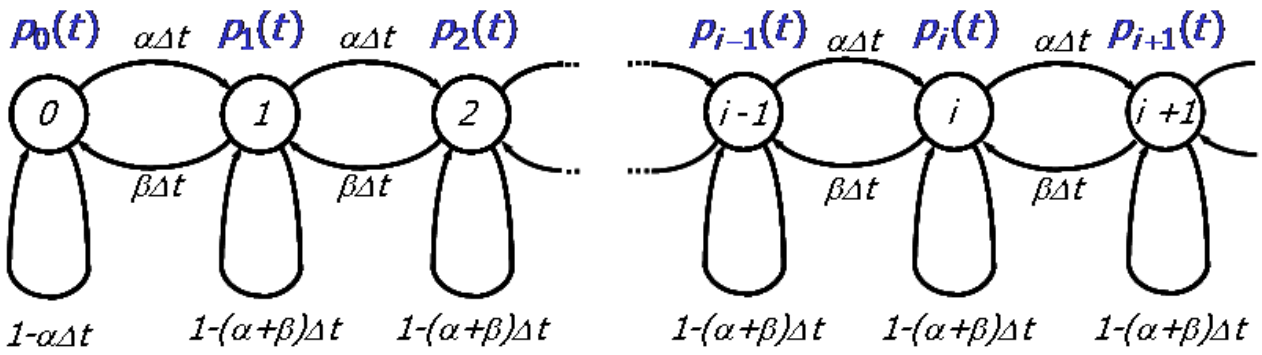
Pretpostavimo da je Δt tako mali da je to vjerojatnost upravo jednog odlaska.

(**ispitno pitanje – crtanje Markovljeva lanca i izvod formula**)

Dakle, za vrlo mali interval Δt vrijedi:

- vjerojatnost jednog dolaska: $\alpha \Delta t$
- vjerojatnost jednog odlaska: $\beta \Delta t$
- vjerojatnost jednog dolaska i jednog odlaska: $\alpha \Delta t \cdot \beta \Delta t = \alpha \beta \Delta t^2 \approx 0$ (zanemarujemo)
- vjerojatnost da nema ni dolaska ni odlaska: $(1 - \alpha \Delta t)(1 - \beta \Delta t) \approx 1 - (\alpha + \beta)\Delta t$
- vjerojatnosti višestrukih dolazaka i/ili odlazaka u Δt zanemarujemo

Markovljev lanac



Za trenutak $t + \Delta t$, za čvor i (stanje i) iz grafa slijedi:

$$p_i(t + \Delta t) = p_i(t) \cdot [1 - (\alpha + \beta)\Delta t] + p_{i-1}(t) \cdot \alpha \cdot \Delta t + p_{i+1}(t) \cdot \beta \cdot \Delta t \tag{7.45.}$$

Iznimka je stanje 0:

$$p_0(t + \Delta t) = (1 - \alpha \Delta t) \cdot p_0(t) + \beta \cdot \Delta t \cdot p_1(t) \tag{7.46.}$$

(**ispitno pitanje do ovdje**)

Sredimo prije puštanja Δt u 0:

$$\begin{aligned} \frac{p_i(t + \Delta t) - p_i(t)}{\Delta t} &= -(\alpha + \beta) \cdot p_i(t) + \alpha \cdot p_{i-1}(t) + \beta \cdot p_{i+1}(t) \\ \frac{p_0(t + \Delta t) - p_0(t)}{\Delta t} &= -\alpha \cdot p_0(t) + \beta \cdot p_1(t) \end{aligned} \tag{7.47.}$$

Kada $\Delta t \longrightarrow 0$ tada je lijeve strana derivacija vjerojatnosti, ali s obzirom na to da su vjerojatnosti konstantne to je 0! Slijedi:

$$\begin{aligned} 0 &= -(\alpha + \beta) \cdot p_i(t) + \alpha \cdot p_{i-1}(t) + \beta \cdot p_{i+1}(t) \\ 0 &= -\alpha \cdot p_0(t) + \beta \cdot p_1(t) \end{aligned} \tag{7.48.}$$

odnosno,

$$\begin{aligned} p_1(t) &= \frac{\alpha}{\beta} \cdot p_0(t) = \rho \cdot p_0(t) \\ p_{i+1}(t) &= (1 + \frac{\alpha}{\beta}) \cdot p_i(t) - \frac{\alpha}{\beta} \cdot p_{i-1}(t) = (1 + \rho) \cdot p_i(t) - \rho \cdot p_{i-1}(t) \end{aligned} \tag{7.49.}$$

Uvrštavanjem redom dobiva se konačna formula: (skraćeno: p_i umjesto $p_i(t)$)

$$\begin{aligned} p_1 &= \rho \cdot p_0 \\ p_2 &= (1+\rho) \cdot p_1 - \rho \cdot p_0 = (1+\rho) \cdot \rho \cdot p_0 - \rho \cdot p_0 = \rho^2 \cdot p_0 \\ p_3 &= (1+\rho) \cdot p_2 - \rho \cdot p_1 = (1+\rho) \cdot \rho^2 \cdot p_0 - \rho \cdot \rho \cdot p_0 = \rho^3 \cdot p_0 \\ &\vdots \\ p_i &= \rho^i \cdot p_0 \end{aligned} \tag{7.50.}$$

Da bi imali sve određeno treba nam samo p_0.

Njega možemo izračunati iz zakonitosti da zbroj svih vjerojatnosti bude 1 (vjerojatnost da je sustav u nekom stanju je 1).

$$\sum_{i=0}^{\infty} p_i = \sum_{i=0}^{\infty} \rho^i \cdot p_0 = p_0 \sum_{i=0}^{\infty} \rho^i = p_0 \cdot \frac{1}{1-\rho} == 1 \quad \Rightarrow \quad p_0 = 1-\rho \tag{7.51.}$$

te konačna formula za vjerojatnost da u sustavu ima i poslova:

$$\boxed{p_i = (1-\rho)\rho^i} \quad \text{pišemo i:} \quad \boxed{p(i=N) = (1-\rho)\rho^N} \tag{7.52.}$$

Očekivanje slučajne varijable i je zapravo prosječan broj poslova $\overline{n}$:

$$\overline{n} = E(i) = \sum_{i=0}^{\infty} i \cdot p_i = \sum_{i=0}^{\infty} i \cdot (1-\rho)\rho^i = (1-\rho) \sum_{i=0}^{\infty} i \cdot \rho^i = (1-\rho)\frac{\rho}{(1-\rho)^2} = \frac{\rho}{1-\rho} \tag{7.53.}$$

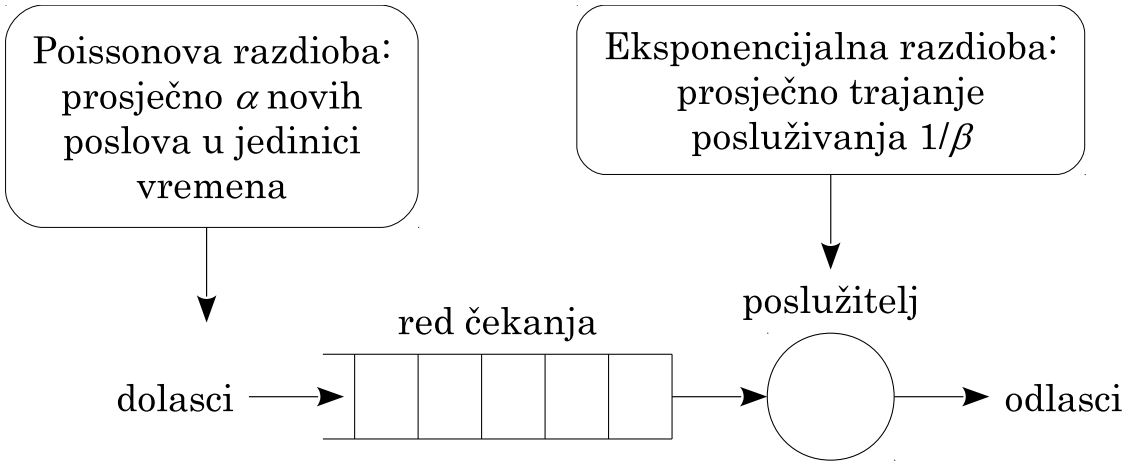
$$\boxed{\overline{n} = \frac{\rho}{1-\rho} = \frac{\alpha}{\beta-\alpha}} \tag{7.54.}$$

Te prema Littleovoj formuli slijedi i:

$$\boxed{\overline{T} = \frac{\overline{n}}{\alpha} = \frac{1}{\beta-\alpha}} \tag{7.55.}$$

7.5. Posluživanje s Poissonovom i eksponencijalnom razdiobom

(info) U literaturi se ovakav sustav obilježava sa: M/M/1, gdje prva oznaka obilježava dolaske, druga obradu, a treća broj poslužitelja. S obzirom na to da su Poissonova i eksponencijalna razdioba povezane i spadaju u tzv. klasu Markovljevih procesa označavaju se s M.



Slika 7.12. Model poslužitelja u nedeterminističkom sustavu

Poslovi dolaze u sustav s Poissonovom razdiobom:

- α – prosječan broj dolazaka novih poslova u sustav u jedinici vremena
- $\frac{1}{\alpha}$ – prosječno vrijeme između dva dolaska
- (vremena između dva dolaska podliježu eksponencijalnoj razdiobi s parametrom $\frac{1}{\alpha}$)

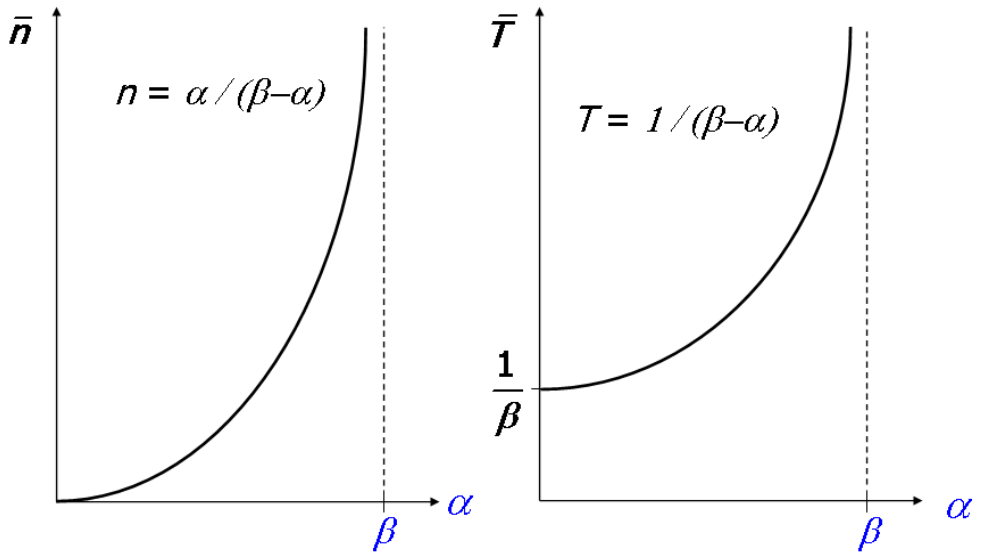
Vjerojatnost da u intervalu t dođe k novih poslova:

$$p(k(t), \lambda) = \frac{(\lambda t)^k}{k!} \cdot e^{-\lambda t} \tag{7.56.}$$

Trajanje obrade podliježe eksponencijalnoj razdiobi

- $\frac{1}{\beta}$ – prosječno trajanje obrade/posluživanja (za zadane poslove)
- β – prosječan broj poslova koje poslužitelj može obaviti u jedinici vremena
  - sposobnost poslužitelja da obavi prosječno β poslova u jedinici vremena
- $\overline{T}$ – prosječno trajanje zadržavanja posla u sustavu
- $\overline{n}$ – prosječan broj poslova u sustavu
- $\overline{n} = \alpha \cdot \overline{T}$ – Littleova formula
- $\rho = \frac{\alpha}{\beta}$ – faktor iskorištenja, prosječno iskorištenje procesora/poslužitelja
- $\overline{n} = \frac{\rho}{1-\rho} = \frac{\alpha}{\beta - \alpha} \quad \Longrightarrow \quad \overline{T} = \frac{1}{\beta - \alpha}$
- $p(i = N) = (1 - \rho)\rho^N$ – vjerojatnost da u sustavu ima N poslova
- $p(i > N) = 1 - p(i \leq N) = \rho^{N+1}$ – vjerojatnost da u sustavu ima više od N poslova

Iz prethodnih se formula vidi da se ne smije dopustiti 100% opterećenje jer tada nazivnik ide u beskonačnost. Povećanjem opterećenja red i prosječno zadržavanje u sustavu se povećavaju.



β je konstantan za graf (ne mijenjamo poslužitelj, nego mu samo dajemo više posla).

Zadatak 7.2.

Pretpostavimo da proizvođač i potrošač komuniciraju preko ograničenog spremnika koji se sastoji od N pretinaca. U sustavu se tada može nalaziti najviše $M = N + 1$ poruka (N poruka nalazi se u redu, a jednu troši potrošač). Pretpostavimo, nadalje, da proizvođač proizvodi poruke tako da su događaji njihovih stavljanja u spremnik podvrgnuti Poissonovoj razdiobi, te da potrošač na obradu poruka troši vremena podvrgnuta eksponencijalnoj razdiobi. Zanima nas koliko međuspremnik mora imati pretinaca da, uz dani faktor ρ, vjerojatnost blokiranja proizvođača bude manja od neke zadane vrijednosti.

Preformulirano: kolika je vjerojatnost da će u sustavu biti manje od M poruka (tada ne dolazi do blokiranja proizvođača), tj. $p(i < M) = p(i \leq N) = ?$

Prema formulama:

$$p(i \leq N) = \sum_{i=0}^{N} p_i = \sum_{i=0}^{N} (1-\rho) \cdot \rho^i = (1-\rho) \sum_{i=0}^{N} \rho^i = (1-\rho) \frac{1-\rho^{N+1}}{1-\rho} = 1 - \rho^{N+1} \tag{7.57.}$$

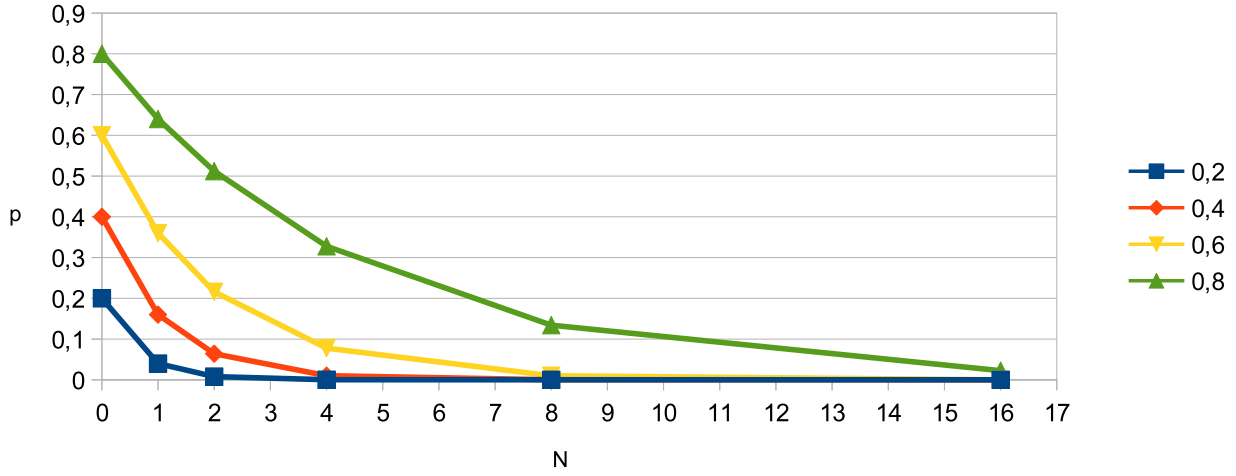
Suprotna vjerojatnost, vjerojatnost blokiranja je:

$$p(i > N) = \rho^{N+1} \tag{7.58.}$$

Brojke (vjerojatnost blokiranja):

N	$\rho = 0{,}2$	$\rho = 0{,}4$	$\rho = 0{,}6$	$\rho = 0{,}8$
0	0,20000	0,40000	0,60000	0,80000
1	0,04000	0,16000	0,36000	0,64000
2	0,00800	0,06400	0,21600	0,51200
4	0,00032	0,01024	0,07776	0,32768
8	5,12E-07	0,00026	0,01008	0,13422
16	1,31E-12	1,72E-07	0,00017	0,02252

Vjerojatnost blokiranja značajno raste s opterećenjem!



Zadatak 7.3.

U nekom sustavu poslovi se javljaju periodički svakih 30 ms i to: P1 u 3. ms, P2 u 5., P3 u 6., P4 u 10., P5 u 20. i P6 u 23. milisekundi (gledano prema početku periode). Svi poslovi traju isto, po 4 ms. Odrediti prosječno zadržavanje poslova u sustavu i prosječan broj poslova u sustavu.

Zadatak 7.4.

U nekom deterministčkom sustavu poslovi P1-P4 se javljaju periodički, svakih 20 ms. P1 se javlja prvi. Slijedi P2 nakon 5 ms, P3 nakon još 2 ms te P4 nakon još 6 ms. P1 i P2 trebaju po 6 ms, a P3 i P4 po 3 ms poslužiteljskog vremena. Izračunati: α, β, ρ, $\overline{T}$, $\overline{n}$.

Zadatak 7.5.

Zahtjevi za obradu podliježu Poissonovoj razdiobi s $\alpha = 2s^{-1}$, a vrijeme obrade ima eksponencijalnu razdiobu. Mjerenjem je ustanovljeno prosječno vrijeme zadržavanja posla u sustavu $\overline{T} = 0,5s$. Kolika je vjerojatnost da se u sustavu nađe više od 5 poslova?

Dodatno:

a) Kolika je vjerojatnost da u sustavu bude bude između 2 i 4 (2, 3 ili 4) poslova?

b) Što ako se poslužitelj ubrza za 30%?

Zadatak 7.6.

Za neki Web sustav s jednim poslužiteljem prosječan broj zahtjeva u minuti je 100, dok je snaga poslužitelja znatno veća, on ih može obraditi 300 u minuti (prosječno). Koliki se najveći postotak poslužiteljskog vremena može rezervirati za druge usluge, a da klijenti i dalje ne čekaju više od dvije sekunde na svoje zahtjeve (prosječno)? (Pretpostaviti da to neće utjecati na razdiobe. Npr. da će se vrijeme za te druge poslove dati u vrlo kratkim intervalima.)

Zadatak 7.7.

U nekom je poslužiteljskom centru napravljena analiza rada poslužitelja. Ustanovljeno je da tri poslužitelja rade s prilično malim opterećenjem. Poslužitelj P1 prosječno dobiva 70 zahtjeva u minuti i njegova prosječna iskoristivost je 20 %, poslužitelj P2 dobiva 200 zahtjeva u minuti s prosječnim opterećenjem od 30 %, dok poslužitelj P3 s prosječno 150 zahtjeva u minuti radi tek s 10 % opterećenja. Poslužitelj P3 je procesorski najjači, 50% jači od P1 te 100% jači od P2. Izračunati kvalitetu usluge (prosječno vrijeme zadržavanja zahtjeva u sustavu) ako bi se svi poslovi preselili na poslužitelj P3.

Zadatak 7.8.

Za neki Web sustav s jednim poslužiteljem prosječan broj zahtjeva u sekundi je 100 (dolazak zahtjeva podliježe Poissonovoj razdiobi). Poslužitelj obrađuje tri tipa zahtjeva: Z_1, Z_2 i Z_3. Obrada zahtjeva podliježe eksponencijalnim razdiobama. Za zahtjeve tipa Z_1 obrada prosječno traje 5 ms, za Z_2 8 ms te za Z_3 10 ms. Ako je postotak zahtjeva za Z_1 30%, za Z_2 40% te za Z_3 30% odrediti prosječnu kvalitetu usluga koje poslužitelj pruža, tj. odrediti prosječno vrijeme zadržavanja zahtjeva u sustavu te vjerojatnost da se u sustavu nalazi više od 10 zahtjeva.

Zadatak 7.9.

U nekom sustavu imamo dva poslužitelja P_1 i P_2 i dva tipa poslova Z_1 i Z_2 koje oni obrađuju (P_1-Z_1, P_2-Z_2). Poslovi Z_1 prosječno dolaze s 30 poslova u minuti, dok poslovi Z_2 dolaze s 90 poslova u minuti. P_1 radi s 30% opterećenjem, a P_2 s 60%.

(ZAD) Kada bismo zamjenili poslužitelje, tj. kada bi P_1 obrađivao poslove Z_2 (umjesto Z_1), opterećenje bi mu iznosilo 80%. Koje bi bilo opterećenje poslužitelja P_2 ako bi on obrađivao poslove Z_1?

Druge inačice zadatka:

(ZAD) Ako sve poslove obrađuje P1 opterećenje mu naraste na 60%. Koliko bi bilo opterećenje P2 ako on obrađuje sve poslove?

(ZAD) Ako sve poslove obrađuje P1 prosječno vrijeme zadržavanja poslova u sustavu naraste na 2 s. Koliko bi prosječno vrijeme zadržavanja poslova u sustavu ako sve poslove obrađuje P2?

Zadatak 7.10.

Poslužitelj koji je radio s prosječnim opterećenjem od 20% zamijenjen je drugim, dvostruko slabijim. Uz to dobiva još 50% istih poslova. Ako su dolasci novih poslova modelirani Poissonovom razdiobom, a obrada eksponencijalnom, koliko će biti opterećenje novog poslužitelja?

7.6. Osnovni načini dodjeljivanja procesora dretvama – raspoređivanje dretvi

7.6.1. Dodjeljivanje po redu prispijeća

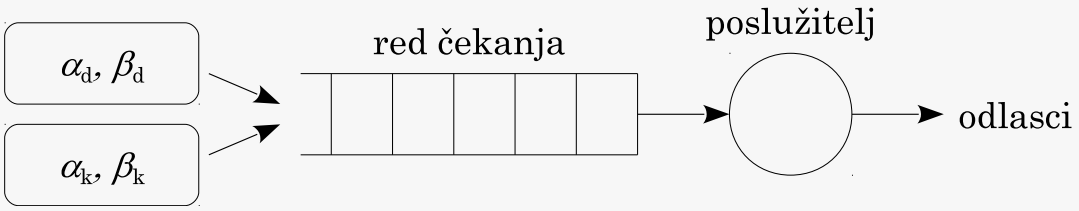
Ovo je najjednostavnije dodjeljivanje (i za ostvarenje). Ali je li najbolje? Kako definirati "najbolje"?

- najefikasnije (najviše se zadataka obavi)
- najpravednije prema zadacima; svi su "jednaki", poslužuju se redom kojim su i došli, ali da li dobiju jednak dio procesora kroz duže vrijeme (ako ostaju u sustavu duže)?
- najmanje zahtjevno za izvođenje (najmanji overhead)

Analizirajmo (prvo) statistički.

Primjer 7.3. Mješanje kratkih i dugih poslova

Neka u sustav dolaze dvije skupine zadataka: kratki s indeksom k te dugi s indeksom d. Kakva je statistika za tu mješavinu? Kako to izgleda za pojedinačni zadatak?



Koliki je ukupni α? Zbroj!

$$\alpha = \alpha_k + \alpha_d \tag{7.59.}$$

Što je s β? Bolje razmišljati i $\frac{1}{\beta}$ (prosječnom trajanju posluživanja mješavine poslova).

$$\frac{1}{\beta} = \frac{\alpha_k}{\alpha_k + \alpha_d} \cdot \frac{1}{\beta_k} + \frac{\alpha_d}{\alpha_k + \alpha_d} \cdot \frac{1}{\beta_d} \tag{7.60.}$$

Ipak jednostavnije je razmišljati o ukupnom opterećenju koje je suma opterećenja koje generiraju kratki i dugi poslovi:

$$\rho = \rho_k + \rho_d \tag{7.61.}$$

te β računati prema:

$$\beta = \frac{\alpha}{\rho} \tag{7.62.}$$

Zadatak 7.11.

Promotrimo dvije skupine poslova koje možemo nazvati kratkim i dugim poslovima. Svojstva tih dvaju skupina možemo prikazati tablično (dolasci Poisson, obrada eksp. razd.):

	kratki	dugi	mješavina
α	10	0,01	10,01
β	50	0,02	14,3
$1/\beta$	0,02	50	0,6993
ρ	0,2	0,5	0,7
$\overline{n}$	0,25	1	2,23
$\overline{T}$	0,025	100	0,233

- Ovo je statistika za mješavinu.
- Kad ih pomiješamo i dalje će obrada kratkog trajati 0,02 a dugog 50 (prosječno!)
- Sama statistika je značajno pogoršanje za kratke poslove! Za red veličine (statistički).
- Najgori slučaj? Kratki posao koji dolazi neposredno iza dugog: on mora čekat da dugi završi, tj. mora čekati i više od 50 jedinica vremena!!!

Analizom prethodnih primjera ustanovljeno je da posluživanje po redu prispijeća nije dobro za kratke poslove.

Kako onda posluživati?

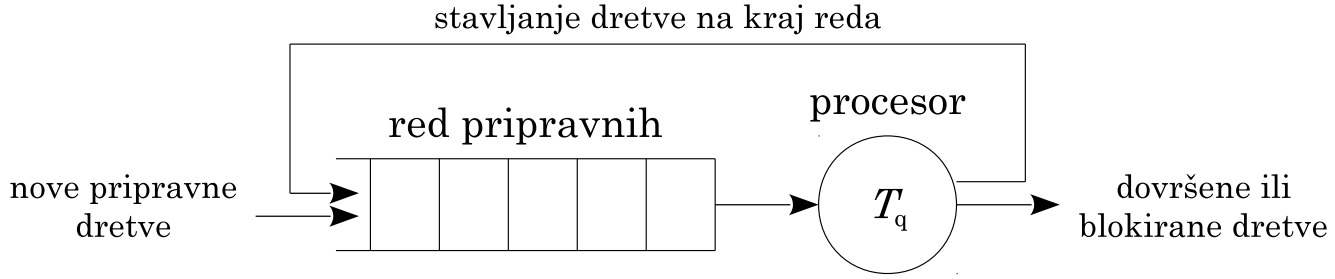
Koristiti prioritete? Dati kratkim poslovima veći priorite? To bi bilo nepravedno prema dugim poslovima koji bi se stalno odgađali.

Drugo rješenje = podjela poslužiteljskog vremena = kružno posluživanje!

7.6.2. Kružno posluživanje (posluživanje podjelom vremena)

Engleski termini: Round-Robin (RR), time-share

Neka različite poslove izvode različite dretve. Posluživanje poslova stoga možemo nazvati posluživanje dretvi, odnosno raspoređivanje dretvi.



- Raspoređivanje pravednom podjelom procesorskog vremena (“svima jednako”)
- Svaki posao dobije kvant vremena T_q
  - ako ne završi u tom kvantu vraća se na kraj reda
- Poslovi se prekidaju u izvođenju!
  - npr. koristi se satni mehanizam za izazivanje periodičkih prekida.
- Pravednije posluživanje naspram kraćih poslova.

Primjer ostvarenja kružnog posluživanja prilagodbom jezgrine funkcije `OTKUCAJ_SATA` (info)

```
j-funkcija OTKUCAJ_SATA ()
{
  pohrani kontekst u opisnik Aktivna_D;
  //raspoređivanje podjelom vremena (dodatak)
  ako je (Aktivna_D se raspoređuje podjelom vremena) {
    Aktivna_D->Otkucaji--;
    ako je (Aktivna_D->Otkucaji == 0) {
      Aktivna_D->Otkucaji = Aktivna_D->Otkucaji_max;
      //stavi dretvu na kraj reda
      stavi_u_red(makni_prvu_iz_reda(Aktivna_D), Pripravne_D);
      stavi_u_red(makni_prvu_iz_reda(Pripravne_D), Aktivna_D);
    }
  }
  //odgođene dretve (isto kao i prije)
  ako je (Odgođene_D->prva != prazno) {
    Odgođene_D->prva.Zadano_kašnjenje--;
    ako je (Odgođene_D->prva.Zadano_kašnjenje == 0) {
      stavi_u_red(makni_prvu_iz_reda(Aktivna_D), Pripravne_D);

      dok je (Odgođene_D->prva != prazno && Odgođene_D->prva.Zadano_kašnjenje == 0)
        stavi_u_red(makni_prvu_iz_reda(Odgođene_D), Pripravne_D);

      stavi_u_red(makni_prvu_iz_reda(Pripravne_D), Aktivna_D);
    }
  }
  obnovi kontekst iz opisnika Aktivna_D;
}
```

7.6.3. Usporedba posluživanja poslova: po redu prispijeća i kružno

Za primjer, uzmimo iste podatke o poslovima kao i u zadatku 7.3. ali sada neka to bude *deterministički sustav*, gdje su vremena apsolutna a ne prosječna (kratki poslovi dolaze svakih 100 ms, dugi svakih 100 s, obrada kratkih traje 20 ms a dugih 50 s).

Primjer 7.4. Posluživanje po redu prispijeća

α i ρ su isti kao i prije (isto razmišljanje i dalje vrijedi!):

$$\alpha = \alpha_k + \alpha_d = 10 + 0,01 = 10,01 \tag{7.63.}$$

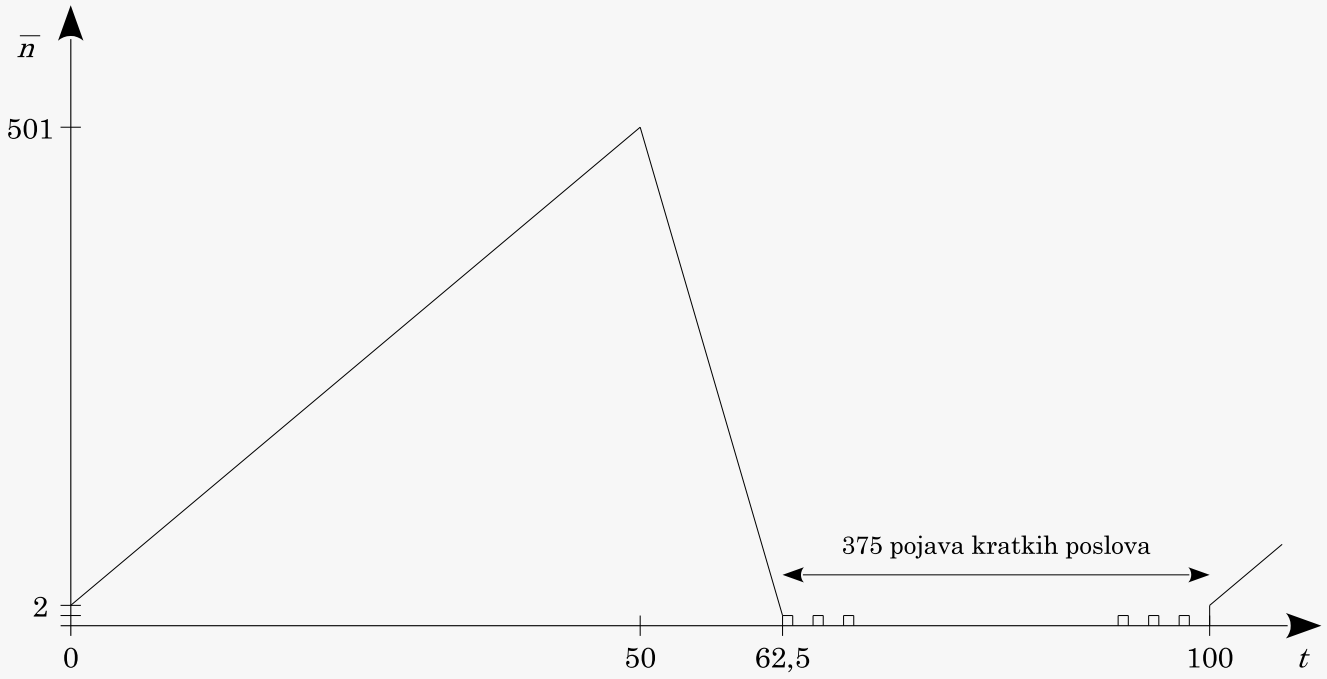
$$\rho = \rho_k + \rho_d = \frac{10}{50} + \frac{0,01}{0,02} = 0,7 \tag{7.64.}$$

Koliki su $\overline{n}$ i $\overline{T}$ (stvarni po zadacima te u prosjeku)?

Razmatrajmo pojedinačne zadatke.

Pretpostavimo da je najprije došao dugi posao i odmah nakon njega kratki.

Obratno razmatranje gotovo da samo pomiče vremensku skalu i neznatno usporava taj dugi posao (20ms).



Slika 7.13. Broj poslova u poslužitelju

U $t = 0$ kreće dugi posao i prvi kratki ide u red (2 ukupno).

Obrada dugog traje 50 s te za to vrijeme dolaze i kratki poslovi (10 u sekundi) i gomilaju se u redu.

U 50. sekundi kada dugi posao završava u sustavu se nagomila 500 kratkih poslova (u redu).

Nakon toga broj poslova pada: poslužitelj obrađuje 1/0,02=50 poslova u sekundi, ali poslovi i dalje dolaze s 10 u sekundi = broj poslova pada s 40 u sekundi.

$\frac{500}{40} = 12,5 \quad \Rightarrow$ za $12,5$ sekundi red se isprazni (u $50 + 12,5 = 62,5$ s).

Nakon toga red je prazan i svaki posao koji dođe dolazi izravno do poslužitelja.

Prosječan broj poslova u sustavu $\overline{n}$ može se približno izračunati prema:

$$\overline{n} = \frac{\text{površina}}{\text{vrijeme}} \approx \frac{\frac{62,5 \cdot 500}{2} + 375 \cdot 0,02 \cdot 1}{100} = 156,325 \tag{7.65.}$$

Opaska: računamo kao da je kontinuirano iako je diskretno!

Prema Littleovu pravilu:

$$\overline{T} = \frac{\overline{n}}{\alpha} \approx \frac{156,325}{10,01} = 15,62\text{s} \tag{7.66.}$$

Navedeno je statistički. Što možemo reći o zasebnim poslovima?

Skicirati graf za T_k (izgled: |__).

Prvi kratki se u sustavu zadržava 50 s !!!

Primjer 7.5. Kružno posluživanje

Neka je kvant vremena 0,01 s (10 ms).

Kratki poslovi trebaju $\frac{0,02}{0,01} = 2$ kvanta, dugi $\frac{50}{0,01} = 5000$ kvantova

Neka dugi opet dođe prvi.

Redoslijed izvođenja: D K D K D D D D D D K D K D D D ...

Dugi se zadržava: $\frac{50 \text{ s}}{80 \text{ ms}} \cdot 100 \text{ ms} = 62,5 \text{ s}$

Kratki:

- prvi: $4 \cdot 0,01$
- daljnjih 624: $3 \cdot 0,01$
- zadnjih 375: $2 \cdot 0,01$

$$\overline{T} = 62,5 \cdot \frac{1}{1001} + 0,04 \cdot \frac{1}{1001} + 0,03 \cdot \frac{624}{1001} + 0,02 \cdot \frac{375}{1001} = 0,0887 \text{ s} \tag{7.67.}$$

$$\overline{n} = \alpha \cdot \overline{T} = 10,01 \cdot 0,0887 = 0,887 \tag{7.68.}$$

Iz primjera je očito je RR pravednije posluživanje.

Ali: treba prekidati obrade poslove (satni mehanizam) + troši se vrijeme na zamjenu dretvi.

Ako su zamjene rijetke onda se vrijeme zamjene dretvi (zamjena konteksta) može zanemariti, ali ako su česte one mogu značajno utjecati na učinkovitost sustava!

7.7. Raspoređivanje dretvi u operacijskim sustavima

Osnovne (teorijske) strategije raspoređivanja:

- raspoređivanje po redu prispijeća (engl. *first-in-first-out – FIFO*)
- raspoređivanje prema prioritetu
- raspoređivanje podjelom vremena (engl. *time-share, round-robin – RR*)

Zadatak 7.12. Osnovne strategije raspoređivanja

U nekom sustavu javljaju se poslovi/dretve A, B, C i D u trenucima 3.5, 0, 2,5 i 6,5 respektivno s trajanjima obrade 5, 5, 3 i 2 (respektivno). Pokazati izvođenje dretvi (raspoređivanje) ako se koristi:

a) raspoređivanje po redu prispijeća

b) raspoređivanje prema prioritetu uz $p_A > p_B > p_C > p_D$

c) kružno raspoređivanje s $t_q = 1$ (jedinica vremena)

Načini raspoređivanja dretvi u operacijskim sustavima često koriste kombinacije gornjih strategija, ali i ovise o tipovima dretvi. Naime, različiti su zahtjevi pojedinih tipova dretvi. Primjeri tipova dretvi:

- dretve koje preko U/I naprava upravljaju nekim procesima (industrija, automobili, zgrade, ...) – kritične dretve koje moraju brzo/pravovremeno reagirati/slati naredbe
- dretve koje obavljaju neke dugotrajne proračune – ako ih se malo i odgodi "neće to primijetiti"
- dretve koje upravljaju sučeljem programa – korisničko iskustvo će biti bolje ako te dretve što prije dođu na red za izvođenje, a uglavnom su vrlo brzo gotove sa svojim poslom u tom trenutku ("interaktivne dretve")

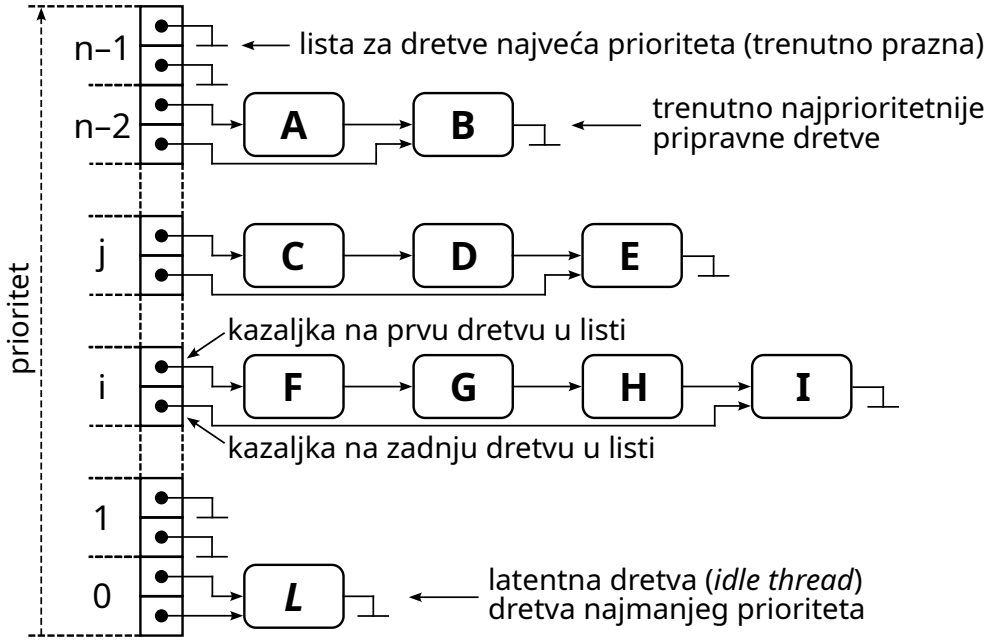
Obzirom na navedeno, dretve u kontekstu raspoređivanja unutar operacijskih sustava se dijele na vremenski kritične i vremenski nekritične (normalne). Raspoređivanje jednih i drugih nije jednako. (Više o raspoređivanju)

7.7.1. Vremenski kritične dretve (engl. *real-time*)

Prioritetno raspoređivanje

- **osnovni kriterij** = prioritet – UVIJEK je aktivna ona dretva s najvećim prioritetom
- kad se u sustavu pojavi nova dretva B s većim prioritetom od prioriteta trenutno aktivne dretve A, dretva B postaje aktivna – istisne dretvu A koja će nastaviti s radom tek nakon što se dretva B makne (završi ili blokira)
- kada osnovni kriterij ne daje samo jednu dretvu (postoji više dretvi ista najveća prioriteta) tada se za odabir jedne od njih koristi **dodatni** kriterij najčešće FIFO ili RR:
  - prema redu prispijeća – FIFO: aktivna dretva se ne mijenja sve se ne završi ili blokira (ili dođe nova dretva većeg prioriteta)
  - podjelom vremena – RR: aktivna se izvodi zadani kvant vremena, potom se miče iza svih dretvi ista prioriteta u redu pripravnih dretvi i uzima iduća pripravna (ista prioriteta)

  - dodatni kriterij se najčešće definira za svaku dretvu zasebno!
- s obzirom na to da se uvijek najprije gleda prioritet, način raspoređivanja (postavljen pojedinoj dretvi) dobiva ima po dodatnom kriteriju; nazivi prema istoimenim UNIX raspoređivačima:
  - SCHED_FIFO (prioritet pa red prispijeća)
  - SCHED_RR (prioritet pa podjela vremena)
- slika 7.14. prikazuje primjer skupa pripravnih dretvi u kojemu su dretve složene u različite redove, u skladu s njihovim prioritetima



Slika 7.14. Sustav pripravnih dretvi raspodijeljen prema prioritetu

- dretve A i B imaju u tom trenutku najveći prioritet $p_A = p_B = n - 1$
- prva dretva u redu je dretva A => aktivna dretva
- ako je drugi kriterij za raspoređivanje dretve A FIFO, ona će se izvoditi dok ne završi ili više ne bude u pripravnom stanju
- ako je drugi kriterij za raspoređivanje dretve A RR, ona će se izvoditi dok ne završi ili više ne bude u pripravnom stanju ili dok ne istekne kvant vremena koji raspoređivač koristi
- korištenje strukture podataka pripravnih dretvi kao na slici 7.14., s jednom listom za svaku razinu prioriteta omogućuje složenost O(1):
  - uz dodatnu kazaljku na zadnji element liste, dodavanje dretve u red pripravnih se radi u jednom koraku
  - uz bitmapu u kojoj jedinice označavaju neprazne liste, odabir naprioritetnije se također obavlja u jednom koraku (jednom instrukcijom se iz bitmape dobije indeks najviše postavljene jedinice, tj. indeks najprioritetnijeg nepraznog reda pripravnih dretvi)
- u višeprocesorskim sustavima za svaki procesor se koristi zasebni red pripravnih
  - razlog je efikasnost iz dva aspekta
    1. korištenje priručnog spremnika – dretva se vraća na isti procesor u čijem priručnom spremniku su možda još uvijek neki njeni podaci – ne treba ih dohvaćati iz memorije (na što bi se inače potrošilo vrijeme)

2. smanjena potrebi zaključavanja reda – svaki procesor ima svoj red pa pri raspoređivanju uglavnom zaključava samo svoj red

– problemi kad se koriste prioritetni raspoređivači: moglo bi se dogoditi da u nekom redu pripravnih bude dretva veće prioriteta od neke druge koja se izvodi na drugom procesoru; to treba spriječiti (push-pull postupci – gurnuti takvu dretvu drugom procesoru, uzeti takvu dretvu iz reda drugog procesora)

Raspoređivanje prema mjeri ponavljanja (info)

- izvorna imena i kratice: *rate monotonic scheduling* – RMS , *rate monotonic priority assignement* – RMPA
- odnosi se na periodičke dretve, koje u svakoj periodi moraju nešto napraviti
- dretvama se pridijeli prioritet tako da one s kraćom periodom dobe veći prioritet
- nakon toga koristi se prioritetno raspoređivanje

Raspoređivanje prema rokovima završetaka (info)

- odnosi se na periodičke dretve, koje u svakoj periodi moraju nešto napraviti
- posao u pojedinoj periodi mora biti gotov do nekog trenutka (gledano od početka periode) – krajnji trenutak završetka – rok završetka (engl. *deadline*)
- pripravne dretve posložene su prema njihovim rokovima
- prva u redu je ona s najbližim rokom
- izvorna imena i kratice: *earliest deadline first* – EDF , *deadline driven scheduling* – DDF
- Linux definira SCHED_DEADLINE sa sličnim načinom raspoređivanja (pogledati sched-deadline.txt na https://www.kernel.org/doc/Documentation/scheduler/)

Raspoređivanje "sporadičnih poslova" (info)

- prioritetno raspoređivanje za periodične poslove
- parametri raspoređivanja: period – T, povlašteni prioritet – PP, manji prioritet – MP te povlašteno vrijeme rada – PV (PV < T)
- unutar jedne periode T dretvi se daje do PV procesorskog vremena s prioritetom PP; ako joj to nije dovoljno, prioritet joj se smanjuje na MP do kraja te periode kada joj se ponovno podiže prioritet na PP
- POSIX ga definira (SCHED_SPORADIC), ali rijetki OS-evi ostvaruju (QNX)

7.7.2. "Normalne" dretve – nekritične dretve

- osnovna ideja: raspoređivanje podjelom vremena

a) pravedna podjela procesorskog vremena

b) prioritet određuje udio vremena koji će dretva dobiti (uglavnom)

c) tip dretve određuje udio vremena koji će dretva dobiti

- heuristika koja se nastoji primijeniti na raspoređivanje nekritičnih dretvi (podjela vremena tipa c)) opisuje se algoritmom naziva *višerazinsko raspoređivanje s povratnom vezom* (engl. *multilevel feedback queue – MFQ*).

MFQ algoritam (info)

- algoritam nastoji:
  - 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
- koristi nekoliko FIFO redova različita prioriteta
- aktivna dretva je prva dretva iz nepraznog reda najveća prioriteta
- dretva dobiva kvant vremena
- ako dretva završi prije isteka kvanta nestaje iz sustava
- ako se dretva blokira prije isteka kvanta privremeno nestaje iz domene raspoređivača (nije među pripravnima); pri odblokiranju, takva dretva se vraća u isti red iz kojeg je i otišla ili čak i viši red (dretva zadržava prioritet ili joj se on i povećava)
- ako je dretva protrošila cijeli kvant, onda ju raspoređivač pri isteku kvanta miče na kraj prvog idućeg reda (manjeg prioriteta)
- u redu najmanjeg prioriteta dretve se poslužuju kružno (svakoj kvant vremena)
- pri pojavi nove dretve ona ide u red najvećeg prioriteta (na kraj)
- sustav redova identičan je onome sa slike 7.14.
- MFQ algoritam brzo procjenjuje/klasificira dretvu:
  - je li procesorski zahtjevna
  - je li kratka ("interaktivna", uglavnom koristi UI)
- ... te se dalje prema njoj odnosi prema tome
  - kratke dretve dobivaju prednost jer brzo izlaze iz sustava
  - zahtjevne dijele procesorsko vrijeme podjelom vremena
- operacijski sustavi (danas) nemaju raspoređivače ostvarene po navedenom algoritmu, ali nastoje (s drugim algoritmima) ostvariti slična načela

7.7.3. Raspoređivanje u Linuxu

- za korisničke dretve Linux ima tri raspoređivača:
  1. prema roku; DEADLINE – SCHED_DEADLINE
  2. prema prioritetu: REALTIME – SCHED_FIFO, SCHED_RR
  3. obične dreteve: CFS/EEVDF – SCHED_OTHER

Raspoređivanje dretvi za rad u stvarnom vremenu (engl. *real time*)

- strategije: SCHED_FIFO, SCHED_RR (striktno)
  - prioriteti od 0 do 99 (veći broj veći prioritet)
- strategija: SCHED_DEADLINE – raspoređivanje prema krajnjim trenucima završetaka
  - parametri: period, vrijeme računanja, krajnji trenutak završetka

- ove dretve uvijek imaju prednost pred običnim dretvama!

Raspoređivanje "običnih" dretvi

- strategije: SCHED_OTHER i slični (_IDLE, _BATCH),
- prioriteti "službeno" od 100 do 139, ali se ne koriste
- koristi se "razina dobrote" (engl. *nice*): -20 do 19
  - manji broj veća dobrota (ekvivalent prioritetu)
  - negativne vrijednosti može postavljati samo povlašteni korisnik
  - pri pokretanju obična procesa njegova dobrota je 0
- cilj raspoređivača jest "pravedno" podijeliti procesorsko vrijeme
- dretve veće dobrote (prioriteta) dobivaju više procesorskog vremena
  - oko 10-15% po prioritetu
  - npr. dretva dobrote 0 treba dobiti oko $1{,}15^5$ više procesorskog vremena od dretve dobrote 5

Algoritam CFS: potpuno pravedan raspoređivač (engl. *Completely Fair Scheduler*) (info)

- koristi se od jezgre 2.6.23 (2007.)
- koristi razliku između:
  - izračunatog vremena koje pripada pojedinoj dretvi s obzirom na njenu dobrotu
  - stvarno dodijeljenog procesorskog vremena pojedinoj dretvi
- razlika određuje položaj opisnika dretve u stablu (crveno-crna stabla)
- aktivna dretva je dretva kojoj sustav najviše duguje ("najlijevija" dretva u stablu)

Algoritam EEVDF: raspoređivanje prema roku spremnih zadataka (engl. *Earliest Eligible Virtual Deadline First*) (info)

- koristi se od jezgre 6.6 (2023.)
- algoritam je izvorno predstavljen u članku 1995. od autora Ion Stoica i Hussein Abdel-Wahab: "Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation"
- svaki zadatak i ima svoju težinu w_i te se prema njoj i ostalim pripravnim zadacima (u skupu A) i njihovim težinama računa koliko mu procesorskog vremena pripada u intervalu $[t_1, t_2)$:

  $$S_i(t_1, t_2) = w_i \int_{t_1}^{t_2} \frac{1}{\sum_{j \in A(\tau)} w_j} d\tau,$$

  primjerice, kad bi N zadataka bilo pripravno u tom intervalu, svi s istom težinom w, svima bi pripadao jednak dio intervala, tj. $S_i(t_1, t_2) = \frac{t_2 - t_1}{N}$; kada težine nisu iste, oni s većom bi trebali dobiti više procesorskog vremena
- ako sa t_0^i označimo trenutak kad se zadatak i pojavio u sustavu, te sa $s_i(t_0^i, t)$ procesorsko vrijeme koje je zadatak dobio do trenutka t onda se definira dug (engl. *lag*):

  $lag_i(t) = S_i(t_0^i, t) - s_i(t_0^i, t)$
- kada je $lag_i(t) < 0$ zadatak je do t dobio više nego što je trebao (npr. u $t = 0$ neki zadatak

dobije kvant vremena od 5 ms; u $t = 5$ ms on je dobio više nego je trebao, jer je u teoriji to vrijeme trebalo ravnomjerno podijeliti svima)

- pri raspoređivanju se gledaju samo oni zadaci s pozitivnim dugom (kojima sustav duguje), tj. definira da su takvi zadaci podobni za izvođenje (engl. *eligible*)
- sam raspoređivač za odabir aktivnog zadatka koristi:
  - $V(e)$ – virtualni trenutak kad zadatak postaje podoban (kada vrijedi $lag_i(t) = 0$),
  - r – potrebno procesorsko vrijeme (kvant vremena za taj zadatak) te
  - $V(d)$ – rok (engl. *virtual deadline*) kad bi on trebao biti gotov, uz njegov udio u procesorskom vremenu prema težinama
- $V(e)$ se računa prema prema:

  $$V(e) = V(t_0^i) + \frac{s_i(t_0^i, t)}{w_i},$$

  tj. kao suma virtualnog vremena kad se zadatak pojavo u sustavu i već dobivenog procesorskog vremena, skaliranog s težinom zadatka
- vremena $V(e)$ i $V(t_0^i)$ su virtualna, dok t, t_0^i, s_i, S_i i lag_i to nisu
- virtualna vremena $V(t)$ se računaju prema:

  $$V(t) = \int_0^t \frac{1}{\sum_{j \in A(\tau)} w_j} d\tau$$

  tj. stvarno vrijeme skalirano je težinama – što imamo više zadataka virtualno vrijeme sporije teče
- $V(d)$ se računa prema:

  $$V(d) = V(e) + \frac{r}{w_i},$$

  tj. od $V(e)$ do budućeg virtualnog trenutka kad bi trebao biti gotov obzirom na zahtjev r i njegov udio u procesorskom vremenu (r skaliran sa w_i)
- raspoređivač odabire zadatak s najmanjim $V(d)$ (najbliži rok) za koji vrijedi $V(e) \leq V(t)$ (među svim zadacima podobnim za izvođenje)
- što je kvant r manji to će i rok biti bliži te će i zadatak doći prije na red – ovo je potrebno zadacima koji trebaju nakon buđenja (ili odblokiranja ili stvaranja) što prije dobiti procesorsko vrijeme (što manju latenciju) – uz ovaj algoritam latencija se smanjuje bez nadodane heuristike (koja je potrebna za CFS)

Sučelja za postavljanje načina raspoređivanja

- sučelja za upravljanje postojećim dretvama:

```
int pthread_setschedparam(pthread_t thread, int policy,
                          const struct sched_param *param);
int pthread_setschedprio(pthread_t thread, int prio);
int nice(int inc);
```

- sučelja za postavljanje parametara za buduće dretve (koje će se tek stvoriti):

```
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,struct sched_param *param);
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
```

```
void *(*start_routine)(void*), void *arg);
```

7.7.4. Raspoređivanje u Windowsima

- koristi se prioritetno raspoređivanje s podjelom vremena (slično SCHED_RR)
- prioritet se formira na osnovu: prioritetne klase procesa i prioritetne razine dretvi prema tablicama 7.1. i 7.2. Imenima klase procesa iz tablice treba dodati `_PRIORITY_CLASS` a imenima razine dretvi prefiks `THREAD_PRIORITY_`.

Tablica 7.1. Prioritetne klase procesa

oznaka klase
IDLE (ID)
BELOW_NORMAL (BN)
NORMAL (N)
ABOVE_NORMAL (AN)
HIGH (H)
REALTIME (RT)

Tablica 7.2. Prioritetne razine dretvi

oznaka razine
IDLE (ID)
LOWEST (L)
BELOW_NORMAL (BN)
NORMAL (N)
ABOVE_NORMAL (AN)
HIGHEST (H)
TIME_CRITICAL (TC)

- tablice 7.3. i 7.4. prikazuju dodjeljivanje prioriteta na osnovu prioritetnih klasa i razina

Tablica 7.3. 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

Tablica 7.4. 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

- prioritet 0 (najniži prioritet) rezerviran je za posebne dretve operacijskog sustava (npr. dretve koje brišu oslobođene stranice u postupku upravljanja spremnikom straničenjem).
- raspoređivanje dretvi za rad u stvarnom vremenu (engl. *real time*):
  - strategija: REALTIME_PRIORITY_CLASS uz kružno posluživanje (identično sa SCHED_RR)
  - prioriteti od 16 do 31 (veći broj veći prioritet)
- raspoređivanje "običnih" dretvi:

  - strategija: kružno (slično kao i SCHED_RR) uz iznimke:
    * radi rješavanja problema izgladnjivanja (da i dretve manjeg prioriteta ipak nešto povremeno rade)
    * dinamičkog dodavanja prioriteta procesima u fokusu (engl. *foreground*)
    * radi osiguravanja kvalitete usluge (ne samo na razini procesora)
  - prioriteti od 0 do 15 (veći broj veći prioritet)
- sučelja:

```
BOOL WINAPI SetPriorityClass(HANDLE hProcess, DWORD dwPriorityClass);
BOOL WINAPI SetThreadPriority(HANDLE hThread, int nPriority);
```

- više o raspoređivanju na windowsima na službenim stranicama

Zadatak 7.13. Raspoređivanje prema SCHED_FIFO, SCHED_RR i SCHED_OTHER

U nekom sustavu javljaju se poslovi/dretve A, B, C i D u trenucima 3.5, 0, 2,5 i 6,5 respektivno s trajanjima obrade 5, 5, 3 i 2 (respektivno). Pokazati rad poslužitelja ako se koristi:

d) raspoređivanje prema SCHED_FIFO uz $p_A > p_B = p_C = p_D$

e) raspoređivanje prema SCHED_RR uz $p_A > p_B = p_C = p_D$ i $t_q = 1$

f) raspoređivanje prema SCHED_OTHER uz kvantove po dretvama $t_{q_A} = 2$, $t_{q_B} = t_{q_C} = t_{q_D} = 1$ (npr. za $d_A = 5, d_B = d_C = d_D = 10$ – razlika od 5 u dobroti: $1{,}15^5 \approx 2$)

Zadatak 7.14. Raspoređivanje na dvoprocesorskom računalu

U nekom sustavu javljaju se poslovi/dretve A, B, C i D u trenucima 3.5, 0, 2,5 i 6,5 respektivno s trajanjima obrade 5, 5, 3 i 2 (respektivno). Pokazati rad **dvoprocesorskog** poslužitelja ako se koristi:

g) raspoređivanje prema SCHED_FIFO uz $p_A > p_B = p_C = p_D$

h) raspoređivanje prema SCHED_RR uz $p_A > p_B = p_C = p_D$ i $t_q = 1$

i) raspoređivanje prema SCHED_OTHER uz kvantove po dretvama $t_{q_A} = 2$, $t_{q_B} = t_{q_C} = t_{q_D} = 1$ (npr. za $d_A = 5, d_B = d_C = d_D = 10$ – razlika od 5 u dobroti: $1{,}15^5 \approx 2$)

Pitanja za vježbu 7

1. Opisati značenja veličina: α, $1/\alpha$, β, $1/\beta$, $\overline{T}$, $\overline{n}$ i ρ u kontekstu sustava s jednim poslužiteljem:
   a) kada se radi o deterministčkom sustavu
   b) kada se radi o nedeterminističkom sustavu s Poissonovom razdiobom dolazaka i eksponencijalnom razdiobom trajanja obrada.

   Kojim su formulama povezane zadane veličine?
2. Napisati i pojasniti Littleovo pravilo.
3. Navesti formulu za izračun vjerojatnosti pojave k događaja u vremenskom intervalu t, ako se koristi Poissonova razdioba za modeliranje dolazaka uz parametar α (prosječan broj dolazaka novih poslova u jedinici vremena).
4. Nacrtati Markovljev lanac koji prikazuje moguća stanja sustava u odnosu na trenutni broj poslova u sustavu ako se promatra jako mali interval Δt u kojem se mogu dogoditi promjene (doći novi posao ili neki posao napustiti sustav). Označiti vjerojatnosti prijelaza iz jednog stanja u drugo. Korištenjem grafa izraziti vjerojatnost da sustav u trenutku $t + \Delta t$ bude u stanju i (tj. $p_i(t + \Delta t)$).
5. Koja formula povezuje $\overline{T}$ s α i β ako se koriste Poissonova i eksponencijalna razdioba za modeliranje dolazaka i trajanja obrade?
6. Zašto kod nedeterminističkog sustava nije dozvoljeno opteretiti poslužitelj sa 100% opterećenja?
7. Ako je zadano prosječno opterećenje poslužitelja s ρ, izračunati vjerojatnost (navesti formule) da u nekom trenutku u sustavu ima:
   a) N poslova
   b) više od N poslova.
8. Opisati osnovne principe raspoređivanja dretvi:
   a) po redu prispijeća (engl. *First-In-First-Out – FIFO*)
   b) podjelom vremena (engl. *Round-Robin – RR*)

   Navesti prednosti i nedostatke navedenih principa raspoređivanja.
9. Opisati principe raspoređivanja dretvi koji se koriste u današnjim operacijskim sustavima (Windows, Linux, SCHED_FIFO, SCHED_RR, SCHED_OTHER).

8. UPRAVLJANJE SPREMNIČKIM PROSTOROM

Osim riječi *spremnik*, tj. *radni spremnik*, jednakovrijedna je i riječ *memorija*, koja se udomaćila u hrvatskom jeziku.

8.1. Uvod

8.1.1. Veličine radnog spremnika, što se sve nalazi u njemu

Veličina spremnika (RAM) za pojedine namjene (okvirno) (info)

- poslužitelj: 16 GB+
- osobno računalo: 4 GB+
- pametni telefoni i tableti: 512 MB+
- ugrađeni sustavi: obično u rasponu od 2 KB do desetke MB

Adresiranje spremničke lokacije – širina sabirnice i moguće veličine spremnika:

- 16 bita $\Rightarrow$ 64 KB = 65536 B
- 32 bita $\Rightarrow$ 4 GB $\approx 4 \cdot 10^9$ B
- 64 bita $\Rightarrow$ 16 EB (exa-) $\approx 2 \cdot 10^{19}$ B
  - ovo je previše i za današnje sustave
  - x64 procesori koriste 48 bita i teoretski mogu koristiti 256 TB spremnika

Pristup spremniku (od strane procesora)

- pri dohvatu instrukcija
- pri dohvatu i pohrani operanada (podataka)
- pri korištenju stoga

Što sve ide u spremnik?

- jezgra:
  - jezgrine funkcije (npr. `Započni_UI`, `ČekajBSEM`, ...)
  - strukture podataka (opisnici dretvi, procesa, naprava, liste, ...)
- programi:
  - instrukcije
  - podaci
  - gomila (heap, za malloc/free)
  - stog (za svaku dretvu zaseban)

8.1.2. Osnovne ideje upravljanja spremnikom

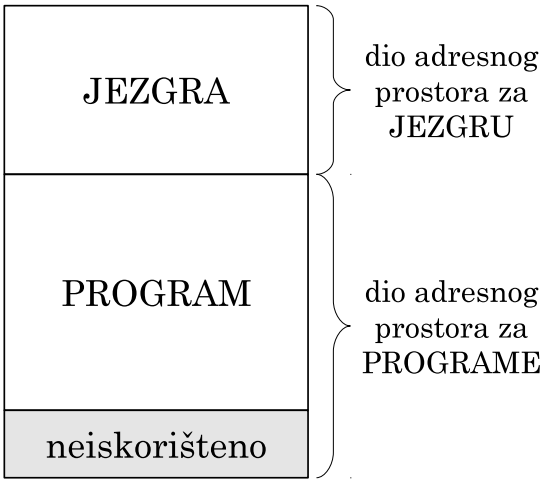
Pojmovi program i proces u ovom kontekstu

- pod pojmom proces obično podrazumjevamo i odvojeni adresni prostor od ostalih procesa
- ipak, ovdje se koristi i za načine upravljanja koji to nemaju
- *program* se odnosi na niz instrukcija i podataka čijim pokretanjem nastaje proces (OS pokreće program i nastaje proces)

Zahtjevi od programa i OS-a prema upravljanju spremnikom

- da sve "potrebno" za rad stane u spremnik
- zaštita "da nitko nikome ne smeta", tj. od grešaka u programima i zlonamjernih programa
  - da se zaštiti jezgra od procesa, proces od drugog procesa
- učinkovito koristiti radni spremnik i po potrebi pomoćne spremnike
- dati preporuke programerima kako učinkovito koristiti spremnik

Najjednostavniji način upravljanja: samo OS i jedan proces (slika 8.1.)



Slika 8.1. Upravljanje spremnikom u sustavima s jednim procesom

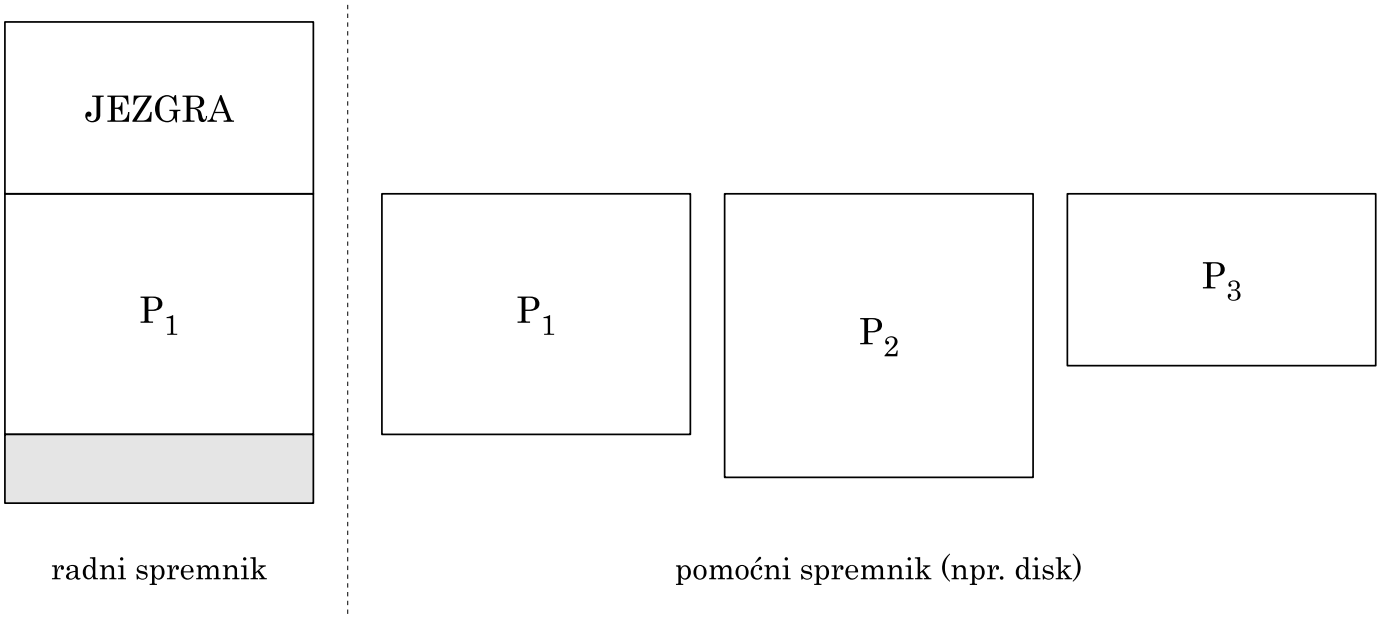
- ovo je dovoljno samo za vrlo jednostavne sustave
  - međutim, takvih sustav ima jako puno – većina ugrađenih sustava je jednostavno i ovo je dovoljno

Više procesa – svi u radnom spremniku

- ovo je "idealno", ali potencijalno traži puno spremnika (stoga i skupo rješenje)
- koristi se tamo gdje su procesi mali i svi stanu u spremnik (ugrađeni sustavi)

Više procesa – jedan u radnom spremniku, ostali na pomoćnom (slika 8.2.)

- na pomoćnom spremniku su pripremljeni svi procesi
- po jedan se učitava u radni spremni i izvodi
- pri zamjeni procesa radi se "velika" zamjena konteksta:
  1. proces iz radnog spremnika miče se na pomoćni spremnik
  2. drugi proces se učitava s pomoćnog u radni spremnik

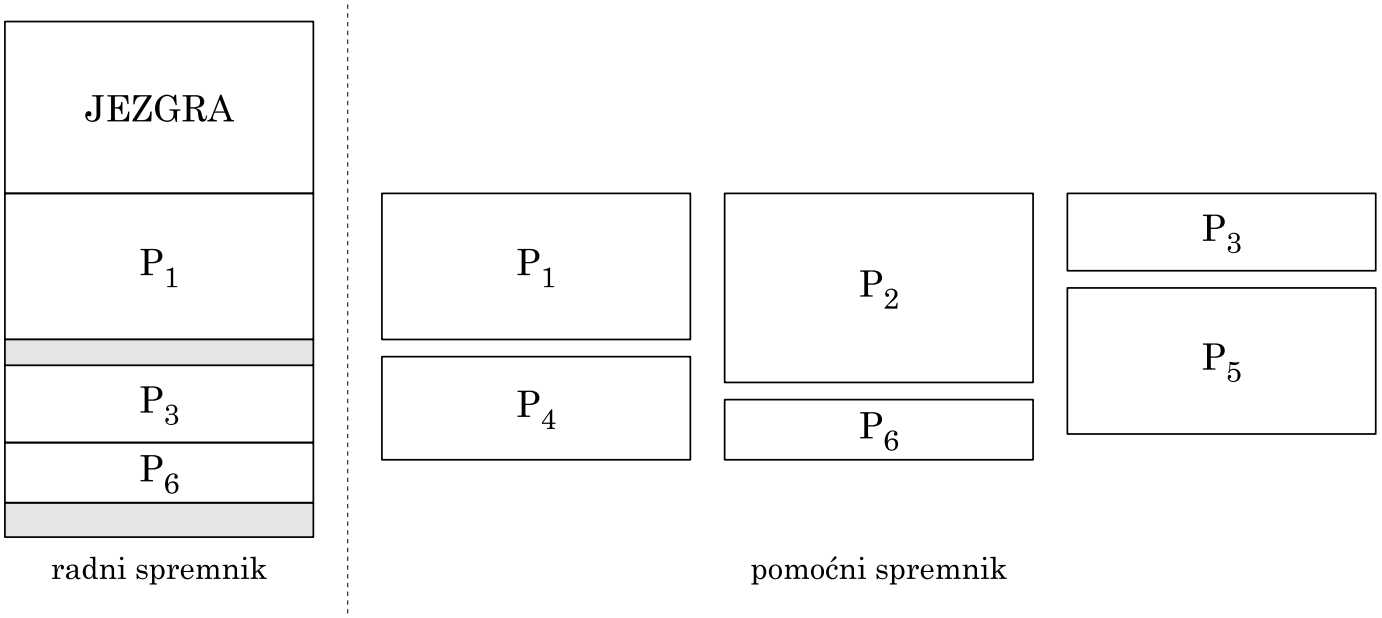


Slika 8.2. Korištenje pomoćnog spremnika za "veliku zamjenu konteksta"

Svojstva pomoćnog spremnika – diska

- disk se detaljnije razmatra u 9. poglavlju, sada samo osnovna svojstva
- vremenska svojstva diska:
  - vrijeme pristupa koda HDD-a (čitanje jednog bloka/sektora): red veličine ≈ 10 ms!
  - vrijeme pristupa koda SSD-a (čitanje jednog bloka/sektora): red veličine ≈ 0,05 ms!
  - čitanje procesa s diska od ≈ 1 ms do 100 ms (i više)!
- spremniku se pristupa znatno brže ≈ red veličine nekoliko ns (bar 1000 puta brže čak i uz SSD)
- disk je prespor da bi navedeni gornji način (prema slici 8.2.) bio zadovoljavajući i učinkovit

Učinkovito korištenje pomoćnog spremnika za upravljanje spremnikom (slika 8.3.)



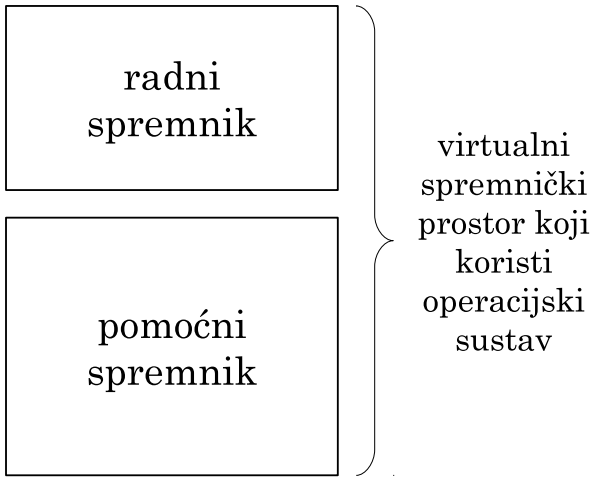
Slika 8.3. Primjer sustava s više procesa u radnom spremniku

- u radnom spremniku treba biti više od jednog procesa (ako svi ne stanu)

- koristiti pomoćni spremnik za privremenu pohranu svih procesa
  - svi se početno pripremaju na pomoćnom disku, a neki učitavaju u radni spremnik
  - za učitavanje/spremanje procesa koristiti sklopove s izravnim pristupom spremniku (DMA)
- zamjena jednog procesa (spremanje prvog pa učitavanje drugog) – velika zamjena konteksta
- za vrijeme obavljanja zamjene (korištenjem DMA načina) procesor može izvoditi neki drugi proces koji je u radnom spremniku – ne gubi se to vrijeme, ovakav sustav je učinkovit

Virtualni spremnički prostor

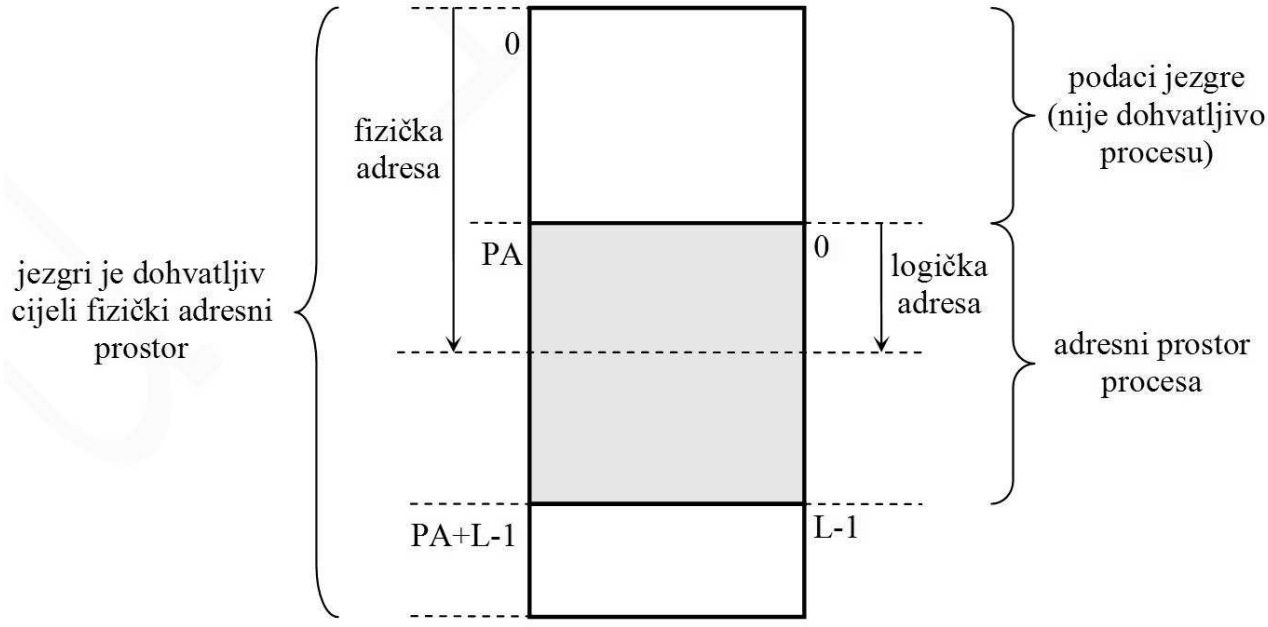
U sustavima koji koriste pomoćni spremnik za upravljanje spremničkim prostorom koristimo pojam virtualni spremnički prostor OS-a (engl. *virtual memory*) koji se sastoji od radnog spremnika i pomoćnog spremnika.



Slika 8.4. Virtualni spremnički prostor OS

Fizičke i logičke adrese

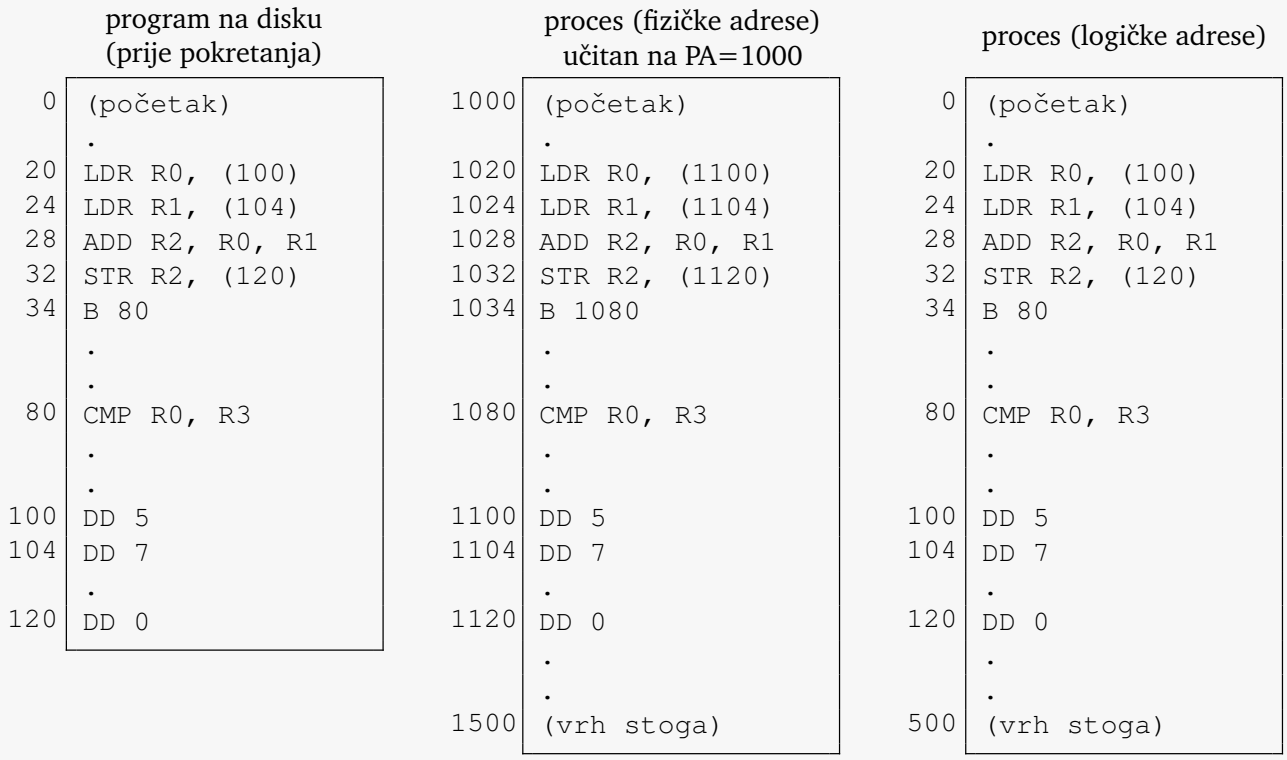
- apsolutne adrese = fizičke adrese → adrese koje idu na sabirnicu
- relativne adrese = logičke adrese → koriste se u procesu



Slika 8.5. Fizička i logička adresa

Primjer 8.1. Fizička i logička adresa

Odnos fizičke i logičke adrese procesa ovisi gdje se on nalazi.



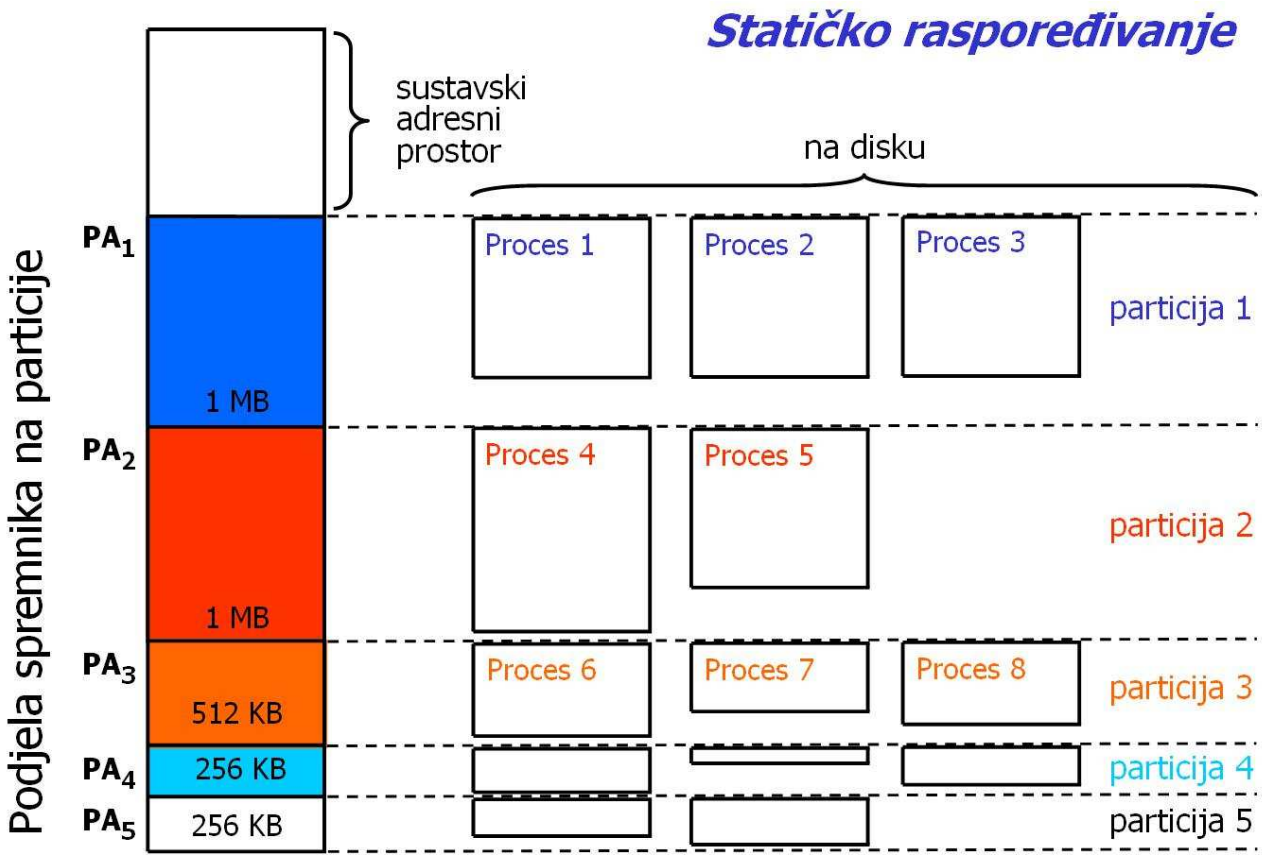
Procesni informacijski blok

- za upravljanje procesom potreban je i opisnik spremničkog prostora procesa
  - sadržaj opisnika ovisan je o algoritmu upravljanja spremnikom
  - opisnik mora opisivati korišteni dio radnog spremnika, ali i korišteni dio pomoćnog spremnika od strane procesa
- za upravljanje procesa, osim opisnika spremničkog prostora procesa potrebni su i opisnici dretvi, opisnici korištenih datoteka i slično
- skup potrebnih informacija za upravljanje procesom (koji uključuje i sve navedeno) naziva se *procesni informacijski blok*

Načini upravljanja spremnikom koji koriste pomoćni spremnik

1. statičko upravljanje: statički podijeliti spremnik na particije
2. dinamičko upravljanje: dinamički dijeliti spremnik prema potrebama
3. straničenje

8.2. Statičko upravljanje spremnikom



- dio spremnika namjenjenog za procese se u početku podijeli na *particije* unaprijed zadanih veličina.
- prilikom pokretanja procesa stvara se proces na pomoćnom spremniku
  - proces se priprema u apsolutnim adresama za tu particiju
  - odabir particije ide na osnovu potrebne veličine procesa i trenutnog zauzeća svih particija
- OS odabire koje će procese učitati – samo po jedan proces u pojedinu particiju
- kada se proces blokira, OS pokreće postupak njegove zamjene, a u međuvremenu, dok se zamjena ne obavi pod utjecajem DMA sklopova, OS odabire neku dretvu iz procesa koji se nalaze u radnom spremniku (u nekoj od drugih particija)

Svojstva statičkog upravljanja

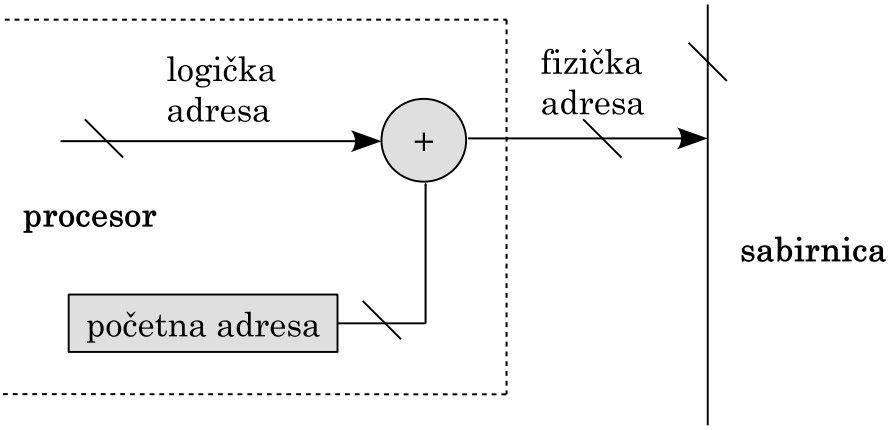
+ jednostavan model
– unutarnja fragmentacija – dio particije koji proces ne koristi ne može se iskoristiti od strane drugih procesa jer niti jedan od njih ne bi ispravno radio na toj lokaciji (procesi su pripremljeni samo za adrese koje počinju na početku odabrane particije)
– vanjska fragmentacija – kada svi procesi pripremljeni za jednu particiju su u nekom trenutku blokirani, tu particiju se ne može iskoristiti za druge procese
– nema zaštite u pristupu izvan procesu dodijeljenog dijela spremnika (particije)
  - greška u jednom procesu može uzrokovati grešku u drugom ili u jezgri
    – npr. zbog krive vrijednosti indeksa mijenja se podatak drugog procesa, što se može manifestirati kao greška u oba procesa

– veliki procesi se ne mogu pokretati

– (manji nedostatak) prilikom pokretanja proces treba prvo na pomoćnom spremniku pripremiti u fizičkim adresama za odabranu particiju (zbrajanjem logičke adrese s adresom particije za koju se priprema)

8.3. Dinamičko upravljanje spremnikom

- proces se priprema za logičke adrese – ostaje u logičkim adresama i u spremniku
- koristi se sklopovska pretvorba logičke adrese u fizičku u trenutku kad adresa izlazi iz procesora
- dovoljno je zbrajalo koje zbraja logičku adresu (LA) generiranu u procesoru s adresom početka segmenta gdje je proces smješten (taj registar postaje dio konteksta dretve).



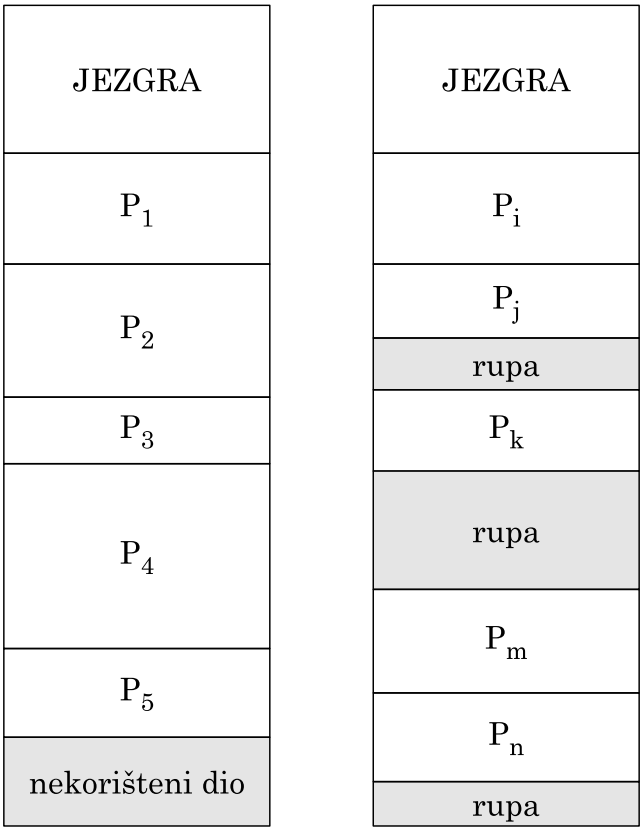
Slika 8.6. Sklop za pretvorbu logičke adrese u fizičku

Problem fragmentacije:

- u dinamičkom okruženju, procesi se stvaraju, rade, završavaju
- kad se proces miče iz spremnika u prostoru koji je on koristio nastaje "rupa"
- kad se neki proces učita u spremnik, učita se u neku rupu, ali često ne zauzme sav prostor rupe
- rupe koje ovako nastaju često nisu jedna do druge (kada bi se spojile u jednu veću), već su raspršene po spremniku – slobodni dio spremnika je fragmentiran
- fragmentacija postaje problem kada niti jedna rupa nije dovoljno velika za novi zahtjev iako ukupno ima dovoljno slobodnog spremnika
- fragmentacije se ne možemo riješiti što pokazuje Knuthovo 50% pravilo (definicija 8.1.)
- fragmentaciju možemo donekle kontrolirati sljedećim postupcima:
  - pri dodjeli uzeti najmanju rupu koja je dovoljno velika za zahtjev
  - pri oslobađanju bloka, nastalu rupu spojiti sa susjednim rupama
  - ako treba i zaustaviti sustav i napraviti "defragmentaciju" – pomicati programe tako da se rupe spoje

Definicija 8.1. Knuthovo 50% pravilo

U stabilnom sustavu s dinamičkim upravljanjem spremnikom broj rupa (n) jednak je 50%



Slika 8.7. Primjer stanja na početku rada te kasnije

broja punih blokova (m):

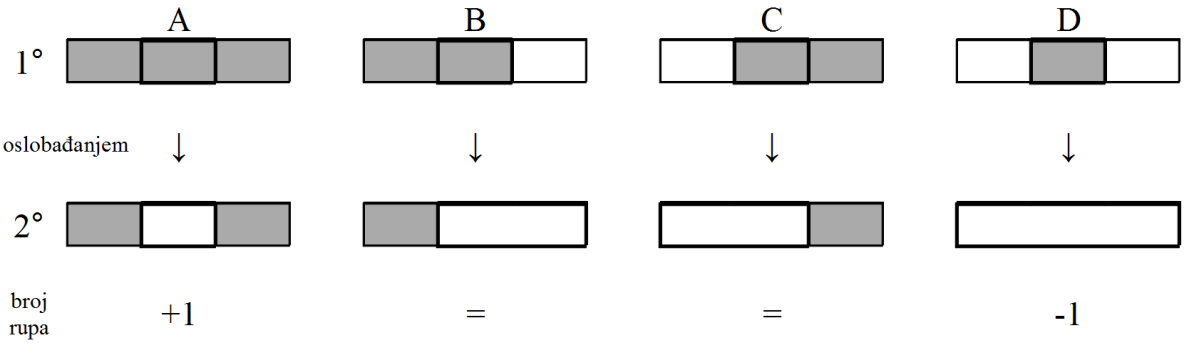
$$n = \frac{m}{2} \tag{8.1.}$$

Obzirom da je ukupan broj blokova zbroj punih blokova i rupa $m + n = 2n + n = 3n$ slijedi da trećinu ukupnog broja blokova čine rupe (po broju, ne po veličini).

Zadatak 8.1. Dokaz Knuthova pravila

Sustav se nalazi u stohastičkoj ravnoteži – vjerojatnost zahtjeva za spremnikom p_Z jednaka je vjerojatnosti zahtjeva za oslobađanje p_O. tj. $p_Z = p_O$.

U sustavu imamo 4 tipa punih blokova (puni blokovi su zasjenjeni):



Iz $1°$ se može izračunati broj punih blokova i broj rupa (a broj punih blokova tipa A, b ...):

- broj punih blokova m i broj rupa n:

$$\begin{aligned} m &= a + b + c + d \\ n &= \frac{b + c + 2d}{2} \end{aligned} \tag{8.2.}$$

Svaka rupa koju blok B vidi kao desnu mora biti ili lijeva od C ili D. Zato se broj rupa dijeli s dva. Isto tako, ako zanemarimo rubne slučajeve, nakon jedne rupe koju B vidi (desno) mora se pojaviti blok tipa C koji tu rupu vidi s lijeve strane, ili blok D – ali onda se konačno nakon D mora pojaviti blok C koji njegovu desnu rupu vidi kao lijevu!

Tj. vrijedi:

$$b = c \quad \Longrightarrow \quad \begin{aligned} m &= a + 2b + d \\ n &= \frac{b + b + 2d}{2} = b + d \end{aligned} \tag{8.3.}$$

Vjerojatnost povećanja broja rupa možemo izračunati (prema gornjoj slici) kao vjerojatnost oslobađanja uz uvjet da oslobođeni blok bude tipa A, tj.:

$$p_+ = p_O \cdot \frac{a}{m} \tag{8.4.}$$

Vjerojatnost smanjenja broja rupa možemo izračunati (prema gornjoj slici) kao vjerojatnost oslobađanja uz uvjet da oslobođeni blok bude tipa D, tj.:

$$p_- = p_O \cdot \frac{d}{m} \tag{8.5.}$$

(zanemarujemo mogućnost zahtjeva koji bi potpuno popunio neku rupu)

U stanju stohastičke ravnoteže te dvije vjerojatnosti moraju biti jednake:

$$p_+ = p_- \quad \Longrightarrow \quad p_O \cdot \frac{a}{m} = p_O \cdot \frac{d}{m} \quad \Longrightarrow \quad a = d \quad \Longrightarrow \begin{matrix} m = 2(a+b) \\ n = a + b \end{matrix} \Longrightarrow \quad n = \frac{m}{2} \tag{8.6.}$$

Knuthovo 50% pravilo: $n = \frac{m}{2}$ – broj rupa jednak je polovici broja punih blokova

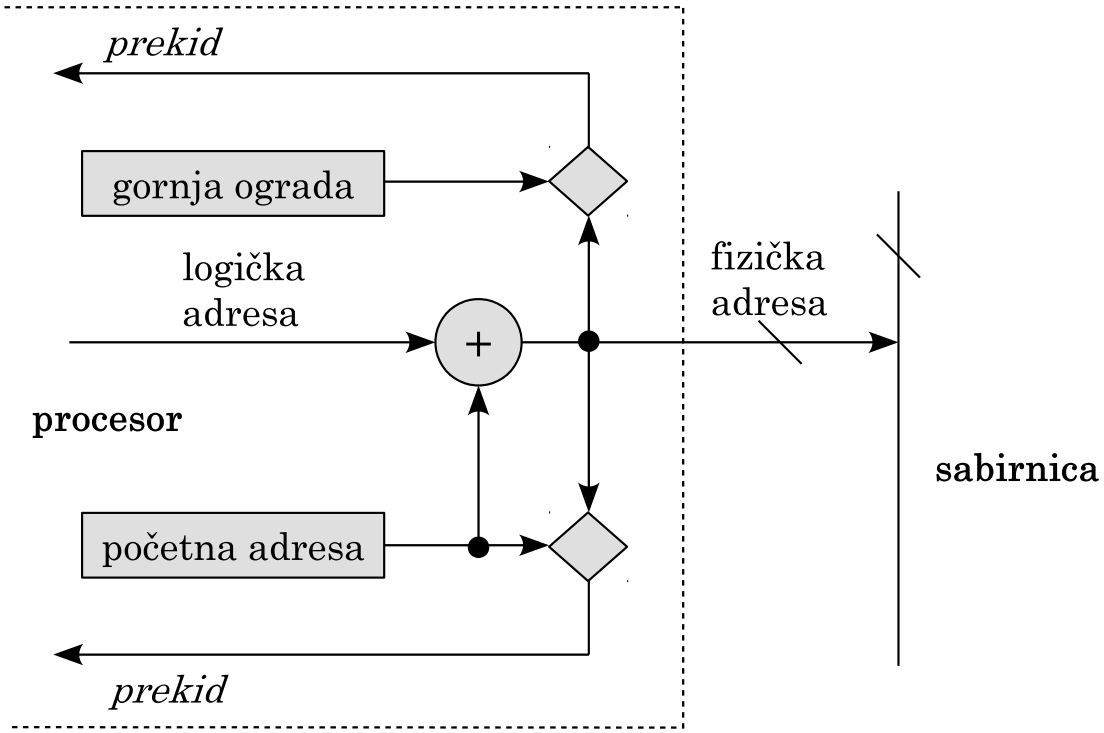
---

Knuthovo pravilo pokazuje problem rupa odnosno rascjepkanog slobodnog prostora koji se javlja u mnogim slučajevima.

Sustavi zahvaćeni Knutovim pravilom (info)

- dinamičko upravljanje spremnikom (na više razina)
  - dodjela prostora za procese (na opisani način dinamičkog upravljanja spremnikom)
  - dodjela prostora za potrebe jezgre (opisnici, međuspremnici i sl.)
  - upravljanje prostorom i kod drugih načina upravljanja (upravljanje okvirima)
  - gomila (heap) – dodjela prostora unutar procesa (malloc/free)
- upravljanje datotečnim sustavom (o njemu više kasnije)
  - zauzeti i slobodni dijelovi diska

Zaštita spremničkog prostora



- uz zbrajalo mogu se dodati dva komparatora (uspoređivača, oduzimala) koja će spriječiti da proces izađe iz svog dodijeljenog segmenta (slika): donja ograda – početna adresa, gornja ograda – najveća adresa koja pripada procesu
  - dobiva se sklopovska zaštita spremnika (engl. *memory protection unit – MPU*)
  - pri pokušaju dohvata adrese izvan ograda sklop izaziva prekid – OS tada prekida izvođenje procesa uslijed takve nepopravljive pogreške
  - ovakva zaštita je jednostavna za ostvariti te se stoga može koristiti u jednostavnijim procesorima

Svojstva dinamičkog upravljanja spremnikom

+ proces ostaje u logičkim adresama
  - proces se može izvoditi s bilo kojeg dijela spremnika – jednom se može učitati na jednu lokaciju, a potom (nakon što je izbačen iz spremnika) na neku drugu
+ zaštita spremnika (uz MPU)
– treba (jednostavna) sklopovska potpora (kod statičkog nije potrebna!)
– fragmentacija
– ne mogu se pokretati procesi koji ne stanu u radni spremnik!

U odnosu na statičko upravljanje:

+ fleksibilnije, nije potrebno unaprijed podijeliti spremnik
+ prosječno veća iskoristivost spremnika
+ nema vanjske fragmentacije, ali ima fragmentacije
- (info) u početku je potrebno znati potrebe procesa za spremnikom i toliko se prostora zauzme – kasnije je teško povećavati ili smanjivati adresni prostor (vrijedi i za statičko i dinamičko upravljanje spremnikom podjednako)

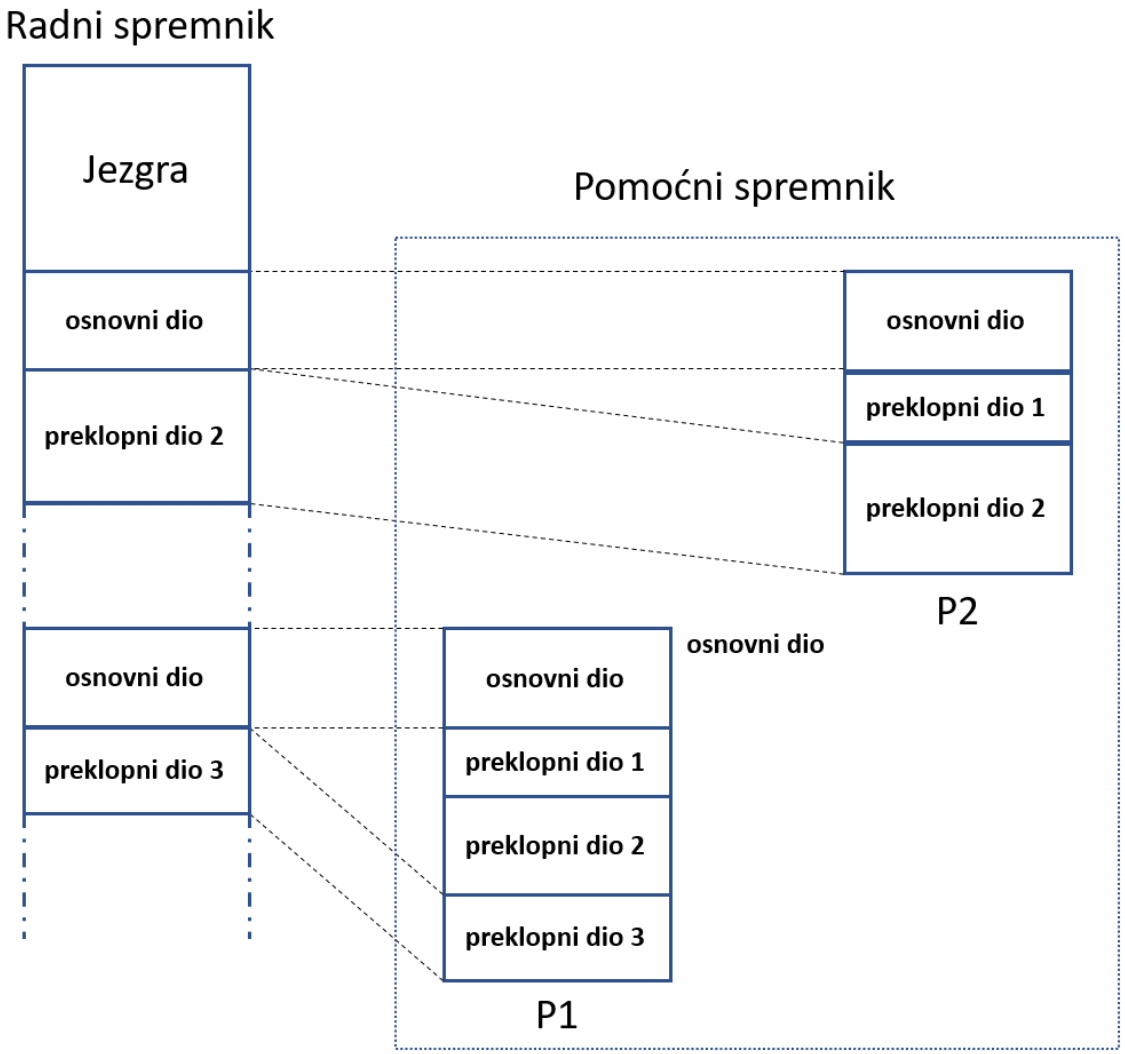
Statičko i dinamičko upravljanje u praksi? (info)

- u praksi se statičko/dinamičko vrlo rijetko koristi
  - u jednostavnim sustavima procesi nisu potrebni
  - u složenijima se koristi straničenje koje ima puno bolja svojstva
- u okviru ovog poglavlja prikazani su kao mogućnosti i da se vide njihove prednosti i nedostaci

Kako riješiti problem velikih procesa?

- osnovna ideja: učitavati samo one dijelove procesa koji su trenutno potrebni, a ostale ostaviti na pomoćnom spremniku
- ali kako? pod djelovanjem samog programa ili OS-a?

Preklopni način rada (slika 8.8.)



Slika 8.8. Preklopni način rada

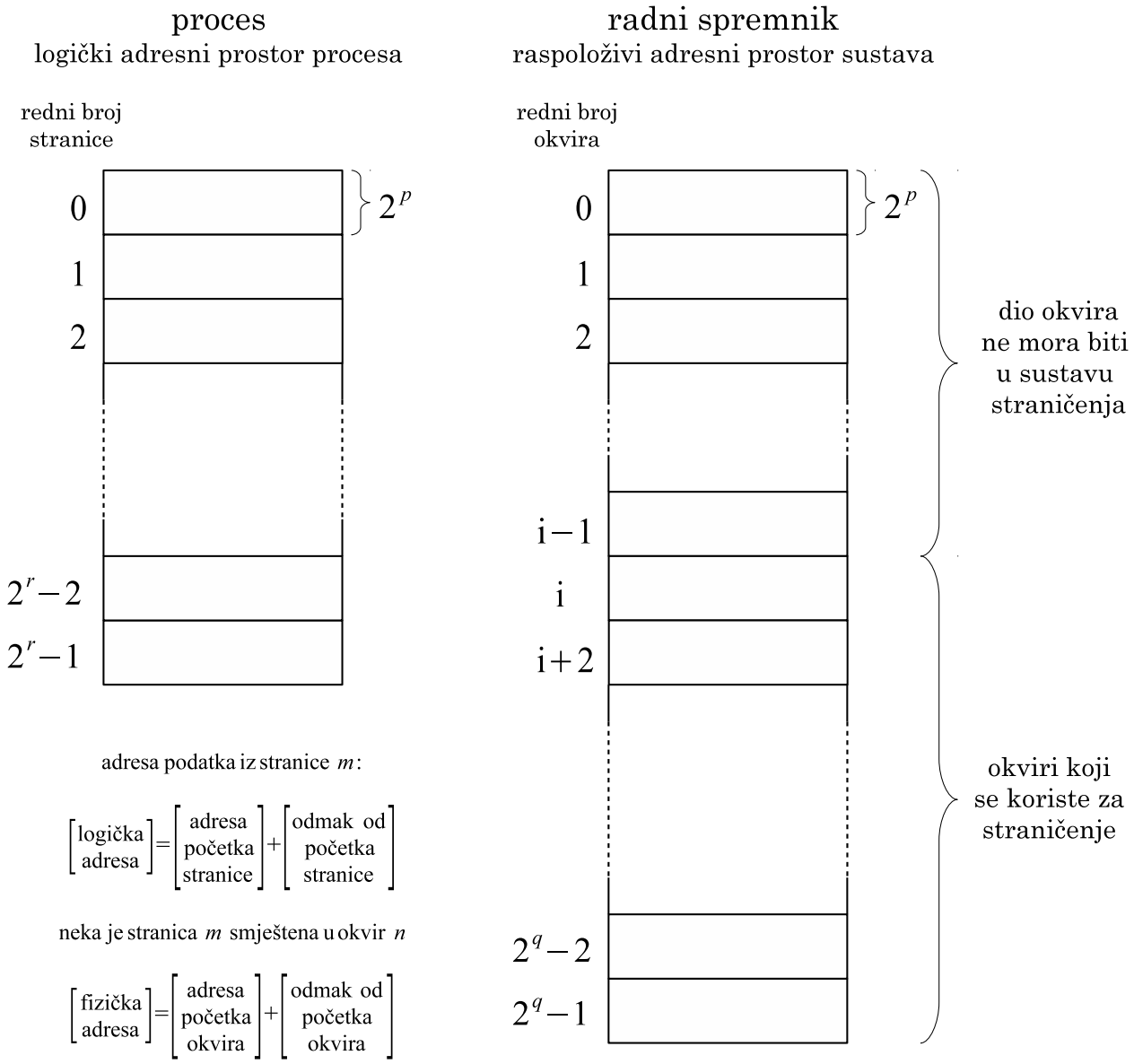
- pod utjecajem procesa zatražiti zamjenu djelova procesa (jedan dio van iz spremnika na pomoćni spremnik, a drugi u spremnik)
- problem: presloženo, zahtijeva od programera da tako izgradi program i predviđa potrebe kada što izbaciti i ubaciti ...
- rješenje: neka OS upravlja time koji dijelovi procesa će biti u spremniku u pojedinom trenutku – neka prati koji su potrebni i neka ih učitava po potrebi → *straničenje*

8.4. Straničenje

8.4.1. Stranice, okviri, tablica prevođenja

Osnovne ideje:

- proces se dijeli na *stranice*
- radni spremnik se dijeli na *okvire*
- jedna stranica stane u jedan okvir (istih su veličina, uobičajeno 4 KB)
- pretvorba adresa obavlja se dodatnim *sklopom* uz odgovarajuću *strukturu podataka* – tablicu prevođenja, koju održavaju jezgrine funkcije
- u radnom spremniku nalaze se samo trenutno potrebne stranice procesa
- u pomoćnom spremniku nalaze se sve stranice procesa (u ovom modelu; u stvarnim sustavima se samo stranice procesa koje nisu u radnom spremniku nalaze u pomoćnom spremniku)
- stranice se učitavaju prema potrebi – kad su potrebne za rad procesa

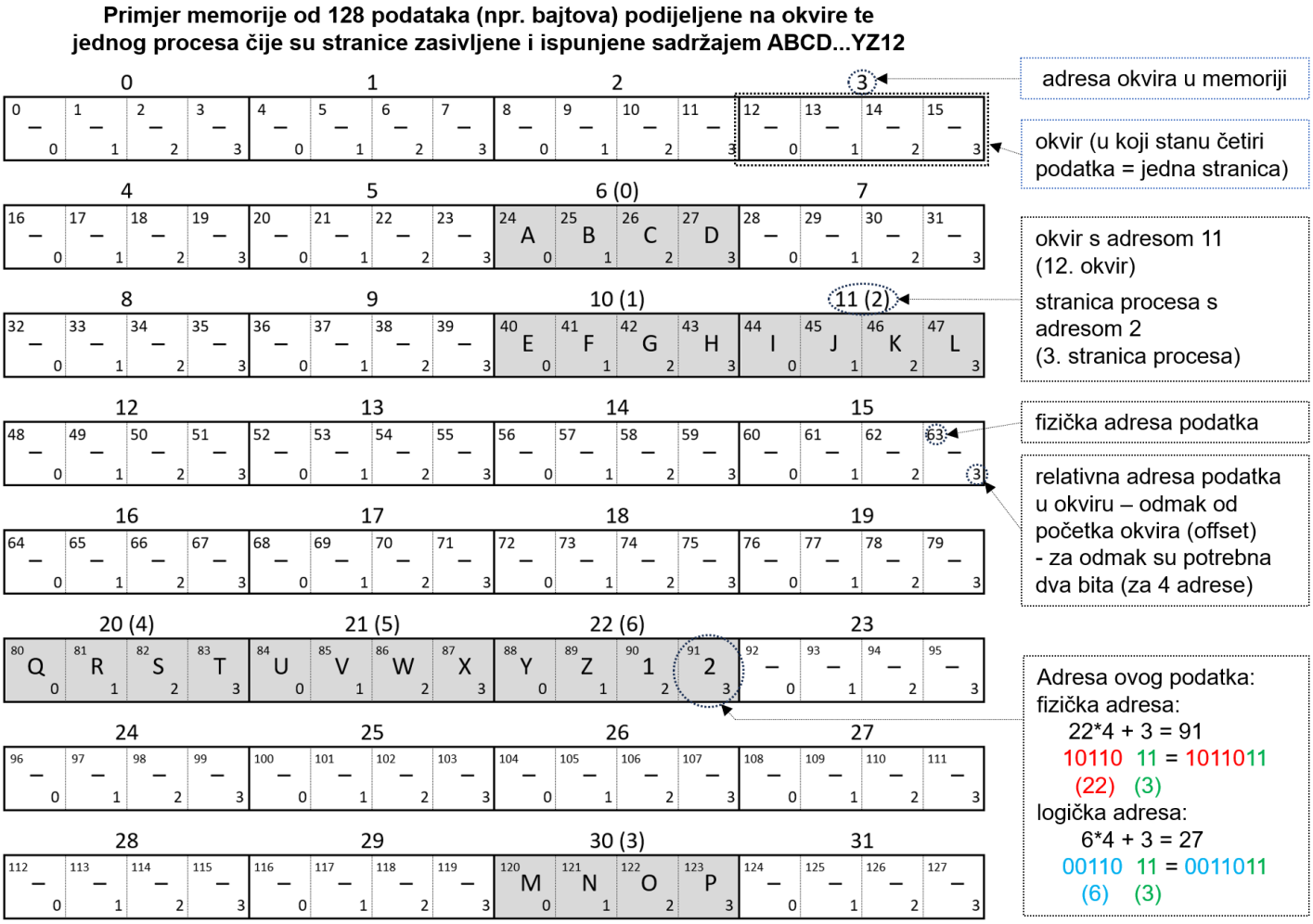


Slika 8.9. Podjela na stranice i okvire

Veličina stranice mora biti potencija broja 2 da bi se adresa (logička i fizička) mogla podijeliti

na dva dijela

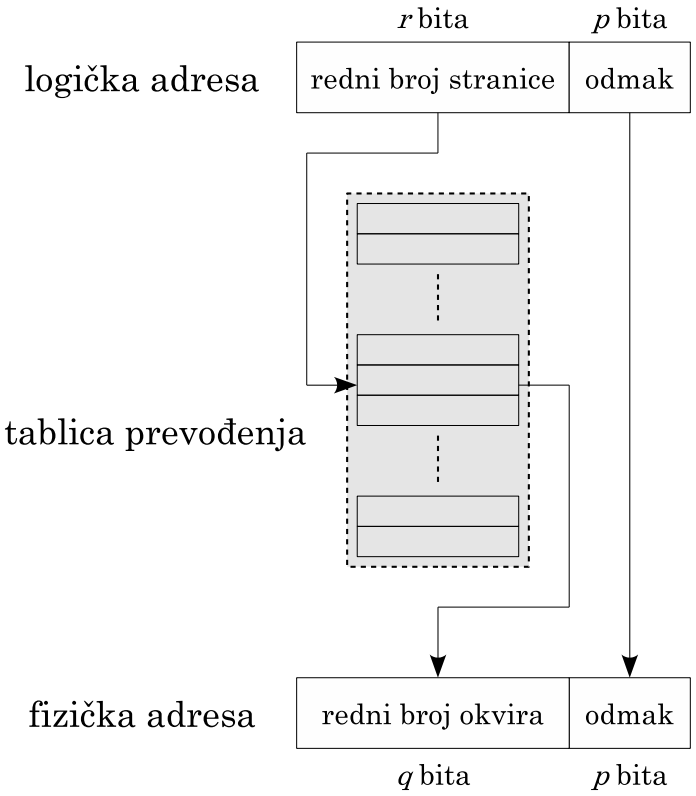
- logička adresa:
  - *redni broj stranice* (indeks stranice) – viših r bita adrese
  - *odmak* od početka stranice – nižih p bita adrese
- fizička adresa:
  - *redni broj okvira* (indeks okvira) – viših q bita adrese
  - *odmak* od početka okvira – viših p bita adrese



Slika 8.10. Ilustrativni primjer s malim spremnikom i malim okvirima

Tablica prevođenja

- za svaku stranicu procesa postoji jedan opisnik – jedan redak u tablici prevođenja
- zapisana je u opisniku procesa, za svaki proces treba zasebna tablica
- izgrađuje ju i održava OS, koristi sklop za pretvorbu adresa



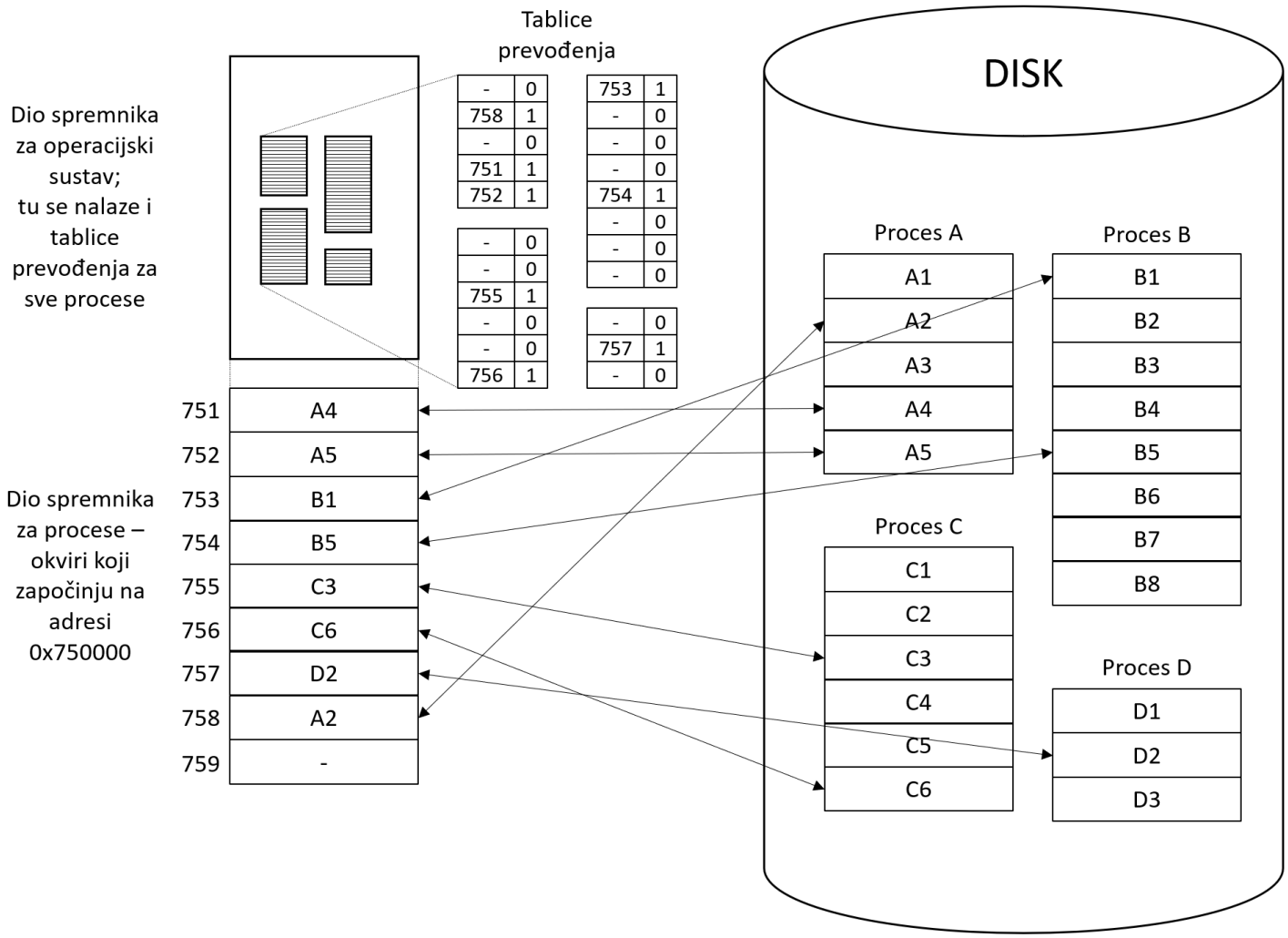
Slika 8.11. Pretvorba adrese kod straničenja

- svaki opisnik stranice se sastoji od dva dijela:
  - adrese okvira u kojem se stranica nalazi (ako se nalazi)
  - zastavice – detalji o stranici
    - bita prisutnosti: nalazi li se stranica u radnom spremniku (1) ili ne (0)
    - bitova koji označavaju pristup stranici, promjenu sadržaja, zaštitu od promjene/pristupa i sl. (zastavice su detaljnije opisane kasnije)

Pretvorba logička $\Longrightarrow$ fizička:

- *odmak* se prekopira
- viši bitovi logičke adrese – *redni broj stranice* – koristi se kao indeks u tablici prevođenja iz koje se dohvaća opisnik te stranice
- u opisniku stranice piše *redni broj okvira* u kojem se stranica nalazi (kada je bit prisutnosti postavljen, ako nije sklop izaziva prekid zbog promašaja – o tome kasnije)

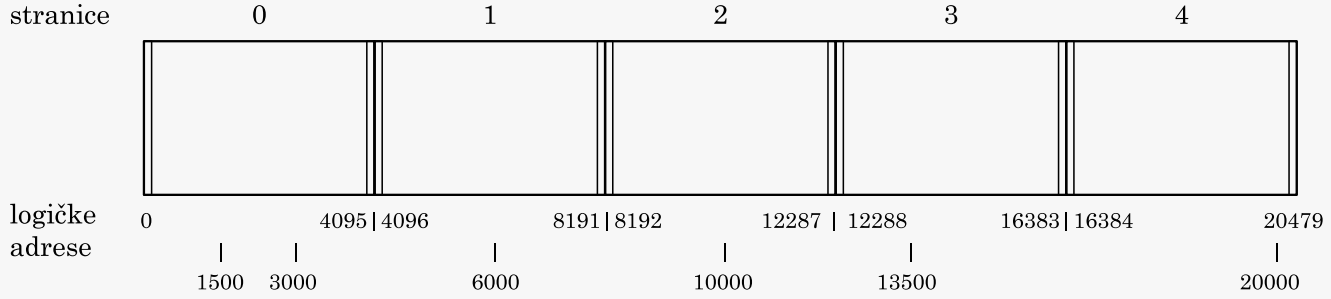
Prevođenje adresa mora biti obavljeno sklopom – inače bi sustav bio prespor



Slika 8.12. Ilustrativni primjer straničenja sa svim elementima i njihovim smještajem

Primjer 8.2. Stranice, okviri, logičke i fizičke adrese

Neki proces se sastoji od pet stranica, svaka veličine 4096B prema slici 8.13. Uz tablicu prevođenja 8.1. pokazati kako će se prevesti izdvojene logičke adrese (1500, 3000, ...).



Slika 8.13. Proces, u logičkom adresnom prostoru

Tablica 8.1. Tablica prevođenja

stranica	indeks okvira	bit prisutnosti
0	0x108B	1
1		0
2	0x3039	1
3		0
4	0x673	1

Pri pretvorbi iz logičke u fizičku, adresa se najprije dijeli na dva dijela, gornji dio koji označava redni broj stranice procesa te donji dio koje označava odmak od početka stranice. Obzirom da je stranica velika 4 KB, za odmak je potrebno 12 najnižih bita adrese, što se u heksadekadskom zapisu zapisuje s tri znamenke. U nastavku će se adrese najprije prikazati u heksadekadskom obliku, a da bi se ovo rastavljanje pojednostavilo.

Npr. logička adresa 10000 = 0x2710 se rastavlja na {0x2, 0x710} te preko tablice prevođenja prevodi u fizičku adresu {0x3039, 0x710}, tj. 0x3039710.

Tablica 8.2. Preslikavanje adresa

logička adresa			fizička adresa	
LA	hex(LA)	{stranica, odmak}	{okvir, odmak}	hex(FA)
1500	0x05DC	{0x0, 0x5DC}	{0x108B, 0x5DC}	0x108B5DC
3000	0x0BB8	{0x0, 0xBB8}	{0x108B, 0xBB8}	0x108BBB8
6000	0x1770	{0x1, 0x770}	promašaj (prekid)	
10000	0x2710	{0x2, 0x710}	{0x3039, 0x710}	0x3039710
13500	0x34BC	{0x3, 0x4BC}	promašaj (prekid)	
20000	0x4E20	{0x4, 0xE20}	{0x673, 0xE20}	0x673E20

Stranice 1 i 3 nisu radnom spremniku pa se adrese 6000 i 13500 ne mogu pretvoriti u fizičke (sklop će izazvati prekid u pokušaju pretvorbe).

8.4.2. Struktura i organizacija logičkog adresnog prostora procesa (info)

Dijelovi spremnika koje koriste dretve, redom kojim se oni slažu u logičkom adresnom prostoru procesa:

1. segment instrukcija
   - svaka dretva izvodi nekakav niz instrukcija (npr. kod u početnoj funkciji dretve)
   - više dretvi može izvoditi iste instrukcije (više dretvi može imati istu početnu funkciju)
   - svaka dretva zasebno izvodi "svoje" instrukcije (ima svoj "kontekst")
   - u procesu se sve instrukcije grupiraju u segment instrukcija
2. segment statički alociranih podataka
   - dretve koriste podatke s različitih dijelova spremnika: konstante, globalne varijable, lokalne varijable, gomilu
   - razni podaci se nalaze u različitim segmentima procesa
   - segment statički alociranih podataka, koji se u logičkom adresnom prostoru procesa nalazi nakon instrukcija sadrži konstante i globalne varijable

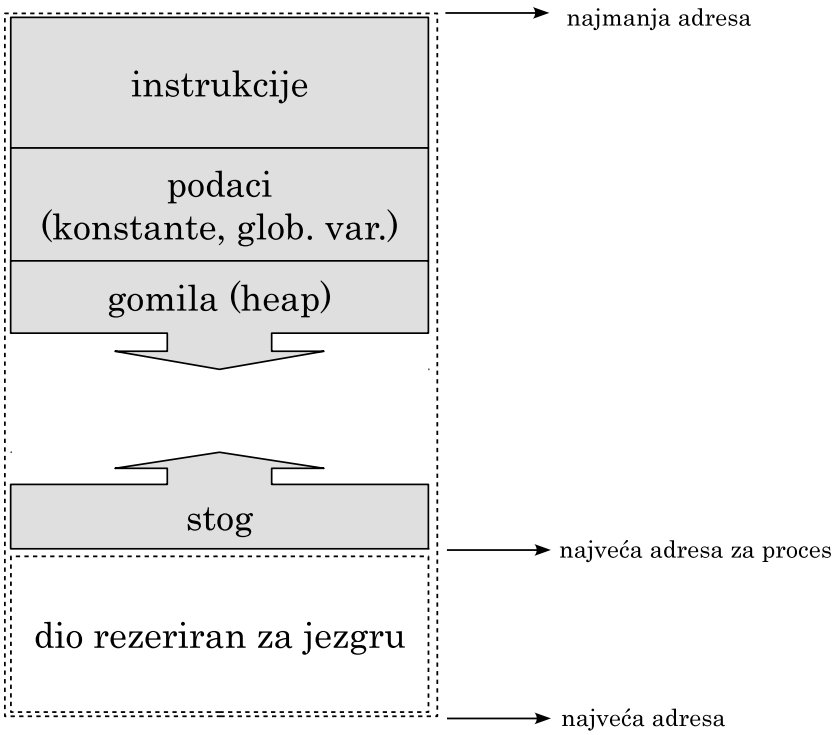
   a) konstante
      - nizovi znakova (stringovi) i slične konstante
      - npr. u naredbi: `printf("najveci broj je %d\n", najveci);` se niz znakova `"najveci broj je %d\n"` mora negdje spremiti pri učitavanju programa
      - sve se konstante spremaju u zajednički prostor i dohvaćaju preko adresa

   b) globalne varijable
      - globalne varijable su varijable definirane izvan funkcija
      - globalne varijable su zajedničke za sve dretve
3. gomila (heap)
   - gomila nastaje radom procesa i može biti proizvoljno velika u pojedinom trenutku
   - obzirom na dinamičko svojstvo promjene veličine, gomila se ne nalazi među globalnim varijablama, nego iza njih, u dijelu procesa koji može rasti
   - gomila je zajednička za sve dretve istog procesa
   - gomila se koristi preko kazaljki (a kazaljke mogu biti definirane lokalnim ili globalnim varijablama)
4. stog
   - stog se koristi pri pozivu funkcija na sljedeći način (najčešće):
     a) na stog se stavljaju argumenti za funkciju
     b) na stog se sprema povratna adresa (za povratak iz funkcije)
     c) na stog se spremaju registri koji će se mijenjati u funkciji (da bi se očuvao njihov sadržaj prije povratka iz te funkcije)
     d) na stogu se rezervira mjesto za lokalne varijable
   - sve navedeno što se sprema na stog pri pozivu funkcije naziva se "okvir stoga"

- stog "raste" prema manjim adresama
- lokalne varijable
  - lokalne varijable su varijable definirane u početnoj funkciji dretve ili u nekoj od funkcija koje se pozivaju iz nje
  - varijable nestaju izlaskom iz funkcije u kojoj su definirane

Organizacija tablice prevođenja

- kada bi tablica prevođenja bila linearna i za svaki proces opisivala cijeli mogući adresni prostor tog procesa, onda bi bila jako velika (zauzimala bi znatan dio spremnika)
  - npr. za 4 GB adresnog prostora i stranice veličine 4 KB treba 2^{20} redaka!
- takva linearna tablica nije potrebna jer većini procesa ne treba cijeli adresni prostor (npr. pogledati zauzeća spremničkog prostora procesa u task manageru)
- međutim, radi fleksibilnosti koristi se početak i kraj adresnog prostora procesa
  - na početku su instrukcije i podaci
  - iza njih raste gomila (koristi se kod malloc/free, new/delete operacija)
  - na kraju je stog koji raste prema manjim adresama



Slika 8.14. Struktura procesa (logički adresni prostor)

Dio procesa može opisivati (mapirati) dio same jezgre

- na Windowsima i Linuxu dio jezgre je mapiran u zadnjem dijelu adresnog prostora (npr. zadnja 2 GB kod 32-bitovnih Win32 sustava, od 2^{47} na 64-bitovnim)
- taj dio ne može koristiti proces iz svojih dretvi, već je dohvatljiv samo u jezgrinim funkcijama (dretve nemaju privilegije za korištenje tih stranica – to se može postaviti u opisniku tih stranica)
- npr. u taj dio se može smjestiti kod za prihvat prekida

Primjer 8.3. Primjer programa koji ispisuje adrese raznih dijelova procesa

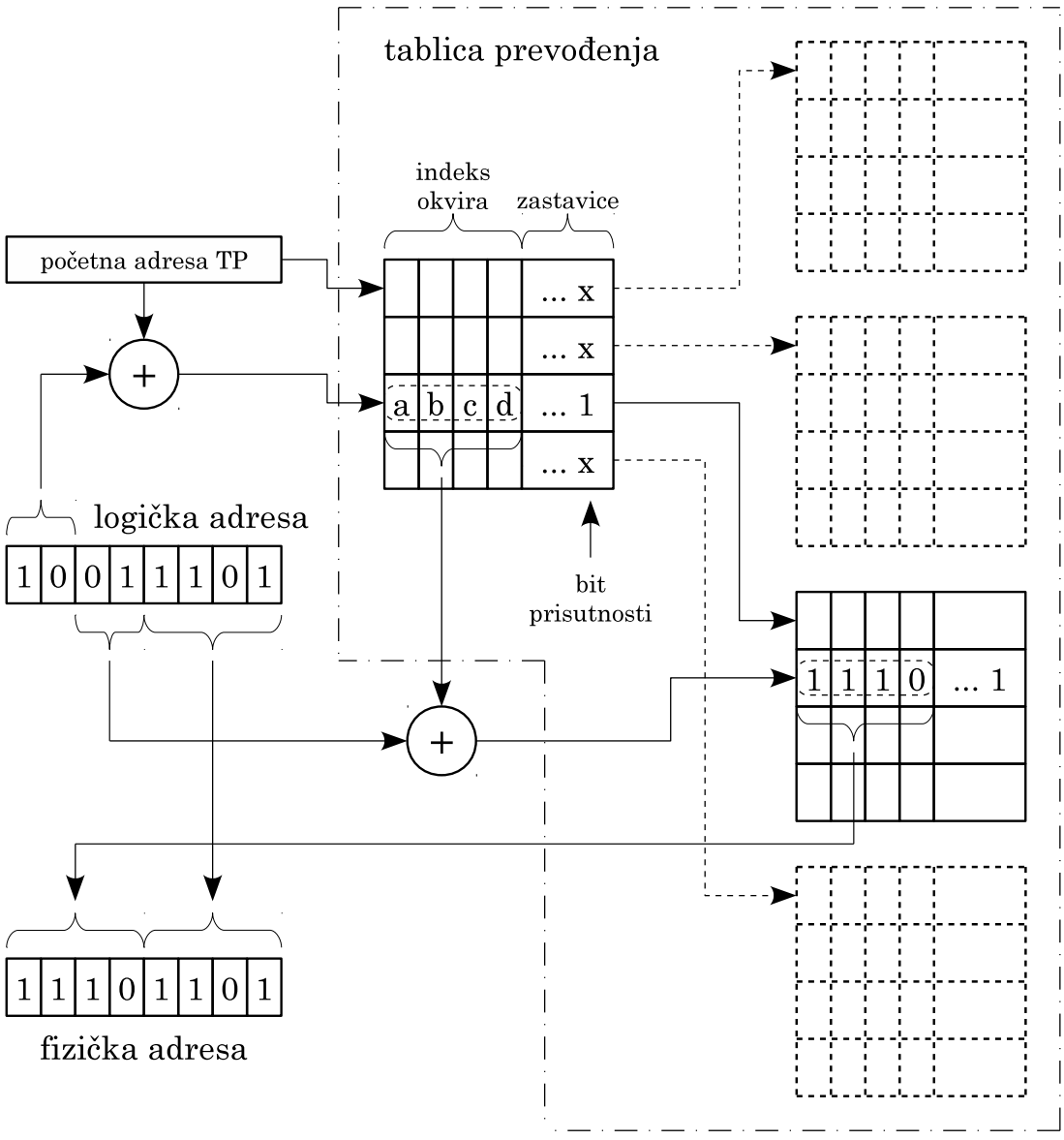
```
#include <stdio.h>
int a = 1;
int main() {
    int b = 2;
    printf("gl. var. => %p\n", &a);
    printf("lok.var. => %p\n", &b);
    printf("f. main  => %p\n", main);
    return 0;
}
```

```
# na 32-bitovnom Linuxu
gl. var. => 0x804a014
lok.var. => 0xbfb4f1bc
f. main  => 0x80483e4

# na 64-bitovnom Linuxu
gl. var. => 0x601020
lok.var. => 0x7fff1164fb1c
f. main  => 0x4004f4
```

8.4.3. Hijerarhijska organizacija tablice prevođenja

- tablicu podijeliti na manje tablice koje su hijerarhijski povezane – one u višoj razini sadrže opisnike tablica iz niže razine, a tablice u nižim sadrže opisnike stranica, ili opisnike tablica u još nižim razinama
- tablice u nižim razinama možda i nisu potrebne ako taj dio procesa još ne postoji

Slika 8.15. prikazuje trivijalni sustav s dvorazinski organiziranom tablicom.



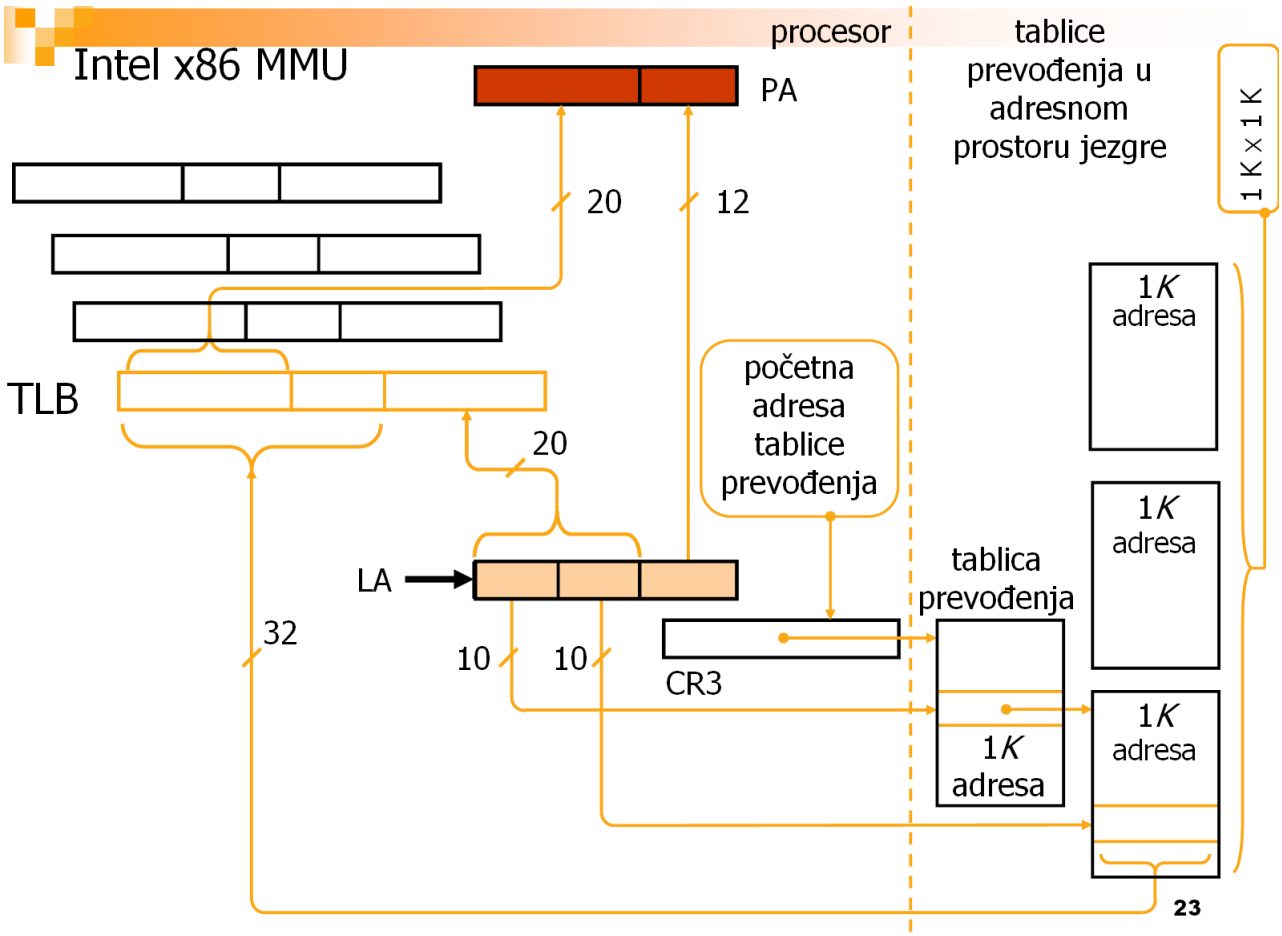
Slika 8.15. Primjer dvorazinske organizacije tablice prevođenja

Primjer prevođenja kod arhitekture x86

U stvarnim sustavima (32-bitovnim arhitekturama) potrebno je tek nekoliko dijelova tablice prevođenja: početni dio u hijerarhiji te nekoliko dijelova u drugoj razini (za opis instrukcija i podataka na početku, te za opis stoga na kraju)

Tablica prevođenja je u radnom spremniku te ju sklop mora čitati pri prevođenju adresa

- dohvat opisnika ide u dva koraka – dva dohvata iz spremnika
  - prvo se dohvaća adresa tablice u drugoj razini (iz tablice u prvoj)
  - iz tablice u drugoj razini dohvaća se opisnik
- za jedan dohvat podataka za proces bila bi potrebna tri pristupa spremniku!
- da se to izbjegne u velikoj većini slučajeva koristi se priručni spremnik za nedavno dohvaćene opisnike stranica (engl. *translation lookaside buffer – TLB*)
- tek ako opisnik već nije u TLB on se mora dohvatiti iz spremnika (uz dva dodatna dohvata)



Slika 8.16. Primjer dvorazinske organizacije kod x86 arhitekture

Elementi opisnika stranice:

- bitovi 31-12 – indeks okvira u kojem se stranica nalazi
- bitovi 11-0 su zastavice:
  - V – bit prisutnosti (engl. *validity bit*) – je li stranica u radnom spremniku?
  - A – je li se stranici pristupalo (engl. *accessed*)
  - D – je li stranica mijenjana, "prljava" (engl. *dirty*)
  - W – zaštita od promjene (engl. *write protect*)

  - O – stranica je za OS ne za proces
  - Wt – “write through” (svaka promjena stranice pokreće njenu pohranu i na pomoćni spremnik tako da i u slučaju nestanka napajanja sustav ostaje zapamćen na pomoćnom spremniku)
  - Gl – globalna (npr. za ostvarenje dijeljenog spremnika)

Neke arhitekture koriste dvorazinsku organizaciju tablice prevođenja (npr. i386), a neke i višerazinsku.

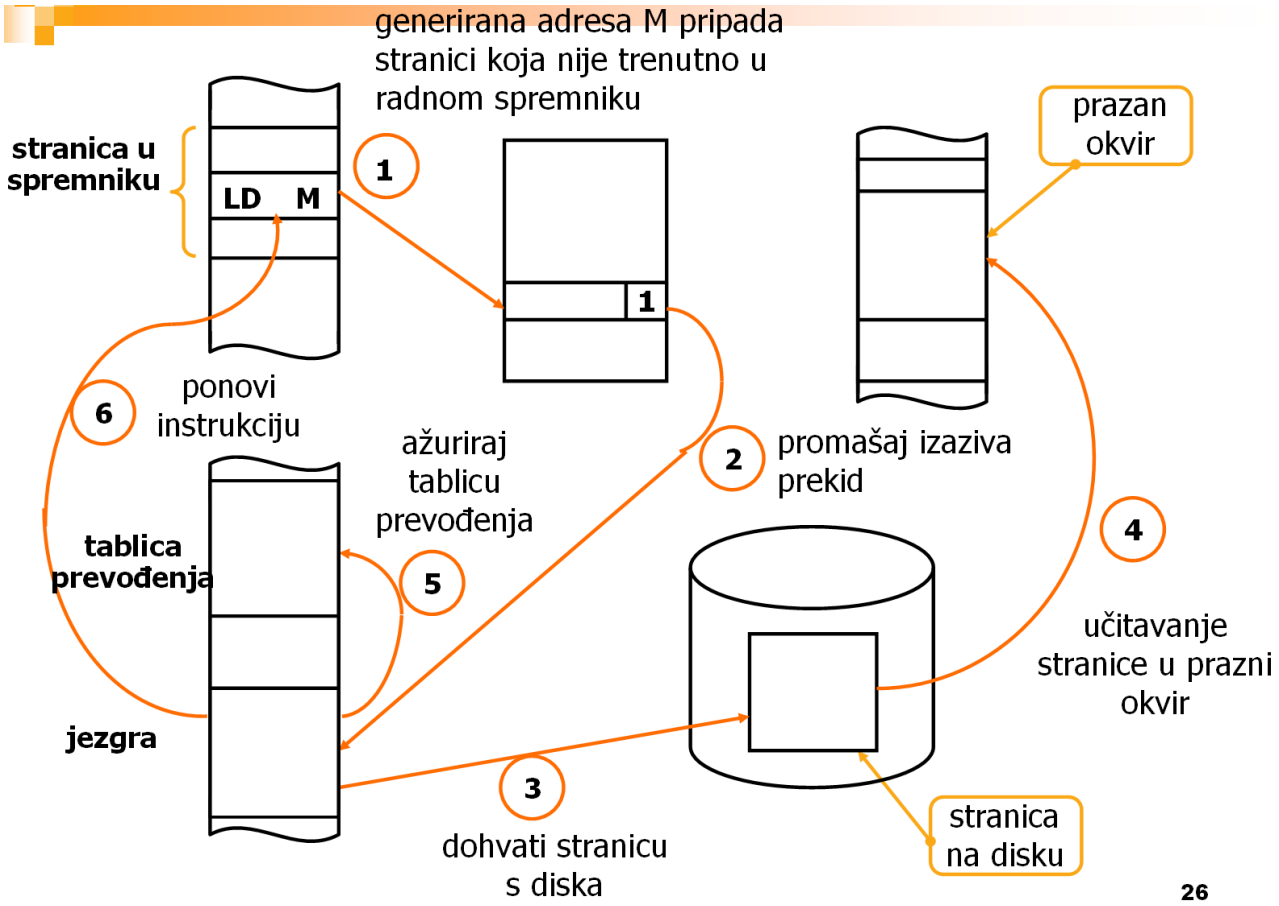
Primjerice, kod 64-bitovne arhitekture (x86_64, amd64):

- tablica prevođenja ima čak četiri razine hijerarhije
  - najnižih 12 bita adrese se prepisuje u fizičku adresu
  - idućih 4x9 bitova služi za adresiranje u tablicama različitih razina
  - najviši bitovi adrese (49-64) se ne koriste
    - i bez njih je adresni prostor za radni spremnik golem 2^48=256 TB
    - korištenje dužih adresa tražilo bi još složenije sklopovlje

8.4.4. Straničenje na zahtjev

Što kada proces generira adresu za stranicu koja nije u radnom spremniku?

- kažemo da se dogodio *promašaj* – sklop za upravljanje spremnikom izaziva *prekid*
- u obradi prekida OS dohvaća stranicu s pomoćnog spremnika i stavlja ju u radni



Slika 8.17. Operacije po promašaju

Straničenje na zahtjev (engl. *demand paging*) je način upravljanja spremnikom kod kojeg se

početno samo mali dio procesa učita u radni spremnik, a ostale stranice se učitavaju tek na zahtjev – pri promašajima.

Po dohvatu stranice, instrukcija koja je izazvala promašaj mora se *ponoviti*

- procesor mora imati pomoćne registre koji pohranjuju međurezultate, a koji se mogu odbaciti ako se dogodi promašaj
- npr. neka postoji instrukcija `DIV a,b,d,r` koja cjelobrojno dijeli `a` i `b`, kvocijent sprema u `d`, a ostatak u `r`; ako bi se promašaj dogodio pri spremanju ostatka `r` (u spremnik), niti `d` se ne smije pohraniti, već odbaciti – obzirom da će se nakon dohvata te stranice instrukcija ponoviti, sustav treba dovesti u stanje u kojem je bilo i prije prvog pokretanja instrukcije (npr. `d` je isto što i `a` i/ili `b`)

8.4.5. Usporenje rada procesa zbog straničenja

- pretpostavimo da se za pomoćni spremnik koristi tvrdi disk i da nam treba oko 10 ms za dohvat stranice s diska.
- promašaj će usporiti rad procesa, tj. odgoditi njegovo izvođenje za to vrijeme dok se stranica ne dohvati

Primjer 8.4. Usporenje procesa

Pretpostavimo da sabirnički ciklus traje $T_B = 10$ ns, te da dohvat stranice s diska traje $T_D = 10$ ms. Ako na svakih N instrukcija (sabirničkih ciklusa) imamo jedan promašaj, koliko će se proces usporiti (u postocima)?

Prosječno trajanje sabirničkog ciklusa može izraziti sa: $\overline{T}_B = \frac{(N-1) \cdot T_B + T_D}{N}$

Tablica 8.3. Usporenje rada procesa zbog straničenja

N	10^3	10^4	10^5	10^6	10^7
$\overline{T}_B$	10,01 μs	1,01 μs	110 ns	20 ns	11 ns
$\overline{T}_B/T_B$	1001	101	11	2	1,1

U stvarnosti je usporenje još i manje jer se pri promašaju uglavnom ne dohvaća samo jedna stranica već više njih.

Uz modernije računalo, uz T_B =1 ns i $T_D = 10$ μs rezultat $\overline{T}_B/T_B = 1,1$ ("prihvatljivo usporenje" od 10%) postiže se za $N = 10^6$.

8.4.6. Strategije zamjene stranica

Što ako u obradi promašaja nema praznih okvira kamo bi učitali stranicu?

- treba odabrati jedan okvir i isprazniti ga – KAKO?
- koji se okvir "isplati" isprazniti?
- iskoristiti svojstvo "prostorno-vremenske lokalnosti" procesa

  - sukcesivni zahtjevi procesa prema spremniku su većinom za lokacije bliske prethodnim zahtjevima
    * instrukcije koje se izvode su blizu jedna drugoj, jedna iza druge
    * podaci nad kojima instrukcije nešto rade su većinom također blizu jedni drugima
    * stog je kompaktan
- načini korištenja tog svojstva:
  - korištenje zastavica A i D iz opisnika stranice
    * zastavica A – "nedavno" korištene stranice (njih pokušaj ostaviti)
    * zastavica D – "čiste" i "nečiste" stranice, trošak njihove zamjene nije isti
  - teorijske strategije: FIFO, LRU, LFU, OPT
  - satni algoritam (ono što se koristi)

Teorijske strategije zamjene stranica

FIFO – First-In-First-Out

- izbaciti najstariju stranicu – onu koja je najduže u radnom spremniku

LRU – Least-Recently-Used

- izbaciti stranicu koja se najdulje nije koristila
- jedina koja je donekle ostvariva i nudi najveću učinkovitost (ne računajući OPT)

LFU – Least-Frequently-Used

- izbaciti stranicu koja se najmanje puta koristila

OPT – optimalna strategija

- izbaciti stranicu koja se najduže neće koristiti (u budućnosti)
- nije ostvariva, ali može služiti za usporedbu

Navedene strategije su presložene za praktično ostvarenje.

Satni algoritam (engl. *clock algorithm, second chance algorithm*)

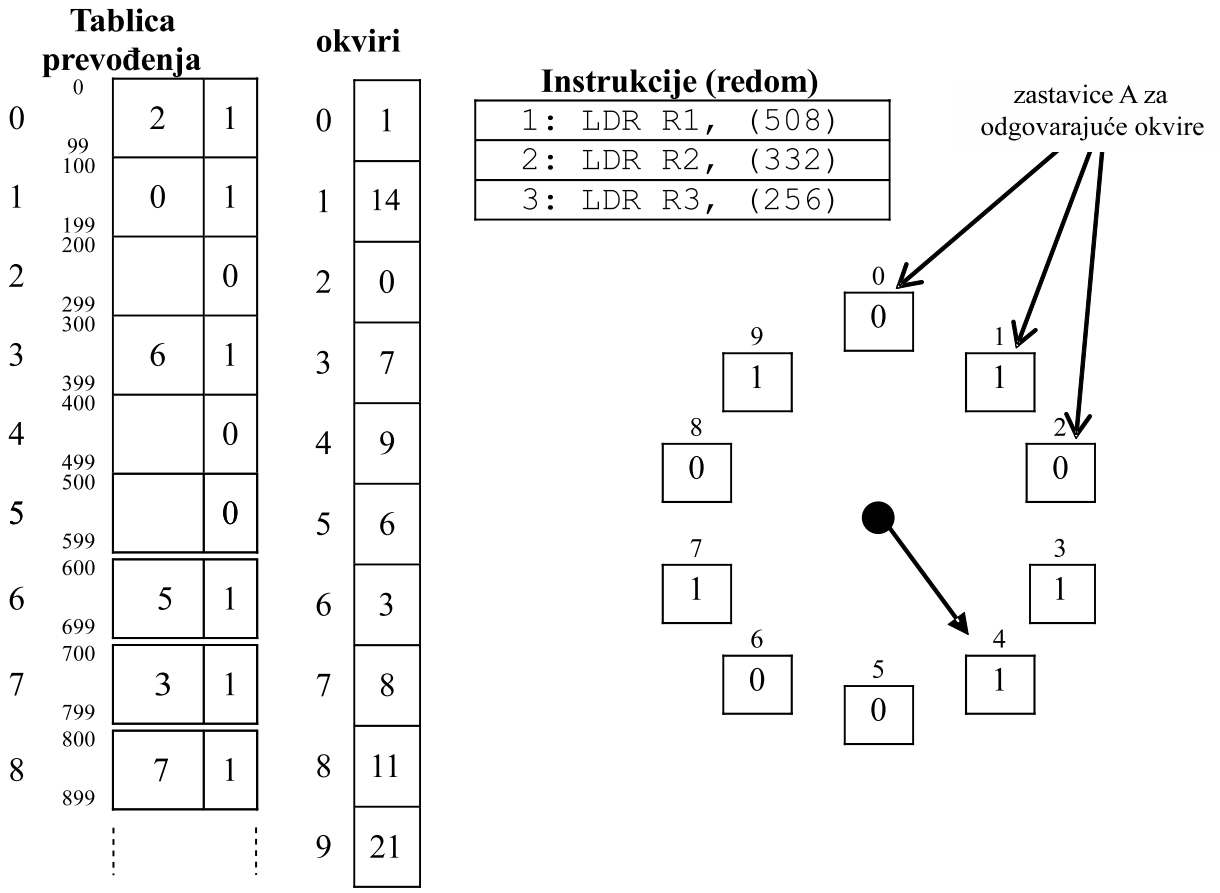
- inačica algoritma LRU
- nije samo teoretska – koristi se (UNIX, Windows) (uz neke prilagodbe)
- ideja algoritma:
  - koristi se zastavica A iz opisnika stranica kao oznaka da se stranica koristi
  - pri svakom korištenju – pri dohvatu ili pohrani u tu stranicu, sklop za prevođenje adresa automatski postavlja tu zastavicu u 1 (u opisniku te stranice)
  - jednom kazaljkom kružno obilazimo te opisnike, brišemo jedinice ili izbacujemo stranice
- kratki opis rada algoritma (poziva se samo kad treba prazan okvir a svi su puni):
  1. ako je A==0 u opisniku stranice na koju pokazuje kazaljka
     - stranica se nije koristila od kada je kazaljka zadnji puta ovuda prošla
     - izbaci tu stranicu iz njenog okvira i u njega učitaj potrebnu stranicu

– (kazaljka se ne pomiče – do idućeg poziva algoritma, vidi 2.)

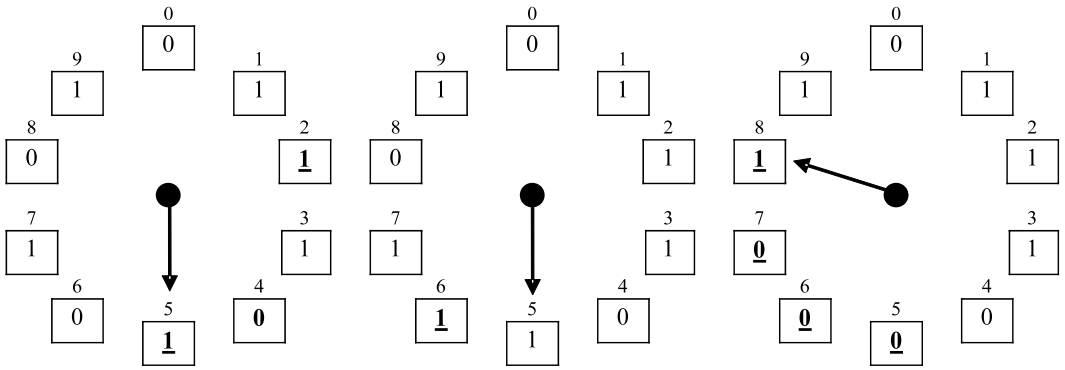
2. ako je A==1 u opisniku stranice na koju pokazuje kazaljka ima
   – stranica se koristila bar jednom od kada je kazaljka zadnji puta ovuda prošla
   – postavi A=0 u tom opisniku
   – pomakni kazaljku na idući opisnik
   – ponovi postupak (od koraka 1.)

Zadatak 8.2. Satni algoritam

Za neki sustav koji koristi straničenje i metodu satnog algoritma, zadana je tablica prevođenja za jedan proces, stanje okvira sustava, stanje kazaljke i zastavica A. Ako zadani proces treba izvesti zadane tri instrukcije (koje se nalaze u stranici 0), kako će se sustav mijenjati?



Rješenje:



a) nakon 1. instr. b) nakon 2. instr. c) nakon 3. instr.

Objašnjenje: Dohvat prve instrukcije izazvat će pogodak (0. stranica je u 2. okviru). Izvođenjem prve instrukcije dogodit će se prekid zbog promašaja podataka (traži se stranica 5 procesa). Zastavica četvrtog okvira A(4) postavit će se u 0 te će se kazaljka pomaknuti. Zastavica A(5)

je 0 te će se taj okvir osloboditi i u njega staviti 5. stranica procesa. Tada se može obaviti prva instrukcija. Njenim izvođenjem (čitanjem iz 5. okvira) postavlja se zastavica A(5) u 1 (sl. a)).

Druga instrukcija traži 3. stranicu koja se nalazi u okviru 6. Njenim izvođenjem (čitanjem podatka iz 6. okvira) postavit će se zastavica A(6) u 1 (sl. b)) (kazaljka se ne miče).

Treća instrukcija traži podatak iz 2. stranice koja nije u radnom spremniku, pa će se kazaljka pomaknuti, najprije na 6. mjesto, pa na 7. (pritom postavlja A(5), A(6) i A(7) u nulu) i tek na 8. pronalazi A(8)=0, izbacuje stranicu koja se tu nalazi i učitava stranicu 2 procesa. Nakon toga može se izvesti instrukcija 3. Izvođenjem 3. instrukcije postavlja se zastavica A(8) u 1 (sl. c)).

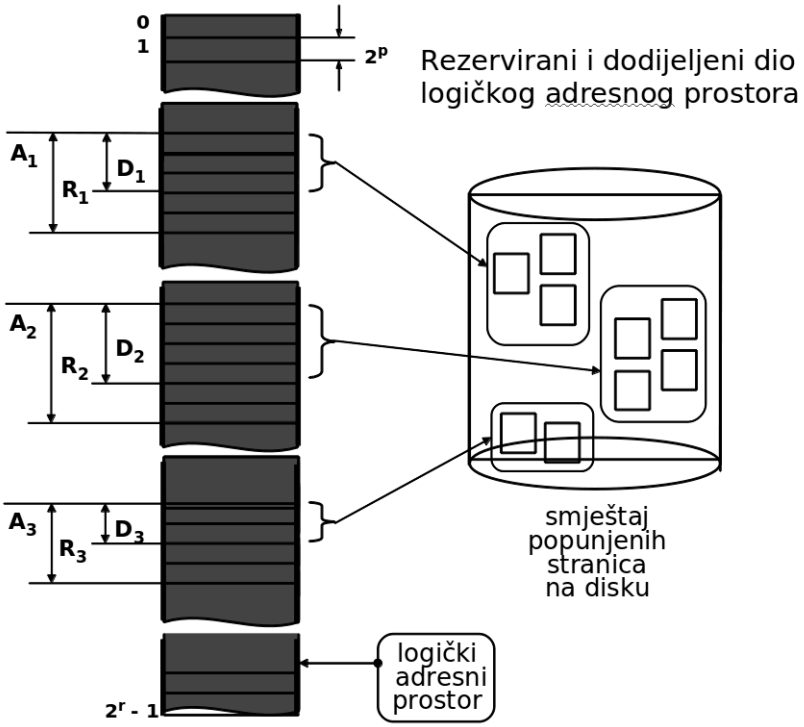
Zadatak 8.3. Teorijske strategije

U sustavu sa straničenjem proces veličine 400 riječi (1-400) generira slijed adresa: 23, 47, 333, 81, 105, 1, 400, 157, 30, 209, 289, 149, 360. Proces ima na raspolaganju 200 riječi radnog spremnika. Napisati niz referenciranja stranica veličine 50 riječi. Koliki je postotak promašaja za sve četiri navedene strategije izbacivanja stranica? Prikazati trenutni izgled tablice prevođenja na kraju primjene LFU strategije.

8.4.7. Rezervirani i dodijeljeni dijelovi procesa

Opisnik spremničkog prostora procesa – *informacijski blok* sastoji se:

- tablice prevođenja
- opisa gdje je proces smješten na pomoćnom spremniku
- dodatnog opisa (slika 8.18.):
  - koji su dijelovi procesa dodijeljeni – D (i opisani tablicom prevođenja)
  - koji su dijelovi procesa rezervirani – A (a još nisu posve opisani i tablicom prevođenja)



Slika 8.18. Rezervirani i dodijeljeni dijelovi adresnog prostora procesa

Navedeni opisi rezerviranog i dodijeljenog prostora omogućavaju:

- stvaranje potrebnih stranica kada one postanu neophodne (uz popunjavanje tablice prevođenja)
- detekciju greške (engl. *segmentation fault*) te prekid izvođenja procesa
  - primjerice, ako je tražena adresa (u nekoj instrukciji) izazvala prekid zbog toga što stranica nije u radnom spremniku ili nije čak ni opisana u tablici prevođenja, pregledom navedene strukture podataka može se ustanoviti je li tražena adresa unutar rezerviranog prostora procesa ili nije
    * ako jest, onda se stvara takva stranica i opisuje u tablici prevođenja
    * ako nije, prekida se proces jer je izazvao kritičnu grešku (kako se oporaviti od nje?)

8.4.8. Upravljanje okvirima

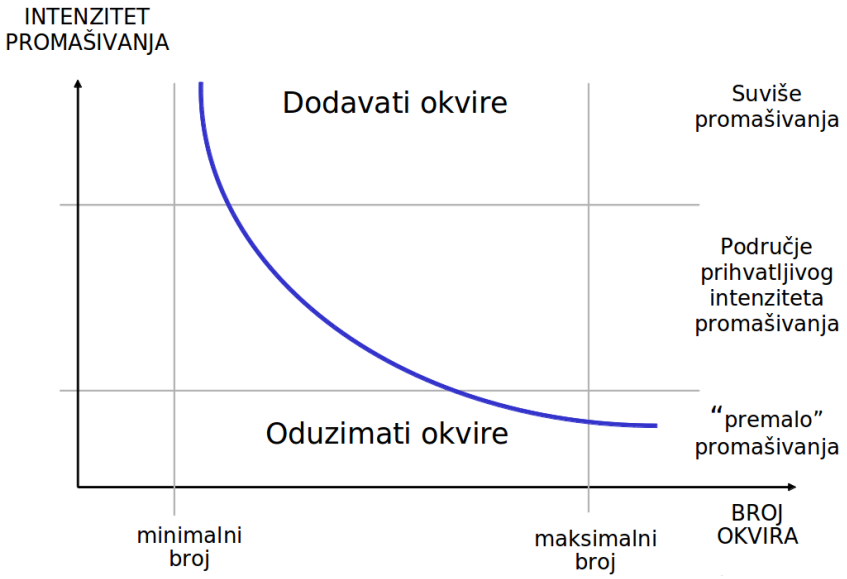
OS mora voditi evidenciju o svim okvirima

Stanja u kojima se okviri mogu naći:

- aktivno – sadrži stranicu koja se koristi
- slobodno – trenutno se ne koristi, ali ima neki sadržaj – potrebno ga je obrisati prije dodjele (osim ako se ne prepisuje drugom stranicom)
- slobodno s obrisanim sadržajem – spreman za dodjelu
- neispravno – greška u tom djelu spremnika te se oni ne koriste

Dodjela okvira

- dio okvira koristi OS za svoje potrebe (strukture podataka i međuspremnike)
- koliko okvira dati procesima? opcije:
  - koliko traže (možda, ako ima toliko slobodnih okvira)
  - fiksni broj (npr. svima isto)
  - prema postotku promašaja: (slika 8.19.)



Slika 8.19. Dodjela okvira prema intenzitetu promašaja

Skup radnih stranica

Radi povećanja učinkovitosti straničenja razni sustavi koriste dodatne postupke.

Primjerice, na Windows temeljenim sustavima koristi se postupak temeljen na skupu radnih stranica (engl. *working set*).

Osnovna ideja postupka jest da se procesu definira potrebna veličina skupa radnih stranica s kojom će on imati dovoljno dobru učinkovitost, tj. zadovoljavajući broj promašaja. Ako mu se takav skup stranica (okvira za te stranice) ne može osigurati onda njega u tom trenutku treba cijelog maknuti na pomoćni spremnik a da bi ostali procesi mogli koristiti svoje skupove radnih stranica (u radnom spremniku). U protivnom, zbog malog broja stranica tog procesa u radnom spremniku imali bi preveliki broj promašaja (slično može vrijediti i za druge procese), a time bi i sustav bio manje učinkovit. Proces koji je u jednom trenutku maknut na pomoćni spremnik, kasnije će se učitati u radni sa cijelim skupom radnih stranica, a neki drugi će se pohraniti na pomoćni spremnik da napravi mjesta.

Ako se ne koristi nekakav algoritam upravljanja sličan opisanom, sustav može cijelo vrijeme intenzivno raditi s diskom zbog učestalih promašaja (engl. *thrashing*) i značajno smanjiti učinkovitost sustava.

Sučelje OS-a za upravljanje straničenjem

OS ima i sučelje za procese koji preko njega mogu utjecati na straničenje

- proces može zatražiti da neke stranice (dijelovi procesa) ostanu u radnom spremniku
- uglavnom su to samo preporuke koje OS ne mora poštovati
- primjeri (POSIX, Win):

```
int mlock(const void * addr, size_t len);
int mlockall(int flags);
bool VirtualLock(void *address, void *size);
bool SetProcessWorkingSetSize(void *process, size_t min, size_t max);
```

8.4.9. Prostorno-vremenska lokalnost

- koristiti podatke slijedno, ne "šarati" po spremniku
- smanjuje se broj promašaja
- povećava iskoristivost priručnih spremnika
- dobitak na učinkovitosti može biti vrlo velik (za nekoliko reda veličine!)

Primjer 8.5. Inicijalizacija velike matrice

Incijalizacija matrice $A[N][N]$ po recima ili stupcima? Neka jedan redak matrice stane u jednu stranicu i neka na raspolaganju stoji samo jedan okvir za podatke matrice.

```
za i = 1 do N radi
{
    za j = 1 do N radi
    {
        A[i][j] = 0; ili A[j][i] = 0;   !!!
    }
}
```

Inicijalizacija po retcima = N promašaja – nakon promašaja zbog dohvata prvog elementa $A[i][0] = 0$ svi ostali upisi su pogodci ($A[i][j > 0] = 0$). Ovo je ujedno i minimalan broj promašaja – veći broj okvira ne bi pomogao, matricu u konačnici treba dohvatiti (postaviti svugdje nule), a ona je velika N stranica.

Inicijalizacija po stupcima = N^2 promašaja – svaki susjedni zahtjev je za različiti redak što je ujedno i različita stranica, svaki stupac generira N promašaja. Broj promašaja se ovdje ne bi smanjio ni kada bismo imali više okvira za matricu (osim kada je broj okvira blizu N ili veći). Naime, nakon što sve okvire iskoristimo za nekoliko redaka, morali bi jedan okvir isprazniti za idući redak. Ali taj redak nam treba kasnije i trebat će ga ponovno dohvaćati – svaki se redak dohvaća N puta!

Zadatak 8.4.

U sustavu s virtualnim spremnikom, veličina okvira je N riječi, a okviri se pune na zahtjev. Algoritam zamjene stranica je LRU. Poredak $A[N, N]$ je pohranjen po retcima (na susjednim lokacijama se mijenja desni indeks). Koliko promašaja će izazvati prikazani proces ako za poredak A u radnom spremniku postoji a) samo jedan okvir; b) dva okvira; c) tri okvira; d) N okvira.

```
t = 0;
za i=1 do N-1 {
    za j=i+1 do N {
        t = t + A[i,j];
        t = t * A[j,i];
    }
}
```

Napomena: Zanemariti promašaje zbog dohvata instrukcija samog procesa i pristupa pomoćnim varijablama. (Na primjer, neka je cijeli program u priručnom spremniku za instrukcije, a pomoćne varijable i, j, t u registrima.)

Zadatak 8.5. Upravljanje spremnikom

U nekom sustavu trebaju se obaviti četiri procesa: P1, P2, P3 i P4 koji su već pripremljeni na pomoćnom spremniku i zauzimaju redom 5 MB, 8 MB, 3 MB, 10 MB. Događaji pokretanja i završetka procesa znani su unaprijed i mogu se iskazati sljedećim nizom događaja: P1 pokrenut; P2 pokrenut; P1 završava; P3 pokrenut; P4 pokrenut; P3 završava; P2 završava; P4 završava. Sustav na raspolaganju ima 20 MB spremnika rezervirana za korisničke procese. Prikazati stanje radnog spremnika ako se koriste metode upravljanja spremnikom:

a) statičko upravljanje s veličinom segmenta od 10 MB

b) dinamičko upravljanje

c) straničenje, uz veličinu stranice od 1 MB.

Zadatak 8.6. Upravljanje spremnikom

U nekom jednoprocesorskom sustavu se javljaju četiri zadatka P1 u 10., P2 u 20., P3 u 30., i P4 u 40. jedinici vremena. Zadaci se raspoređuju prema prioritetu. Prioritet zadatka jednak je broju (P4 ima najveći prioritet). P1 treba 10 MB, P2 20, P3 30 te P4 40 MB. Pohrana jednog procesa (neovisno o veličini) neka traje 5 jedinica vremena. Isto toliko traje i učitavanje jednog procesa. Pretpostaviti da sustav ne paralelizira učitavanje/spremanje procesa (najviše jedan

proces se učitava s pomoćnog spremnika ili sprema na njega). Procesi se već pripremljeni na pomoćnom spremniku.

a) Pokazati rad sustava, ako se koristi statičko upravljanje spremnikom s dvije particije od 20 i 40 MB (P1 i P2 za manju particiju, P3 i P4 za veću).

b) Pokazati rad sustava, ako se koristi dinamičko upravljanje spremnikom sa spremnikom od 60 MB.

Zadatak 8.7. Dinamičko upravljanje spremnikom

U sustavu koji koristi dinamičko upravljanje spremnikom ukupna kapaciteta 20 MB pojavljuju se sljedeći zahtjevi/događaji: pokretanje procesa P1 (koji treba 5 MB), pokretanje procesa P2 (12 MB), blokiranje procesa P1 (npr. na UI napravi), pokretanje procesa P3 (7 MB), pokretanje procesa P4 (4 MB), završetak P2, odblokiranje P1, završetak P3, završetak P1, završetak P4. Pretpostaviti da su procesi već pripremljeni na pomoćnom spremniku i da zahtjevi koji se ne mogu ostvariti u trenutku pojave čekaju i ostvaruju se kada to bude moguće (ne odbacuju se). Pokazati stanje spremnika (grafički) nakon SVAKOG događaja i SVAKE promjene u spremniku.

Zadatak 8.8. Straničenje (algoritam 1)

Zadani algoritam (množenja) koristi kvadratne matrice dimenzija NxN i izvodi se u sustavu koji koristi straničenje s veličinom stranice od N riječi (redak matrice stane u stranicu, svaka matrica treba N stranica). Pretpostaviti da se koristi optimalna strategija zamjene stranica, te da za podatke procesa (elemente svih matrica) na raspolaganju stoje tri okvira. (Dohvat instrukcija i lokalnih varijabli (i,j,k) neće izazivati promašaje.) Koliko će promašaja izazvati prikazani dio procesa, ako:

a) (1) N=2

b) (1) N=3

c) (1) općenito (za N)

```
za i = 1 od N
  za k = 1 od N
    za j = 1 od N
      C[i,j] += A[i,k] * B[k,j];
```

Zadatak 8.9. Straničenje (algoritam 2)

Koliko će promašaja izazvati prikazani algoritam, ako za matricu A u radnom spremniku postoji a) samo jedan okvir, b) 2 okvira? Veličina okvira jest 2N riječi, a algoritam zamjene stranica je OPT.

```
za i = 1 do N
  za j = 1 do N
     A[i+1,j] = A[i,j] + A[i+1,j]
```

Zadatak 8.10. Straničenje (algoritam 3)

Koliko će promašaja izazvati prikazani algoritam, ako za matricu A u radnom spremniku postoje 4 okvira? Veličina okvira jest N riječi, a algoritam zamjene stranica je OPT.

```
za i = 1 do 2N-1
  za j = 1 do 2N-1
     x = A[i,j]
     A[i,2N] = A[i,2N] + x
     A[2N,j] = A[2N,j] + x
```

8.4.10. Zaključne napomene o straničenju

Prednosti:

- nema fragmentacije
- zaštita jezgre i procesa
- pokretanje i velikih procesa – učitavaju se samo trenutno potrebne stranice
- podržano sklopovljem i operacijskim sustavom
- transparentno za program (ali dobar program može biti učinkovitiji)
- ostvarenje dijeljenog spremnika između procesa – tablice prevođenja oba procesa pokazuju na iste stranice
- dupliciranje procesa (fork) – nije potrebno fizički kopirati dijelove koji se samo čitaju

Nedostaci:

- potreban je složeni sklop
- moguće usporenje zbog promašaja
  - promašaj može odgoditi izvođenje procesa što u nekim okruženjima može izazvati nedopušteno kašnjenje (npr. u slanju upravljačkih naredbi)

Što arhitekt/programer treba/može napraviti?

- teoretski ništa – upravljanje je transparentno – potpuno rješeno sklopovljem i OS-om
- međutim, korištenjem načela "prostorno-vremenske lokalnosti" smanjuje se broj promašaja i povećava učinkovitost
  - vrijedi općenito, ne samo radi straničenja
  - putevi podataka u računalu
    - [disk] $\Leftrightarrow$ [radni spremnik] $\Leftrightarrow$ [procesor]
    - [disk]: medij (magnetske ploče i sl.) $\Leftrightarrow$ međuspremnik diska
    - [procesor]: priručni spremnik procesora (L3 $\Leftrightarrow$ L2 $\Leftrightarrow$ L1) $\Leftrightarrow$ registri procesora
- korištenje u sustavima za rad u stvarnom vremenu (RT)
  - zaključati stranice kritičnih procesa u radni spremnik (preko sučelja OS-a)
    - `mlock/mlockall`, `VirtualLock`, `SetProcessWorkingSetSize`

O korištenju pomoćnog spremnika

- u prikazanome modelu OS pri pokretanju programa priprema proces najprije na pomoćnom spremniku, a onda ga učitava u radni
- u stvarnim sustavima se pomoćni spremnik koristi tek po potrebi

  - na pomoćnom spremniku se ne mora nalaziti proces
  - na pomoćnom spremniku može biti i samo dio procesa (po potrebi)

Dijeljene biblioteke

- za obavljanje zadanog posla mnogi programi koriste već pripremljene dijelove koda za pojedine operacije = operacije iz pojedinih *biblioteka*
- biblioteke se ne moraju ugrađivati u datoteku s programom ako su one prisutne (instalirane) na operacijskom sustavu (engl. *shared libraries*)
- dijelovi biblioteka se mogu dinamički učitavati pri pokretanju programa
- dijelovi biblioteka koji sadrže samo kod (instrukcije) mogu se učitati u stranice koje se dijele među procesima koji ih koriste – nije potrebno da se za svaki proces ponovno učitaju ti dijelovi
- primjeri:
  - .dll datoteke na Windows sustavima (engl. *dynamic-link library*)
  - .so datoteke na UUNIX sustavima (engl. *dynamically linked shared object libraries*)

8.5. Detaljnije o dinamičkom upravljanju spremnikom (info)

- U poglavlju 8.3. je prikazana metoda dinamičkog upravljanja spremnikom, ali na razini upravljanja procesima i to samo idejno, bez naznake potrebnih struktura podataka i načina njihovog korištenja.
- U ovom odjeljku se spominju i neki drugi detalj iz područja dinamičkog upravljanja spremnikom.

Potreba za metodama dinamičkog upravljanja spremnikom:

1. na razini jezgre
   a) za procese (njihove adresne prostore) ako se koristi dinamičko upravljanje
   b) za upravljanje okvirima ako se koristi straničenje
   c) za opisnike svih elemenata jezgre (dretvi, procesa, semafora, . . .)
   d) za međuspremnike (za naprave i druge potrebe)
2. na razini procesa
   a) pri radu procesa dinamički se javljaju zahtjevi za dodatnim spremničkim prostorom (za pohranu novih struktura podataka)

- Iako su gornje potrebe za dinamičkim dodjeljivanjem spremnika vrlo različiti (po veličini zahtjeva, po učestalosti, . . .), slični algoritmi će zadvoljiti njihove potrebe.

Svojstva algoritama za dinamičko upravljanje spremnikom

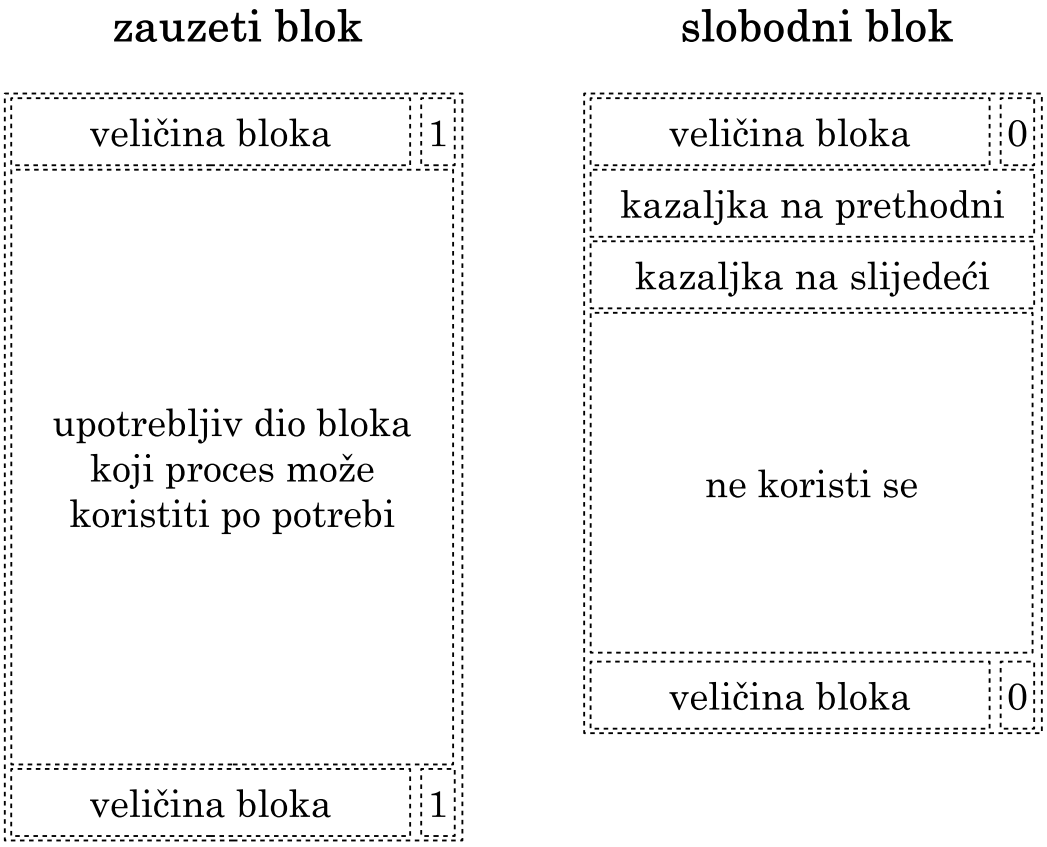
- Što sve treba uzeti u obzir pri analizi algoritama?
- Mnogo toga ovisi o primjeni – koje okruženje se razmatra:
  – je li najbitnija (prosječna) učinkovitost s obzirom na vrijeme izvođenja
  – je li najbitnija (prosječna) učinkovitost s obzirom na korištenje spremničkog prostora (za sustave s ograničenim spremničkim prostorom)
  – je li najbitnija vremenska određenost – da se može odrediti koliko će operacije trajati (da se algoritam može koristiti i u sustavima za rad u stvarnom vremenu)
- Različiti kriteriji uvjetuju i različite odabire
  – Primjerice, slobodne blokove možemo imati posložene od manjih prema većim radi smanjenja utjecaja fragmentacije i boljeg iskorištenja spremničkog prostora.
    * Međutim, slaganje slobodnih blokova (u listi) zahtijeva dodatno procesorsko vrijeme što utječe na smanjenu učinkovitost (proces se duže izvodi).

8.5.1. Jednostavni algoritmi za dinamičko upravljanje spremnikom

- nazovimo dodijeljene dijelove spremnika *blokovima*
- neka se za upravljanje spremnikom koriste *liste*
  – postoji barem jedna lista sa slobodnim blokovima
  – dodijeljeni (zauzeti) blokovi mogu biti u (zasebnoj) listi, ali nije neophodno
- kazaljke na početne blokove liste neka su "globalne" varijable
- na početku i kraju blokova nalazi se zaglavlje i podnožje koje se koristi za ostvarenje liste i

druge potrebe dinamičkog upravljanja spremnikom

- koristan prostor u bloku je između zaglavla i podnožja
- zaglavlje i podnožje slobodnih i zauzetih blokova može se razlikovati
  - primjerice, ako zauzeti blokovi nisu u listi, zaglavlje i podnožje može se sastojati samo od veličine bloka i oznake zauzetosti, dok slobodni blokovi dodatno trebaju imati kazaljke na prethodni i sljedeći blok (prema slici 8.20.)



Slika 8.20. Primjer zaglavlja zauzetih i slobodnih blokova

Korištenjem liste slobodnih blokova (npr. prema prethodno opisanoj strukturi podataka) mogu se izraditi sljedeći jednostavni algoritmi:

1. dodjeljivanje prema redu prispijeća (engl. *first-in-first-out – FIFO*)
2. dodjeljivanje prema obrnutu redu prispijeća (engl. *last-in-first-out – LIFO*)
3. dodjeljivanje metodom najbolji-odgovarajući (engl. *best-fit*)
4. dodjeljivanje metodom grupiranja sličnih blokova (engl. *good-fit*)

Dodjeljivanje prema redu prispijeća

- Načelo rada:
  1. pri zahtjevu za novi blok:
     a) lista se pretražuje od početka
     b) prvi blok koji je dovoljno velik dodjeljuje se u cijelosti ili dijeli na dva dijela: jedan dio se dodjeljuje zahtjevu a drugi se vraća u listu slobodnih blokova
  2. pri oslobađanju bloka
     a) oslobođeni blok se prvo pokušava spojiti sa susjednim slobodnim blokovima u sprem-

niku (ako takvi postoje oni se najprije moraju maknuti iz liste slobodnih blokova)

b) potom se blok (izvorni ili spojeni) stavlja na kraj liste slobodnih blokova

- Svojstva:
  - − povećana fragmentacija
  - + složenost oslobađanja je O(1) (blok ide na kraj liste)
  - − složenost pretrage O(n), gdje je n broj elemenata u listi
    - * u najgorem slučaju treba doći do zadnjeg elementa liste

Dodjeljivanje prema obrnutom redu prispijeća

- Načelo rada:
  - – Jedina razlika od prethodnog dodjeljivanja je da se pri oslobađanju bloka on stavlja na početak liste (ne na kraj).
- Svojstva:
  - − povećana fragmentacija
  - + složenost oslobađanja je O(1) (blok ide na početak liste)
  - − složenost pretrage O(n), gdje je n broj elemenata u listi
    - * u najgorem slučaju treba doći do zadnjeg elementa liste
  - – Moguća bolja svojstva zbog korištenja priručnog spremnika
    - * za očekivati je da se oslobođeni blok nedavno koristio te je još uvijek u priručnom spremniku procesora – njegovo korištenje bit će brže naspram nekog drugog bloka koji se već neko vrijeme nije koristio

Dodjeljivanje metodom najbolji-odgovarajući

- Načelo rada:
  - – Razlika od prethodnih dodjeljivanja je u korištenju uređene liste slobodnih blokova – pri oslobađanju bloka on se stavlja u uređenu listu na mjesto koje odgovara njegovoj veličini (na početku je najmanji slobodni blok).
- Svojstva:
  - + minimalna fragmentacija (bolje ne može bez preslagivanja)
  - − složenost oslobađanja je O(n)
  - − složenost pretrage O(n)

Dodjeljivanje metodom grupiranja sličnih blokova

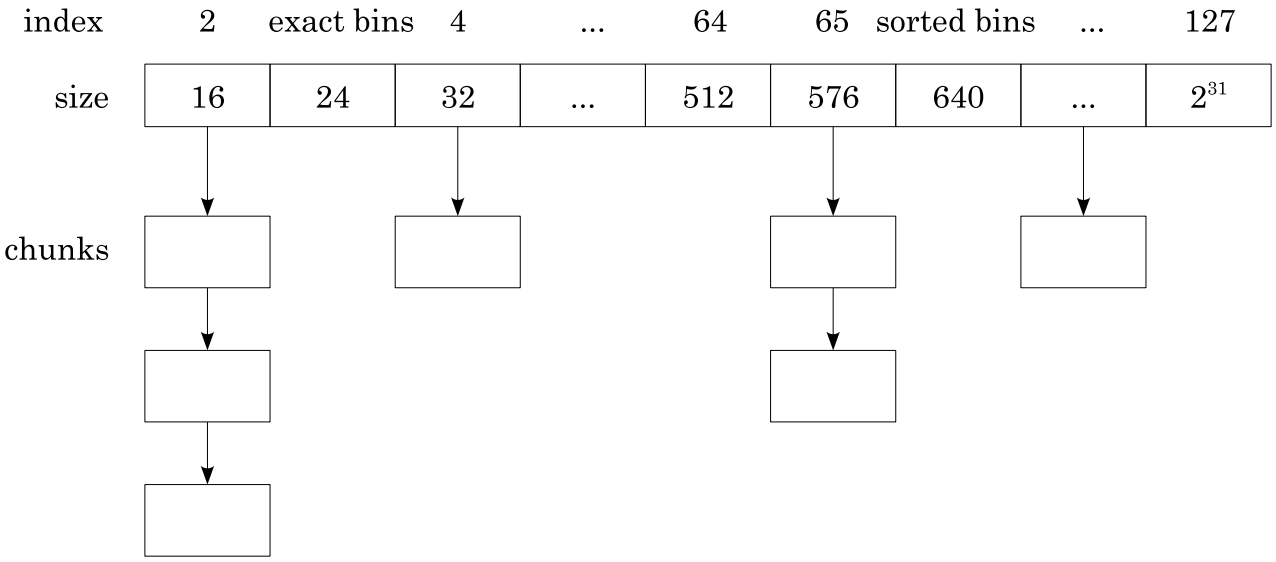
- Načelo rada:
  - – Za slobodne blokove postoji nekoliko lista, ovisno o veličini
    - * primjerice po jedna za blokove veličina:
      - · 1 B – 128 B
      - · 129 B – 256 B
      - · 257 B – 1 kB

. . .

- Unutar jedne liste blokovi su složeni po redu prispijeća
- Pri zauzimanju najprije se pretražuje lista koja bi mogla imati blokove odgovarajućih veličina, a tek ako tamo nema takvog bloka kreće se s idućom listom koja sadrži veće blokove.
- Pri oslobađanju te nakon spajanja bloka sa susjednim praznim on se stavlja u listu koja sadrži takve blokove

- Svojstva:
  - ± smanjena fragmentacija
  - \+ složenost oslobađanja je O(1)
  - ± složenost pretrage O(m) (m < n)
  - \+ u prosjeku, dobro posloženo ovakvo upravljanje može imati značajno bolja svojstva od prethodno navedenih, a utjecaj fragmentacije može ostati neznatno veći od metode "najbolji odgovarajući"

Najpoznatija metoda koja spada u kategoriju dodjeljivanje metodom grupiranja sličnih blokova jest Doug Lee malloc koja se jedno vrijeme koristila na sustavima s jezgrom Linuxa. Postoje neke razlike između gornjeg opisanog postupka i Doug Lee metode. Kod njega neke liste su uređene a neke nisu (one s manjim blokovima).



Slika 8.21. Liste slobodnih blokova kod *Doug Lee malloc* metode

Dodjeljivanje metodom grupiranja sličnih blokova može se promijeniti na način da su veličine blokova po listama određene algoritmom te da se koriste bit-maske za bržu pretragu nepraznih listi. Ako se pritom uvijek traži u listama čiji najmanji blokovi zadovoljavaju zahtjeve, može se postići složenost O(1) što omogućuje njihovu primjenu u sustavima za rad u stvanom vremenu. O(1) u ovom slučaju ne znači da je ovaj algoritam brži od drugih (prosječno) već da njegovo vrijeme rada ne ovisi o broju slobodnih blokova. Najčešće je njegovo vrijeme rada i veće od prethodno opisanih metoda, kada se gledaju prosječna vremena tih metoda. Primjer takvog algoritma jest TLSF (engl. *two level segregated fit*).

Pitanja za vježbu 8

1. Kada, iz kojih razloga, procesor pristupa spremniku?
2. Koliko adresnog prostora može adresirati sustav koji koristi 36-bitovnu adresnu sabirnicu?
3. Od čega se sastoji *virtualni spremnički prostor* koji koristi operacijski sustav?
4. Navesti dobra i loša svojstva algoritama:
   - statičkog upravljanja spremnikom
   - dinamičkog upravljanja spremnikom
   - upravljanje spremnikom straničenjem.
5. Što su to fizičke a što logičke adrese?
6. Kod kojih algoritama upravljanja spremnikom je proces u fizičkim a kod kojih u logičkim adresama?
7. Objasniti pojmove: unutarnja i vanjska fragmentacija.
8. Koji se postupci koriste kod dinamičkog upravljanja spremnikom radi smanjenja fragmentacije?
9. Izvesti i objasniti Knuthovo 50% pravilo.
10. Nacrtati sklop koji se koristi za pretvorbu adresa kod dinamičkog upravljanja spremnikom. Koja su proširenja tog sklopa potrebna da bi se dodala i zaštita?
11. Objasniti pojmove: stranica, okvir, tablica prevođenja u kontekstu straničenja.
12. Čemu služi i od čega se sastoji tablica prevođenja?
13. Zašto se koristi hijerarhijska organizacija tablice prevođenja?
14. Čemu služe zastavice V (bit prisutnosti), A (oznaka korištenja) te D (oznaka izmjene) u opisniku stranice?
15. Što je to “promašaj” u kontekstu straničenja?
16. Opisati upravljanje spremnikom metodom “straničenje na zahtjev”.
17. Opisati strategije zamjene stranica: FIFO, LRU, LFU, OPT i satni algoritam.
18. Opisati mogućnosti upravljanja okvirima (načini dodjele).
19. Što je to “prostorno vremenska lokalnost” i kako ona utječe na učinkovitost sustava?
20. Kakva sučelja operacijski sustavi nude programima radi upravljanja straničenjem?

9. DATOTEČNI SUSTAV

Datotečni sustavi (engl. *File Systems – FS*) su uglavnom ostvareni na diskovima, pa se prije razmatranja samih datotečnih sustava razmatraju diskovi.

Iako se koristi pojam *diskovi* u ovu kategoriju su uključeni i spremnici podataka koji nisu diskovi u stvarnosti, npr. SSD disk nije disk iako se tako zove. Osim na diskovima, datotečni sustavi su i na CD-u/DVD/*, USB ključiću, memorijskim karticama.

9.1. Diskovi

Uloga diska u računalnom sustavu:

- kao skladište za trajno spremanje podataka (i kada se računalo ugasi)
- kao pomoćni spremnik pri upravljanju spremnikom

9.1.1. Svojstva HDD i SSD diskova

- tvrdi disk – HDD (engl. *hard disk drive, hard disk, hard drive*)
- SSD disk – SSD (engl. *solid-state drive*)
- interno su potpuno različiti dok prema van (OS-u) daju isto (slično) sučelje
- u idućem poglavlju je detaljnije razmatran HDD obzirom na njegovu složenu građu
- u ovom su poglavlju samo ukratko navedena svojstva oba tipa diskova

Svojstva tvrdog diska – HDD

- diskovi su elektro-mehaničke naprave
- podaci su pohranjeni na magnetiziranim pločama (koje se vrte sa 5000-15000 okr/min)
- ručica s glavama se pomiče po dijagonali i čita/piše s pojedine ploče (površine)
- organizacija podataka:
  1. staza – podaci na jednoj površini jednako udaljeni od osi rotacije – mogu se pročitati/zapisati bez pomaka glave (samo s rotacijom ploče)
  2. sektor – staza se dijeli na manje jedinice – sektore, tipično velike 512 B (ili 4 KB za diskove za spremanje podataka, video nadzor i slično)
  3. cilindar – staze na različitim pločama jedako udaljene od osi rotacije – mogu se dohvatiti bez pomicanja glave, samo aktivacijom druge glave za drugu površinu

  – jedinica podataka je sektor

  – "adresa sektora": {broj površine (glave), broj staze na površini, broj sektora na stazi}

  – izvorno CHS *Cylinder-Head-Sector* (C=broj staze, H=broj površine)
- disk je spor zbog mehaničkih djelova
- ublažavanje sporoće – podatke kompaktno smjestiti, dok se ne pohrani sva datoteka koristiti redom:

1. uzastopne ili sve sektore iste staze
2. ostale staze istog cilindra (bez pomaka glave)
3. susjedne cilindre (što manji pomak glave)

– OS nastoji tako smjestiti datoteke, ali kad je disk dosta popunjen to više nije moguće i javlja se fragmentacija: različiti dijelovi datoteke su na različitim mjestima diska – čitanje/pisanje se tada usporava
– defragmentacija nastoji premjestiti sadržaje tako da budu kompaktnije smješteni
– elektronika diska "skriva" internu arhitekturu (broj površina, staza, sektora po stazi) i prema OS-u sve sektore prikazuje kao linearno polje sektora
– u takvom linearnom polju susjedni sektori jesu i susjedni na disku (osim iznimno kad se prelazi na stazu na drugom cilindru ili susjednu stazu, što je rijetko)

- OS poslužuje skupinu zahtjeva i optimira njihovo posluživanje radi manjeg pomaka glave
- uobičajena svojstva diskova:
  – kapaciteti: od nekoliko stotina GB do desetke TB (nekoliko TB je uobičajeno)
  – brzine čitanja/pisanja kompaktno smještenih podataka: 100-200 MB/s
  – čitanje/pisanje nasumičnog bloka: 5-15 ms !
  – priključak SATA (Serial ATA; ATA: AT Attachment; AT: Advanced Technology (u Ĩ986.))
  – komunikacija procesor – disk kontroler: AHCI *(Advanced Host Controller Interface)*
    * optimiran za rad s diskovima preko SATA ožičenja
    * half-duplex – prijenos samo u jednom smjeru u jednom trenutku
- OS, tj. datotečni sustavi obično rade s *blokom* podataka koji se sastoji od susjednih sektora (uobičajeni blok je 4 KB – osam uzastopnih sektora od 512 B)

Svojstva SSD-a

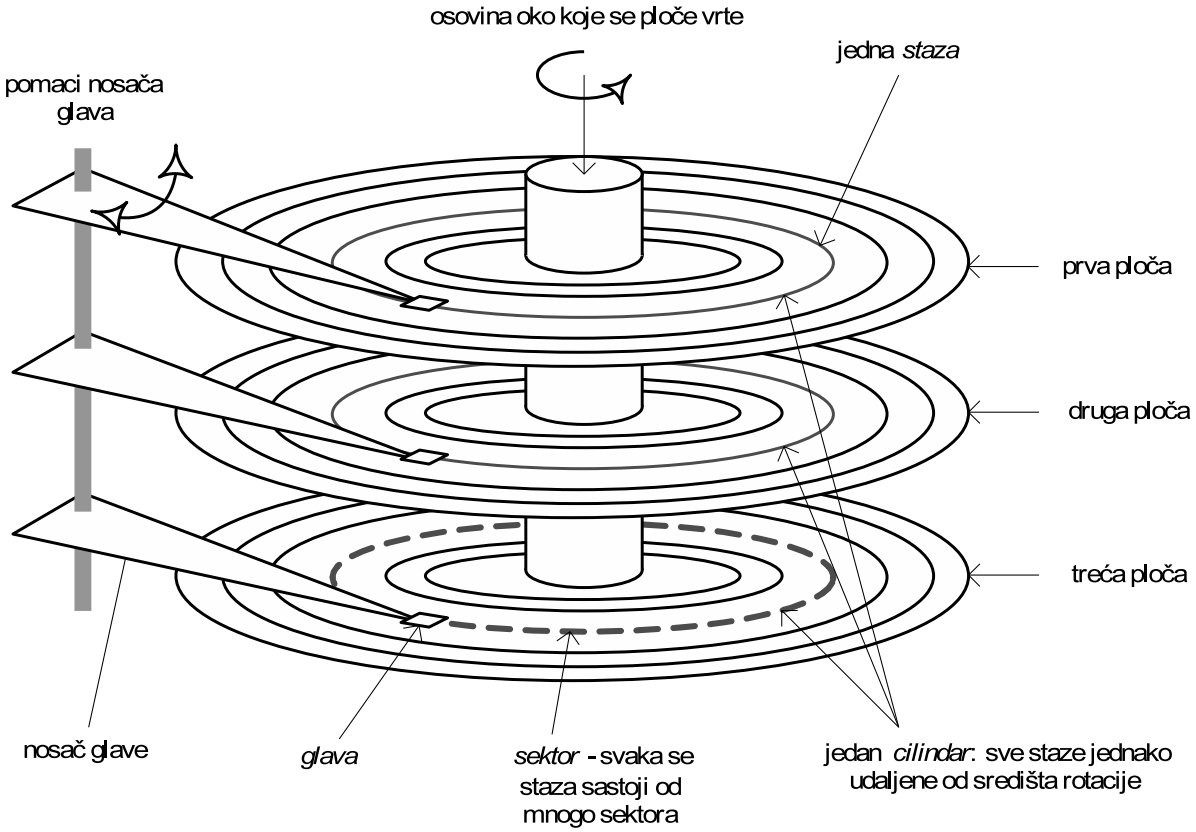
- poluvodički "diskovi" – koriste se poluvodičke ćelije za spremanje naboja
- ćelija: može ovisno o tehnologiji spremiti od 1 do 4 bita
  – 1-bit: SLC–*Single Level Cells* – najbolji ali i najskuplji (za poslužitelje)
  – 2-3 bita: MLC–*Multi Level Cells* – jefitinji (za obične korisnike)
  – 4-bita: QLC–*Quad-bit Cells* – jefitinji (za obične korisnike)
  – 3D XPoint: novija tehnologija – umjesto naboja "sprema" otpor (postavlja otpor)
- jedinica podataka je sektor/blok (kao i za HDD)
- adresiranje je linearno – nema površina, staza, sektora
- dohvat je efikasniji ako se radi u blokovima
  – dohvat jednog nasumičnog sektora može trajati i do 0,1 ms
  – dohvat susjednih sektora (već dohvaćenom) je puno brži
  – stoga se dohvaća blok, a ne pojedinačini sektor
  – npr. za dohvat nasumičnog bloka od 4 KB (8 sektora) trajanje je neznatno veće od dohvata jednog nasumičnog sektora

- pisanje zahtjeva dvije operacije: brisanje ćelije, pa tek onda pisanje
- pisanje nije sporije ako se radi u blokovima
- mješoviti zahtjevi čitanje/pisanje za nasumičnim blokovima znatno usporavaju rad
- uobičajena svojstva SSD-ova:
  - kapaciteti: od nekoliko stotina GB do nekoliko TB
  - brzine čitanja/pisanja kompaktno smještenih podataka:
    * preko SATA sučelja (AHCI): 200-550 MB/s
    * preko M.2 ili PCIe (NVMe): 1000 do 7000 MB/s
  - čitanje/pisanje nasumičnog bloka: 0,1 ms ! znatno sporije od gornjih brzina kad se čitaju/pišu nasumični mali dijelovi (<4KB)
  - komunikacija procesor – disk kontroler: NVMe *(Non-Volatile Memory Host Controller Interface Specification)*
    * optimiran za rad s SSD-ovima preko M.2 i PCIe
    * full-duplex – prijenos u oba smjera istovremeno
    * više redova za zahtjeve od AHCI-ja

9.1.2. Fizička svojstva tvrdih diskova (HDD)

HDD je elektromehanička naprava. Sastoji se od:

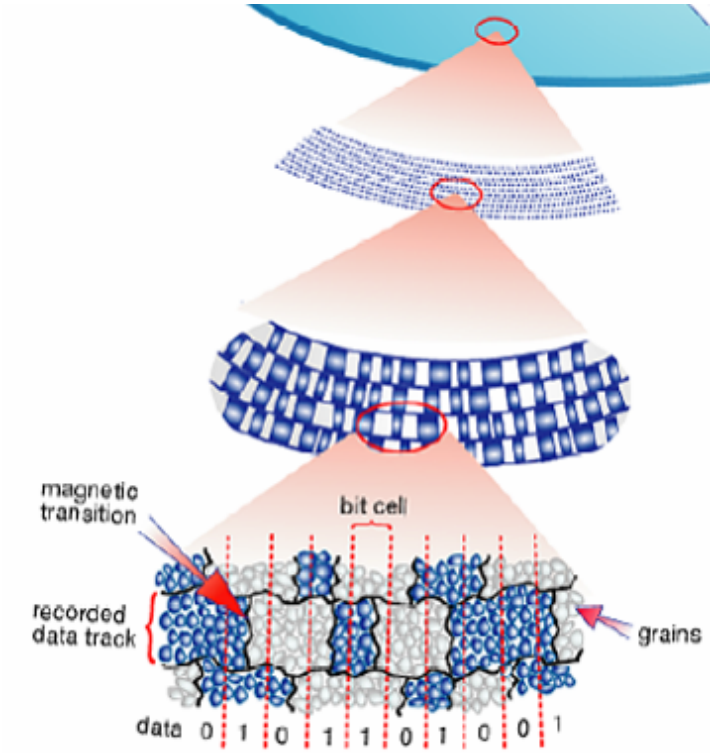
- pokretnih djelova:
  - magnetiziranih ploča
  - pokretnih glava
  - elektromotora koji pokreću ploče (vrte ih)
  - elektromotora koji pokreću glave (od ruba prema centru rotacije i obratno)
- nepokretnog dijela: upravljački sklop



Slika 9.1. Shematski prikaz mehaničkog dijela diska

Magnetska ploča pod povećalom (info)

Podaci se na magnetskim pločama pohranjuju korištenjem različite polarizacije vrlo malih površina ploča. Najjednostavniji pristup bi bio da je jedinica predstavljena jednim smjerom a nula drugim. Međutim, diskovi koriste različite načine kodiranja. Slika 9.2. prikazuje jedan takav primjer kod kojeg je jedinica predstavljena promjenom stanja.



Slika 9.2. Magnetski materijal pod povećalom[1]

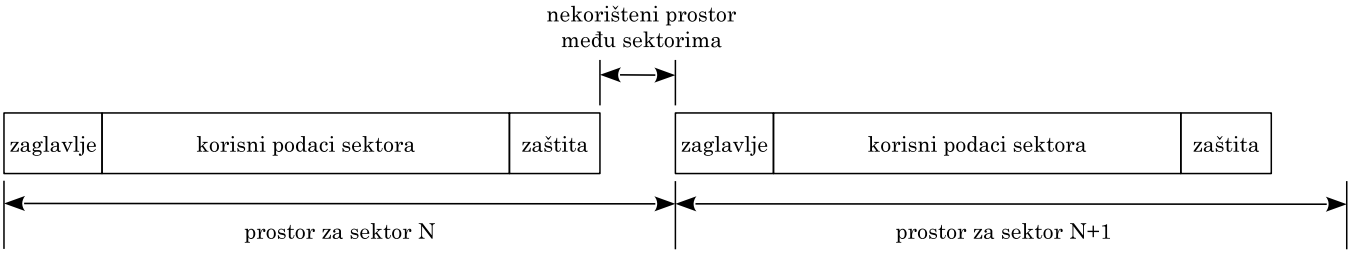
[1]Izvori: http://www.cs.virginia.edu/~gurumurthi/courses/HDD_Basics.ppt,

Čitanje, pisanje, dodatni i zaštitni bitovi (info)

Podaci se čitaju preko glave koja je zapravo jednostavan strujni krug. Pri čitanju, svaka će magnetizirana površina inducirati napon koji se onda može interpretirati kao nula ili jedinica. Međutim, zbog velike gustoće zapisa, očitani signal ipak nije lijepi "pravokutni" već ga treba obraditi da bi dobili stvarno stanje (susjedni bitovi donekle utječu na očitanje trenutne "male površine". Nadalje, s obzirom na te i druge specifičnosti, koriste se posebni kodovi (npr. RLL) za zapis podataka.

Pisanje se ostvaruje tako da se kroz glavu propusti struja određena smjera u trenutku kada je glava točno iznad željenog područja. Tada će se magnetski materijal ispod glave polarizirati u željenome smjeru (i time zapisati željeni sadržaj).

Jedna staza se sastoji od mnogo sektora. Osim korisnih podataka pohranjenih u sektoru, uz sam sektor moraju biti i dodatni podaci. Primjerice, zaglavlje koje uključuje identifikaciju sektora, identifikaciju staze, podnožje sa zaštitnim bitovima (da se može detektirati i možda ispraviti greška u čitanju). Također se između dva sektora može ostaviti i malo prazna prostora ... Proizvođači diskova nastoje smanjiti dodatno trošenje prostora na ostale podatke tako da mogu pohraniti više korisnih podataka.



Slika 9.3. Sektor i popratni podaci na stazi

S obzirom na to da je staza kružnica te da su kružnice dalje od centralne osi rotacije dužeg opsega, na njima je moguće pohraniti više podataka (bitova) nego na onim stazama bliže centru. Stoga se na udaljenijim stazama nalazi i veći broj sektora (te je i čitanje podataka brže s obzirom na konstantnu brzinu vrtnje).

U nastavku razmatranja diskova (prvenstveno u zadacima), koristit će se jednostavan model diska kod kojeg se zanemaruju ostali podaci na stazi, osim korisnih bitova pripadnih sektora. Također, pretpostavit će se da svaka staza ima jednaki broj sektora.

Podatkovna jedinica koju disk daje na zahtjev je “sektor”

Adresa sektora (“fizička adresa”) (engl. *cylinder-head-sector – CHS*) sastoji se od:

1. rednog broja površine
2. rednog broja staze
3. rednog broja sektora

Upravljački sklop diskovne jedinice

- upravlja mehaničkim i elektroničkim dijelovima
- ima procesor (očitani signali nisu lijepi uglati)
- spaja se na sabirnicu

http://www.hitachigst.com/hdd/research/storage/pm/index.html

- ima međuspremnik
- pretvara “linearnu” ili “logičku” adresu u CHS i obratno

Logička adresa sektora (engl. *logical block addresing – LBA*)

- svi sektori su predstavljeni kao jedno polje
- jednostavniji prikaz i upravljanje diskom za OS
- pretvaranje adresa radi sklop diskovne jedinice

Svojstva današnjih diskova (okvirne vrijednosti)

- gustoća zapisa do 1,34 Tbita/in^2 ($\approx$2 Gbita/mm^2)
- kapaciteti: od $\approx$ 100 GB do 20 TB
- promjeri ploča: 3,5"; 2,5"; 1,8"; 1"
- brzine okretanja: 5000 do 15000 okr/min; 5400; 5900; 7200, 10000
- brzina prijenosa (kompaktno smještenih podataka): 100 – 200 MB/s
- vrijeme pristupa (od zahtjeva do posluživanja): 2 do 15 ms ($\approx$ 8 ms)
  - prosječno pomicanje glave za “slučajni zahtjev”
  - računa se kao pomicanje glave za 1/3 staze
- veličine sektora: 512 B, 4 kB

9.1.3. Vremenska svojstva diskova

Komponente trajanja prijenosa podataka ("detaljno"):

1. postavljanje glave na početak podataka (engl. *head position time*)
   - trajanje traženja staze, vrijeme postavljanja (engl. *seek time*)
     - obično zadano:
       - formulom $T_D = \ldots$ ili
       - prosječnim trajanjem – vrijeme prijelaza preko 1/3 staza = *prosječno vrijeme traženja staze*
     - ovisi o početnom i konačnom položaju
     - sastoji se od vremena:
       a) ubrzavanja ručice glave
       b) kretanja konstantnom brzinom (maksimalnom)
       c) usporavanja
       d) finog pozicioniranja na stazu
   - rotacijsko kašnjenje (engl. *rotation latency*) $= \overline{T}_R = T_R/2$
2. čitanje podataka
   - trajanje čitanja dijela staze ili cijele staze
   - po potrebi uzeti u obzir “faktor preplitanja” (info)

3. prijenos podataka u radni spremnik
   - ovisno o disku ova akcija može ići paralelno sa:
     - postavljanjem glave na iduću stazu ili sektor
     - čitanjem idućih sektora
     - zadano u zadatku, ništa nije "pretpostavljeno"

Trajanje prijenosa podataka (ukratko)

a) postavljanje glave na stazu (npr. T_{seek})

b) postavljanje glave na početak staze = rotacijsko kašnjenje $\overline{T}_R = T_R/2$
(Teoretski bi mogli pretpostaviti da čitanje staze može započeti bilo gdje – čitamo cijelu stazu, ne moramo čekati početak! Međutim, u zadacima se ipak uzima i ovaj dio.)

c) čitanje staze = vrijeme jednog okreta diska T_R

d) prijenos staze ili njenog dijela u radni spremnik

Kada treba čitati podatke s više staza, paralelno s operacijom (d) može ići i idući pomak glave (ako je potreban) te se u proračunu uzima veća vrijednost od (d) i (a).

Npr. ako treba pročitati stazu X, X+1, X+2 i pola staze Y (Y je daleko do X-a) uz zadane vrijednosti T_{seek} (prosječan pomak glave), T_1 (pomak glave na susjednu stazu), T_R, V_S (veličina sektora), N (broj sektora na stazi), V_P (brzina prijenosa podataka iz međuspremnika diska u radni spremnik (i obratno)) trajanje bi računali prema:

$$
\begin{aligned}
T_P &= V_S \cdot N / V_P \\
T &= T_{seek} + T_R/2 + T_R + \max(T_P, T_1) \\
&\quad + T_R/2 + T_R + \max(T_P, T_1) \\
&\quad + T_R/2 + T_R + \max(T_P, T_{seek}) \\
&\quad + T_R/2 + T_R + T_P/2
\end{aligned}
\tag{9.1.}
$$

Zadatak 9.1. Čitanje dvije datoteke

Operacijski sustav treba učitati dvije datoteke velike po 4 MB u radni spremnik. Koliko će mu vremena za to trebati ako su datoteke kompaktno smještene na disku (ali svaka na svom dijelu diska, udaljene jedna od druge) te ako su svojstva diska: dvije obostrano magnetizirane ploče (4 glave), 512 staza po površini, 1024 sektora po stazi, veličina sektora je 512 B, 6000 okretaja u minuti, prijenos cijele staze u radni spremnik traje $T_P = 5$ ms, prosječno postavljanje glave traje $T_{seek} = 10$ ms, a premještanje na susjednu stazu $T_1 = 1$ ms.

Rješenje:

Prvo treba ustanoviti koje sektore zauzima datoteka (koliko njih, gdje, ...)

1024 sektora * 512 B = 512 KB po stazi => potrebno 8 staza za svaku datoteku => dva puna cilindra (svaki sa četiri staze)

vrijeme čitanja se sastoji od nekoliko komponenata:

0. postavljanje glave na početnu stazu datoteke (T_{seek}) +
1. postavljanje glave na početak staze ($\overline{T}_R$) +
2. čitanje staze (T_R) +
3. prijenos staze u radni spremnik (T_P) +

Zadnje tri linije treba dodati još dva puta, za 2. i za 3. stazu na istom cilindru.

Čitanje zadnje staze na istom cilindru je jednako, uz razliku da se paralelno s prijenosom staze u radni spremnik pomiče i glava na susjednu stazu:

4. postavljanje glave na početak staze ($\overline{T}_R$) +
5. čitanje staze (T_R) +
6. duže od: (prijenos staze u radni spremnik (T_P), pomak na susjednu stazu (T_1) +

Čitanje drugog cilindra je jednako kao i za prvi, uz razliku da u zadnjem koraku treba paralelno s prijenosom zadnje staze u radni spremnik obaviti pomak na početak druge datoteke – s obzirom na to da je ona daleko uzima se T_{seek}

Potom slijedi čitanje druge datoteke, slično kao i prve (koraci 1-6, bez računanja pomaka u koraku 6).

Konačna formula za ovaj zadatak bi bila:

$$\begin{aligned} t_{\check{c}} = {} & T_{seek} + (\overline{T}_R + T_R + T_P) \cdot 3 + (\overline{T}_R + T_R + \max(T_P, T_1)) \\ & + (\overline{T}_R + T_R + T_P) \cdot 3 + (\overline{T}_R + T_R + \max(T_P, T_{seek})) \\ & + (\overline{T}_R + T_R + T_P) \cdot 3 + (\overline{T}_R + T_R + \max(T_P, T_1)) \\ & + (\overline{T}_R + T_R + T_P) \cdot 4 \end{aligned} \tag{9.2.}$$

Uz $\max(T_P, T_{seek}) = T_{seek}$ i $\max(T_P, T_1) = T_P$ formula se svodi na:

$$t_{\check{c}} = T_{seek} + (\overline{T}_R + T_R + T_P) \cdot 15 + (\overline{T}_R + T_R + T_{seek}) \tag{9.3.}$$

Čitanje jedne staze T_R računa se iz brzine vrtnje diska kao trajanje jednog okreta:

$$T_R = 1/\omega = 1/(6000 \text{ okr/min}) = 1/(6000/60 \text{ okr/s}) = 1/100 \text{ s} = 10 \text{ ms} \tag{9.4.}$$

Rotacijsko kašnjenje $\overline{T}_R$ jest $\overline{T}_R = T_R/2 = 5$ ms

Uvrštavanjem dobiva se:

$$t_{\check{c}} = 10 + (5 + 10 + 5) \cdot 15 + (5 + 10 + 10) = 10 + 300 + 25 = 335 \text{ ms} \tag{9.5.}$$

Zadatak 9.2.

Disk ima 500 sektora po stazi, 30000 staza, 3 površine i vrti se brzinom 7200 okretaja u minuti (engl. *rpm – rotations per minute*). Veličina sektora je 512 B. Upravljački sklop pročita jednu cijelu stazu u interni spremnik, a zatim prenosi potrebne sektore u glavni spremnik. Prijenos u glavni spremnik odvija se brzinom od 300 Mbit/s, a za to vrijeme sklop ne može čitati s diska (ali može pomicati glavu ako je potrebno za iduće zahtjeve).

a) Koliki je kapacitet tog diska?

b) Koliko prosječno traje prebacivanje kompaktno smještene datoteke veličine 5 MB ako je vrijeme postavljanja 10 ms i vrijeme premještanja sa staze na stazu 1 ms?

Zadatak 9.3.

Program A veličine 5 MB i program B veličine 9200 kB su dvije datoteke kompaktno smještene na disk (svaka je zasebno kompaktno smještena). Disk ima 128 sektora po stazi, veličina sektora

je 1 kB, a disk se okreće brzinom 7200 rpm. Upravljački sklop pročita jednu cijelu stazu u interni spremnik, a zatim prenosi potreban dio u glavni spremnik. Prijenos u glavni spremnik odvija se brzinom od 400 Mbita/s, a za to vrijeme sklop ne može čitati s diska. Vrijeme traženja staze je 10 ms, a vrijeme premještanja sa staze na stazu 1 ms. Koliko vremena protekne od istovremenog izdavanja naredbi za pokretanjem programa A i B pa do trenutka kada se oba programa izvode ako:

a) se prvo program A u cijelosti učita u radnu memoriju, a zatim se učitava program B

b) se u radni spremnik prvo prenosi jedna staza programa A, a zatim jedna staza programa B i tako dalje naizmjenično.

Pretpostaviti da vremenom dominira vrijeme potrebno da se program učita u radni spremnik, a sve drugo se zanemaruje npr. vrijeme potrebno da se procesi/dretve stave u liste.

Zadatak 9.4.

Ista fotografija kompaktno je smještena u dvije različite datoteke na disku u nekomprimiranom (128 MB) i komprimiranom formatu (896 kB). Disk ima 128 sektora po stazi, veličine sektora je 1 kB, a disk se okreće brzinom 7200 rpm. Upravljački sklop pročita jednu cijelu stazu u interni spremnik, a zatim je prenosi u glavni spremnik (potrebne sektore). Prijenos u glavni spremnik odvija se brzinom od 100 Mb/s, a za to vrijeme sklop ne može čitati s diska. Vrijeme traženja staze je 10 ms, a vrijeme premještanja sa staze na stazu 1 ms. Ako procesor može dekomprimirati komprimiranu sliku brzinom od 5 Mb/s prikazuje li se brže slika koja se učita iz nekomprimirane ili komprimirane slike?

9.1.4. Posluživanje zahtjeva

- disk je SPOR pa je moguće koristiti postupke optimiranja dohvata
- jedan od oblika optimiranja je optimiranje nad skupom zahtjeva
- ne posluživati zahtjev po zahtjev redom, već iz skupine zahtjeva naći najbolji način redoslijeda posluživanja
  - optimiranje može raditi OS ali i upravljački sklop diska (nad dobivenim zahtjevima)

Strategije posluživanja:

- FCFS – redom prispijeća (engl. *First Come First Served*)
- SSTF – s najkraćim vremenom premještaja glave (engl. *Smallest Seek Time First*)
- SCAN – pregledavanje: ide u jednom smjeru do kraja (zadnje staze u tom smjeru) i staje na svakoj stazi na kojoj postoji zahtjev
- LOOK – pregledavanje: ide u jednom smjeru do zadnjeg zahtjeva u tom smjeru i staje na svakoj stazi na kojoj postoji zahtjev
- C-SCAN, C-LOOK – slične gornjim, samo što se posluživanje obavlja samo u jednom smjeru; kad se dođe do kraja ili zadnjeg zahtjeva, brzim potezom se glava postavlja na prvu stazu (ili prvi zahtjev za C-LOOK) s druge strane.

Zadatak 9.5.

Disk s pokretnim glavama ima 100 staza (0 - 99). Neka se glava trenutno nalazi na stazi 29, s tim da je prije bila na stazi 8. Zahtjevi za pristup pojedinim stazama svrstani po redu prispjeća su: 3, 7, 40, 98, 71, 68, 70, 21, 5 i 49.

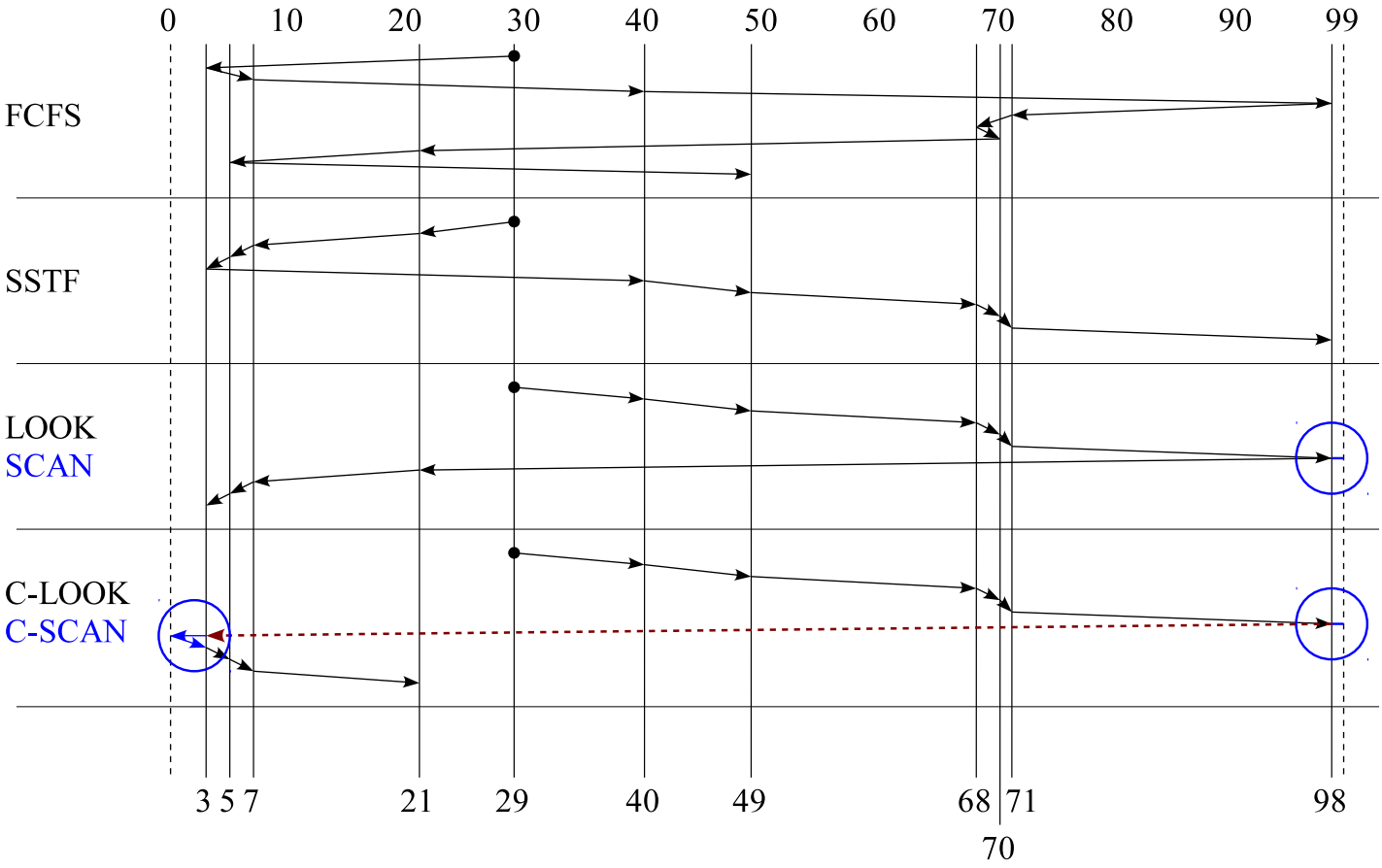
a) Opisati svaku od navedenih strategija!

b) Grafički prikazati kretanje glave prilikom obrade zahtjeva za sve strategije!

c) Koliki je ukupan pomak glave pri izvršenju tih zahtjeva za strategije posluživanja: redom prispjeća (FCFS), s najkraćim vremenom premještaja glave (SSTF), pregledavanje (LOOK i SCAN) i kružno pregledavanje (C-LOOK i C-SCAN)?

d) Kod kojih može doći do izgladnjivanja (beskonačnog odgađanja posluživanja nekih zahtjeva zbog posluživanja novo pristiglih zahtjeva)?

Rješenje:



FCFS: $(29-3)+(98-3)+(98-68)+(70-68)+(70-5)+(49-5)=262$
SSTF: $(29-3)+(98-3)=121$
LOOK: $(98-29)+(98-3)=164$ (SCAN: $(99-29)+(99-3)=166$
C-LOOK: $(98-29)+(21-3)=87$ (C-SCAN: $(99-29)+(21-0)=91$
SCAN i C-SCAN uvijek idu do kraja (bez obzira na zahtjeve)
LOOK i C-LOOK idu do zadnjeg zahtjeva u smjeru.
Zahtjevi su u ovom zadatku svi unaprijed poznati. U stvarnosti novi zahtjevi dolaze stalno, pa je i rad strategija ponešto drukčiji (ponekad se vidi izgladnjivanje kod SSTF strategije).

9.2. Datotečni sustav

CHS $\Rightarrow$ LBA

- adresiranje korištenjem numeracije staza/ploča/sektora (engl. *cylinder-head-sector – CHS*) je kompliciran na novijim diskovima (različiti broj sektora na stazama i sl.)
- noviji diskovi (tj. svi) najčešće i ne nude (prave) informacije o broju staza/ploča/sektora
- oni nude sučelje za korištenje "polja sektora" (engl. *logical block addressing – LBA*)
- sektori su dostupni preko samo jednog broja = rednog broja sektora.
- elektronika pretvara LBA $\Leftrightarrow$ CHS

Blok (nakupina sektora, klaster)

- veličina sektora je svojstvo diska (ne može se mijenjati npr. formatiranjem)
- veličine sektora: 512 B (uobičajeno), 4 KB (noviji diskovi)
- datotečni sustav definira novu jedinicu podataka = blok (engl. *cluster*)
- blok je niz uzastopnih sektora (nakupina sektora)
- niz čini 1 ili 2 ili 4 . . . ili 2^n uzastopnih sektora
- što je blok veći potrebna struktura podataka za opis je manja, ali je gubitak zbog fragmentacije veći

9.2.1. Datoteke

- Podaci na disku su organizirani u datoteke
- Datoteka: skup povezanih informacija koje čine cjelinu (logičke tvorevine)
- Datoteke obično sadrže:
  - programe, npr.: .exe; .out; .bat; .sh; .dll; .so; .jar; . . .
  - podatke, npr.
    - dokumente (word, tekstualne datoteke, HTML, . . .)
    - multimediju (slike, video, muziku, . . .)
    - postavke programa i sustava
  - ostalo
    - privatne podatke OS-a (dijelove pomoćnog spremnika)
    - privatne podatke datotečnog sustava
- formati datoteka:
  - binarna (.exe, .doc, .dll, .zip, .mp3)
  - tekstualna (.txt, .html, .pls, .srt, .bat) => ASCII ili sličan format (npr. UTF-8)
- OS se učitava iz datoteka!
- Sve mora biti na disku u datotekama (osim za ugrađene sustave koji ne moraju imati disk)

9.2.2. Datotečni sustav

Pojam "datotečni sustav" koristimo:

- za oznaku tipa datotečnog sustava (NTFS, FAT, UDF, ISO 9660, . . .)
- za dio operacijskog sustava – točnije bi bilo reći "datotečni podsustav"
- za neki konkretni datotečni sustav (na primjeru ili stvarnom računalu, "na tom disku")

Iz konteksta je uvijek jasno na što se odnosi pojam.

Datotečni sustav daje odgovore na pitanja:

- kako su datoteke smještene na disku
  - fizičko smještanje: gdje, u kojim sektorima/blokovima, kojim redoslijedom
  - logičko smještanje: kako su datoteke organizirane, grupirane, kako im se pristupa, pronalazi, . . . ?
  - atributi: tko im smije pristupiti, sigurnost, učinkovito korištenje diska (fragmentacija), . . . ?
- koji dijelovi diska su slobodni

Datotečni sustav definira kako smjestiti podatke na disk i kako do njih doći

Disk se može i podijeliti u više particija (svezaka)

- svaka particija je zasebni datotečni sustav
  - npr. part1: blokovi 0-10000, part2: blokovi 10001-20000

Datotečna tablica (engl. *file table*)

- tablica sadrži:
  - podatke koji definiraju disk, slobodni prostor (ponekad su ove informacije u zasebnim strukturama izvan tablice)
  - opisnike datoteka
- svaka datoteka ima svoj opisnik u datotečnoj tablici
- datoteke se nastoje spremiti u kontinuirani dio
  - smanjenje fragmentacije – datoteka se brže učitava

OS koristi datotečni sustav – preko datotečnog podsustava – za operacije:

- stvori, obriši, preimenuj, premjesti datoteku ili direktorij
- otvori datoteku, čitaj, piši, pomakni kazaljku, . . .

9.2.3. Opisnik datoteke

- svaka datoteka ima svoj opisnik u datotečnoj tablici
- osnovni dijelovi opisnika:
  - ime datoteke
  - direktorij gdje je datoteka smještena (u logičkoj org. diska)
  - tip datoteke

  - veličina datoteke
  - vrijeme stvaranja, zadnje promjene, zadnjeg korištenja
  - podaci o “vlasniku” (kojem korisniku pripada), prava pristupa
  - ...
  - opis smještaja na disku (u kojim blokovima)

9.2.4. Direktoriji

- datoteke su logički organizirane preko stabla direktorija
- direktoriji su logička tvorevina – povezuju datoteke iste namjene, istog korisnika i slično
- Windows pristup:
  - svaka particija ima svoje ime (C:, D:, E:, ...)
  - particija na kojoj je OS (načešće C:) naziva se sustavska
  - uobičajeni direktoriji i njihov sadržaj:
    * `C:\Windows\` – operacijski sustav
    * `C:\Program Files\` – programi
    * `C:\Program Files (x86)\` – 32-bitni programi na 64-bitovnom OS-u
    * `C:\ProgramData` – postavke programa
    * `C:\Users\korisničko_ime` – korisnički direktoriji, postavke i podaci
    * `C:\pagefile.sys` – pomoćni spremnik za straničenje
  - druge particije, CD/DVD i sl.: svaki ima svoju oznaku (D:, E:, ...)
- UNIX pristup:
  - `/` – početni direktorij (korijen, *root*)
  - `/home/korisničko_ime` – korisnički direktoriji (postavke i podaci)
  - `/etc/` – većina postavki sustava
  - `/bin/`, `/sbin/`, `/usr/bin/`, `/usr/local/bin/` – OS i programi
  - i još puno njih sa svojim posebnim funkcijama
  - pogledati: http://en.wikipedia.org/wiki/Unix_filesystem
  - particije:
    * tipično (najjednostavnije)
      · jedna particija za `/` (i sve na njoj)
      · jedna particija za `swap` (pomoćni spremnik za straničenje, opcionalno)
    * “naprednije” postavke s više particija, npr.:
      · jedna particija za `/home`
      · jedna particija za `/boot`
      · jedna particija za `/` (sve ostalo)

        - jedna particija za `swap` (pomoćni spremnik za straničenje)
    - druge particije, CD/DVD i sl.:
        - spajaju se na neku "točku" datotečnog sustava (engl. *mount point*)

Na jednoj particiji (jednom datotečnom sustavu) nalaze se blokovi sa sadržajima:

- "opisnik" particije (veličina i broj blokova, . . .)
- datotečna tablica (opisnici datoteka i direktorija)
- opisnici slobodnog prostora
- blokovi sa sadržajem datoteka
- slobodni blokovi

9.3. Primjeri datotečnih sustava

9.3.1. NTFS

- NTFS – skraćenica od *New Technology File System*
- NTFS sadrži datotečnu tablicu koja se zove *Master File Table* – MFT (*glavna tablica datoteka*)
  - svaka datoteka ima opisnik u MFT, pa i sama MFT
  - u opisniku se nalaze podaci o datoteci
- numeriranje blokova u NTFS-u:
  - LCN – Linear Cluster Number
    - logička adresa bloka na particiji
    - particija se dijeli u blokove, linearno numerirane, počevši s LCN=0
  - VCN – Virtual Cluster Number
    - logička adresa bloka datoteke
    - svaka se datoteka sastoji od skupine blokova (osim onih vrlo malih, čiji je sadržaj pohranjen u samom opisniku)
    - VCN predstavlja adresu bloka unutar datoteke
      - prvi dio datoteke je u bloku s VCN=0, drugi u VCN=1, itd.
  - Povezivanje VCN-a u LCN definirano je u opisniku datoteke
  - datoteka koja je kompaktno smještena na disku ima samo jedan zapis u tablici (gdje je prvi blok i koliko ih ukupno ima)
- jako male datoteke (manje od $\sim$900 B) pohranjuju se u sam opisnik, ne zauzimaju dodatne blokove na disku

Primjer 9.1. Datoteka na disku

Zadana je datoteka sa sadržajem i prikazom svih blokova na disku.

Tablica 9.1. Datoteka – logički prikaz

blok	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
sadržaj	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Tablica 9.2. Datotečni sustav (blokovi particije)

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18 C	19 D	20 E	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37 A	38 B	39	40
41	42	43	44	45	46	47	48
49	50	51 K	52 L	53 M	54 N	55 O	56
57 F	58 G	59 H	60 I	61 J	62	63	64

Primjer 9.2. NTFS

Opis smještaja datoteke za primjer 9.1. prema NTFS pravilima:

VCN	LCN	#
1	37	2
3	18	3
6	57	5
11	51	5

– broj uzastopnih blokova – nakupina blokova

Prvi blok datoteke (VCN=1) nalazi se u 37. bloku particije (LCN=37). S obzirom na to da datotečni sustav nastoji datoteke održati kompaktnima, jedan red tablice može opisati i više **uzastopnih** blokova. Koliko ih opisuje kazuje nam zadnji stupac. Prvi red tako opisuje blok 1 i blok 2, s time da se blok 2 (VCN=2) nalazi u bloku LCN=38.

Primjer 9.3. NTFS (2)

Neka datoteka pohranjena je kompaktno po dijelovima u blokovima:

1. $526 - 587$
2. $124 - 225$
3. $432 - 449$.

Prikazati sadržaj dijela opisnika te datoteke koji opisuje njen smještaj, ako se radi o datotečnom sustavu NTFS.

Rješenje:

1. dio: $526 - 578 \Rightarrow 578 - 526 + 1 = 62$ bloka (uključen je i $526.$ i $578.$!)
2. dio: $124 - 225 \Rightarrow 225 - 124 + 1 = 102$ bloka
3. dio: $432 - 449 \Rightarrow 449 - 432 + 1 = 18$ blokova

VCN	LCN	#
1	526	62
63	124	102
165	432	18

Zadatak 9.6. NTFS (3)

Neka datoteka je smještena na disku po dijelovima: prvi MB je kompaktno smješten počevši od 725. bloka diska, druga dva MB su kompaktno smještena počevši od bloka 2001. Ako je veličina bloka 4 KB, prikazati dio sadržaj opisnika datoteke, koji opisuje smještaj datoteke na disku. U kojem se bloku na disku nalazi bajt s adresom 2000000 unutar datoteke?

Rješenje:

Zad. 9.6. NTFS

prvi MB => 725. bloka diska

druga dva MB => 2001.

vel. bloka = 4 KB

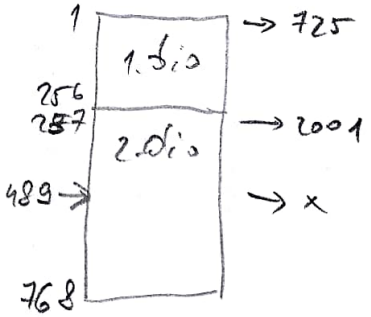
VCN	LCN	#
1	725	256
257	2001	512

$\frac{1\,MB}{4\,KB} = \frac{1024\,KB}{4\,KB} = 256$ blokova

2 MB => 512 blokova

$\frac{2000000}{4\,KB} = \frac{2000000}{4096} = 488{,}28$ => bajt 2000000 je u bloku dat 489

gdje je na disku blok 489 ?



489 − 257 = x − 2001

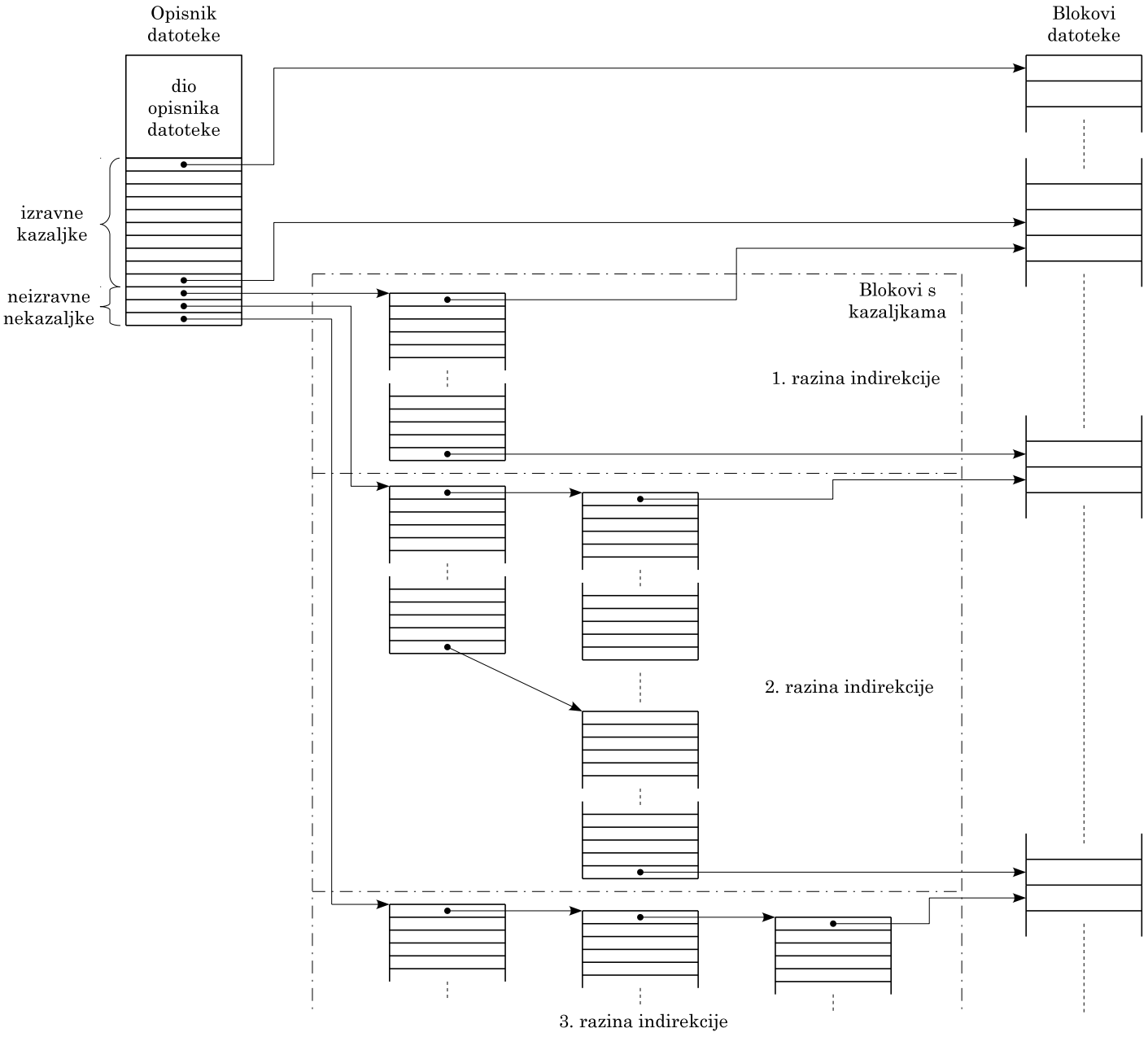
x = 2001 + 489 − 257

= 2001 + 222

= 2223

9.3.2. UNIX i-node

- Opisnik datoteke = i-node = index node
- Za opis blokova koriste se sljedeće kazaljke:
  - deset izravnih kazaljki
    - kazaljke koje izravno pokazuju na blokove datoteke (LCN indeksi)
    - npr. 5. kazaljka pokazuje na blok koji sadrži 5. blok datoteke
  - jedna jednostruko indirektna kazaljka
    - kazaljka na blok s kazaljkama na blokove datoteke
  - jedna dvostruko indirektna kazaljka
    - kazaljka na blok s kazaljkama na blokove s kazaljkama na blokove datoteke
  - jedna trostruko indirektna kazaljka
    - kazaljka na blok s kazaljkama na blokove s kazaljkama na blokove s kazaljkama na blokove datoteke



Primjer 9.4. UNIX i-node za datoteku iz primjera 9.1.

U opisniku datoteke:

opisnik	37	38	18	19	20	57	58	59	60	61	X	-	-

U bloku X (na disku):

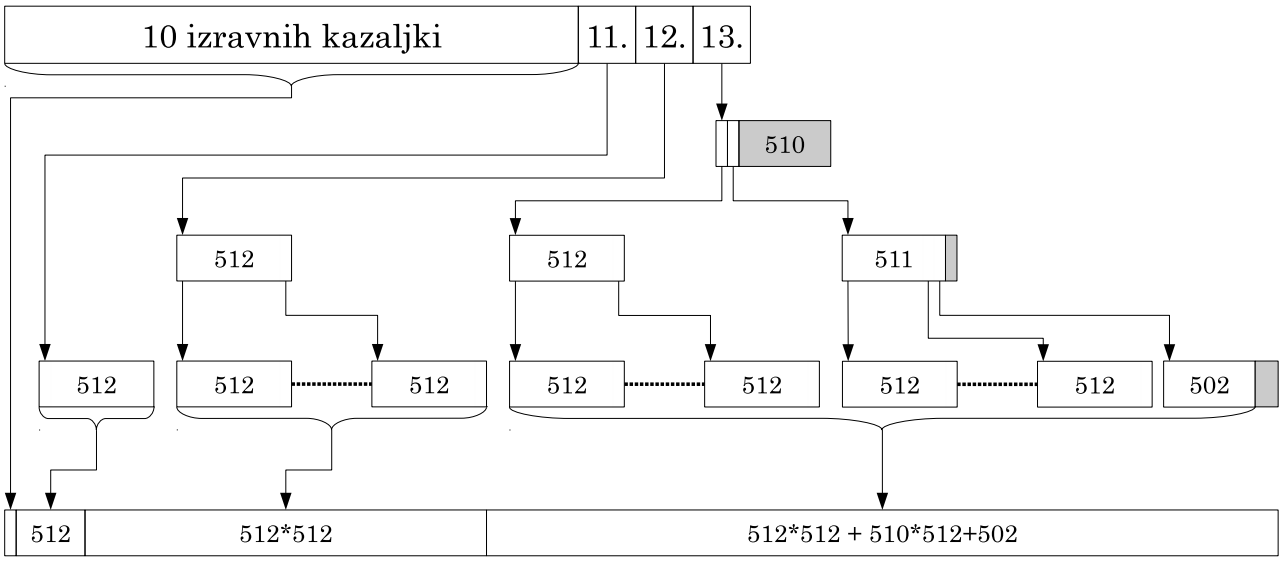
51	52	53	54	55	(ostatak kazaljki je neiskorišten)

Zadatak 9.7.

Ako je na UNIX datotečnom sustavu pohranjena datoteka veličine 3GB koliko spremničkog prostora zauzimaju kazaljke za tu datoteku? Skicirajte organizaciju tih kazaljki. Veličina bloka je 4 kB, a veličina kazaljke 64 bita.

Rješenje:
broj blokova $= 3$ GB $/4$ kB $= 3 \cdot 1024 \cdot 1024$ kB $/4$ kB $= 3 \cdot 256 \cdot 1024 = 786432$ blokova
u jedan blok stane 4 kB $/64$ bita $= 4096/8 = 512$ kazaljki

- prvih 10 kazaljki opisuje prvih 10 blokova datoteke: ostaje $786432 - 10 = 786422$ blokova
- 11. kazaljka pokazuje na blok s 512 kazaljki: ostaje $786422 - 512 = 785910$ blokova
- 12. kazaljka pokazuje na blok s 512 kazaljki na blokove s kazaljkama
  - ukupno $512 \cdot 512 = 262144$ kazaljki
  - ostaje $785910 - 262144 = 523766$ blokova
- 13. kazaljka pokazuje na blok s 512 kazaljki na ...
  - $523766/(512 \cdot 512) = 1,998 \Rightarrow$ jedan cijeli blok s $512 \cdot 512$ kazaljki te još jedan skoro cijeli
  - $523766 - (512 \cdot 512) = 261622 = 510 \cdot 512 + 502$
  - $523766 = (512 \cdot 512) + (510 \cdot 512 + 502)$



Zadatak 9.8.

Ako je na UNIX datotečnom sustavu pohranjena datoteka veličine 555 kB koliko spremničkog prostora zauzimaju kazaljke za tu datoteku? Skicirajte organizaciju tih kazaljki. Veličina bloka je 1024 okteta (1kB), a veličina kazaljke 32 bita.

9.3.3. FAT (informativno)

- koristi:
  - "tablicu direktorija" (engl. *directory table*) i
  - "tablicu zauzeća" (engl. *file allocation table – FAT*)
- tablica direktorija sadrži:
  - opisnike datoteka i direktorija
    - ime datoteke
    - ostali atributi
    - adresa prvog bloka datoteke (u kojem se bloku particije nalazi)
- tablica zauzeća je zajednička za sve datoteke (jedna za sve)
  - tablica ima onoliko zapisa koliko ima blokova na particiji
  - svaki zapis tablice (broj, 32-bitovni broj za FAT32):
    - pokazuje na idući blok datoteke, ili
    - -1 ako je ovo zadnji blok u datoteci, ili
    - 0 ako je blok slobodan (ne koristi se)

Primjer 9.5. FAT

Za datoteku iz primjera 9.1. u opisniku datoteke, koji se nalazi u tablici direktorija, piše samo adresa prvog bloka = 37.

Tablica zauzeća (engl. *file allocation table*):

	19	20	57				
				38	18		
		52	53	54	55	-1	
58	59	60	61	51			

9.3.4. EXT2 (informativno)

- nastao po uzoru na MINIX/UNIX uz poboljšanja
- opisnik datoteke = inode (indeksni čvor)

Globalni podaci na particiji:

- superblock - svaka particija sadrži po jedan opisnik particije
- blokovi su grupirani u "grupe blokova" (iste veličine)
  - jednu grupu čine "susjedni" blokovi
- tablica s opisom svih grupa blokova

Svaka grupa blokova ima:

- opisnik grupe - gdje su smješteni idući elementi (ovi ispod)
- bitmapa za opis zauzetih i slobodnih blokova
- bitmapa za opis zauzetih i slobodnih inode-a
- inode tablica za ovu grupu (tablica opisnika dateteka, direktorija, ...)
- kopija superblock-a - radi mogućnosti oporavka od kvara (ne moraju sve grupe ovo imati)
- blokovi za podatke datoteka (i dodatne opisnike)

Direktoriji:

- opisnik direktorija je također inode
- sadržaj blokova na koje inode (direktorija) pokazuje su parovi {inode, ime} koji predstavljaju sadržaj direktorija (tj. datoteke i direktorije u njemu)

Ideja grupiranja blokova u grupe:

- smanjiti problem fragmentacije držanjem povezanih podataka zajedno
- pokušati držati sadržaje jedne datoteke unutar grupe == blizu!
- sadržaj direktorija u jednoj grupi

Primjer za disk od 47 GB ($\approx 12 \cdot 10^6$ blokova): (`dumpe2fs -h /dev/*particija*`)

- vel. bloka = 4 KB
- vel. grupe = 32768 blokova
- broj inode-ova po grupi = 8192
- broj blokova za inode-ove = 512

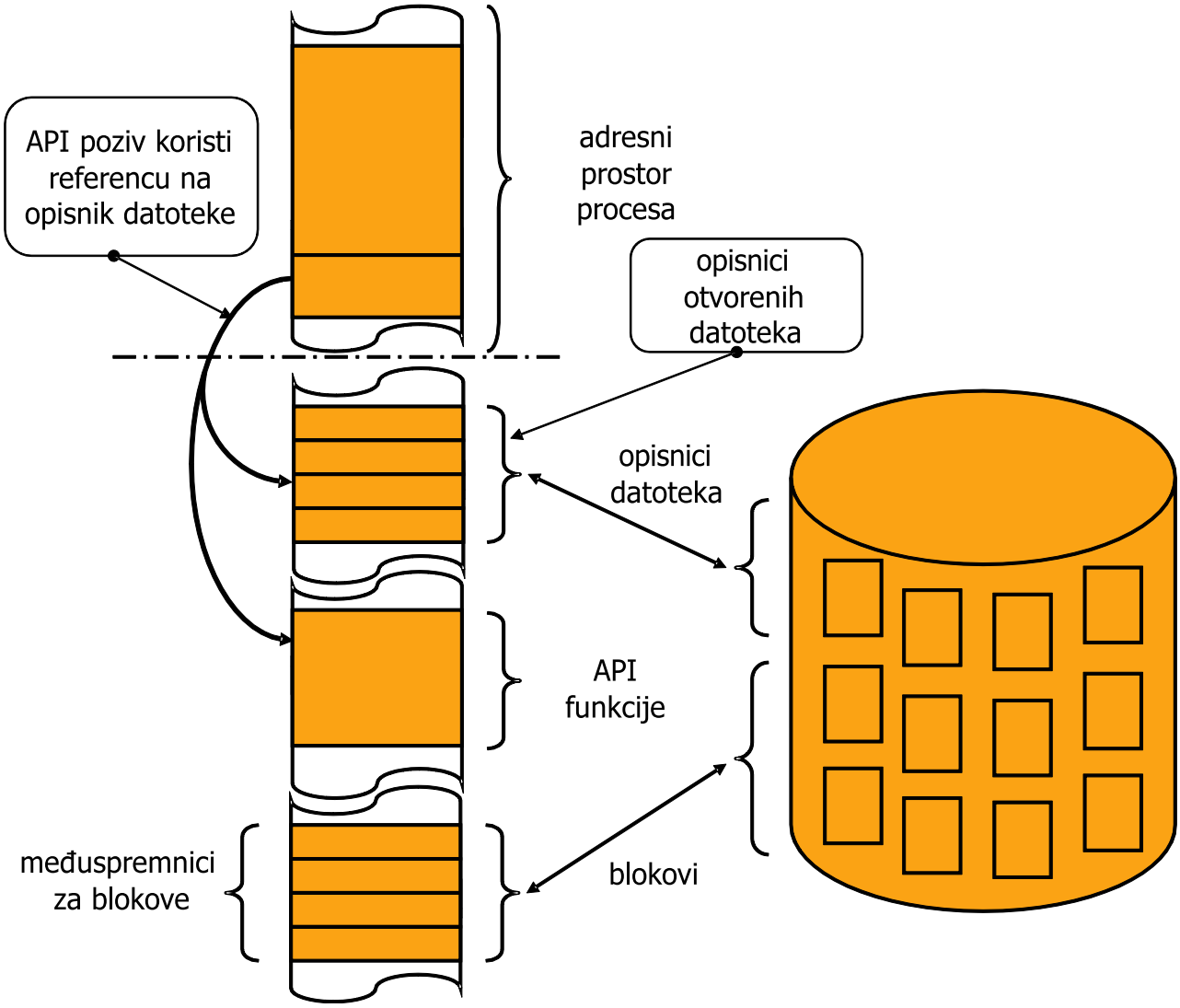
Dohvat sadržaja koje inode opisuje:

- 12 izravnih kazaljki + 3 indirektine (jednostruko, dvostruko i trostruko)

Noviji ext3 i ext4 su slični, barem po ovim opisanim svojstvima

9.4. Datotečni podsustav operacijskog sustava

- Operacijski sustav treba omogućiti korištenje datotečnog sustava – kroz datotečni podsustav
- OS zato:
  - kopira datotečnu tablicu u radni spremnik (ili samo dijelove koje koristi)
  - za svaku datoteku koja se koristi stvara kopiju opisnika i proširuje ga:
    * kazaljkom
    * međuspremnicima (za brži rad)
    * ...
- Pri radu s datotekama (čitanje/zapisivanje) koristi se datotečna kazaljka (engl. *file pointer*)
  - prilikom "otvaranja datoteke" kazaljka pokazuje na početak datoteke
  - čitanjem i pisanjem se kazaljka pomiče prema naprijed



- Korištenje datoteka
  - OS pruža sučelje za korištenje datoteka
  - uobičajena sučelja uključuju:
    * `int open (char *filename, int access, int perm);`
    * `int close (int handle);`

* `int read (int handle, void *buffer, int nbyte);`
* `int write (int handle, void *buffer, int nbyte);`
* `int lseek (int fildes, int offset, int whence);`

Primjer programa:

```
include <stdio.h>

int main()
{
    FILE *fp;

    fp = fopen("hello.txt", "w");
    fwrite("Hello world!\n", 13, 1, fp);
    fclose(fp);

    return 0;
}
```

9.4.1. Jezgrine funkcije datotečna podsustava (informativno)

Načelno ostvarenje operacija datotečnog podsustava kroz jezgrine funkcije skicirano je u knjizi

- `Stvoriti_datoteku(ime, atributi)`
- `Otvoriti_datoteku(ime, način_pristupa, id)`
- `Čitati_datoteku(id, logička_adresa, broj_bajtova)`
- `Stvoriti_datoteku(ime, atributi)`
- pored navedenih potrebne su i operacije za zatvaranje datoteke, brisanje, premještanje, preimenovanje, pomicanje unutar datoteke, promjena veličine i slično

U načelnom rješenju su zanemarini neki problemi:

- korištenje podataka s diska traži:
  - započinjanje operacije
  - dovršavanje operacije nakon prekida naprave
- za obavljanje jedne operacije dohvata/pohrane može biti potrebno obaviti više operacija s diskom!
- u istom računalu može biti više uređaja s datotečnim sustavima
  - različiti uređaji (trebaju različite upravljačke programe)
  - različiti tipovi datotečnih sustava (npr. NTFS, FAT, UDF, . . .)
- navedeno povećava složenost izvedbe takvih operacija
- zato se i sam datotečni podsustav najčešće ostvaruje u slojevima

9.5. Primjer slojevitog datotečnog podsustava (info)

- datotečni podsustav je složen
- zato se ostvaruje u slojevima
- podsjetnik na slojeve računalnog sustava
  - korisnik $\Longleftrightarrow$ programi $\Longleftrightarrow$ API $\Longleftrightarrow$ jezgra OS-a $\Longleftrightarrow$ sklopovlje
- API sučelje prema programima
  - prilagodba jezgrinih funkcija
  - dodatno korištenje međuspremnika radi ubrzanja
  - neće se razmatrati ovdje (u sklopu dat. sust.) – razmatra se samo jezgra
- slojevi datotečna podsustava (u jezgri OS-a) i njihova svrha
  - VFS $\Longleftrightarrow$ LFS $\Longleftrightarrow$ BDEV $\Longleftrightarrow$ DEV
  - npr. pogledati mapu jezgre Linuxa na slici 1.2. na stranici 6 (pod *storage* stupcem)

1. VFS – virtualni datotečni sustav
   - ostvaruje sučelje prema programima (ili API podsloju)
   - OS može koristiti više različitih datotečnih sustava, npr.
     - na disku imamo NTFS ili ext2 ili slično
     - USB štapić je možda FAT32
     - DVD je u UDF formatu
     - mrežnom disku pristupamo preko Samba-e ili NFS-a ili ?
     - ...
   - kad se pozove `otvori_datoteku` iz programa to sučelje mora moći otvoriti datoteku bez obzira gdje se ona nalazila
     - zato takvo sučelje (na vrhu) je "virtualno", "u ostvarenju" mora koristiti neko stvarno (npr. NTFS)
2. LFS – logički datotečni sustav
   - "upravljački programi" koji znaju raditi s "konkretnim" tipom datotečnog sustava
     - tu su npr. FAT32, NTFS, ISO*, UDF, Samba, ...
     - "znaju" kako su podaci zapisani u opisniku, gdje su opisnici i sl.
   - isti tip datotečnog sustava može biti na raznim medijima
     - npr. NTFS može biti i na disku i na USB štapiću i ...
     - za svaki različiti uređaj potreban je različiti upravljački program
   - naprave koje služe za pohranu datotečnih sustava su spore (u usporedbi sa spremnikom)
     - rad s njima svodi se na:
       * započinjanje operacije
       * dojava kraja operacije preko mehanizma prekida

  - stoga bi operacije nad datotečnim sustavom bile slične opisanim UI operacijama (u 5. poglavlju)
  - međutim, za jednu operaciju nad datotečnim sustavom često treba više UI operacija
    * primjerice: dohvat dijela datoteke (jedan poziv `read` ili `write`) može tražiti podatke iz više blokova datoteke
    * zahtjev jednog `read`-a može biti za više blokova
    * različite dretve (procesi) mogu imati zahtjeve nad istim diskom
    * posluživanje zahtjeva nije (općenito) FIFO
    * pristup: "blokiraj dretvu nad napravom + odblokiraj prvu u obradi prekida" ovdje nije dovoljan ni prikladan
  - jedan od načina rješavanja ovakva problema jest da se jezgrina dretva (koja izvodi jezgrinu funkciju) i sama tretira kao dretva (vezana uz dretvu koja je pozvala jezgrinu funkciju) te da se i ona može blokirati
    * poslije dovršetka prve operacije i odblokiranja takve dretve ona može započeti i drugu UI operaciju ...
    * za to je potreban i poseban kontekst takve dretve - jezgrin kontekst dretve
    * opisnik obične dretve tada treba proširiti opisnikom jezgrine dretve povezane s njom
    * o takvoj dretvi više u samom primjeru koda
- pojedini upravljački program LFS-a ne koristi izravno upravljački program naprave (npr. diska) već se koristi međusloj – blok naprave

3. BDEV – blok naprave
   - posebnost naprava koje služe za pohranu datotečnih sustava jest što je jedinica podataka "blok"
     - blok je veličine 512 B, 1 KB, 2 KB, 4 KB, ...
     - operacije koje se traže nad takvim napravama su:
       * pročitaj blok
       * zapiši blok
     - operacije nisu (općenito) odmah gotove!
       * naprava završetak zadane operacije javlja prekidom
       * u obradi prekida rade se dodatni poslovi
   - zato se uvodi dodatni sloj "blok naprava" s dvije osnovne operacije:
     - `dohvati_blok` te `pohrani_blok`
     - dio tih operacije izvodi se po prihvatu prekida tih naprava
     - zato je uz gornja dva sučelja potrebno i `obradi_prekid_blok_naprave`
   - blok naprava kao sloj može biti neovisna o uređaju i tipu datotečnog sustava
     - s jedne strane koristi upravljački program naprave (preko sučelja sloja BDEV)
     - s druge strane daje sučelje prema LFS-u

- zašto navedene operacije (ovo što sučelje "blok naprava" pruža) nisu izravno uključene u upravljački program naprave?
  - da se iste operacije ne dupliciraju u svakom upravljačkom programu
  - da se smanji složenost upravljačkih programa

4. DEV – sloj upravljačkog programa naprave (npr. diska)
   - "zna" komunicirati s napravom
     - slati podatke
     - čitati podatke
     - dohvatiti status naprave i informacije o zadnjim posluženim zahtjevima
5. primjer/skica ostvarenja u "dat-sust" (na webu)

9.6. Uloga međuspremnika u povećanju učinkovitosti (sažetak)

Korištenje međuspremnika (engl. *cache*) u povećanju učinkovitosti (sažetak)

- ideja: korištenjem međuspremnika postići brzinu pristupa jednaki najbržoj komponenti spremnika (npr. brzini L1 međuspremnika procesora!)
- [disk: mag. ploče⇔međuspremnik] $\Longleftrightarrow$ [spremnik] $\Longleftrightarrow$ [procesor: L3⇔L2⇔L1⇔registri]
- straničenje:
  - TLB (pamti opisnike zadnjih korištenih stranica)
  - u radnom spremniku samo “potrebne” stranice
- korištenje diska (i drugih UI naprava)
  - čitanje malo više podataka od traženih (susjedni blokovi će možda trebati)
  - zadržavanje korištenih blokova jedno vrijeme (možda će opet trebati)
- uloga “arhitekta” programske potpore je značajna
  - hijerarhijska organizacija spremnika MOŽE biti vrlo učinkovito iskorištena (idealno se efektivna brzina svodi na najbrži spremnik = L1; tome se možemo jako približiti!)
  - isto tako loše posloženi sustavi mogu biti vrlo spori (i na najbržem sklopovlju) upravo zbog lošeg korištenja priručnih spremnika, tj. zbog "šaranja" po spremniku

Pitanja za vježbu 9

1. Od kojih se komponenata sastoji disk (HDD)?
2. Kako se na disk pohranjuju podaci? Koji se materijali i principi koriste?
3. Kako se iskazuje adresa jednog sektora (kako se sektor jedinstveno identificira)?
4. Što je to “cilindar”?
5. Koje operacije obavlja upravljački sklop diskovne jedinice?
6. Od kojih se komponenata sastoji vrijeme čitanja jednog sektora?
7. Opisati pojmove: vrijeme postavljanja, rotacijsko kašnjenje, faktor preplitanja.
8. Ako je brzina prijenosa podataka zadana s 100 Mb/s (ili Mbita/s) koliko je to b/s (bita/s)?
9. Opisati strategije posluživanja: FCFS, SSTF, SCAN, LOOK, C-SCAN, C-LOOK.
10. Što je to “datoteka”?
11. Što je to datotečni sustav? Što sadrži?
12. Navesti elemente opisnika datoteke.
13. Opisati kako se opisuje smještaj datoteke na NTFS te UNIX i-node sustavima.

10. VIRTUALIZACIJA

10.1. Uvod

Virtualizacijom se unutar stvarna sustava (operacijska sustava, sklopovlja) stvara okruženje (virtualno računalo) sličnih ili različitih svojstava od stvarna sustava, prema potrebama.

Moguće prednosti virtualizacije:

- virtualno okruženje bolje odgovara potrebama
  - sklopovlje u virtualnom sustavu može biti i različito od postojećeg, stvarnog
  - programska potpora može biti različita (npr. operacijski sustav, programi, datotečni sustav, ...)
  - kod projektiranja drugih sustava, simulacija takvih sustava kroz virtualizaciju omogućava jednostavniji razvoj (lakše povezivanje s alatima za ispitivanje ispravnosti, pronalaženje i ispravljanje pogrešaka)
- bolja iskorištenost sklopovlja
  - na istom sklopovlju može se pokrenuti više istih/različitih virtualnih sustava
  - umjesto N stvarnih računala/poslužitelja koristi se jedno (možda snažnije)
  - smanjeni troškovi održavanja, potrošnje, nadogradnje
  - proširivost
    - jednostavno dodavanje novih virtualnih okolina/računala
    - modularno sklopovlje se često može nadograditi (npr. poslužitelj koji se koristi za virtualizaciju se može proširiti dodatnim diskovima ili dodatnim poslužiteljima s kojima će tvoriti logičku cjelinu radi ostvarenja još jačeg sklopovlja koja je podloga virtualizaciji – dodatno sklopovlje se najčešće može samo ugraditi, a virtualizacijski programi će ga automatski uklopiti u postojeći sustav)
  - zastarijeli poslužitelji se mijenjaju jednim novim, koji preuzima usluge prethodnih korištenjem virtualizacije (i u novom sustavu mogu postojati jednaki sustavi, samo što su oni sada virtualni, ali i dalje jednako funkcionalni ili čak i bolji)
- izolacija
  - izolacijom se štite sustavi izvan virtualnog od sustava u virtualnom okruženju, i obratno
  - dostupnost samo dijela sustava u virtualnom okruženju
    - neki se podsustavi operacijska sustava mogu i izostaviti u virtualnom okruženju ili ograničiti pristup samo dijelu nekih podsustava/operacija
  - razvoj novih sustava/programa/operacija u zaštićenom okruženju bez rizika za ostatak sustava

10.1.1. Osnovni pojmovi

- domaćin (engl. *host*) – operacijski sustav (OS) unutar kojeg se ostvaruje virtualno okruženje, ili upravljač virtualnim strojevima ako nema OS-a
- gost (engl. *guest*) – OS koji se pokreće u virtualnom okruženju
- virtualni stroj (VS), virtualno računalo (engl. *Virtual Machine – VM*) – okruženje u kojem se pokreće gost, a s kojim upravlja upravljač virtualnim strojevima
- upravljač virtualnim strojevima – hipervizor (engl. *hypervisor*) – program na domaćinu koji stvara virtualno okruženje u kojem se mogu pokretati gostujući OS-evi

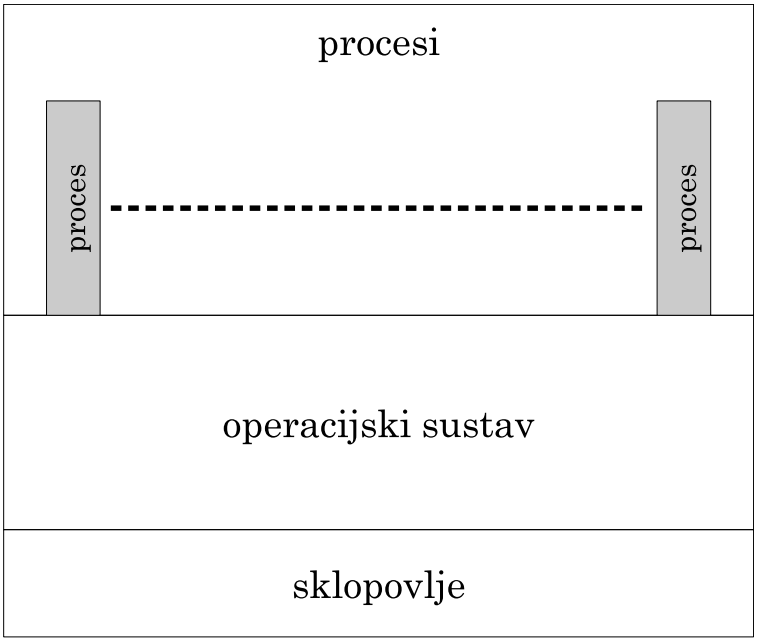
- emulacija – umjesto da se koristi pravo sklopovlje (procesor, IO naprave), ono se oponaša u svim detaljima (npr. sa svim skupom internih registara)
- simulacija – umjesto da se koristi pravo sklopovlje ono se oponaša samo u vanjskom ponašanju (npr. ako mu nešto pošaljemo simuliramo odgovor bez da emuliramo svo njegovo interno stanje)
  - kod virtualizacije se ponekad upotrebljava emulacija, a ponekad simulacija
- paravirtualizacija – u sustavu virtualizacije koristi se prilagođeni gost koji je svjestan da se izvodi u virtualnom okruženju te neke elemente sustava ne koristi na uobičajeni način (kao da sam upravlja tim sklopovljem) već koristi dodatna sučelja koja mu nudi hipervizor, sve radi učinkovitijeg rada (da se što više preskoče dupli poslovi – prvi puta u gostu sa simuliranim sklopovljem, a onda u hipervizoru sa stvarnim)
  - npr. ako hipervizor nudi korištenje diskova kroz jednostavnije sučelje, gost neće slati naredbe (simuliranom) kontroleru (koristiti PCIe sučelje, način pakiranja i slično), već će na jednostavniji način hipervizoru reći što želi, a hipervizor će odraditi taj posao (koji bi i inače napravio)

10.2. Oblici virtualizacije i slični mehanizmi

1. procesi (ostvareni mehanizmima upravljanja spremnikom kao što je to straničenje)
2. virtualno okruženje za pokretanje jednog procesa (npr. Java virtualni stroj – *Java VM*)
3. virtualno okruženje za pokretanje skupa procesa (npr. *Linux Containers*)
4. virtualno okruženje za pokretanje operacijskog sustava (“prava virtualizacija”)
   - Tip-0: tanak sloj s hipervizorom nalazi se nad samim sklopovljem, u “firmwareu”
   - Tip-1: poseban OS s hipervizorom nalazi se nad samim sklopovljem
   - Tip-2: hipervizor je program koji se pokreće u “običnom” OS-u

10.3. Procesi u operacijskom sustavu

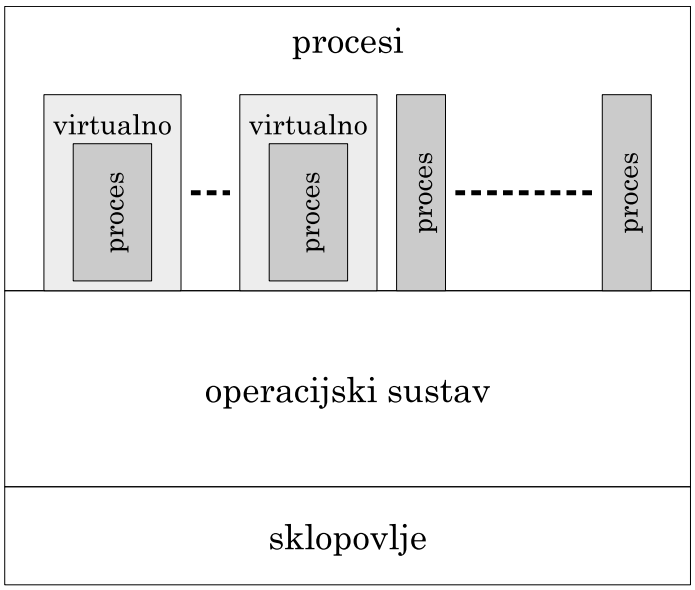
- pokretanjem programa stvara se proces
- proces ima izolirani adresni prostor od ostalih procesa i operacijska sustava (straničenje to omogućava)
- svi procesi unutar istog operacijskog sustava dijele sam operacijski sustav
- zajednički elementi unutar jednog operacijskog sustava za procese:



Slika 10.1. Sustav bez virtualizacije

  - datotečni (pod)sustav (i povezani objekti)
  - mrežni podsustav
  - ulazno-izlazne naprave (grafički podsustav, ...)
  - zajednički spremnik, redovi poruka, cjevovodi, ...
- jedan proces svojim akcijama neizravno, preko zajedničkih elemenata, može utjecati na druge procese

10.4. Virtualizacija na razini aplikacije

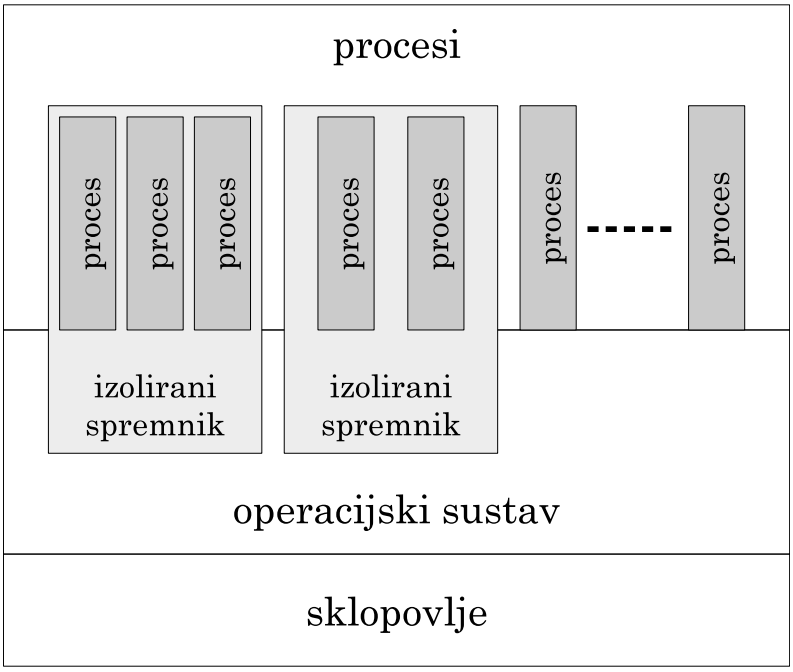


Slika 10.2. Virtualizacija na razini aplikacije

- programi koji nisu pripremljeni u strojnom obliku za operacijski sustav i računalo na kojem se izvode
- stvara se virtualno okruženje samo za taj program
- primjeri: Java programi, programi koji se interpretiraju (npr. skripte, programi u Python-u, Perl-u, JavaScript-u, ...)

- virtualno okruženje može imati ograničeni pristup sredstavima sustava (disku, mreži, napravama, ...)
- virtualno okruženje je u takvu sustavu samo jedan dodatni proces

10.5. Virtualno okruženje za skup procesa – spremnici



Slika 10.3. Virtualizacija na razini spremnika

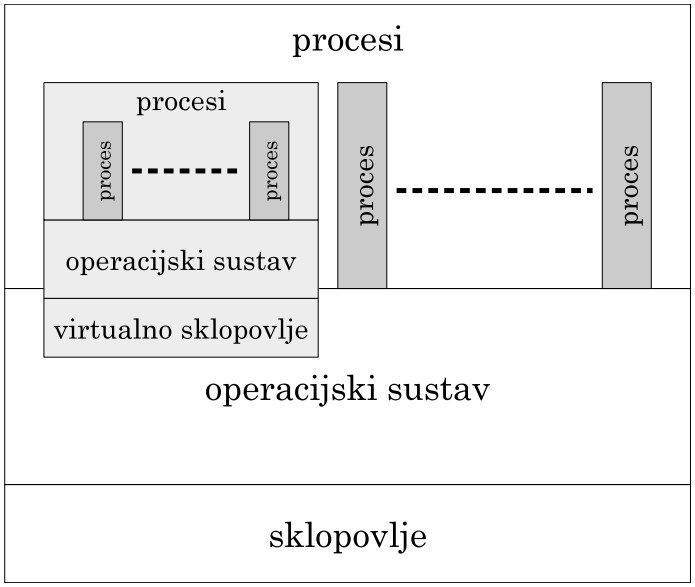
- podržavaju operacijski sustavi temeljeni na jezgri Linuxa (engl. *Linux Containers*), ali rade i na Windowsima uz WSL (Windows Subsystem for Linux – zapravo Linux u virtualnom okruženju)
- skup sredstava sustava se izolira/duplicira (disk, mreža, radni spremnik, procesori, ...)
- skup takvih sredstava nazovimo *spremnikom* ili kontejnerom (engl. *container*)
  - u kontekstu virtualizacije termin spremnik se ne odnosi na radni spremnik – memoriju računala, već na opisano virtualno okruženje
- u takvom okruženju pokreću se novi procesi
- procesi (preko jezgre) koriste pravo sklopovlje, samo su pojedina sredstva zasebna i ograđena – virtualizacija se ostvaruje pri korištenju jezgre za pristup takvim sredstvima
- temeljni operacijski sustav je jednak, ali korištenje sredstava je izolirano
- u svakom spremniku se može pokrenuti zasebni niz usluga
  - web poslužitelj, poslužitelj elektroničke pošte, korisnici, sve može biti u zasebnim spremnicima
  - različiti web poslužitelji mogu biti u različitim spremnicima (npr. pružatelji usluga iznajmljivanja web poslužitelja, mogu za svakog klijenta napraviti zasebni spremnik i u njemu dati prostor za web poslužitelj)
  - razne usluge u različitim spremnicima su izolirane – ne utječu jedne na drugu s aspekta sigurnosti
  - svaki spremnik može imati dodijeljeno i procesorsko vrijeme (npr. broj procesora, postotak procesorskog vremena), prostor u memoriji, vlastiti datotečni sustav koje je bar

u nekim dijelovima različiti od ostalih, svoj mrežni stog (adrese, način prosljeđivanja i slično), ...

- mehanizam spremnika se sve češće koristi ne samo radi virtualizacije i odvajanja već radi pakiranja svih potrebnih dodataka uz aplikaciju
  - uobičajeni način instaliranja aplikacije je da se neke dijeljene biblioteke koje su potrebne aplikaciji instaliraju na razini sustava (npr. DLL-ovi na Windowsima)
  - sa spremnikom se one sve mogu "upakirati" zajedno s programom – nije potrebno ih instalirati na razini sustava
  - na ovaj način se puno lakše pokreću aplikacije – ne treba ih instalirati već samo pokrenuti takav već pripremljeni spremnik (npr. Docker container)

10.6. Tipovi (prave) virtualizacije

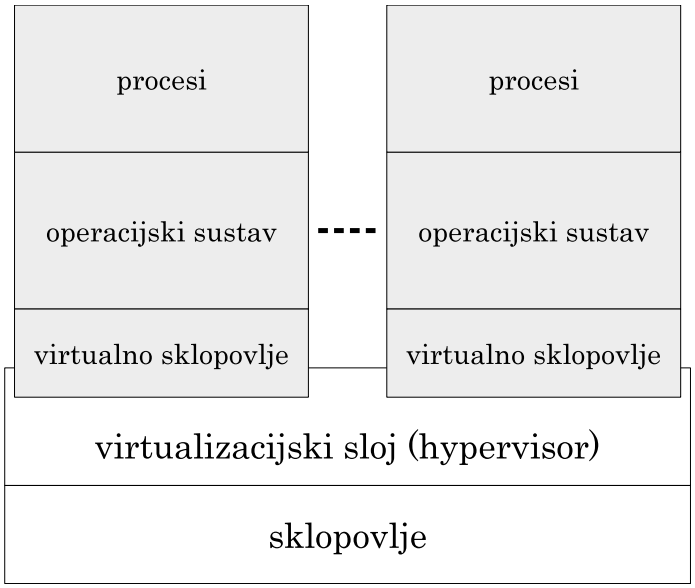
10.6.1. Tip–2: Virtualno okruženje na razini programa



Slika 10.4. Virtualizacija na razini programa

- nad operacijskim sustavom domaćina pokreće se hipervizor koji stvara virtualno okruženje – virtualno računalo, a u njemu se pokreće gostujući operacijski sustav
- *hipervizor je program* – dodatni program instaliran na "normalan" OS
- domaćin i gost mogu biti različiti operacijski sustavi (npr. domaćin je Windows 10, a gost Ubuntu)
- hipervizor osigurava pristup sklopovlju gostu korištenjem raznih mehanizama (stvaranjem virtualnih uređaja, dozvolom izravna pristupa nekom sklopovlju i slično)
- pristup sklopovlju prolazi kroz "debeli sloj": najprije u virtualiziranom gostujućem OS-u, a potom i kroz pravi OS domaćina – stoga se ponešto gubi na učinkovitosti pri intenzivnim UI operacijama
- studenti koji na svom računalu nemaju neki Linux/UNIX sustav mogu preko odgovarajućih programa (VMware, VirtualBox, WSL) uspostaviti virtualno računalo za laboratorijske vježbe za ovaj predmet

10.6.2. Tip–1: Virtualno okruženje na razini operacijskog sustava



Slika 10.5. Virtualizacija na razini operacijskog sustava i sklopovlja

- posebno pripremljeni operacijski sustav upravlja sklopovljem i ostvaruje hipervizor unutar sebe
- virtualizacijski sloj = OS + hipervizor → *hipervizor je u OS-u*
- OS+hipervizor predstavljaju tanak sloj između sklopovlja i virtualnih računala
- iako u virtualnom okruženju, učinkovitost operacijskih sustava znatno je veća nego kad se za virtualizaciju koristi operacijski sustav domaćina
- hipervizor omogućuje pokretanje više sustava iznad sebe, upravlja dodijeljenim sredstvima, stvara nova virtualna okruženja, ...
- ukupno virtualno dodijeljena sredstva (procesori, memorija) su najčešće i veća od dostupnih sredstava – hipervizor dinamički dodjeljuje sredstva, prema potrebama – ne koriste svi virtualni strojevi sve resurse u isto vrijeme (slično kao i kod straničenja, isto načelo se ovdje može primjenjivati za sva sredstva)
- primjeri: VMware ESX/ESXi, Xen Project, Oracle VM Server, Microsoft Hyper-V
- najčešće se koristi u poslužiteljskim centrima (podatkovnim centrima, sustavima koji pružaju infrastrukturu za računarstvo u oblaku)
  - virtualno računalo ovog tipa se može zakupiti (npr. Amazon, Google, Microsoft Azure)
- hipervizor mora imati mehanizme slične onima koje ima i operacijski sustav
  - virtualnim računalima treba dodijeliti procesorsko vrijeme, memoriju, ...
  - hipervizor zato često uključuje i jezgru OS s kojom je čvrsto povezan

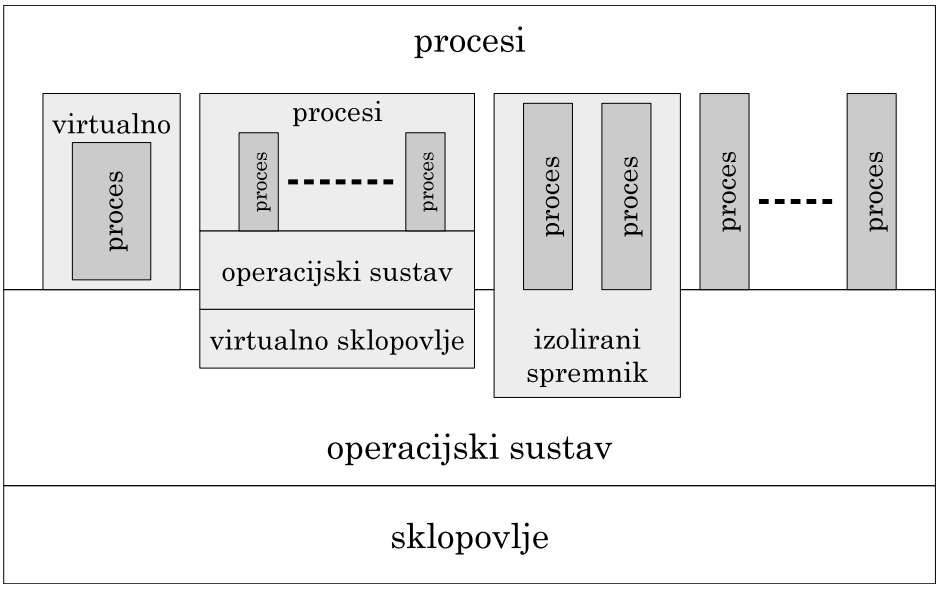
10.6.3. Tip–0: Virtualno okruženje na razini sklopovlja

- sličan tipu-1 (ista slika), ali je hipervizor značajno jednostavniji s manje mogućnosti dinamičkog upravljanja dodjelom sklopovlja
- hipervizor je ugrađen u upravljački sustav ("firmware") → *hipervizor je u sklopovlju*
- resursi sustava se statički dodjeljuju virtualnim računalima

- hipervizor samo sudjeluje u “mapiranju”
- npr. procesori 0-7 virtualnom računalu 1, procesori 8-9 virtualnom računalu 2, itd.
- s aspekta performansi ovo je najbolje (gotovo da i nema virtualizacije)
- ali zbog statičke dodjele resursa učinkovitost iskorištenja sredstava sustava je puno manja nego kod tipa-1

10.7. Korištenje različitih oblika virtualizacije istovremeno

Navedeni oblici se mogu kombinirati, koristiti paralelno ili čak jedan ugraditi u drugi.



Slika 10.6. Paralelno korištenje različitih tipova virtualizacije

10.8. Problemi ostvarenja virtualizacije

- osnovni problem: gostujuće računalo se izvodi u korisničkom načinu rada
- u gostujućem računalu se OS (njegova jezgra) želi ostvariti korištenjem privilegiranog načina rada
  - računalo domaćin ne smije dozvoliti da gost bude u privilegiranom načinu rada jer onda može kompromitirati i domaćina (i ostale programe/podatke na njemu)
  - pokušaj ulaska u privilegirani način rada iz program na gostu, kao i pokušaj izvođenja privilegiranih načina rada može se detektirati u računalu (operacijskom sustavu) domaćina (kao prekid zbog nedozvoljenih operacija)
    - način "uhvati-i-emuliraj"
    - u obradi takvih događaja, uz provjeru da se zaista radi o procesu koji omogućuje virtualno računalo, "problematične instrukcije/operacije" gostujućeg računala se ručno odrađuju – simulira se njihov rad
  - problem: neke instrukcije se drukčije izvode ovisno o načinu rada
    - npr. instrukcija POPF će u korisničkom načinu rada sa stoga obnoviti samo "obične zastavice" u registar stanja, dok će u privilegiranom načina rada obnoviti sve zastavice (i one koje utječu na prihvat prekida i na promjenu načina rada procesora!)
    - takva instrukcija neće izazvati prekid u korisničkom načinu rada procesora, neće ju se

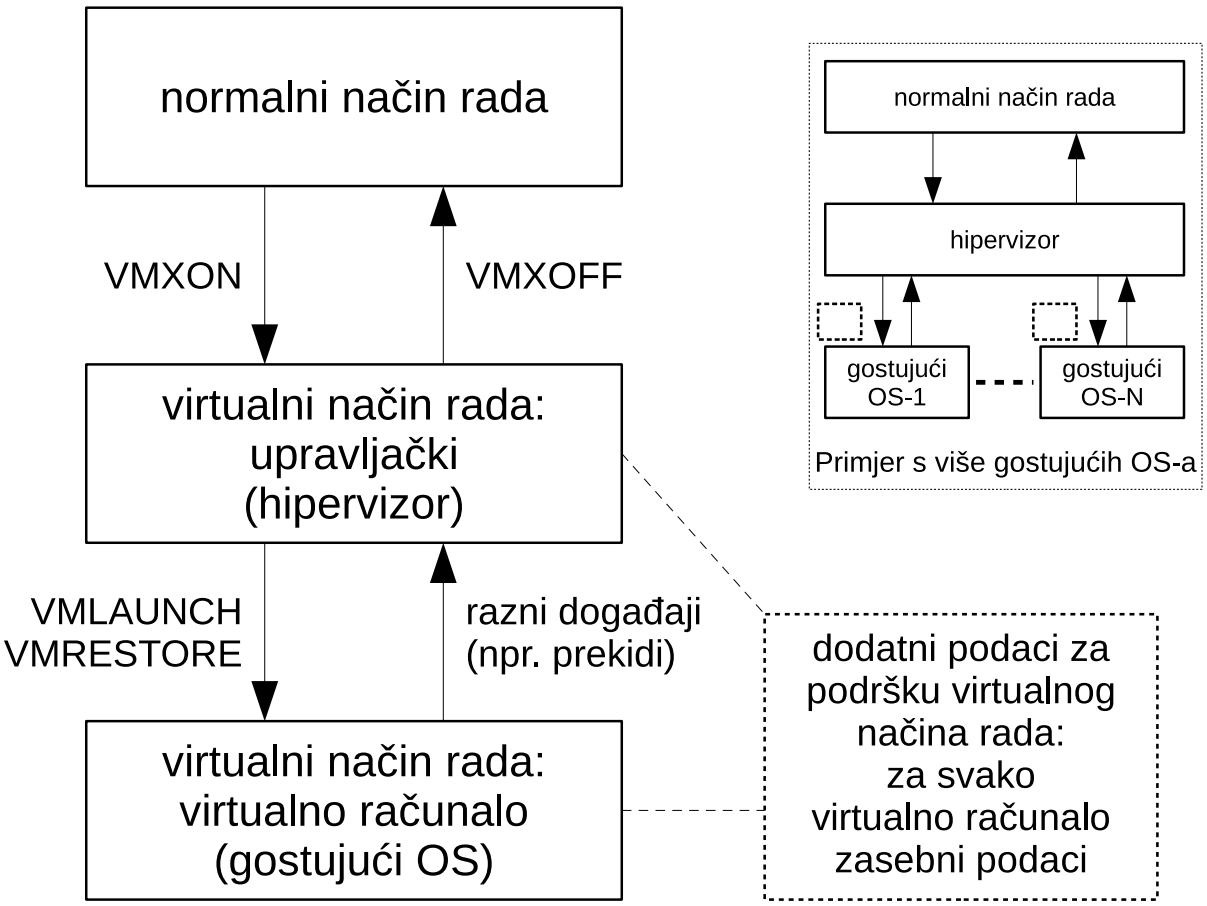
na taj način moći ispraviti

* stoga se pri zahtjevu za prelazak u jezgrin način rada u gostujućem računalu, što se neće dogoditi već treba simulirati, prije nastavka rada treba "proći" krozi instrukcije koje bi se obavile te po potrebi napraviti korekcije
  - binarna translacija
  - jedna instrukcija se zamjenjuje drugom ili čak i nizom instrukcija

- problem s upravljanjem spremnikom (straničenje!), rad s UI napravama, ...
- mnogi problemi se mogu riješiti ako procesor ima sklopovsku potporu za virtualizaciju

10.8.1. Sklopovska podrška virtualizaciji

- danas većina procesora ima nekakvu podršku virtualizaciji
- podrška virtualizaciji kod Intelovih procesora:
  - dodatni načini rada procesora i dodatne instrukcije i strukture podataka
  - osim dva normalnog načina rada (korisnički i privilegirani), procesor dodatno ima i dva "virtualna načina rada" za podršku virtualizacije
    1. upravljački virtualni način rada - za rad hipervizora
    2. virtualni način rada za gosta
       * u ovom načinu se lakše upravlja virtualim korisničkim i jezgrinim načinom rada
       * neke stvari procesor može sam napraviti ispravno, a one za koje to ne može prekida izvođenje i prepušta hipervizoru da to odradi za njega



Slika 10.7.

- posebnim instrukcijama se ulazi u te načine rada i pokreće virtualna računala (slika 10.7.)
- dodatna struktura podataka omogućuje pohrane/obnove ključnih podataka da virtualno računalo ne bi utjecalo na računalo domaćina i ostala virtualna računala
- slično upravljanju procesima u OS-u, samo što se ovdje radi o virtualnim računalima kojima upravlja hipervizor

Pitanja za vježbu 10

1. Navedite nekoliko razloga primjene virtualizacije.
2. Ukratko opišite virtualizacije tipa 0, 1 i 2 te navedite za koje se primjene koristi pojedini tip.
3. Što su to spremnici (kontejneri) te koje su njihove prednosti i nedostatci naspram virtualizacije (npr. tip-1)?
4. Navedite nekoliko problema u ostvarenju virtualizacije i postupke koji se koriste radi njih.
5. Što uključuje sklopovska potpora virtualizaciji (današnjih procesora)?
6. Što je to paravirtualizacija?

Literatura

[Budin, 2011] Leo Budin, Marin Golub, Domagoj Jakobović, Leonardo Jelenković, *Operacijski sustavi*, Element, 2011.

[Silberschatz, 2002] Abraham Silberschatz, Greg Gagne, Peter Baer Galvin, *Operating System Concepts, 6th edition*, Wiley, 2002.

[Tanenbaum, 2015] Andrew S. Tanenbaum, Herbert Bos, *Modern Operating Systems, Fourth Edition*, Prentice-Hall, 2015.

[Jelenković, 2010] Leonardo Jelenković, *Osnovni koncepti operacijskih sustava, prezentacija i primjeri*, 2010., http://www.zemris.fer.hr/∼leonardo/os/Osnovni_koncepti_OSa.

[NOS, 2024] Leonardo Jelenković, *Napredni operacijski sustavi: ulazno-izlazne naprave, višeprocesorski sustavi*, skripta za predavanje, 2024. https://www.zemris.fer.hr/ leonardo/nos/skripta/

[OSUR, 2024] Leonardo Jelenković, *Operacijski sustavi za ugrađena računala*, skripta za predavanje, 2024. https://www.zemris.fer.hr/ leonardo/osur/doc/

[SRSV, 2024] Leonardo Jelenković, *Sustavi za rad u stvarnom vremenu*, skripta za predavanje, 2024. https://www.zemris.fer.hr/ leonardo/srsv/skripta/

[POSIX] *POSIX, 8. izdanje*, 2024., https://pubs.opengroup.org/onlinepubs/9799919799/.

[Thread-Safety] *POSIX: Thread-Safety*, 2008., http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html.

[Linux] *The Linux Kernel Archives*, http://www.kernel.org.

[Linux, 2021] *Opisi raznih mehanizama u raspoređivačima Linuxa*, 2021. https://www.kernel.org/doc/Documentation/scheduler/

[Linux scheduling] *Linux Programmer's Manual: sched – overview of scheduling APIs*, http://man7.org/linux/man-pages/man7/sched.7.html.

[Futex] *Futex – fast user-space locking*, 2011., https://man7.org/linux/man-pages/man2/futex.2.html.