

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

Skripta iz predmeta

Napredni operacijski sustavi

Ulazno-izlazne naprave,
višeprocessorski sustavi

Leonardo Jelenković

Zagreb, 2022.

Sadržaj

I Upravljački programi	1
1. Korištenje naprava preko operacijskog sustava	3
1.1. Sučelja OS-a za korištenje naprava	5
Pitanja za vježbu	7
2. Jednostavni model OS-a za upravljanje napravama	9
2.1. Radno čekanje, prekidi, DMA	9
2.2. Jednostavna jezgra OS-a	11
2.3. Neki mehanizmi stvarnih sustava	13
Pitanja za vježbu	15
3. Model naprava	17
3.1. Opći model naprave	17
3.2. Pristup napravama preko adresa	19
3.3. Korištenje međuspremnik	19
Pitanja za vježbu	24
4. Primjeri protokola za komunikaciju s napravama	27
4.1. Serijska i paralelna veza – usporedba svojstava	27
4.2. Serijska veza RS-232	27
4.3. Paralelna veza LPT	28
4.4. USB	29
4.5. PCI Express	34
4.6. Serial ATA	39
Pitanja za vježbu	40
5. Podrška za ostvarenje naprava u Linuxu	43
5.1. Izvođenje jezgrina koda	44
5.2. Dozvoljene i nedozvoljene operacije u kodu jezgre	46
5.3. Moduli	46
5.4. Upravljački programi	47
Pitanja za vježbu	50

6. Primjeri upravljačkih programa u Linuxu	53
6.1. Priprema radnog okruženja	53
6.2. Moduli	53
6.3. Jednostavna virtualna znakovna naprava	55
6.4. Logiranje, parametri, liste, odgoda	62
6.5. Alarm, red poslova	65
6.6. Višedretvena obrada prekida	68
6.7. Sučelja <code>poll</code> i <code>ioctl</code>	69
6.8. Dodatne operacije s napravama	72
6.9. Kako spriječiti i popraviti “oops”	73
Pitanja za vježbu	74
II Višeprocessorski sustavi	77
7. Svojstva višeprocessorskih sustava	79
7.1. Zašto su višeprocessorski sustavi danas svugdje?	79
7.2. Osnovne vrste višeprocessorskih sustava	79
7.3. Dodatne metode povećanja učinkovitosti	82
7.4. Okruženja primjene višeprocessorskih sustava	83
Pitanja za vježbu	84
8. Problemi pri ostvarenju operacijskog sustava za višeprocessorske sustave	85
8.1. Očuvanje konzistentnosti podataka kroz paralelni rad	85
8.2. Strukture podataka	91
Pitanja za vježbu	92
9. Raspoređivanje dretvi u višeprocessorskim sustavima	93
9.1. Ukratko o raspoređivačima	93
9.2. Posebnosti višeprocessorskih sustava	94
Pitanja za vježbu	96
10. Primjeri iz implementacije raspoređivanja u Linuxu	99
10.1. Raspoređivači	99
10.2. Raspoređivač CFS	100
Pitanja za vježbu	103
Literatura	105

Dio I

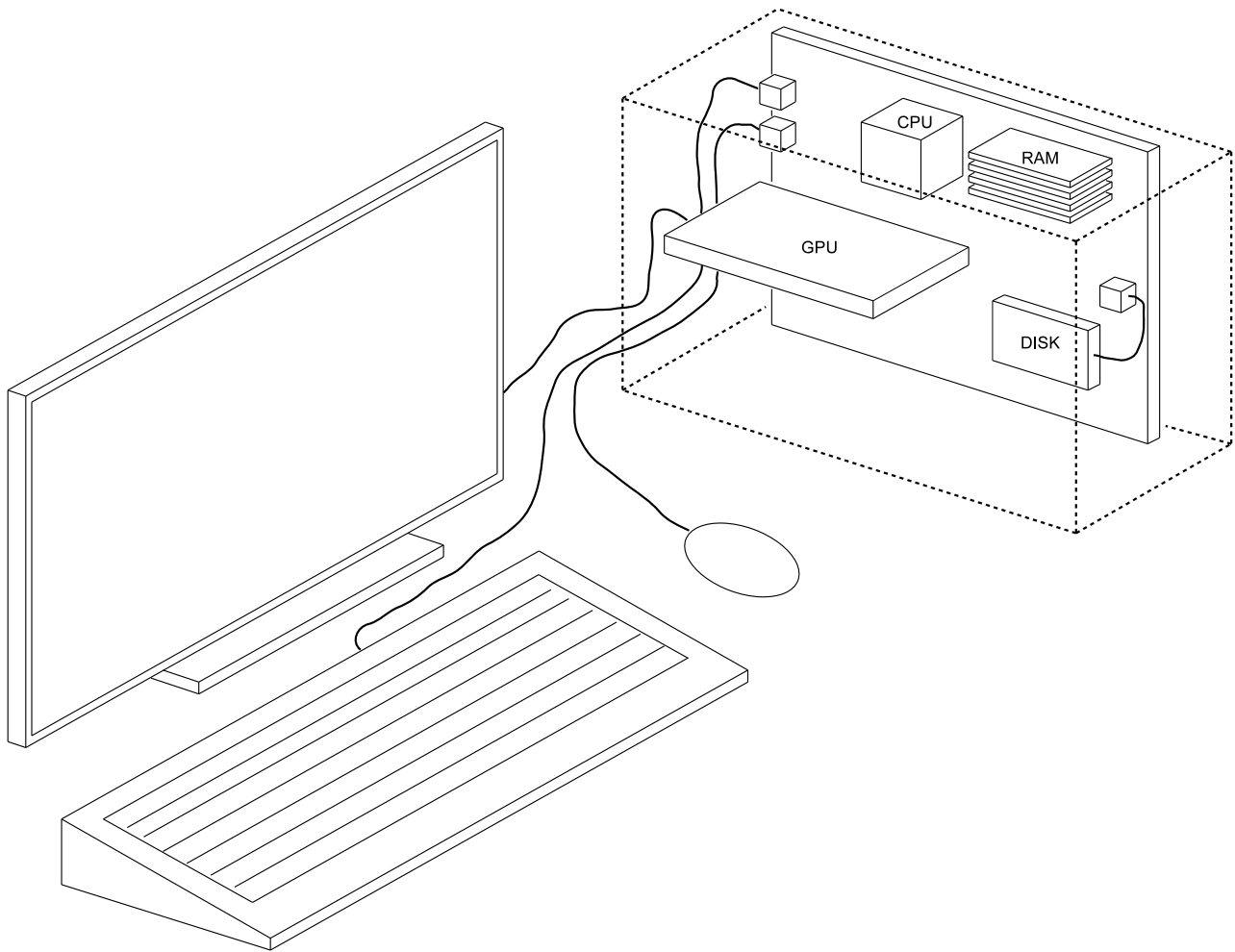
Upravljački programi

Sažetak

U prvom dijelu skripte se opisuje korištenje naprava u računalima. Najprije su prikazana sučelja operacijskog sustava kojima programi mogu koristiti naprave. Slijedi kratki opis jednostavnog modela jezgre za upravljanje napravama koji je već prikazan u okviru predmeta Operacijski sustavi i nedostatci tog modela. U idućem poglavlju dan je prikaz modela naprave i mogućih načina korištenja naprava. Slijede primjeri naprava, tj. protokola za komunikaciju s napravama (serijska veza, USB, PCI-E, SATA). Zadnja dva poglavlja prikazuju elemente operacijskog sustava Linux koji se odnose na upravljanje napravama te primjere izrade i pokretanja upravljačkih programa.

1. Korištenje naprava preko operacijskog sustava

U kontekstu računarstva, pod pojmom *naprava* podrazumijevamo dodatne elemente računala (sklopovlja) koji, uz procesor i radni spremnik, omogućuju potrebne funkcionalnosti. Primjerice, tipkovnica, miš, zaslon, pisač i slično su vanjske naprave koje služe za interakciju korisnika s računalom, disk, mrežna kartica, grafička kartica su unutarnje komponente koje dodaju svoje mogućnosti sustavu (pohranu podataka, komunikaciju s drugim računalima, oblikovanje prikaza za zaslon). Vanjske naprave se na računalo spajaju preko prikladnih priključaka (npr. USB, HDMI) koji se onda unutar računala povezuju s odgovarajućim međusklopovima (upravljačkim sklopovima, kontrolerima) koji rade translaciju protokola u nešto što se može koristiti unutar računala.



Slika 1.1. Računalo s vanjskim i unutarnjim napravama

- Naprave se mogu podijeliti u kategorije po različitim svojstvima:
 - fizičke dimenzije, načine spajanja na/u računalu
 - brzina rada
 - smjer podataka: ulazna, izlazna, ulazno-izlazna
 - dohvat podataka: znak po znak ili blokovi podatak (npr. sektor, paket)
 - čitanje je slijedno ili je moguć dohvat bilo kojeg podatka
 - sinkrono ili asinkrono (čeka se na završetak operacije ili ne)

- može li se naprava istovremeno koristiti od strane više dretvi ili ne
- OS naprave svrstava u zasebne klase (npr. na Linuxu su klase: znakovne, blokofske, mrežne)

Naprave su vrlo raznolike, različitih svojstava. Neki primjeri naprava i njihovih brzina:

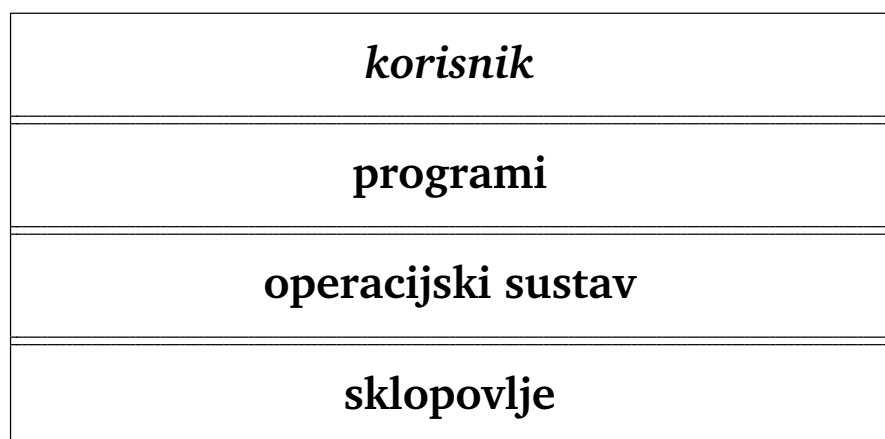
- tipkovnica i miš (USB): 125-1000 Hz
- audio podsustav (analogni ulaz/izlaz): 44, 48, 96, ... kHz
- disk (SATA): čitanje jednog podatka ~ 10 ms za HDD (~ 0.05 ms za SSD); prijenos kompaktno smještenih podataka ~ 100 MB/s (i više)
- mrežna kartica (PCI): npr. 1 Gbit/s \Rightarrow 125 MB/s; odziv mreže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

Iz perspektive operacijskog sustava (OS-a) upravljanje napravama obavlja se preko upravljačkih sklopova na koje su naprave spojene. Npr. upravljanje tipkovnicom ide preko USB međusklopa, upravljanje diskom preko SATA kontrolera i slično. Stoga se u ovim materijalima pojednostavnjuje terminologija i pod pojmom naprava se najčešće misli na upravljački sklop na koji je naprava spojena, a ne na samu napravu.

Upravljanje napravama ostvaruje se kroz *upravljačke programe naprava* (engl. *device drivers*). Međutim, obzirom da operacijski sustavi već nude mnoge generičke usluge za upravljanje napravama (u sklopu podustava za naprave), dodatno potreban “upravljački program” koji se postavlja (instalira) s novom napravom samo popunjava potrebne “rupe” koje generičke usluge ne pokrivaju, a specifične su za napravu.

Prikaz povezivanja upravljačkog programa s operacijskim sustavom prikazano je u idućim poglavljima. U ovom poglavlju razmatra se samo korištenje naprava iz korisničkih programa preko sučelja koje mu operacijski sustav nudi.

- Naprave se iz programa koriste kroz usluge operacijskog sustava.
- OS nudi sučelje za korištenje naprava (skriva naprave)



Slika 1.2. Slojevi računalnog sustava

- OS upravlja napravama:
 - operacije OS-a (jezgrine funkcije) izvode se u privilegiranom načinu rada koji je potreban za manipulaciju napravama (programi/procese se izvode u korisničkom načinu rada)
 - OS zna kako ih koristiti preko zasebnih “upravljačkih programa”

- OS programima dozvoljava korištenje naprava kroz jednostavno sučelje
- OS rješava probleme “paralelnog korištenja” od strane više programa
- OS kontrolira koji programi smiju koristiti koje naprave

1.1. Sučelja OS-a za korištenje naprava

Naprave se iz programa uglavnom koriste preko podsustava kome pripadaju. Primjerice, disk se koristi kroz datotečni podsustav, mrežna kratica preko sučelja mrežnog podsustava, zaslon preko grafičkog podsustava i slično. Tek se naprave koje nisu već pridjeljene u neki podsustav mogu “izravno” koristiti preko generičkog sučelja, sučelja koje se razmatra u nastavku. I ovdje je “izravno” zapravo kroz operacijski sustav koji gotovo sve skriva od programa i (korištenjem upravljačkih programa naprave) na jednostavan način omogućuje korištenje naprave iz programa.

U nastavku su prikazani primjeri sučelja za rad s napravama uz kratki opis funkcionalnosti gdje to nije očito iz imena i argumenata.

1.1.1. Osnovne operacije

Osnovne operacije za rad s napravama su slične kao i za rad s datotekama: otvori, čitaj/piši, zatvori.

- `int open(const char *pathname, int flags);`
- `int open(const char *pathname, int flags, mode_t mode);`
- `int close(int fd);`
- `ssize_t read(int fd, void *buf, size_t count);`
- `ssize_t write(int fd, const void *buf, size_t count);`

1.1.2. Dodatne operacije

Dodatne operacije omogućuju odabir tražene informacije (pomak), upravljanje s više naprava odjednom (čekanje/provjera da se bar negdje nešto dogodi), stvaranje posebnih datoteka u datotečnom sustavu koje su povezane s napravama, postavljanje posebnih svojstava takvim datotekama te slanje naredbi upravljačkom programu naprava.

- `off_t lseek(int fd, off_t offset, int whence);`
 - pomak po podacima (ako to naprava podržava)
- `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);`
 - provjera/čekanje da se nešto negdje dogodi na nekom od otvorenih opisnika datoteka/naprava/*
- `int poll(struct pollfd *fds, nfds_t nfds, int timeout);`
 - provjera/čekanje da se nešto negdje dogodi na nekom od otvorenih opisnika datoteka/naprava/*

- `int fsync(int fd);`
 - pražnjenje međuspremnik koji se koriste za komunikaciju s napravama (slanje podataka napravi ako ima još nešto u međuspremniku)
- `int mknod(const char *pathname, mode_t mode, dev_t dev);`
 - stvaranje posebnih tipova datoteka (npr. cjevovod, poveznica za naprave i sl.)
- `int fcntl(int fd, int cmd, ... /*arg */);`
 - operacije nad opiscima datoteka (npr. prava pristupa)
- `int ioctl(int fd, unsigned long request, ...);`
 - upravljanje napravama (izravno slanje naredbi upravljačkom programu naprave)
- `ssize_t readv/writev(int fd, struct iovec *iov, int iovcnt);`
 - čitaj/piši u/iz više međuspremnik

1.1.3. Asinkrone operacije

Osim uobičajenog načina rada s datotekama i napravama koji uključuje blokiranje dretve dok se operacija ne obavi do kraja, postoji i drukčije sučelje koje ne čeka kraj operacije i na taj način (u nekim slučajevima) omogućuje efikasniji rad.

- Ideja: pokreni operaciju, ali ne čekaj njen kraj
- Nekoliko načina da se čeka/dozna za kraj operacije
 - čekaj dodatnim pozivom
 - provjeravaj status
 - primitak signala
- Struktura u kojoj se zadaje asinkrona operacija:

```
struct aiocb {
    int          aio_fildes;      /* File descriptor */
    off_t        aio_offset;     /* File offset */
    volatile void *aio_buf;      /* Location of buffer */
    size_t       aio_nbytes;     /* Length of transfer */
    int          aio_reqprio;     /* Request priority */
    struct sigevent aio_sigevent; /* Notification method */
    int          aio_lio_opcode;  /* Operation to be performed;
                                lio_listio() only */
    ...
};
```

- `int aio_read/aio_write(struct aiocb *aiocbp);`
 - započni asinkronu operaciju čitanja/pisanja
- `int aio_fsync(int op, struct aiocb *aiocbp);`
 - sinkroniziraj operacije ("zapiši iz međuspremnik na naprave")
- `int aio_error(struct aiocb *aiocbp);`
 - dohvati grešku povezanu s zadanom operacijom

- `ssize_t aio_return(struct aiocb *aiocbp);`
– dohvati status operacije
- `int aio_suspend(struct aiocb *aiocb_list[], int nitems, struct timespec *timeout);`
– čekaj kraj operacija
- `int aio_cancel(int fd, struct aiocb *aiocbp);`
– zaustavi operacije
- `int lio_listio(int mode, struct aiocb *const aiocb_list[], int nitems, struct sigevent *sevp);`
– zadaj više operacija s jednim pozivom

Pitanja za vježbu 1

1. Zašto se naprave ne koriste izravno iz programa već preko operacijskog sustava? Koji su sve razlozi (implementacijski, sigurnosni, ...)?

Odgovor:

implementacijski: samo OS treba imati upravljačke programe, programi koriste naprave kroz OS jednostavnim sučeljem (ne trebaju poznavati detalje naprave); nadalje potrebno je da procesor bude u privilegiranom načinu rada za rad s IO napravama; u tom načinu mu je sve dostupno (sve adrese, svi registri i slično)

*sigurnosni: ne dopustiti svim aplikacijama izravno korištenje svim napravama – programi mogu imati greške ili biti iskorišteni od strane zlonamjernih osoba
ostalo: OS omogućava da se naprave koriste od strane svih procesa, ali kroz kontrolirano okruženje*

2. Naprave se kroz operacijski sustav koriste (uglavnom) istim sučeljima kao i datoteke. Međutim, neka od tih sučelja imaju upravljačku ulogu pri komunikaciji s upravljačkim programom napravama. Navesti neke od takvih sučelja i kratko navesti čemu služe.

Odgovor:

*ioctl – slanje „naredbi“ upravljačkom programu iz korisničkog programa
poll – provjeravanje/čekanje ima li koja naprava iz liste nešto novo
mknod – stvaranje datoteke za napravu
fsync – pražnjenje međuspremnika korištenog pri komunikaciji s napravom
aio_* – asinkrone operacije*

3. Što se može preko funkcije `poll`, a što preko `ioctl`?

Odgovor:

*poll – provjeravanje/čekanje ima li koja naprava iz liste nešto novo
ioctl – slanje „naredbi“ upravljačkom programu iz korisničkog programa*

4. Što su to asinkrone operacije s napravama?

Odgovor:

Asinkrone operacije su operacije čiji se kraj ne mora čekati – naredba se zada, ali se tada ne čeka na njen završetak već se nastavlja s radom (s drugim poslom). Naknadno se posebinim sučeljem provjerava je li operacija dovršena.

5. Navesti nekoliko mogućih podjela naprava ovisno o svojstvima, načinu spajanja, načinu komunikacije, smjeru podataka,

Odgovor:

opća podjela: naprave koje ostvaruju sučelje prema korisniku, naprave za spremanje podataka, naprave za komunikaciju

način spajanja: serijska veza, paralelna veza

način komunikacije: sinkrono – nakon svake operacije (miš, tipkovnica, audio, zaslon), asinkrono – može se poslati/primiti više zahtjeva (disk)

smjer podataka: ulazna (tipkovnica), izlazna (zaslon), ulazno-izlazna (disk)

način razmjene podataka: znakovna naprava – jedinica podataka je oktet koji se slijedno čita/piše (nema preskakanja ili dohvata proizvoljnog podatka), blok naprava – jedinica podataka je blok, mrežna naprava...

dijeljenje naprave među programima: samo jedan program istovremeno (npr. pisač), više programa istovremeno (audio, zaslon)

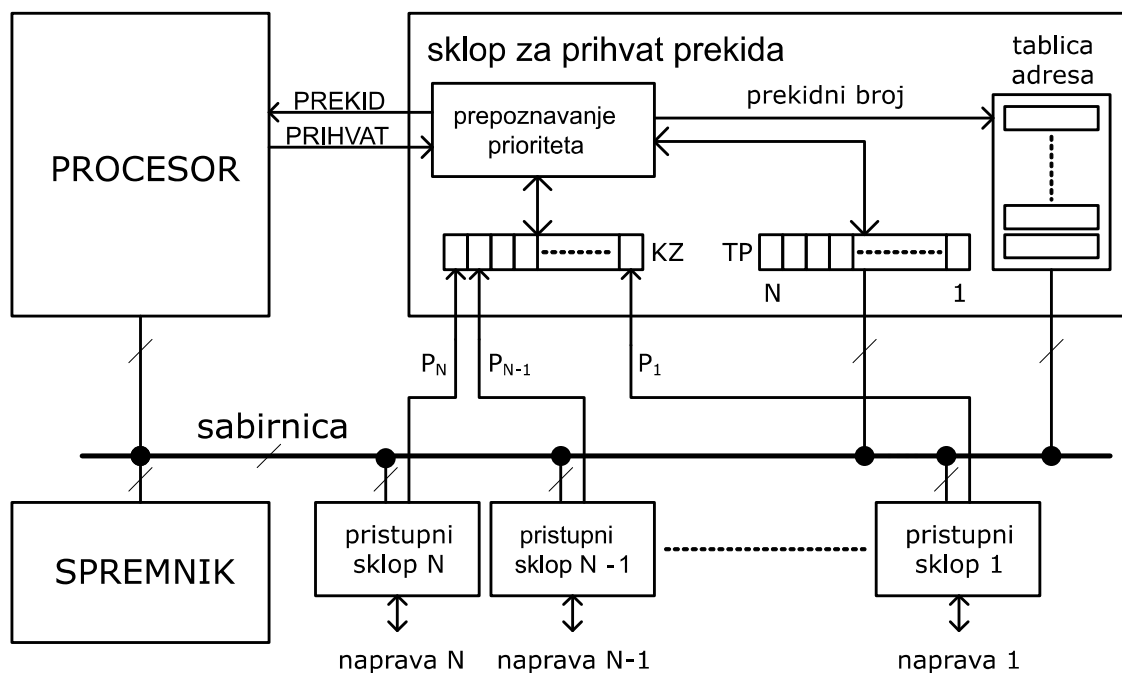
2. Jednostavni model OS-a za upravljanje napravama

Sadržaj ovog poglavlja je većinom sažetak već prikazanog u okviru predmeta Operacijski sustavi. Ovaj prikaz pretpostavlja jednostavne naprave koje mogu operaciju obaviti jednokratnim "programiranjem" pristupnog sklopa naprave.

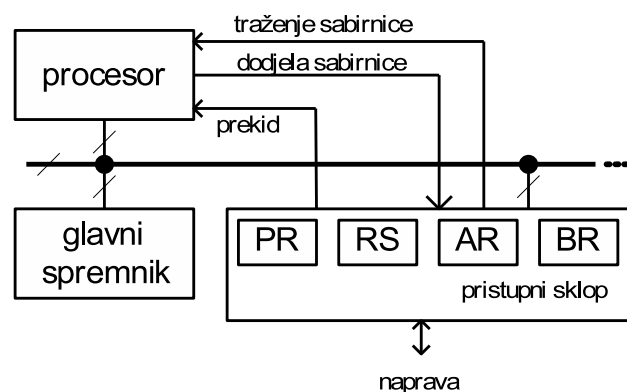
2.1. Radno čekanje, prekidi, DMA

Osnovni načini upravljanja napravama:

- radno čekanje – petlja u kojoj se čeka da naprava bude spremna, tj. petlja u kojoj se neprestano provjetava statusni registar naprave
- prekidi – naprava javlja kad je spremna (slika 2.1.)
- izravan pristup spremniku (engl. *direct memory access* – *DMA*) – naprava sama prenosi podatke u radni spremnik ili iz njega uzima podatke (slika 2.2.)



Slika 2.1. Procesor sa sklopom za prihvat prekida



Slika 2.2. Sklop s izravnim pristupom spremniku

Primjer Intelove arhitekture za prihvata prekida (pojednostavljeno)

a) Prihvata prekida od strane procesora

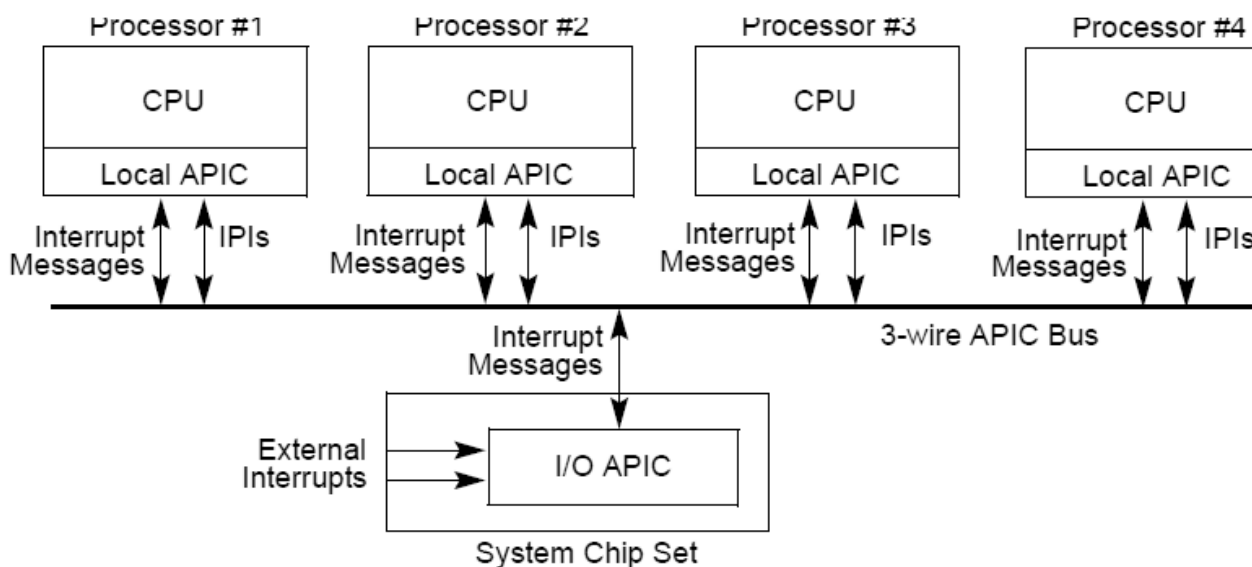
- Obrada prekida se programira preko zasebne tablice – IDT (interrupt description table)
- U registar procesora IDTR se stavlja adresa te tablice (IDTR dostupan samo u prekidnom, tj. privilegiranom načinu rada procesora)
- Kad se dogodi prekid, uz prekid dolazi i informacija o uzroku – broj prekida → N
- Broj prekida se koristi kao indeks za IDT, uzima se N -ti redak u kojem piše koju funkciju treba pozvati za obradu tog prekida

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

- Kratica APIC – advanced programmable interrupt controller
- Uz procesor se nalazi 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

Upravljanje prekidima naprava, npr. koji se prihvaćaju a koji ne:

- na razini procesora (zastavica IE)
- 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



Slika 2.3. Lokalni APIC i I/O APIC na višeprocorskim sustavima (izvor Intel)

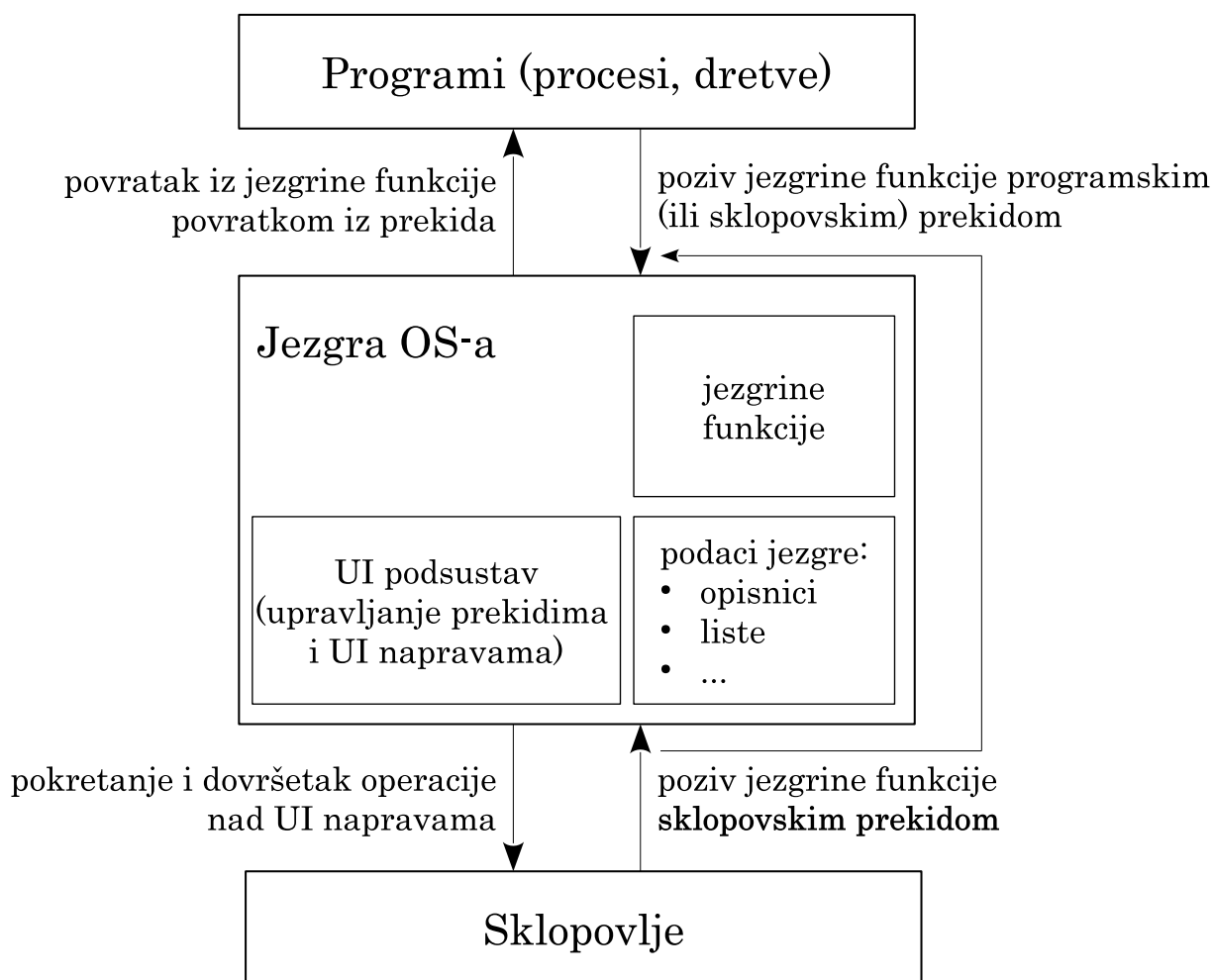
2.2. Jednostavna jezgra OS-a

Jezgrine funkcije pozivaju se mehanizmom prekida:

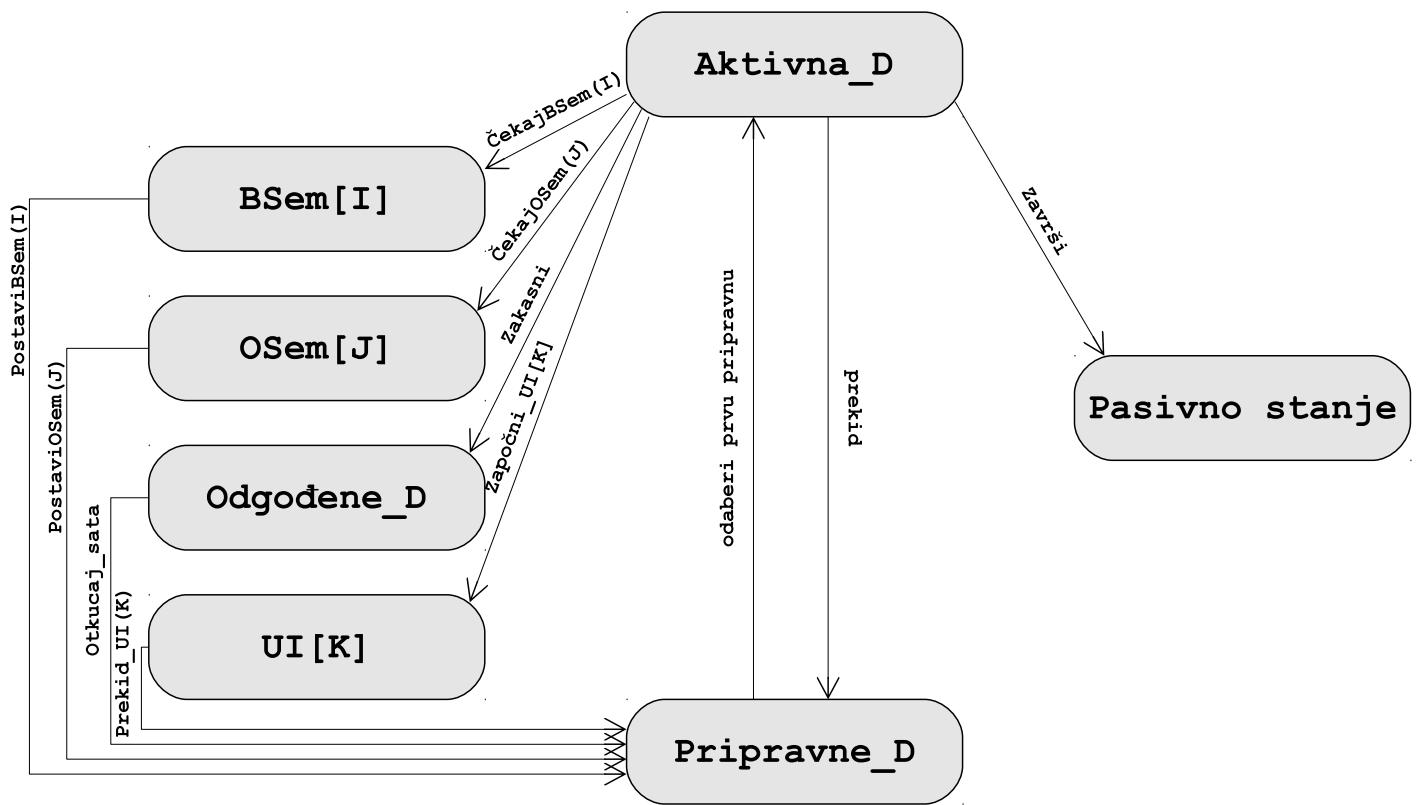
- naprave izazivaju sklopovske prekide kada se na njima nešto dogodi
- programi posebnim instrukcijama izazivaju programske prekide kada žele pozvati neku jezgrinu funkciju
- na ovaj način su jezgrine funkcije i strukture podataka jezgre zaštićene od izravnog korištenja iz programa (uz pretpostavku korištenja nekog oblika upravljanja memorijom).

Nositelji aktivnosti u sustavu su dretve koje obavljaju za korisnika potrebne poslove (ili pomoćne poslove potrebne korisničkim dretvama).

- korisnik preko sučelja OS zadaje naredbu pokretanja programa
- OS stvara proces i u njemu početnu dretvu
- ta dretva u nekom trenutku dobiva procesor (kad na nju dođe red prema odabranim načinima raspoređivanja dretvi)
- dretva izvodi instrukcije, ali samo one koje su joj dozvoljene
- za sve povlaštene operacije treba pomoć OS-a kojeg poziva mehanizmom programskog prekida



Slika 2.4. Mehanizam poziva jezginih funkcija



Slika 2.5. Moguća stanja dretve, jezgrine funkcije

- Primjeri jednostavnih jezgrinih funkcija za upravljanje UI napravama

```

j-funkcija ZAPOČNI_UI(K, parametri)
{
    stavi_u_red(makni_prvu_iz_reda(Aktivna_D), UI[K])
    pokreni UI operaciju na napravi K
    stavi_u_red(makni_prvu_iz_reda(Pripravne_D), Aktivna_D)
}

```

```

j-funkcija PREKID_UI(K)
{
    stavi_u_red(makni_prvu_iz_reda(Aktivna_D), Pripravne_D)
    dovrši UI operaciju na napravi K
    stavi_u_red(makni_prvu_iz_reda(UI[K]), Pripravne_D)
    stavi_u_red(makni_prvu_iz_reda(Pripravne_D), Aktivna_D)
}

```

- Gornje funkcije koriste pretpostavku da naprava obrađuje zahtjeve prema redu prispjeća
- Neki problemi prikazanog pojednostavljenja:
 - naprave možda ne moraju posluživati zahtjeve po redu prispjeća
 - operacija tražena od procesa može zahtijevati više UI operacija, možda i s više naprava

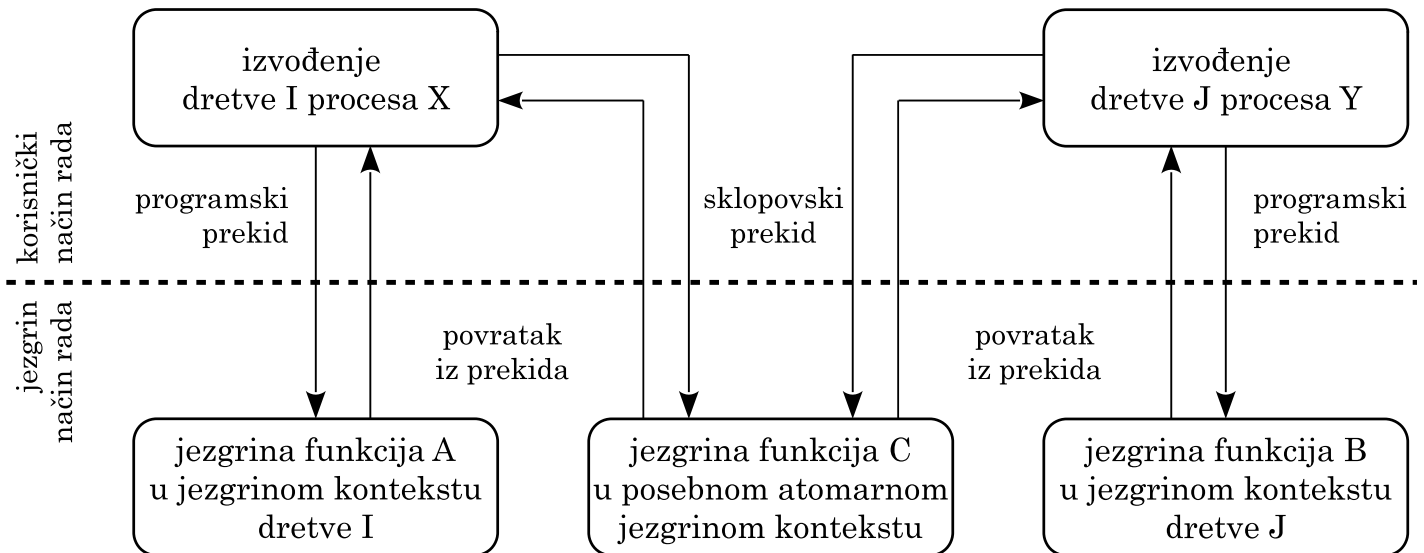
2.3. Neki mehanizmi stvarnih sustava

1. Redovi zahtjeva za UI operacije

- svaki zahtjev se oblikuje kao struktura i ide u listu zahtjeva
- obrada pojedinog zahtjeva može tražiti izvođenje više operacija
- tek kad je zahtjev gotov (odrađen do kraja od strane naprave i jezgre), dretva koja je dala taj zahtjev može nastaviti s radom
- zahtjev tako može duže ostati u sustavu, tražiti složenije operacije nad napravama

2. Jezgrin kontekst dretve

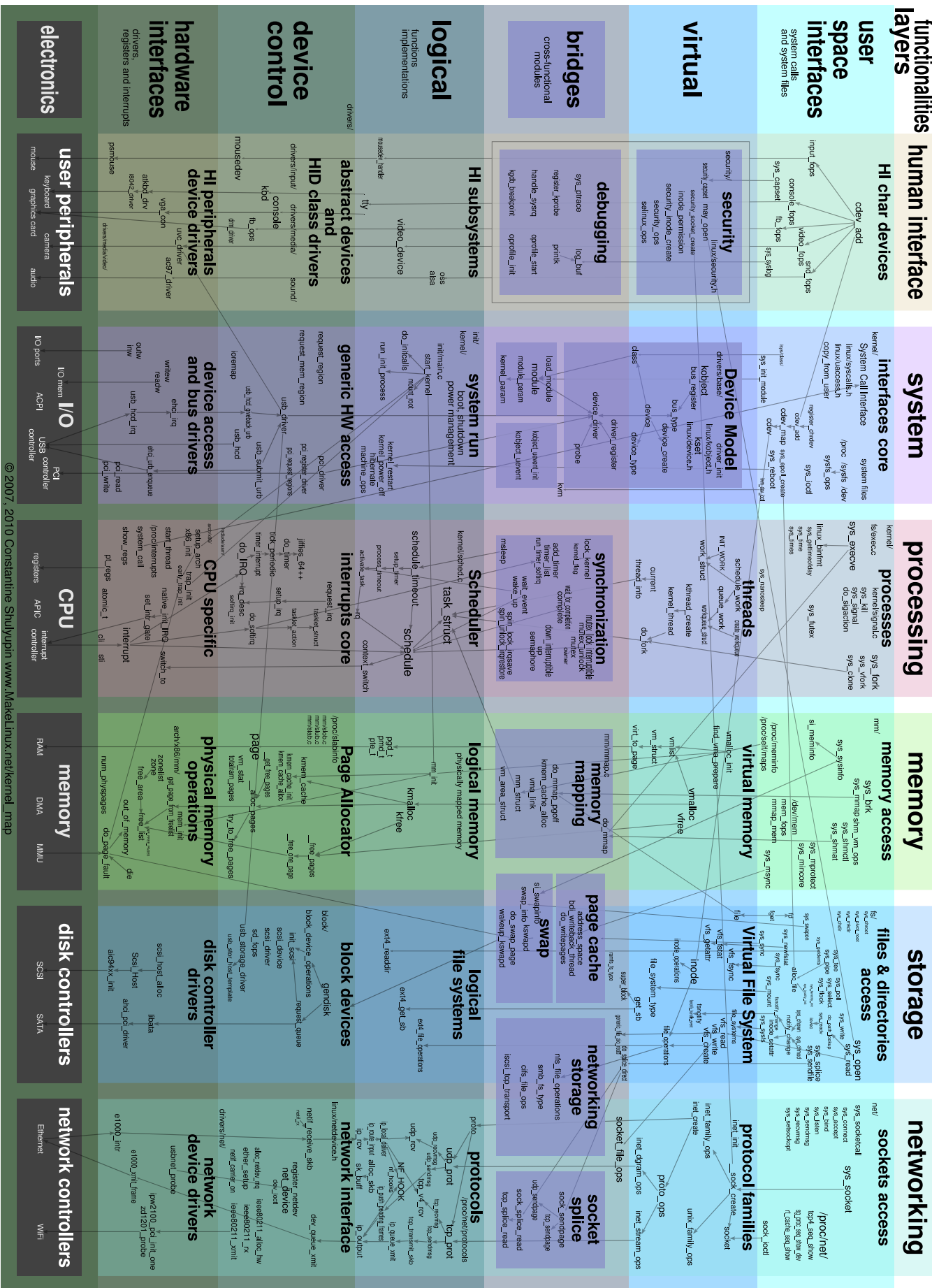
- nakon ulaska u jezgru mehanizmom programskog prekida, aktivira se jezgrin kontekst za zadanu dretvu
- takva jezgrina dretva može biti blokirana u jezgri
- pri blokiranju takve dretve, neka druga dretva nastavlja s radom (jezgrina ili korisnička)
- istovremeno se može izvoditi više jezgrinih funkcija ako im nisu potrebna ista sredstva (zaključavanje se svodi na minimum)



Slika 2.6. Primjer prelaska u jezgrin način rada programskim i sklopovskim prekidima

- Stvarni sustavi su značajno složeniji, i u sklopovlju i programskoj potpori
- Složenost se u stvarnim sustavima “rješava” podjelom na podsustave i slojeve
- Slika 2.7. prikazuje internu organizaciju Linuxa, podjeljenog na podsustave i slojeve

Linux kernel map



Slika 2.7. Interna arhitektura jezgre Linuxa (http://www.makelinux.net/kernel_map/)

Pitanja za vježbu 2

1. Opisati osnovna načela upravljanja radnim čekanjem, prekidima te korištenjem sklopova s izravnim pristupom spremniku (DMA).

Odgovor:

radno čekanje: u petlji se provjerava ima li promjene na nekom ulazu; kada ima onda obraditi taj ulaz (dohvatiti podatak, poslati podatak, ...)

prekidi: raditi nešto drugo (nešto korisno, "glavni program", neka dretva) dok nema ništa na ulazima; kad se nešto negdje pojaviti ta naprava generira zahtjev za prekid: on privremeno prekida dretvu koja se izvodi, sprema se njen kontekst, skače se na funkciju za obradu prekida, prije povratka se obnavlja kontekst prekinute dretve

DMA: programi se DMA sklop da kad dođe podatak taj sklop traži od procesora upravljanje sabirnicom i kad je dobije prenese podatak u memoriju ili obratno

2. U jednostavnom modelu jezgre ulazno-izlazne operacije može se pretpostaviti da naprava obavlja zadane joj naredbe slijedno. Stoga bi procesi koji su tražili takve operacije mogli biti u jednom uređenom redu, čekajući dovršetke svojih operacija. Koji su problemi ovog modela zbog specifičnosti stvarnih sustava? Kako se u stvarnim sustavima rješavaju takvi problemi, tj. ostvaruju složenije operacije u jezgri?

Odgovor:

Operacije koje programi zadaju preko sučelja OS-a može zahtijevati više UI operacija nad napravama. Stoga se takve operacije obavljaju na dva načina (ovisno o OS-u, operaciji):

1. pri zahtjevu za operacijom stvara se opisnik u kojem piše što treba napraviti; taj se opisnik onda stavlja u red i obrađuje (dio po dio); tek kad je obrađen do kraja propušta se dretva koja je tražila tu operaciju (tad je operacija gotova)

2. pri ulasku u jezgru prelazi se u jezgrin kontekst/dretvu koja je povezana s procesom pozivajuće dretve; u takvom kontekstu se ta jezgrina dretva može i blokirati pri čekanju na završetak dijela operacije

3. Kako se rješava problem složenosti ostvarenja operacijskog sustava u stvarnim sustavima, npr. Linuxu?

Odgovor:

princip "podijeli i vladaj"

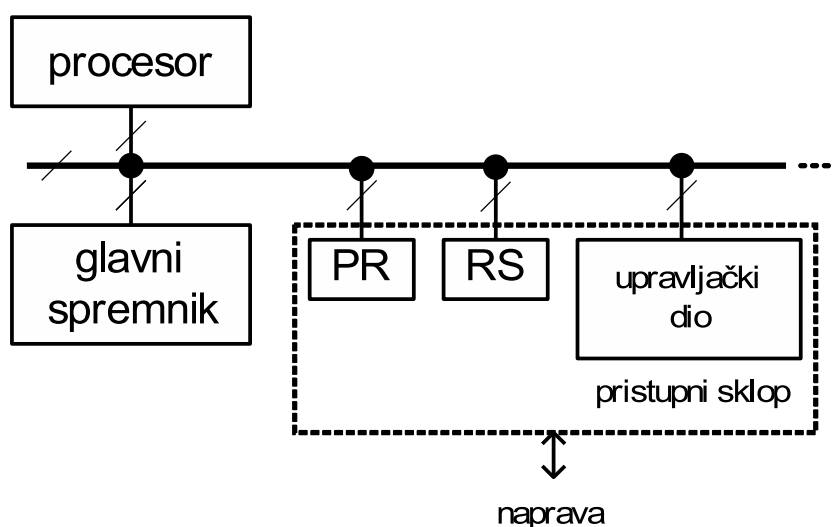
OS se dijeli na podsustave i slojeve

pod sustavi: UI, memorija, procesi/dretve (raspoređivanje, sinkronizacija, komunikacija, ...), datotečni podsustav, mrežni, ...

3. Model naprava

3.1. Opći model naprave

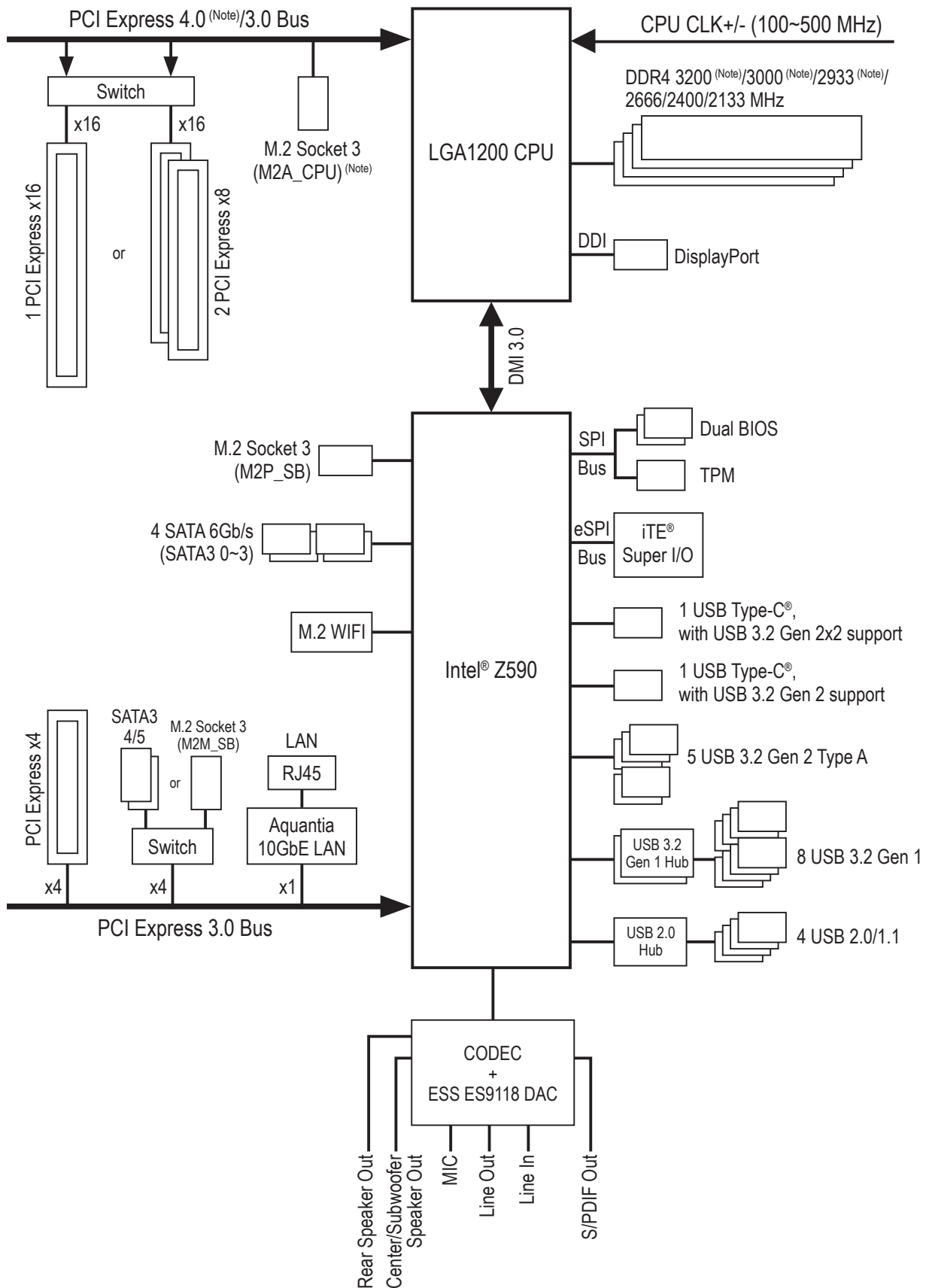
- Za pristup napravi se koriste ili adrese ili UI vrata (portovi)
 - obične adrese kao i za pristup memoriji (najčešće korišteni mehanizam)
 - * npr. `mov r0, [0xe800111c]`
 - UI vrata s posebnim instrukcijama za pristup
 - * npr. `in r0, 0xf8`
- Za upravljanje napravama koriste se njeni registri: upravljački, podatkovni, statusni



Slika 3.1. Jednostavni model pristupnog sklopa UI naprave

- Stvarni sustavi su uglavnom složeniji od primjera na slici 3.1., tj. najčešće imaju više registara i memorijskih lokacija
- Napravama se pristupa izravno ili preko drugog međusklopa/kontrolera (npr. preko PCI kontrolera, USB međusklopa, ...)
- Slika 3.2. prikazuje jednu stvarnu matičnu ploču s hijerarhijski povezanim napravama, gdje su najbrže naprave na vrhu, a sporije ispod
- Ideja je da se sporije naprave stave na sporije sabirnice, da ne utječu na performanse brzih naprava (koje bi inače morale čekati duge sabirničke cikluse sporijih naprava)

Z590 AORUS MASTER Motherboard Block Diagram



Slika 3.2. Primjer arhitekture matične ploče, povezujuće komponente računala (izvor: GIGABYTE, Z590 AORUS MASTER, User manual)

3.2. Pristup napravama preko adresa

- Napravama se najčešće može pristupiti korištenjem adresa
- Adrese mogu biti prave ili virtualne
- Ako OS koristi straničenje (npr. kao što Linux koristi i u jezgrinom načinu rada), onda te adrese treba mapirati u tablici prevođenja
- Potrebna je posebna pažnja pri korištenju adresa naprava da se nešto ne optimira:
 - priručni spremnik procesora
 - * treba upisati u napravu, ne samo u priručni spremnik
 - * bitno kad je redoslijed upisivanja bitan
 - * ponekad ne smeta
 - optimiranje izmjenom redoslijeda izvođenja susjednih nezavisnih instrukcija (engl. *out-of-order*)
 - * ako se takvim instrukcijama upisuje u mapirane registre naprave takve promjene redoslijeda mogu biti problem
 - da se izbjegnu ovakvi problemi ubacuju se posebne instrukcije
- Adrese koje se koriste za komunikaciju s napravama su često mapirane u adresni prostor upravljačkog sklopa - kontrolera na koji je naprava spojena, primjerice PCI-E, USB, SATA i slično. Kontroler tada komunicira napravom na neki način (konverzijom protokola i slično).

3.3. Korištenje međuspremnika

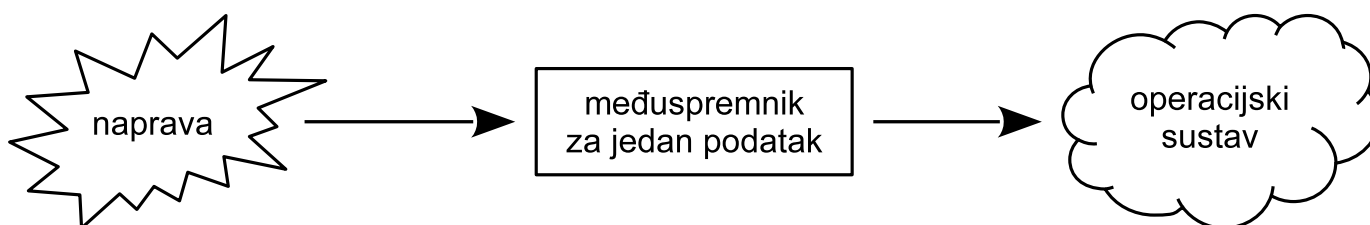
3.3.1. Razlozi korištenja međuspremnika

- Međuspremnici se koriste radi povećanja učinkovitosti
 - oni koji ih koriste mogu raditi raznim brzinama – međuspremnik dozvoljava “gomilanje” podataka i “praznog” prostora
 - ponekad tek skup podataka čini cjelinu koji ima smisla prenositi
 - neke provjere ispravnosti prijenosa treba napraviti nad skupom podataka (paket, blok, ...)
- Naprave mogu imati zasebne, interne, međuspremnike
 - moguće i skrivene od sustava – koriste se implicitno preko sučelja naprave (npr. međuspremnik diska)
 - ili im je omogućen pristup preko registara – s iste adrese se čitaju svi podaci – međuspremnik nije vidljiv (npr. FIFO kod UART sklopa)
 - ili im je omogućen pristup preko mapiranih adresa – svaki bajt ima zasebnu adresu – međuspremnik je vidljiv (npr. PCIe naprava)
- Korištenje (dodatnih) međuspremnika u radnom spremniku
 - obično taj međuspremnik stvara upravljački program naprave
 - mogućnost puno većeg kapaciteta

- za ulazno/izlazne naprave međuspremnik može biti podijeljen na dva dijela: za ulazne i za izlazne podatke
- efikasnije korištenje – asinkrona komunikacija s napravama
- Mnoge naprave su optimirane za blokovski prijenos podataka (kao i pristup memoriji)
 - prijenos bloka podataka je puno efikasniji nego pojedinačno prenošenje
 - stoga se često dohvaća više podataka nego je traženo u programu, očekujući da će program u nastavku rada trebati i te podatke; vrlo često se to i zbiva

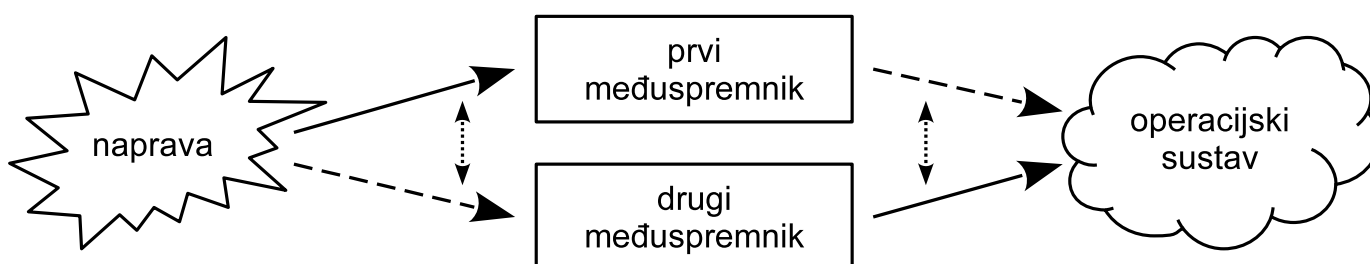
3.3.2. Tipovi međuspremnika

1. Međuspremnik za jedan podatak – u njega stane samo jedan podatak (bajt, paket, blok)
 - a) ulazna naprava upiše podatak u međuspremnik
 - b) tek nakon toga OS može čitati taj podatak
 - c) idući podatak naprava može upisivati tek nakon što OS pročita prethodni
 - d) za izlaznu napravu je obrnuto: OS prvi upisuje, naprava čita nakon toga



Slika 3.3. Međuspremnik za jedan podatak pri komunikaciji s ulaznom napravom

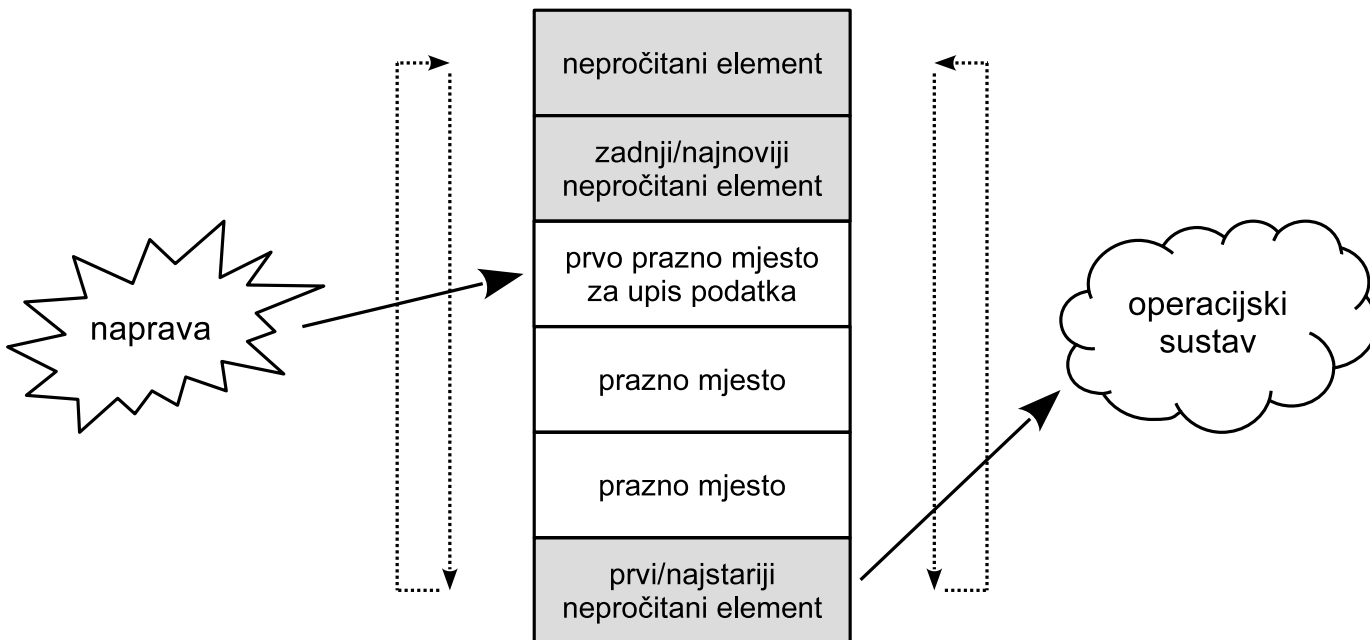
2. Dvostruki međuspremnik (engl. *double buffer*) – u prvi se piše dok se iz drugog čita, uz zamjenu uloga kad se čitanje/pisanje obavi; npr.:
 - a) ulazna naprava piše u prvi, a OS može čitati iz drugog međuspremnika
 - b) kad su obje operacije gotove, međuspremnici se zamjenjuju: naprava piše u drugi, a OS čita iz prvog
 - c) ne moraju operacije biti paralelne, ali onda netko mora čekati; npr. ako je OS pročitao drugi, ali naprava još nije stavila novi podatak u prvi, OS mora čekati; i obratno, kada naprava mora čekati da OS pročita podatak



Slika 3.4. Dvostruki međuspremnik pri komunikaciji s ulaznom napravom

3. Kružno korištenje međuspremnik

- proširenje koncepta dvostrukog međuspremnik na N međuspremnik
- kad se jedno mjesto u međuspremniku popuni, idući podatak stavlja se na iduće (slijedeće) mjesto, ako je "prazno"; ako nije mora se čekati
- kad se dođe do zadnjeg mjesta, iduće mjesto je prvo u međuspremniku (mjesto međuspremnik se kružno obilaze)



Slika 3.5. Primjer kružnog međuspremnik pri komunikaciji s ulaznom napravom

- pri kružnom korištenju međuspremnik potrebna je operacija: $i = (i + 1) \bmod N$
- kako bi se izbjegla skupa operacija moduliranja (dijeljenja) obično se veličina međuspremnik postavi na 2^n (višekratnik broja dva) te se tada povećanje može napraviti jednostavnom operacijom bitovnog I, tj.: $i = (i + 1) \& X$, gdje je $X = (2^n - 1)$ (konstanta)

3.3.3. Kružni međuspremnik s dvostruko mapiranim dijelovima

1. Problem "klasičnog" kružnog međuspremnik je korištenje modulo aritmetike te pri pisanju i čitanju preko granica, kad se mora krenuti od početka.

Primjer 3.1. Kopiranje u međuspremnik koje prelazi gornju granicu

Početno stanje (x – popunjeni dijelovi međuspremnik):

-----xxxxxxxx-----

Primjer dodavanja: 13 novih elemenata (bajtova):

1. dodavanje 8 elemenata na kraj (1. memcpy)

-----xxxxxxxxyyyyyy

2. dodavanje 5 elemenata na početak (2. memcpy)

```
yyyyyy-----xxxxxxxxxxxxyyyyyyyyyy
```

2. Straničenje omogućava da se isti dio memorije mapira na različite adrese

- isti segment memorije (fizičke) se dvaput mapira u procesu, prvi puta na adresu X te drugi puta na adresu X + veličina međuspremnik
- veličina međuspremnik mora biti višekratnik veličine stranice, a da bi prelaskom granice prešli na iduću stranicu u virtualnom adresnom prostoru, a zapravo se vratili na početak međuspremnik
- npr. kad bi stvarna veličina međuspremnik bila jedna stranica, onda bi u tablici prevođenja procesa dva uzastopna opisnika (koja opisuju međuspremnik) pokazivala na isti okvir

Primjer 3.2. Kopiranje u međuspremnik s dvostrukim mapiranjem

Početno stanje (x – popunjeni dijelovi međuspremnik):

```
-----xxxxxxxx-----|-----xxxxxxxx-----  
\ 1. puta mapiran međuspremnik / \ 2. puta mapiran međuspremnik /
```

Dodavanje 13 elemenata u jednom koraku (samo jedan memcopy):

```
yyyyyy-----xxxxxxxxxxxxyyyyyyyyyy|yyyyyy-----xxxxxxxxxxxxyyyyyyyyyy  
                                     \-----/  
                                     za memcopy ovo je kontinuirani dio
```

Skica ostvarenja ovakva međuspremnik, prema:

<https://lo.calho.st/posts/black-magic-buffer/>

```
struct kms { /* kružni međuspremnik */  
    void *ms; /* kazaljka na dvostruko mapirani međuspremnik */  
    size_t velicina; /* veličina rezervirana za ms */  
    size_t izlaz; /* prvo nepročitano mjesto */  
    size_t ulaz; /* prvo slobodno mjesto, ulaz >= izlaz uvijek! */  
};  
  
int inicijaliziraj_ms(struct kms *kms, size_t velicina)  
{  
    /* velicina mora biti višekratnik veličine stranice */  
    [...]  
    x = nađi mjesta u adresnom prostoru procesa za 2 * velicina bajtova  
    kms->ms = mmap(x, velicina, ...); /* zauzmi prostor za ms */  
  
    /* mapiraj isti prostor na susjednim adresama */  
    mmap(kms->ms + velicina, velicina, ...);  
    kms->izlaz = 0;  
    kms->ulaz = 0;  
    kms->velicina = velicina;  
  
    return 0;  
}  
  
size_t stavi_u_ms(struct kms *kms, void *podaci, size_t za_staviti)  
{  
    size_t slobodno = kms->velicina - (kms->ulaz - kms->izlaz);
```

```

    if(slobodno < za_staviti)
        return 0; /* ili: za_staviti = slobodno */

    memcpy(&kms->ms[kms->ulaz], podaci, za_staviti);
    kms->ulaz += za_staviti;

    return za_staviti;
}

size_t uzmi_iz_ms(struct kms *kms, void *podaci, size_t za_uzeti)
{
    size_t ima = kms->ulaz - kms->izlaz;
    if(ima < za_uzeti)
        return 0; /* ili: za_uzeti = ima */

    memcpy(podaci, &kms->ms[kms->izlaz], za_uzeti);
    kms->izlaz += za_uzeti;
    if(kms->izlaz > kms->velicina) {
        kms->izlaz -= kms->velicina;
        kms->ulaz -= kms->velicina;
    }

    return za_uzeti;
}

```

Za gornji primjer kazaljke/vrijednosti bi bile:

a) prije dodavanja 13 elemenata:

```

-----xxxxxxxx-----|-----xxxxxxxx-----
      |                ^kms->ulaz
      kms->izlaz

```

b) nakon dodavanja 13 elemenata:

```

yyyyy-----xxxxxxxxxyyyyyyy|yyyyy-----xxxxxxxxxyyyyyyy
      ^kms->izlaz                ^kms->ulaz

```

c) kada izlaz pređe veličinu međuspremnik, nakon čitanja 20 bajtova:

```

--yyy-----|-----
  | ^kms->ulaz
  kms->izlaz

```

3.3.4. Međuspremnik i priručni spremnik

1. Međuspremnik (engl. *buffer*)

- Pojam međuspremnik se obično odnosi na spremnik koji se koristi za prijenos podataka između dviju strana, npr. naprave i operacijskog sustava.
- Pri prijenosu se podaci miču s one strane koja ih je kopirala u međuspremnik
- Pri preuzimanju podataka iz međuspremnik, oni se miču iz njega
- Podaci su samo na jednom mjestu (nema kopija podataka)

2. Priručni spremnik (engl. *cache*)

- Priručni spremnik se koristi za ubrzanje rada

- Kopija podataka (i instrukcija) se dohvaća da bude bliže onome tko ih koristi (procesor, naprava)
- Postoji više kopija podataka
- Upravljanje priručnim spremnikom je najčešće ostvareno sklopovljem (npr. priručni spremnik procesora)
- U slučaju korištenja naprava, moguće je “ručno” upravljanje – podaci se namjerno kopiraju iz međuspremnika naprave bez da se miču iz nje
- Priručni spremnik podataka može ostvariti kod jezgre, ali i programi, u prostoru procesa
 - pri operaciji `read` OS može dohvatiti više nego se tražilo funkcijom (npr. i susjedne blokove datoteke), ali procesu dati samo ono što je tražio
 - pri operaciji `fread` sama implementacija funkcije može od OS-a tražiti više i dobiveno držati u priručnom spremniku procesa koji je ona napravila; idući zahtjev možda može biti poslužen s tim podacima, bez da se traži “pomoć” OS-a – bez poziva jezgrine funkcije (`read`)
- Kod brzih naprava treba razmisliti o načinu korištenja međuspremnika – pretjerano kopiranje podataka može usporiti operacije koje trebaju biti brze
 - podaci se najprije kopiraju u međuspremnik jezgre (npr. u obradi prekida naprave) a potom od tuda u međuspremnik procesa (u pozivu jezgrine funkcije od strane tog procesa)
 - ideja jest izbjeći ovakvo dvostruko kopiranje, primjerice mapiranjem međuspremnika izravno u prostor procesa, tako se se podaci izravno s naprave kopiraju u njega

Pitanja za vježbu 3

1. Na koje se sve načine može pristupiti napravama, njihovim registrima, memoriji?

Odgovor:

*na isti način kao i memoriji – preko adresa
ili posebnim instrukcijama (in/out)*

2. Zašto su stvarne arhitekture računala hijerarhijski građene, s premosnicima (mostovima) između različitih dijelova?

Odgovor:

želi se izbjeći da spore naprave ne usporavaju brže, zato postoje različite sabirnice

3. Nekim se napravama pristupa korištenjem adresa. Koji sve problemi zbog toga mogu nastati, tj. što treba “reći procesoru” u tom slučaju?

Odgovor:

*procesor je optimiran za rad s memorijom:
može izvrnuti redoslijed izvođenja susjednih "nepovezanih" instrukcija*

može koristiti priručni spremnik – podatak ne ide odmah prema napravi već neko vrijeme ostaje u priručnom spremniku (i od tamo se koristi)
ove i slične probleme treba riješiti dodatnim instrukcijama koje će osigurati ispravan redoslijed, tj. da se podaci odmah i upišu u napravu a ne samo u priručni spremnik

4. Zašto se koriste međuspremnici? Koju funkcionalnost obavljaju?

Odgovor:

efikasnost:

– dretva ne mora čekati da se podatak zaista i pošalje (sporoj) napravi
– kad dođe podatak on se privremeno sprema u međuspremnik dok ga netko ne zatraži – naprava može nastaviti s radom

iako i naprave većinom imaju interne međuspremnike, oni u memoriji mogu biti puno veći

međuspremnici preuzimaju podatke iz naprave / za napravu – nema kopija podataka

5. Navesti vrste međuspremnika i opisati kako se oni koriste.

Odgovor:

jednostruki, dvostruki, kružni

6. Kod korištenja kružnih međuspremnika, često se koristi “trik” u dvostrukom mapiranju istog fizičkog bloka na dva susjedna virtualna bloka. Što se time postiže?

Odgovor:

kopiranje preko "ruba" samo s jednom naredbom copy

7. Kružni međuspremnik je najčešće korišten pri komunikaciji s napravama. Međutim, njegovo korištenje zahtjeva operacije modula (npr. $ULAZ = (ULAZ + 1) \text{ MOD } N$) te dvije operacije kopiranja ako se prelazi granica (npr. kad treba kopirati X podataka, a $N - ULAZ < X$). Kako se efikasno rješavaju ti problemi (bez korištenja operacije modula i samo s jednim kopiranjem)?

Odgovor:

koristi se međuspremnik veličine potencije broja dva pa se MOD rješava s bitovnim operacijama: $ULAZ = ULAZ \& (2^n - 1)$

međuspremnik se mapira na dvije uzastopne lokacije u virtualnom memoriji (kad se koristi straničenje) pa se kopiranje obavlja samo s jednom naredbom copy (i kad ide preko ruba)

8. Usporediti međuspremnik (engl. *buffer*) i priručni spremnik (engl. *cache*). Za što se koriste?

Odgovor:

međuspremnik – za prihvatanje podataka od naprave ili prema njima (nema kopija podataka)

priručni spremnik – dohvatanje podataka u bržu memoriju (imamo kopije podataka)

9. U komunikaciji s napravama često se koristi dvostruki međuspremnik (engl. double buffer). Prikazati korištenje takvog međuspremnika u primjeru u kojem OS treba poslati tri podatka napravi. Pretpostaviti da operacija naprave nad jednim podatkom iz međuspremnika traje 5 ms dok trajanje upisa podatka u međuspremnik od strane OS-a zanemariti.

Odgovor:

Pretpostavka je da OS operaciju napravi brzo (gotovo trenutno) dok napravi treba 5 ms.

<i>vrijeme</i>	<i>OS</i>	<i>naprava</i>
<i>t = 0 ms</i>	<i>stavlja u M1</i>	<i>čeka OS</i>
<i>t = 0 ms</i>	<i>stavlja u M2</i>	<i>započinje sa čitanjem iz M1</i>
<i>t = 0-5 ms</i>	<i>čeka napravu</i>	<i>čita iz M1</i>
<i>t = 5 ms</i>	<i>čeka napravu</i>	<i>završava s M1</i>
<i>t = 5 ms</i>	<i>stavlja u M1</i>	<i>započinje sa čitanjem iz M2</i>
<i>t = 5-10 ms</i>	<i>–</i>	<i>čita iz M2</i>
<i>t = 10 ms</i>	<i>–</i>	<i>završava s M2</i>
<i>t = 10 ms</i>	<i>–</i>	<i>započinje sa čitanjem iz M1</i>
<i>t = 10-15 ms</i>	<i>–</i>	<i>čita iz M1</i>
<i>t = 15 ms</i>	<i>–</i>	<i>završava s M1</i>
<i>t = 15 ms</i>	<i>–</i>	<i>čeka OS</i>

4. Primjeri protokola za komunikaciju s napravama

Radi razumijevanja problema i složenosti komunikacije s napravama, u ovom poglavlju su opisani neki protokoli koji se za to koriste. U ovom prikazu ima jako puno detalja. Međutim, oni su navedeni samo radi lakšeg razumijevanja načela rada protokola.

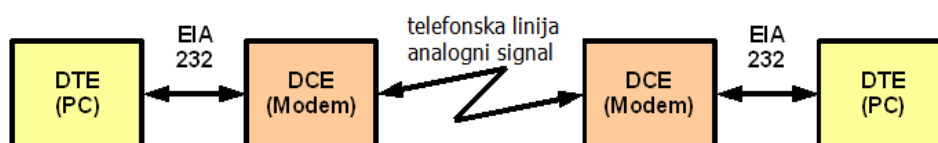
Terminologija: žica = vodič = linija

4.1. Serijska i paralelna veza – usporedba svojstava

- Serijska veza
 - prijenos informacije bit-po-bit preko jedne žice u jednom smjeru
 - po jednom ciklusu (taktu signala) prenosi se jedan bit
 - sinkronizacija/provjera ispravnosti nakon svih prenesenih bitova
 - manje žica, veće udaljenosti, manje brzine prijenosa – u prošlosti
 - danas, uz poboljšanja, serijska veza je brža (za komunikaciju s napravama)
- Paralelna veza
 - paralelni/istovremeni prijenos bitova preko više žica
 - po jednom ciklusu prenosi se više bitova
 - sinkronizacija nakon svakog ciklusa
 - načelno brži prijenos od serijskog jer se paralelno prenosi više bitova, ali ...
 - previše je žica potrebno (podatkovnih i za sinkronizaciju), stvara puno smetnji
 - previše sinkronizacije nakon svakog bita ograničava brzinu prijenosa na većim udaljenostima
- danas se sve više koristi serijska veza: USB umjesto starije serijske RS232 i paralelne; PCIe umjesto PCI, SATA umjesto PATA
- izuzetak je vrlo brza sabirnica procesor – memorija; ali je zato memorija jako blizu procesora

4.2. Serijska veza RS-232

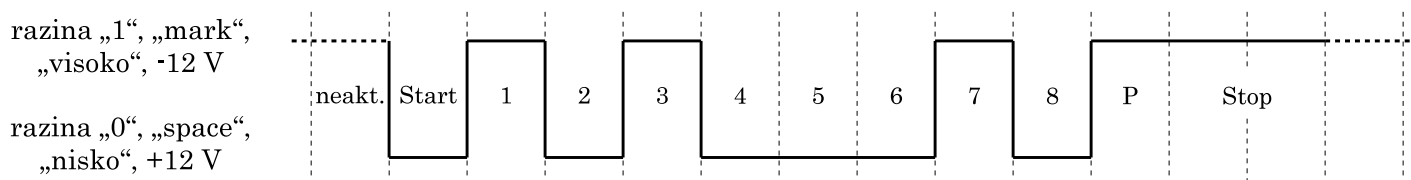
- Serijska veza (RS-232) se uglavnom koristila za komunikaciju s modemom



Slika 4.1. Serijska veza pri komunikaciji s modemom (slika s Wikipedije)

- Jedinica podataka je znak (5–8 bita) – grupa uzastopnih bitova

- Potrebno je sinkronizirati početak/kraj (start/stop bit)
- Bitove jednog znaka moguće je zaštititi paritetnim bitom (ali se to ne mora koristiti)



Slika 4.2. Poruka 01000101_2 duljine 8 bita, parnog pariteta uz 2 stop bita

- 2 vodiča se koriste za prijenos podataka – po jedan za svaki smjer
- Za upravljanje su minimalno potrebna još dva upravljačka (RTS/CTS); pinovi standardnog priključka DB9 prikazani su u tablici 4.1.

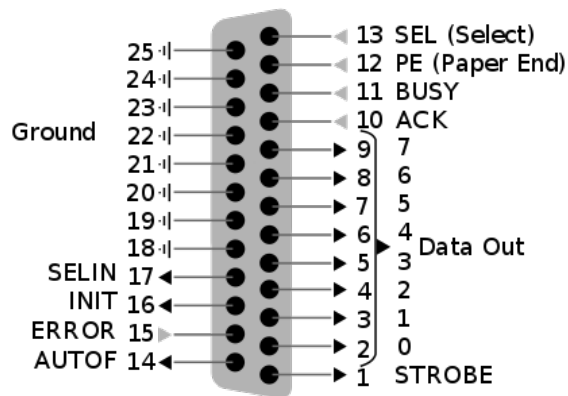
Tablica 4.1. Priključci za serijsku vezu

Izvorni naziv	kratica	DTE	DCE	Za što se koristi
Transmitted Data	TxD	izlaz	ulaz	Prijenos podataka: DTE \Rightarrow DCE
Received Data	RxD	ulaz	izlaz	Prijenos podataka: DCE \Rightarrow DTE
Request To Send	RTS	izlaz	ulaz	DTE traži da DCE počne slati podatke
Ready To Receive	RTR	izlaz	ulaz	DTE je spreman za prihvatanje podataka
Clear To Send	CTS	ulaz	izlaz	DCE je spreman za prihvatanje podataka
Data Set Ready	DSR	ulaz	izlaz	DCE je spreman za prihvatanje i slanje
Data Carrier Detect	DCD	ulaz	izlaz	DCE javlja da je veza uspostavljena
Data Terminal Ready	DTR	izlaz	ulaz	DTE je spreman za komunikaciju s DCE
Ring Indicator	RI	ulaz	izlaz	DCE je detektirao signal za poziv

- Sklop 16550 UART koristi međuspremnike veličine 16 bajtova za ulaz i izlaz
 - može se postaviti da generira prekid kad je unutra 1, 4, 8, ili 14 bajtova
 - svrha međuspremnika jest da se ne generira prekid prečesto
- Veća grupa podataka – znakova se šalje slijedno, znak po znak
- Pri slanju niza znakova potrebno je posebnim znakovima označiti početak i kraj niza (XON/XOFF/DLE)
- Obje strane trebaju se dogovoriti o svim ovim parametrima

4.3. Paralelna veza LPT

- Osam bitova se šalje paralelno, svaki preko svoje žice
- Sinkronizacija dodatnim signalnim linijama ide nakon svakog ciklusa
- LPT: paralelni port DB-25 prema slici 4.3. (za pisač)



Slika 4.3. Oznake priključaka za DB-25 (slika s Wikipedije)

4.4. USB

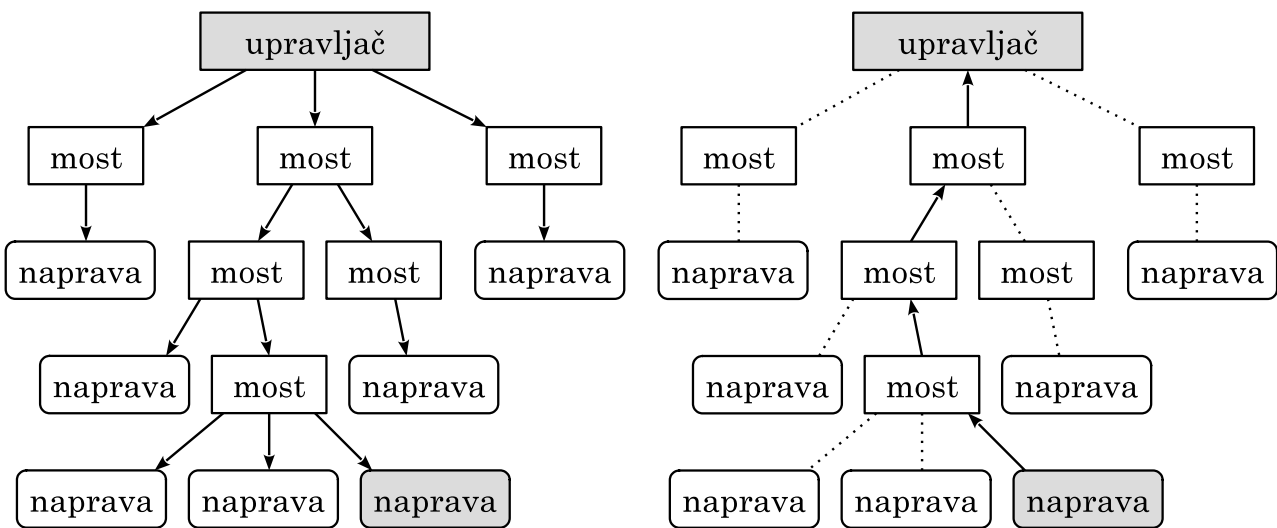
Universal Serial Bus – Univerzalna serijska sabirnica

4.4.1. Zašto USB?

- Uvedena da omogući jednostavan (i jeftin) način spajanja razne periferije na računalo (tipkovnica, miš, pisač, memorijska kartica, ...)
- Razni priključci, prilagođeni napravama, jednostavni za korištenje
- Napredovala s vremenom: USB 1, 2, 3, 4
- Novije verzije su donijele veće brzine, složenije priključke (npr. Type-C)
- U nastavku će biti ukratko opisano kako USB radi, od fizičkog sloja do prijenosa poruka

4.4.2. “Sabirnica”

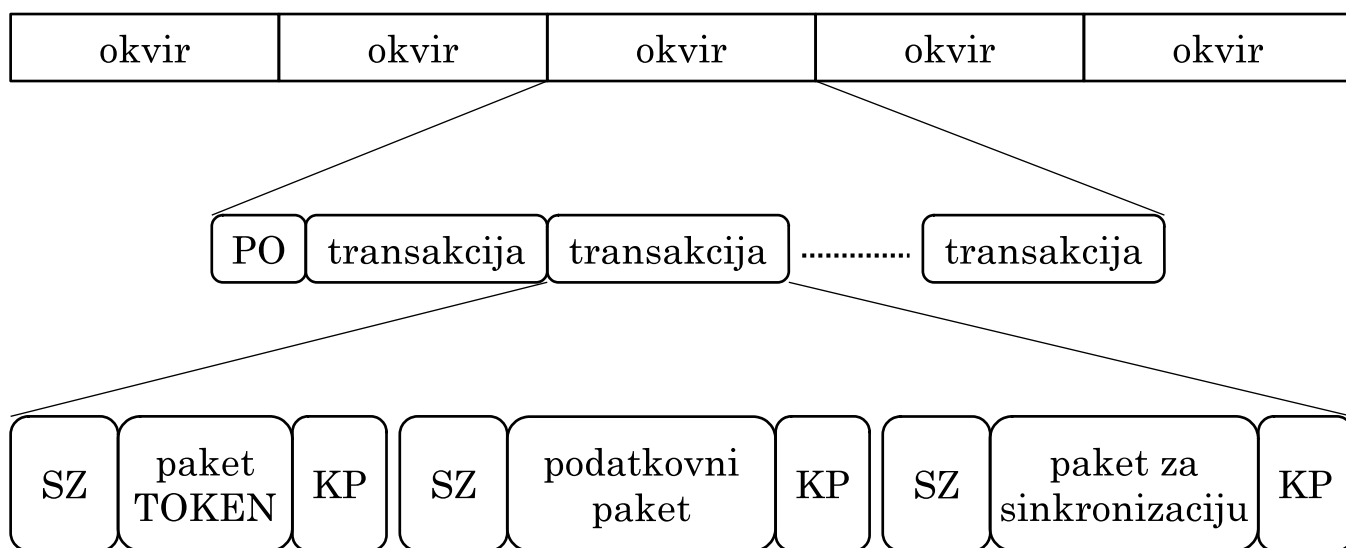
- Sabirnica povezuje naprave na računalo
- Naprave mogu biti obične ili mostovi (engl. *hub*) na koje se mogu spojiti i druge naprave i mostovi
- Sabirnica je upravljana s jednog, glavnog/upravljačkog čvora – u nastavku *upravljač*
 - Izvorni naziv za upravljač: host ili root hub (tj. host upravlja preko root hub-a)
- Upravljač započinje svaku razmjenu poruka: komunikacija se uvijek odvija samo između upravljača i neke naprave
- Logički, sve su naprave na zajedničkoj sabirnici
- Fizički, naprave su najčešće neizravno spojene preko mostova, čineći stablo s korijenom u upravljaču i listovima u napravama (slika 4.4.)
- Zbog vremenskih ograničenja, najviše pet mostova može biti između upravljača i naprave
- Pri slanju podataka od upravljača prema napravi, oni se pošalju svima – most primljene podatke odozgora (od upravljača) prosljedi svim napravama ili mostovima spojenim na njega (slika 4.4. lijevo)
- Pri slanju podataka od naprave prema upravljaču koristi se samo jedna staza (slika 4.4. desno)



Slika 4.4. Primjer USB stabla, s oznakom puta podataka pri slanju i primanju

4.4.3. Komunikacija

- Svaka naprava dobiva (nakon spajanja) adresu: 7-bitovni broj
- Na jedan upravljajč ukupno može biti spojeno 127 naprava
- Upravljajč svaku napravu periodički proziva – pita ju ima li nešto za javiti, podatke ili status
- Koristi se period od 1 ms ili kraći, 125 μs za USB 2+
- Komunikacija je podijeljena u okvire (engl. *frame*)
- Svaki okvir se sastoji od oznake početka okvira PO (engl. *Start of Frame – SOF*) te jedne ili više transakcija
- Svaka transakcija se sastoji nekoliko paketa, ovisno o tipu transakcije
- Slika 4.5. prikazuje primjer prijenos podataka kod kojeg se transakcija sastoji od tri paketa: paketa TOKEN, podatkovnog paketa i paketa sinkronizacije



Slika 4.5. Primjer prijenosa okvira, transakcija i paketa

- Svaki se paket (na fizičkoj razini) sastoji od tri dijela
 1. sinkronizacijskog zaglavlja (SZ, sync)
 2. tijela paketa (paket podatkovne razine)

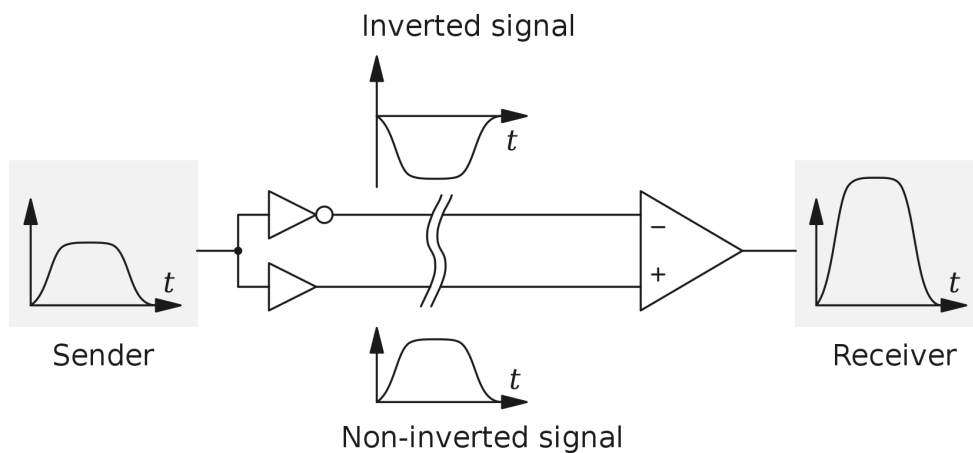
3. oznake kraja paketa (KP, End-of-Packet:EOP)
- Tip paketa određen je prvim bajtom paketa (prvi bajt nakon SZ) – PID – identifikator paketa:
 1. TOKEN paketi (IN, OUT, SETUP, PING, Start-of-frame:SOF – oznaka početka okvira PO)
 2. paketi za prijenos podataka (DATA0/1/2, cirkularno)
 3. paketi za sinkronizaciju (handshake, ACK, NAK, ...)
 - Pakete tipa TOKEN šalje upravljač na početku transakcije
 - TOKEN IN – traži se odgovor od naprave
 - TOKEN OUT – šalje se podatak napravi (paket odmah iza ovoga)
 - Veličina podataka u podatkovnom paketu može biti do 1 KB (ili manje ovisno o protokolu)
 - Uobičajena komunikacija za slanje/primanje podataka uključuje tri paketa:
 1. TOKEN-IN/OUT – šalje upravljač
 2. paket s podacima – ovisno o smjeru: OUT-upravljač, IN-naprava
 3. sinkronizacijski paket (ACK) – ovisno o smjeru: OUT-naprava, IN-upravljač
 - TOKEN paketi sadrže u sebi adresu “naprave” s kojom žele komunicirati
 - Adresa se sastoji od same adrese naprave (7-bitovne adrese) te od broja funkcije naprave (engl. *end-point*)

4.4.4. Logička komunikacija s napravom

- Naprava može imati više “funkcija”
- Svaka funkcija (engl. *endpoint*) ima svoj redni broj (počevši s nulom), koji je dodijeljen pri inicijalizaciji naprave
- Do 32 funkcije po napravi, 16 ulaznih, 16 izlaznih
- Veza upravljač–naprava je zapravo veza upravljač–funkcija
- Veza upravljač–funkcija se ostvaruje preko mehanizma cjevovoda (na višoj razini!)
- Za razliku od funkcija, cjevovodi se otvaraju i zatvaraju, po potrebi
- Dva tipa cjevovoda:
 1. cjevovod za razmjenu poruka (dvosmjerni)
 2. cjevovod za protok podataka (jednosmjerni)
 - a) Izokroni prijenosi (engl. *Isochronous*) – garantira se propusnost, ali s mogućim gubicima podataka
 - b) Prekidni prijenosi – za brze odgovore na događaje
 - c) Veliki prijenosi (engl. *Bulk transfers*) – za prijenos veće količine podataka – ne garantiraju se ni propusnost ni kašnjenja – koriste se slobodni ciklusi na sabirnici
- Funkcije mogu biti grupirane u sučelja (engl. *interface*) koje definira neku operaciju naprave

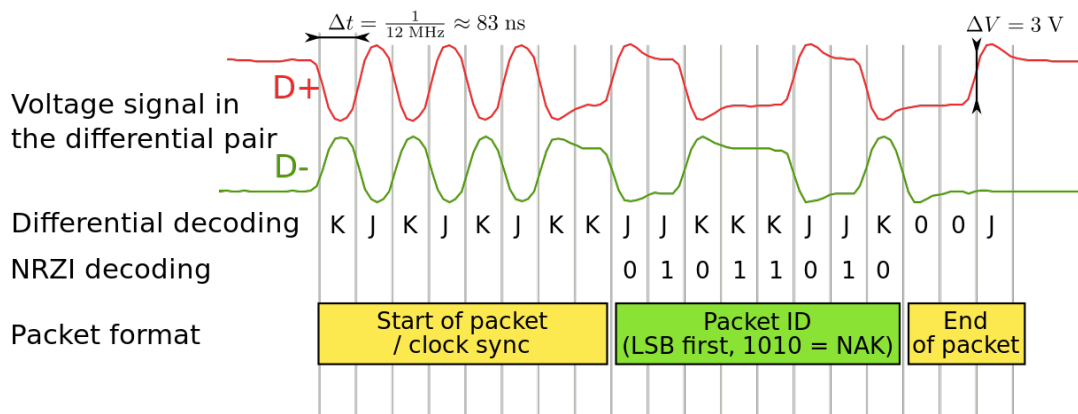
4.4.5. Fizička razina

- USB 1 i 2 koriste 4 žice za spoj: $V_{CC} = 5V$, GND , $D+$ i $D-$
- $D+$ i $D-$ – za prijenos podataka u jednom ili drugom smjeru (ali ne istovremeno)
- USB 3 i 4 koriste dodatne vodiče za brži i istovremeno obostrani prijenos (+2/3 para)
- Signal se prenosi u "normalnom" i "invertiranom" stanju (engl. *differential signalling*) (slika 4.6.)
- Ideja je smanjiti utjecaj signala na okolinu
- USB 3+ koristi obložene vodiče (zbog većih frekvencija)



Slika 4.6. Diferencijalna signalizacija (slika s Wikipedije)

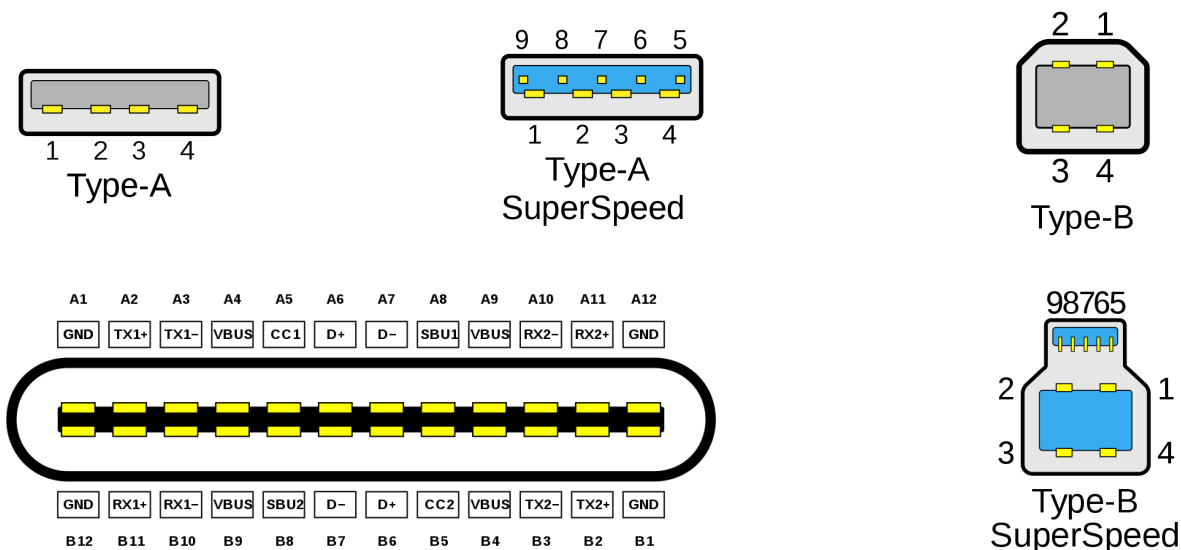
- Na fizičkoj razini koristi se kodiranje Non-return-to-zero inverted (NRZI)
 - Uz definiranu frekvenciju, logička nula predstavlja promjenu u tom taktu te bez promjene za logičku jedinicu
- Svaki prijenos paketa započinje sa sinkronizacijskim zaglavljem (SZ)
 - njegova je namjena da određena strana sinkronizira svoj sat
 - broj koji se razmjenjuje je 00000001_2
 - svaka nula predstavlja promjenu stanja
- Slijedi podatkovni dio paketa (zapочиnje s PID te ostatak ovisno o tipu paketa)
- Kraj prijenosa (kraj paketa, KP, End-of-Packet:EOP) označen je tri ciklusa sata: u prva dva oba su signala $D+$ i $D-$ nisko; u trećem je promjena u jednom.
- Slika 4.7. prikazuje signale pri prijenosu jednog NAK paketa
- Kodiranje podataka (između SZ i KP):
 - USB 1/2/3: 8b/10b kodiranje
 - * za svaku kombinaciju od 8 bita postoji "znak" od 10 bita koji se šalje umjesto 8 bita
 - * potrebno radi minimalne aktivnosti signala
 - USB 3.1: 128b/130b; USB 4: 64/66



Slika 4.7. Primjer signala za prijenos paketa (slika s Wikipedije)

4.4.6. Tipovi naprava – podrška operacijskog sustava

- USB definira nekoliko tipova naprava – klasa naprava, korištenjem 8-bitovnog koda
- Taj broj se razmjenjuje s napravom pri njenoj inicijalizaciji
- Ideja je da posebni upravljački programi nisu potrebni za napravu koja spada u neku kategoriju – da se za nju mogu koristiti uobičajeni upravljački programi za takvu klasu naprave (koje OS već ima)
- Primjeri klasa: audio naprava, naprave za umrežavanje, naprava sučelja prema korisniku (miš, tipkovnica), pisač, podatkovna naprava, video, ...
- USB standard definira nekoliko brzina rada (koje se dogovaraju s napravama):
 1. niska brzina (engl. *Low speed, LS*) – 1,5 Mbit/s
 2. puna brzina (engl. *Full speed, FS*) – 12 Mbit/s
 3. velika brzina (engl. *High speed, HS*) – 480 Mbit/s
 4. super brzina (engl. *SuperSpeed, SS*) uz dva dodatna oklopljena para vodiča – 5 Gbit/s
 5. super brzina plus (engl. *SuperSpeed+, SS+*) uz dva dodatna para vodiča – 10 Gbit/s (20/40 uz USB 3.2/4 i Type C priključak)

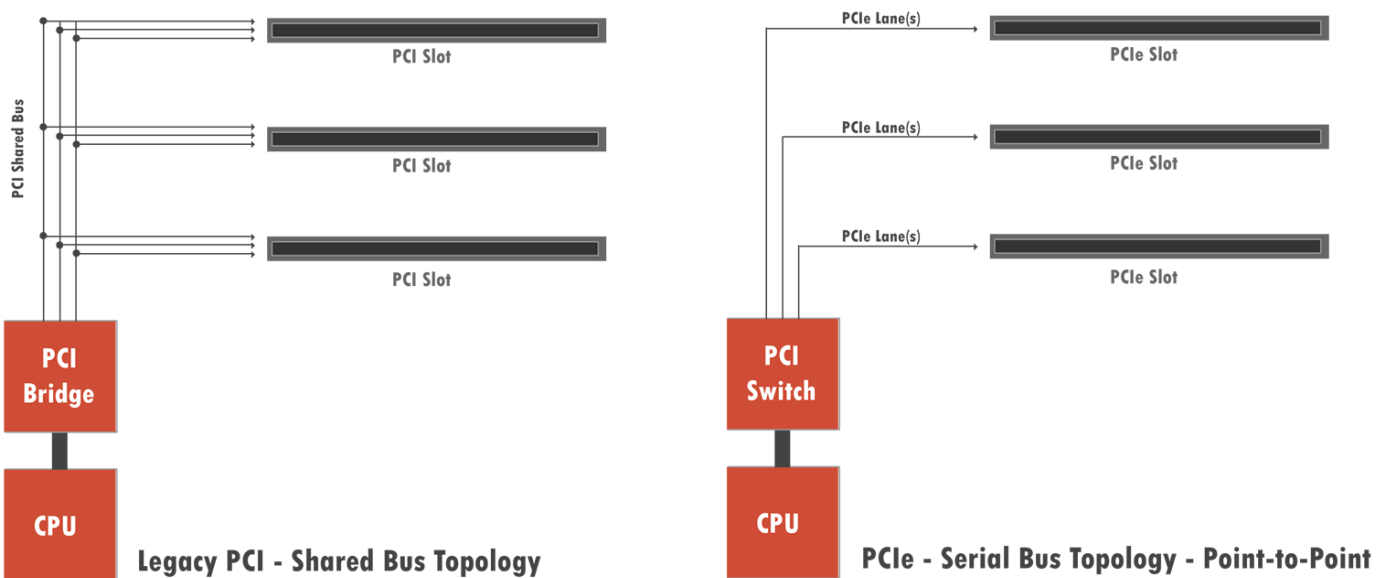


Slika 4.8. Type A, B i C priključci (slike s Wikipedije)

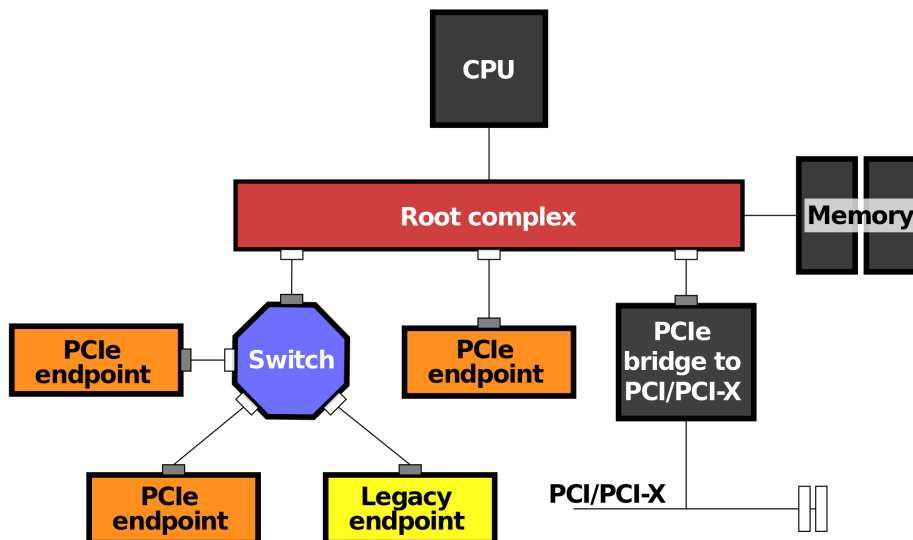
4.5. PCI Express

PCIe, Peripheral Component Interconnect Express

- Zamjenjuje PCI, na nižoj razini
- Problemi sabirnice PCI:
 - zajednička sabirnica (usko grlo), potreban i arbiter
 - komunikacija samo u jednom smjeru istovremeno
 - previše signala, loše za sinkronizaciju, ograničava brzinu
 - zbog sinkronizacije takta, najsporija naprava diktira tempo
- PCIe:
 - izravna komunikacija između dviju strana (npr. naprave i PCIe kontrolera)
 - istovremena dvosmjerna komunikacija (engl. *full duplex*)
 - serijska komunikacija po pojedinim stazama (engl. *lane*)



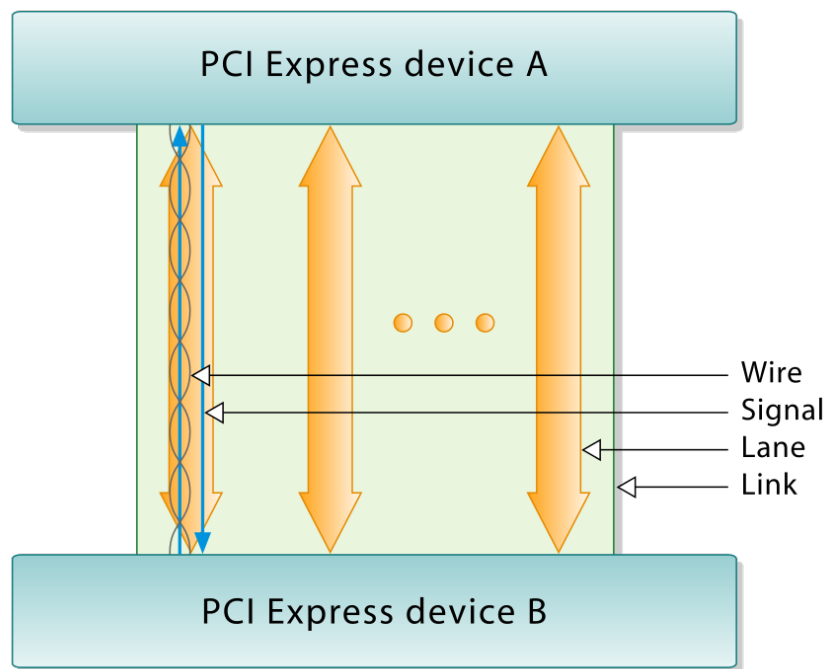
Slika 4.9. Usporedba PCI i PCIe prema načinu spajanja (slika s Wikipedije)



Slika 4.10. Primjer povezivanja naprava na PCIe (slika s Wikipedije)

Detalji PCIe komunikacije preko staza

- Svaka veza (engl. *link*) između dviju naprava povezanih preko PCIe se sastoji od jedne ili više staza (engl. *lane*)
- Oznake uz PCIe utore: x1, x2, x4, x8, x16, x32 – toliko može biti staza
- Dvosmjerna komunikacija na svakoj stazi
- Za svaki smjer dva vodiča (slika 4.6.), ukupno četiri vodiča za svaku stazu (lane)
 - signal se prenosi u "normalnom" i "invertiranom" stanju (engl. *differential signalling*)
 - na taj način najmanje "zagađuje" okolinu
- 8b/10b kodiranje za PCIe 1 i 2, 128b/130b za PCIe 3
 - npr. 128b/130b => za 128 bita korisne informacije preko veze se šalje 130 bita
- Podaci se šalju u "paketima"
 - pri slanju, paketi se podijele na dijelove, koji se onda paralelno šalju različitim stazama
 - ona druga strana treba primljene dijelove posložiti u paket



Slika 4.11. PCIe veza (slika s Wikipedije)

Paketi

- Na fizičkoj, podatkovnoj i transakcijskoj razini
 - PCIe Physical Layer (PHY)
 - data link layer packet (DLLP)
 - transaction layer packet (TLP)
- Veličina paketa je višekratnik od 32 bita (u ovom kontekstu 32 bita je *double word*)
- Format paketa:
 - 1 B: startni bajt (fizička razina)
 - 2 B: redni broj – svaki idući je za jedan veći (podatkovna razina)

- 12-16 B: TLP zaglavlje (transakcijska razina)
- 0-4 KB: podaci koji se prenose (transakcijska razina)
- ECRC 4 B: opcionalna zaštita TLP razine (transakcijska razina)
- LCRC 4 B: zaštita DLLP razine (podatkovna razina)
- završni bajt (fizička razina)
- Svaki paket se potvrđuje s ACK ili NAK porukama (paketima) ako je bilo grešaka u prijenosu
- Da ne bi došlo do zagušenja, naprava javlja stanje svog međuspremnika – "kredit" – ona druga strana treba paziti da ne pošalje više od toga; kad se u međuspremniku oslobodi nešto mjesta ponovno se šalje osvježena vrijednost kredita
- svojstva PCIe 3.0:
 - 8 GHz => 8 GT/s \approx 1 GB/s u jednom smjeru (2 GB/s dvosmjerno)
 - za 16 staza (x16), dvosmjerno \approx 32 GB/s
- Preko PCIe sučelja (opisanog) razmjenjuju se PCI zahtjevi te zahtjevi za prekid

4.5.1. PCI adrese, naredbe, postavljanje

- Identifikacija pojedine naprave, tj. funkcije naprave sastoji se od:
 - adresa segmenta (engl. *PCI Segment Group*) – 16 bita (samo PCIe)
 - adrese PCI sabirnice (engl. *bus*) – 8 bita
 - adrese naprave – 5 bita
 - adrese funkcije – 3 bita
- Inicijalizacija naprave (funkcije) (npr. BIOS ili OS):
 - mora se napraviti "ručno", "prozivanjem" (šalju se poruke)
 - kad se ustanovi da je neka naprava prisutna treba ju inicijalizirati:
 - * odrediti adrese, tj. regije na kojima će ona biti vidljiva (veličine od 16 B do 3 GB)
 - * kontroler dobiva te adrese da bi mogao na njih odgovarati (od procesora, tj. glavne sabirnice) i proslijediti dalje prema napravi
 - * adrese se upisuju u napravu, tj. u zaglavlje konfiguracijskog bloka (slika 4.12.)
 - * osim navedenih 64 B, konfiguracijski blok sadrži ukupno 256 B, tj. do 4096 za PCIe
 - * ostatak konfiguracijskog bloka je ovisan o napravi
 - * može se koristiti i za opise sposobnosti (engl. *capabilities*)
- Naprave se dalje koriste preko njihovih adresa (logički, i dalje se koristi PCIe)
- Naredbe napravama – ima ih 12 ([sve](#)); 4 najvažnije: čitaj/piši, čitaj/piši iz/u postavke

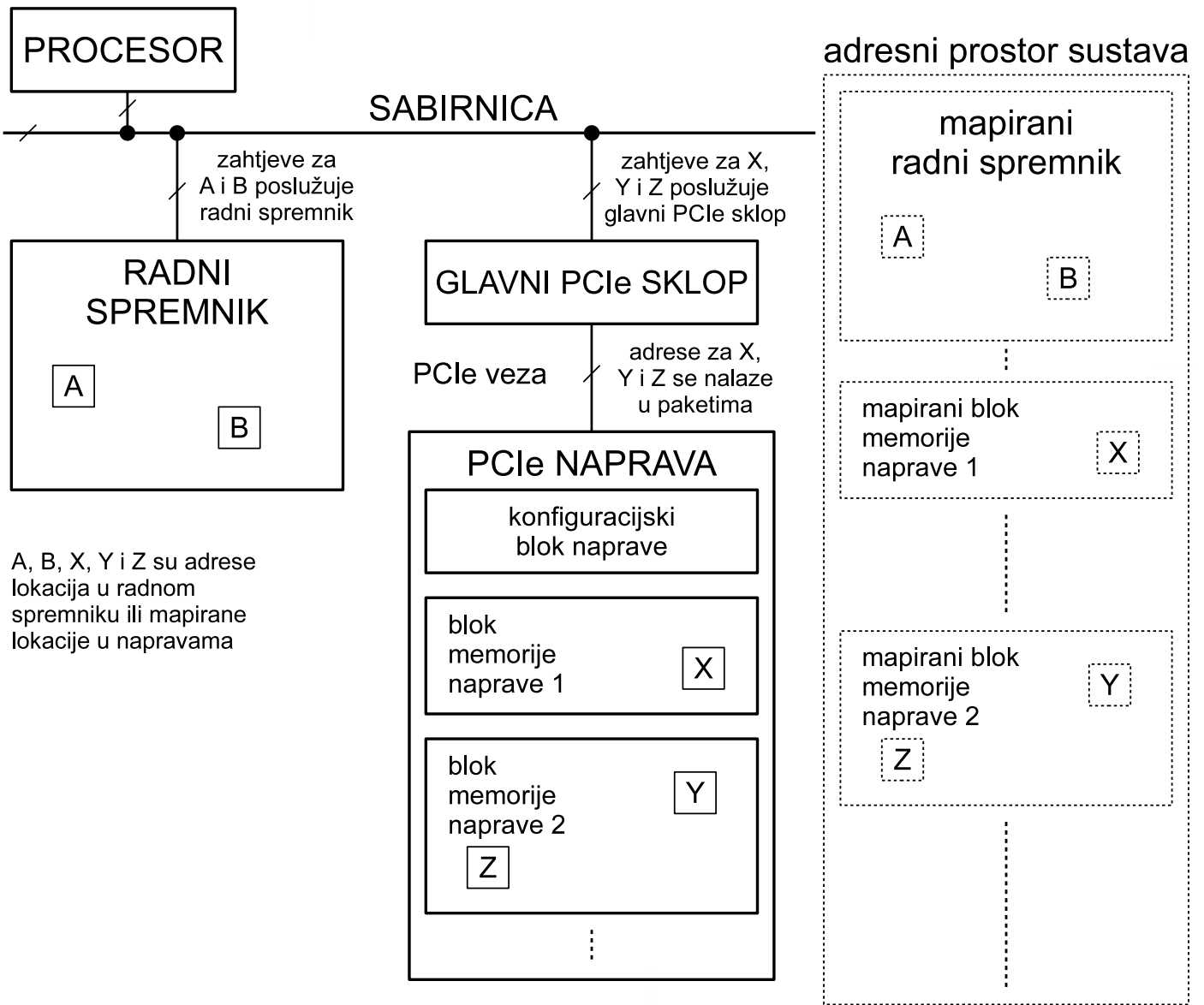
Najvažnija polja zaglavlja konfiguracijskog bloka

- Device ID, Vendor ID, Subsystem ID, Subsystem Vendor ID
 - da OS (ili BIOS) dozna o kakvoj se napravi radi
- Status – za odgovor na podržane mogućnosti i dojavu greške

31		16 15		0
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Lat. Timer	Cache Line S.	0Ch
Base Address Registers				10h
				14h
				18h
				1Ch
				20h
				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved			Cap. Pointer	34h
Reserved				38h
Max Lat.	Min Gnt.	Interrupt Pin	Interrupt Line	3Ch

Slika 4.12. Zaglavlje konfiguracijskog bloka za PCI (za tip 0 – naprave) (slika s Wikipedije)

- Command – bitmaska, preko koje se mogu omogućiti ili onemogućiti neke značajke
- Header Type – Type 0 za naprave, Type 1 za root, switch, bridge, ...
- Interrupt Pin, Interrupt Line – koji prekid da naprava koristi (šalje)
- Base Address Registers (BAR)
 - BAR0-BAR5 do 6 32-bitovnih registara (ili manje 64-bitovnih)
 - OS upiše adresu memorije
 - da bi doznao koliko taj segment treba biti velik upiše sve jedinice te pročita vrijednost (i napravi dvojni komplement), pa opet upiše adresu
 - kad se koristi 64 bita onda se koriste dvije takve lokacije
 - ilustracija s adresama prikazana je na slici 4.13.



Slika 4.13. Primjer sustava s jednom PCIe napravom

4.6. Serial ATA

- Ime: Serial AT Attachment (AT=Advanced Technology)
- Slično kao i za PCIe/USB:
 - serijski prijenos bit-po-bit
 - diferencijsko signaliziranje, 2 para žica A+/A- i B+/B-, po jedan za svaki smjer
 - komunikacija 1-na-1: SATA kontroler (host bus adapter, HBA) – disk
 - s diskom se neizravno komunicira preko kontrolera (na matičnoj ploči)
 - protokoli za komunikaciju s kontrolerom: “noviji” AHCI, NVMe; proširenje ATAPI; starije ATA
- AHCI za komunikaciju sa SATA kontrolerom (procesor–kontroler)
 - ime: Advanced Host Controller Interface
 - half duplex – komunikacija samo u jednom smjeru istovremeno
 - optimiran za rad s tvrdim diskovima
 - definira način zadavanja operacija čitanja/pisanja, upravljanja, ...
- NVM Express (NVMe)
 - ime: Non-Volatile Memory Host Controller Interface Specification (NVMHCIS)
 - optimiran za SSD (taj tip memorije)
 - mogućnosti za više redove zahtjeva s duljim nizom
 - omogućuje full duplex
- Neki SSD diskovi se spajaju preko M.2 sučelja (SATA Express => PCIe, veće brzine)
 - umjesto osmišljavanja protokola za još veće brzine iskoristiti PCIe koji to već nudi
 - dodatna proširenja sučelja (USB 3) ...
- Jedinica podataka blok (sektor); najčešće skupina blokova
- Osnovni zahtjevi prema disku: čitanje/pisanje skupine blokova
- Uglavnom se koristi adresiranje blokova (engl. *Logical block addressing – LBA*) – cijeli disk je logičko polje blokova koje poslužuje sam disk (elektronika na njemu) – OS ne mora brinuti o geometriji (glavama, stazama, sektorima)
- Korištenje sklopova s izravnim pristupom spremniku (DMA) za prijenos – manje opterećenje procesora; on može nešto drugo raditi dok prijenos nije gotov (kad dobije prekidni signal DMA sklopa)
- Brzine prijenosa: do 600 MB/s preko SATA priključka, do 2 GB/s preko M.2

Pitanja za vježbu 4

1. Navesti osnovna svojstva komunikacije preko serijske veze (npr. RS232).

Odgovor:

*bitovi idu jedan za drugim po istoj žici (u jednom smjeru)
dodaju se sinkronizacijski i zaštitni bitovi na početak i kraj*

2. Navesti osnovna svojstva komunikacije preko paralelne veze.

Odgovor:

*bitovi se sinkrono paralelno prenose preko više žica
više se prenese u jednom taktu, ali je potrebno sinkronizirati sve signale*

3. Navesti nekoliko protokola koji koriste serijsku i nekoliko koji koriste paralelnu vezu. Gdje se u današnjem računalu koristi serijska a gdje paralelna veza?

Odgovor:

serijska: USB, PCI-e, SATA paralelna: PATA, komunikacija s memorijom

4. Usporediti serijsku i paralelnu komunikaciju. Koje su prednosti serijske?

Odgovor:

*prednosti serijske:
jednostavnija izvedba – manje sinkronizacije među žicama, jednostavnije sinkronizirati strane
zbog navedenog veća brzina za prijenos preko veći duljina*

5. Velika većina komunikacija s napravama se danas obavlja korištenjem serijskih veza (npr. PCIe, USB, SATA). Koji su osnovni razlozi za prijelaz na serijsku vezu (npr. PCI→PCIe, PATA→SATA, USB umjesto paralelnih veza)?

Odgovor:

jednostavnije za izvesti, moguće veće frekvencije (brže na duljim vodičima)

6. Opisati kako su USB naprave spojene u računalu: logički, fizički.

Odgovor:

*logički na zajedničku sabirnicu
fizički odvojeni, stablasto povezani preko mostova do glavnog upravljača
komunikacija uvijek ide samo između naprave i glavnog upravljača (ne između različitih USB naprava)*

7. Opisati osnovni način rada (komunikacije) naprave koja je spojena na USB priključak.

Odgovor:

čeka da ju glavni upravljač prozove pa odgovara na zahtjeve (master-slave načelo)

8. Svaki paket kod USB protokola započinje sinkronizacijskim zaglavljem 0b00000001. Zašto baš takvim brojem s puno nula na početku?

Odgovor:

obzirom da se koristi NRZI kodiranje gdje je svaka nula promjena signala, ove početne nule omogućuju drugoj strani da sinkronizira primitak ostatka signala

9. USB je po nazivu "sabirnica". Jesu li u stvarnosti svi podaci koji se preko tog protokola šalju vidljivi svima (spojenim na tu "sabirnicu")?

Odgovor:

ono što šalje glavni upravljač šalje se svima (svi vide)

ono što šalje naprava prema glavnom upravljaču vide samo mostovi koji sudjeluju u prijenosu takve poruke prema glavnom upravljaču (ne i ostale naprave)

10. Što su to okviri, transakcije, paketi u kontekstu protokola USB?

Odgovor:

sabirnicom se komunicira preko okvira

svaki okvir započinje s oznakom početka okvira, nizom transakcija te oznakom kraja transakcije

svaka transakcija se sastoji od paketa

svaki paket se sastoji od sinkronizacijskog zaglavlja, tijela paketa i oznake kraja paketa

tijelo paketa može biti TOKEN ili podatak ili sinkronizacija (ACK)

11. Što je to adresa naprave, a što adresa funkcije naprave (kod protokola USB)?

Odgovor:

svaka naprava ima svoju adresu (7-bitovnu)

svaka naprava ima "funkcije", koje predstavljaju neki oblik komunikacije

12. Za prijenos podataka USB-om preko protoka (engl. *stream*) koriste se cjevovodi: izokroni, prekidni i veliki. Opisati njihova svojstva i namjenu.

Odgovor:

izokroni: prijenos podataka, garantirana propusnost

prekidni: kada su potrebni brzi odgovori

veliki: za prijenos puno podataka, ali bez garancije propusnosti

13. Zašto za mnoge USB naprave nije potrebno instalirati upravljačke programe, već ih operacijski sustav može koristiti s postojećim?

Odgovor:

naprave su podijeljene po tipovima – za svaki tip je definirano kako se koristi, koje "sučelje" mora naprava imati

OS koristi naprave prema tipovima – već ima upravljačke programe za sve tipove sučelja, nisu potrebna posebna (osim za posebne naprave koja imaju još proširenije funkcionalnosti)

14. Kod serijskih veza (npr. USB, PCI-e, SATA) komunikacija u jednom smjeru se izvodi samo preko dva vodiča (izvorni i invertirani signal). Kako se sinkroniziraju strane (kad se može slati, kojom brzinom, provjera ispravnosti primljenog)? Zašto se dodatno koristi i kodiranje, npr. za 8 bita informacije se šalje 10 bita preko žice?

Odgovor:

*svaki prijenos započinje sa sinkronizacijskim zaglavljem – tako da ona druga strana uskladi svoj sat za primanje podataka
uz podatke šalju se i zaštitni bitovi za provjeru ispravnosti prijenesa
kodiranje 8 na 10 se koristi da se izbjegnu "monotoni" podaci (npr. puno uzastopnih jedinica) koji mogu narušiti sinkronizaciju kod primatelja – podaci se stoga kodiraju u vrijednosti koje nemaju previše uzastopnih jedinica*

15. Kako se prenose podaci preko PCIe? Koliko vodiča se koristi, je li moguć istovremeni prijenos u oba smjera?

Odgovor:

*više staza
preko svake staze podaci mogu u oba smjera paralelno
za svaki smjer se koriste dvije žice preko kojih se prenosi signal i njegov inverz
(radi smanjenja utjecaja na okolinu te otpornosti na utjecaj okoline)*

16. Neka PCIe naprava ima oznaku x4 (koristi četiri staze). Ako se u jednom smjeru preko jedne staze može prenositi 1 GB u sekundi koliko se s tom napravom može podataka razmijeniti u sekundi (ukupno, u oba smjera)?

Odgovor:

*$4 \times 2 = 8$ – preko toliko žica paralelno mogu podaci
 $8 \times 1 \text{ GB} = 8 \text{ GB}$*

17. Kako se prenose podaci u/iz memorije PCIe naprave?

Odgovor:

preko PCIe kontrolera: procesor koristi adrese na koje odgovara PCIe kontroler koji se druge strane preko PCIe protokola kontaktira naprave

18. Usporediti PCI i PCIe. Koje su prednosti sabirnice PCIe?

Odgovor:

*PCI koristi zajedničku sabirnicu na koju su spojene PCI naprave (samo jedna može u istom trenutku prenositi podatke)
PCIe koristi vezu 1 na 1 – svaka naprava zasebno je spojena s upravljačem i može paralelno raditi*

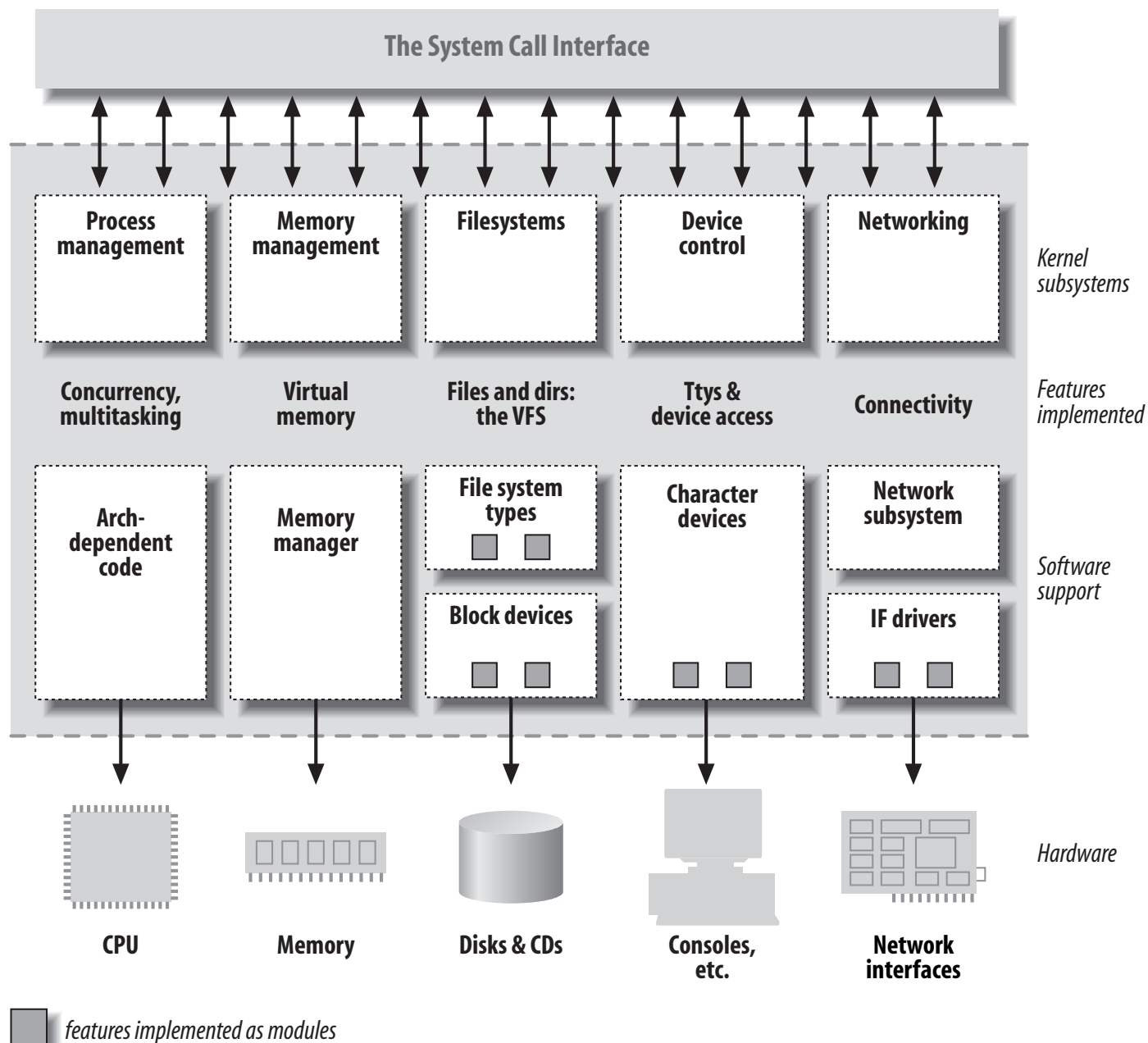
19. Koja je jedinica podataka (ne upravljačkih naredbi) koja se prenosi preko SATA protokola? Zašto nije proizvoljna veličina podataka?

Odgovor:

SATA se koristi za komunikaciju s diskom. Jedinica podataka na disku je sektor a ne oktet. Stoga je i osnovna jedinica podatak iskazana u tim jedinicama.

5. Podrška za ostvarenje naprava u Linuxu

Kao i svaki operacijski sustav, i Linux je iznutra vrlo složen sustav. Upravljanje napravama zahtijeva poznavanje mnogo detalja iz interne organizacije jezgre. Stoga će u uvodu biti riječi o nekoliko njih. Nije cilj potpuno upoznati jezgru već samo osnovne koncepte koji se koriste. Puno više detalja može se pronaći u drugim izvorima, npr. knjizi [Corbet, 2005 – Linux Device Drivers, Third Edition] koja je korištena pri pripremi ovih materijala. Slika 5.1. prikazuje osnove elemente Linuxove jezgre i položaj upravljačkih programa i modula u njima.



Slika 5.1. Podjela Linuxove jezgre prema operacijama (izvor [Corbet, 2005])

5.1. Izvođenje jezgrina koda

Jezgrin kod se može izvoditi u tri “okoline”:

1. u jezgrinom kontekstu procesa (kad proces poziva jezgrinu funkciju)
2. u kontekstu jezgrine dretve (jezgrin proces)
3. bez konteksta dretve – “atomarno” (obrada prekida, alarmi, taskler, softirq)

5.1.1. Poziv jezgrine funkcije iz programa

- kada se jezgrina funkcija zove iz programa, npr. `read()` tada se pri ulasku u jezgru prelazi u jezgrin kontekst procesa
- umjesto dretve koja se izvodila u korisničkom načinu rada, sada se aktivira jezgrina dretva koja izvodi jezgrinu funkciju – ta dretva je “produžetak” korisničke (ali sada u jezgri)
- u tom kontekstu dohvatljiva je jezgra, njene strukture podataka i interne funkcije
- podaci o pripadajućoj dretvi i procesu u jezgri su izravno dohvatljivi, npr. preko varijable `current` koja opisuje dretvu (`struct task_struct`)
- adresni prostor jezgre odvojen je od adresnog prostora procesa (pogotovo nakon otkrivenih sigurnosnih prijetnji Meltdown i Spectre)
- dohvat korisničkih podataka mora se obavljati korištenjem posebnih funkcija (npr. `copy_to_user`, `copy_from_user`)
- jezgrina dretva može se i blokirati posebnim “unutarnjim” mehanizmima, sličnima “vanjskim” (semafori, monitori, ...)

Tablica 5.1. Primjer izvođenja programa u korisničkom i jezgrinom načinu rada

korisnički način (program)	jezgrin način (jezgrine funkcije)
<pre>int fd, a, b; char buf[BUFFER_SIZE]; ssize_t x, y; fd = open("ime-datoteke", O_RDONLY);</pre>	
	<ul style="list-style-type: none">- pronalazak datoteke- provjera prava pristupa- stvaranje opisnika datoteke
<pre>x = BUFFER_SIZE; y = read(fd, buf, x);</pre>	
	<ul style="list-style-type: none">- pronalazak opisnika u procesu- obavljanje operacije (može zahtijevati i čekanje)- kopiranje podataka u buf
<pre>sscanf(buf, "%d %d", &a, &b); printf("a+b=%d\n", a + b);</pre>	
	ispis niza znakova

5.1.2. Poziv jezgrine funkcije preko prekida

- naprave pri svom radu mogu generirati zahtjev za prekid
- pri prihvatu zahtjeva započinje obrada prekida (nakon pohrane konteksta prekinute dretve)
- obrada se izvodi u jezgrinom načinu i nije povezana s prekinutom dretvom
- “obrada” koja je ovako započela ne smije u svom kodu imati blokirajuće pozive, npr. ne smije pozvati ČekajSemafor ili slično, mora završiti “atomarno”
- poželjno je da takav atomaran način rada traje što kraće, da bi se mogli prihvaćati novi zahtjevi
- ako je potrebno više vremena za obradu, onda se posao obrade dijeli na dva dijela
 1. neophodni dio – “top half” u Linux terminologiji (Interrupt Service Routine na Windowsima)
 2. dodatni dio – “bottom half” (Interrupt Service Thread na Windowsima)
- “neophodni dio” je onaj dio koji je započeo s obradom prekida, “atomarni dio”
- “dodatni dio” se obavlja naknadno, s dozvoljenim prekidanjem
- mogućnosti za dodatni dio u Linuxu:
 1. red poslova – “workqueue”
 2. višedretvena obrada prekida – “threaded IRQs”
 3. lagani prekid – “softirq”
 4. zadaćić – “tasklet”
- odgovarajućim sučeljem se unutar obrade prekida (top half) stvori/pripremi posao koji će se na jedan od navedenih načina odraditi kasnije, nakon završetak ove obrade
- idejno: prva dva načina za duže duže poslove, druga dva za kratke
- prva dva načina (workqueue i threaded IRQ) se preporučuju (u novije vrijeme)
 - obavljaju se u kontekstu posebnih jezgrinih dretvi
 - mogu biti odgođeni, blokirani i slično (koristiti blokirajuće pozive)
- druga dva načina (softirq i tasklet) se izvode atomarno
 - u kontekstu neke dretve koja obavlja takve poslove
 - takva dretva obrađuje zahtjeve nekim redom
 - jednom započeti moraju biti dovršeni
 - ne smiju biti blokirani, tj. koristiti pozive koji bi to mogli izazvati
 - preporuka: umjesto njih koristiti prve dvije metode; omogućuju bolje performanse i brži odziv na nove prioritetnije zahtjeve

5.2. Dozvoljene i nedozvoljene operacije u kodu jezgre

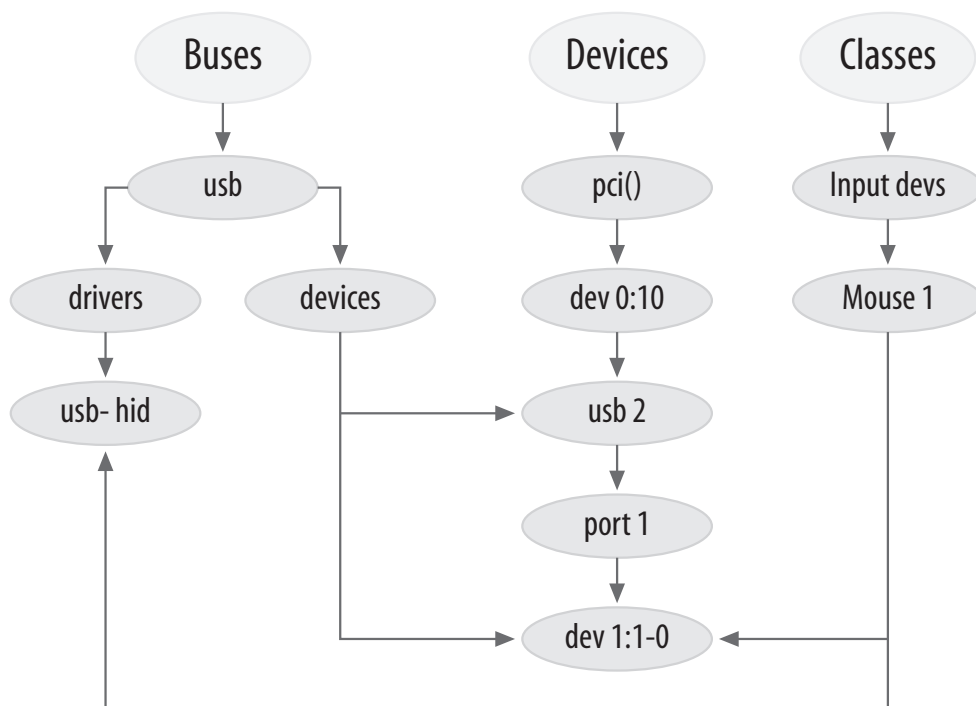
- sinkronizacijske funkcije
 1. atomaran kod (obrada prekida=top half, tasklet, softirq, alarmi)
 - nikakve funkcije koje mogu blokirati
 - nikakve funkcije koje traže kontekst dretve (npr. semafori, monitori)
 - mogu funkcije s radnim čekanjem: spinlock
 2. kod s kontekstom (u kontekstu procesa, workqueue, threaded IRQ)
 - sve funkcije (interne jezgrine)
- pristup adresnom prostoru procesa
 - samo ako se j.f. pozvala iz procesa
 - dohvat/pohrana podataka iz/u proces posebnim funkcijama
- realni brojevi – brojevi koji traže koprocessor (npr. float, double)
 - obzirom da takvi brojevi nisu potrebni u jezgri, oni se ne koriste

5.3. Moduli

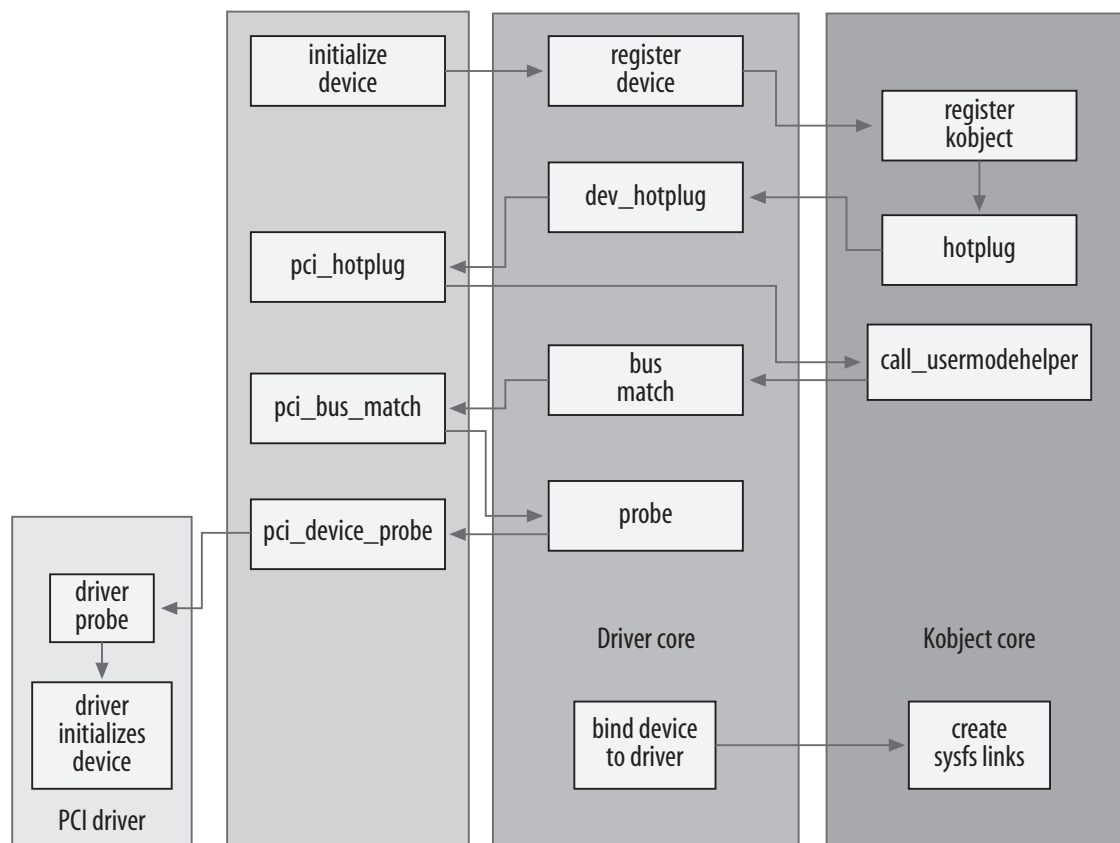
- upravljački program može biti na dva načina uključen u jezgru Linuxa:
 1. permanentno (statički)
 2. dinamički kao modul
- upravljački program može biti pripremljen zajedno s cijelom jezgrom Linuxa ili pripremljen kao “modul” koji se po potrebi, dinamički učitava u jezgru te miče kada više nije potreban
- pri pripremi jezgre može se odrediti koji dijelovi će se učitati s jezgrom (stalno biti prisutni), a koji će se učitavati na zahtjev
- za pripremu (kompajliranje) samo jednog modula nije neophodan cijeli izvorni kod Linuxa već samo njegova zaglavlja (.h datoteke) i postavke
- modul se posebnim naredbama uključuje i isključuje iz jezgre
 1. uključivanje: `insmod ime-datoteke-s-modulom [argumenti]`
 2. isključivanje: `rmmod ime-modula`
- u primjerima idućeg poglavlja koriste se moduli te su tamo objašnjeni detalji pripreme i rada s njima

5.4. Upravljački programi

- upravljački programi upravljaju stvarnim ili virtualnim napravama
- mogu imati i prihvat prekida, ako njihova naprava izaziva prekide koji već nisu obuhvaćeni općenitim mehanizmima
- sastoje se od svoje strukture podataka te funkcija
- funkcije su tzv. registrirane funkcije (engl. *callbacks*)
 - upravljački program registrira svoje funkcije za neke događaje
 - npr. kad se iz programa pozove *piši* koji u konačnici dolazi do te naprave, onda se negdje, u lancu poziva koji se generiraju na taj zahtjev, poziva i funkcija koju je upravljački program registrirao za tu operaciju
 - uobičajene takve funkcije su: *otvari, zatvori, čitaj, piši, pomakni, asinkrono čitaj/piši, šalji upravljačke naredbe, mapiraj memoriju, zaključaj, ...*
 - neke od takvih operacija su prikazane u sklopu primjera u sljedećem poglavlju
- tri su osnovne klase naprava (stvarnih i virtualnih) u Linuxu
 1. znakovne naprave – character devices
 - naprave koje daju/primaju niz bajtova; npr. tipkovnica, miš, terminal, pislač
 2. blokovske naprave – block devices
 - naprave kojima je jedinica podataka blok; najčešće naprave koje ostvaruju datotečne sustave
 3. mrežna sučelja – network devices
 - naprave kojima je svrha ostvarenje komunikacije (koriste se iz mrežnog podsustava)
- u idućem poglavlju su prikazani primjeri sa znakovnim napravama
- interna organizacija naprava u Linuxu je vrlo složena
 - klase/tipovi naprava; sabirnice (stvarne/virtualne); dio sustava, podsustava, sloja, ...
 - hijerarhijska organizacija; integracija s ostalim elementima jezgre
 - detaljnije na: [The Linux driver implementer's API guide] i u knjizi [Corbet, 2005], posebice u poglavlju "Chapter 14: The Linux Device Model"
 - zaglavlja [linux/device.h](#), [linux/device/bus.h](#), [linux/device/class.h](#), [linux/device/driver.h](#)
 - neće se detaljnije opisivati u ovim materijalima; iduće dvije slike 5.2. i 5.3. iz [Corbet, 2005] su dodane samo radi ilustracije problema

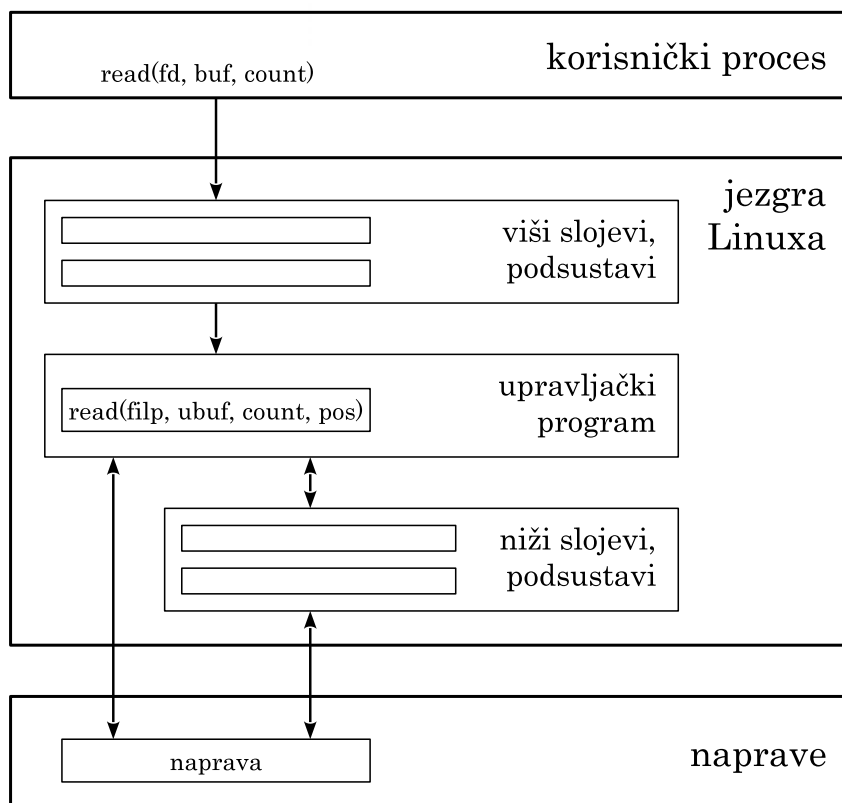


Slika 5.2. Primjer struktura podataka povezanih s jednom napravom (izvor [Corbet,2005])



Slika 5.3. Dodavanje PCI naprave u Linux (izvor [Corbet,2005])

- korištenje naprava potrebno je ostvariti kroz infrastrukturu Linuxa – potrebno je novi upravljački program uklopiti u takvo okruženje
- zahtjev korisnika, npr. čitanje iz naprave, prolazi kroz te slojeve da bi došlo do funkcije upravljačkog programa
- upravljački program, ovisno o napravi, može izravno koristiti napravo, ili pak preko dodatne infrastrukture koju Linux nudi (npr. za korištenje USB naprave, PCIe, ...)
- Slika 5.4. prikazuje takvo slojevito korištenje



Slika 5.4. Slojevitost u korištenju infrastrukture Linuxa

- Pri izradi primjera upravljačkih programa (prikazanih u sljedećem poglavlju) znali su se događati problemi (oops). U takvim događajima se u dnevnik zapisuju dostupne informacije, a jedna od njih je i lanac poziva funkcija unutar jezgre koji su doveli do problema. Primjer jednog takvog lanca funkcija koje se pozivaju kad se iz procesa pozove `write` nad napravom:
 - `do_syscall ⇒ sys_write ⇒ ksys_write ⇒ vfs_write ⇒ shofer_write`
 - ova zadnja je ostvarena u upravljačkom programu, ostalo su slojevi jezgre

Pitanja za vježbu 5

1. Unutar jezgre kôd se može izvoditi u “različitim kontekstima”. Koji su to i koja ograničenja postavljaju pojedini konteksti?

Odgovor:

- a) *bez konteksta – atomarno: prekidi, alarmi; kod se ne može blokirati, nema (jednostavnog) pristupa procesu – nije vezan uz korisnički proces*
- b) *u kontekstu jezgrine dretve: prekidne dretve, odgođeni poslovi; kod se može blokirati, nema (jednostavnog) pristupa procesu – nije vezan uz korisnički proces*
- c) *u jezgrinom kontekstu procesa: kad dretva pozove jezgrinu funkciju; kod se može blokirati, može se pristupiti procesu – izvođenje je povezano s korisničkim procesom*

2. Što treba napraviti/koristiti ako u jezgrinoj funkciji treba pristupiti adresnom prostoru procesa?

Odgovor:

obzirom da su adresni prostori procesa i jezgre odvojeni, onda treba koristiti posebne funkcije koje će kopirati podatke od/prema procesu, npr. `copy_to_user` ili `copy_from_user`

3. Obrada prekida naprave vrlo je bitan dio upravljanja napravom, ali može bitno utjecati i na svojstva sustava. Zbog čega? Koje mogućnosti u Linuxu stoje na raspolaganju za obradu prekida? Koja su njihova svojstva / kada ih koristiti?

Odgovor:

u obradi prekida zabranjen je prihvrat novih prekida (najčešće); to može utjecati na vrijeme odgovora na bitne zahtjeve prema sustavu mogućnosti:

- a) *koristiti sklop za prihvrat prekida s kojim se može ostvariti prekid obrade manjeg prioritet radi obrade zahtjeva većeg prioriteta*
- b) *podijeliti obradu na dva dijela: prvi dio odmah obraditi a ostatak kasnije, s nekom dretvom prikladnog prioriteta*
 - i) *koristiti mogućnosti Linux za to – višedretvena obrada prekida*
 - ii) *"ručno" pripremiti okolinu za to preko redova poslova*

4. Što se smije a što ne koristiti u jezgri? Je li to ovisi o kontekstu u kojem se izvodi kod jezgre? Kako?

Odgovor:

u atomarnom kontekstu ne smiju se koristiti funkcije koje mogu blokirati u kontekstu jezgrine dretve mogu se koristiti sve interne funkcije, pa i one koje mogu blokirati tu dretvu

5. Što je to modul u kontekstu jezgre Linuxa? Čemu služi?

Odgovor:

dio jezgre koji sadrži neku funkcionalnost, npr. upravljanje nekom napravom

6. Navesti tri osnovne klase naprava u Linuxu.

Odgovor:

znakovne, blokvske i mrežne

7. Koja je zadaća upravljačkog programa naprave?

Odgovor:

*popunjava "rupe" u jezgri radi ostvarenja komunikacije s napravom
OS već ima većinu infrastrukture gotovo*

6. Primjeri upravljačkih programa u Linuxu

Izvorni kôd korišten u ovom poglavlju nalazi se na [Shofer]

6.1. Priprema radnog okruženja

- U ovom poglavlju su pokazani primjeri koji su nastali korištenjem virtualnog računala
 - VMware Workstation Player + Ubuntu 20.04
 - alati: gcc, make + neko razvojno okruženje (Atom, Visual Studio Code...)
 - za razvoj modula potreban je izvorni kôd Linuxa, tj. barem njegova zaglavlja
 - upute o pripremi ovakva okruženja su na webu; ukratko:
 - * ažuriranje: `sudo apt-get update && sudo apt-get upgrade`
 - * novo: `sudo apt-get install gcc make linux-headers-generic git`
- Rad u jezgri (a moduli postaju dio jezgre pri učitavanju) je “opasan”!
 - pri testiranju sustav se može zablokirati, srušiti
 - često spremati napravljeno (npr. `git add/commit/push`)
 - najbolje imati i kopiju sustava (virtualnog računala), za svaki slučaj
 - primjeri problema:
 - * upravljački program zaglavi pri pokretanju programa
 - ne može se prekinuti program
 - ne može se izbaciti modul
 - ponekad se ipak (“nasilno”) može ugasiti proces i onda se i modul može izbaciti
 - ponekad treba restartati računalo (ponekad i “hard” načinom)
 - * upravljački program zaglavi u petlji u kojoj ispisuje u dnevnik – dnevnik popuni sav slobodan prostor na disku, sustav se više ne može ni ugasiti, ni normalno pokrenuti

6.2. Moduli

Izvorni kôd korišten u ovom odjeljku je iz direktorija `01-hello-world`

6.2.1. Izvorni kôd

- Izvorni kôd modula mora imati dvije funkcije:
 1. funkciju za inicijalizaciju modula pri učitavanju u jezgru
 2. funkciju za oslobađanje sredstava pri micanju modula iz jezgre
- Takve dvije funkcije treba napisati i posebno označiti s `module_init` i `module_exit`
- Najjednostavniji primjer modula koji samo ispisuje poruku pri učitavanju te pri micanju je naveden u nastavku

Isječak kôda 6.1. Izvorni kôd modula

```
#include <linux/module.h>
#include "config.h"

MODULE_AUTHOR(AUTHOR);
MODULE_LICENSE(LICENSE);

static int __init shofer_module_init(void)
{
    printk(KERN_NOTICE "Module " MODULE_NAME " started\n");
    return 0;
}

static void __exit shofer_module_exit(void)
{
    printk(KERN_NOTICE "Module " MODULE_NAME " unloaded\n");
}

module_init(shofer_module_init);
module_exit(shofer_module_exit);
```

- Uz samu datoteku s kôdom obično ide i zaglavlje s definicijom struktura, konstanti i pomoćnih funkcija/makroa
 - U ovom primjeru se u njoj nalaze samo konstante (AUTHOR, LICENSE, MODULE_NAME)
 - Ime modula: “shofer” => “šofer” kao “driver”, a ima i “fer” u imenu
 - konstrukti MODULE_AUTHOR i MODULE_LICENSE dodatno opisuju modul, ali ne utječu na njegovu funkcionalnost
- Oznake uz funkcije __init i __exit označavaju te funkcije, ali imaju svrhu samo kada se modul priprema za učitavanje s jezgrom (statički)
 - oznaka __init omogućava jezgri da nakon početna izvođenja te funkcije izbaci kôd ove funkciju iz memorije
 - oznaka __exit omogućuje jezgri da pri učitavanju modula ne učita ovu funkciju, jer moduli učitani s jezgrom ostaju cijelo vrijeme u jezgri, “ne izlaze”
- modul se nakon učitavanja pokreće i izvodi u jezgri
- ispisi iz modula uobičajeno se zapisuju u dnevnik sustava
- funkcija za ispis je `printk`, vrlo slična običnom `printf`-u uz nekoliko razlika
 - nema podrške za realne brojeve – oni se u jezgri ne koriste
 - oznaka razine važnosti predaje se kao prvi znak (KERN_NOTICE u primjeru)
 - * sve razine, od najmanje: KERN_DEBUG, KERN_INFO, KERN_NOTICE, KERN_WARNING, KERN_ERR, KERN_CRIT, KERN_ALERT, KERN_EMERG
 - ispis ide u dnevnik jezgre (npr. datoteku /var/log/kern.log)

6.2.2. Izgradnja modula

- Obzirom da se izgrađuje modul za jezgrom, moraju biti prisutne sve informacije o jezgri
- Te se informacije nalaze u izvornom kôdu jezgre Linuxa, odnosno, uz sama zaglavlja – zato je za prevođenje bilo potrebno dohvatiti zaglavlja jezgre, i to baš iste inačice koja se koristi

na sustavu (paket linux-headers-generic)

- Priloženi Makefile je samo dio uputa – ostatak se nalazi uz zaglavlja jezgre

Isječak kôda 6.2. Makefile modula

```
ifneq ($(KERNELRELEASE),)
# call from kernel build system
obj-m := shofer.o
else
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
modules:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
clean:
$(MAKE) -C $(KERNELDIR) M=$(PWD) clean
```

- Obzirom da u početku varijabla `KERNELRELEASE` nije postavljena, prvo je aktivna "else" grana u Makefile-u
 - najprije se postavljaju varijable `KERNELDIR` i `PWD`
 - prvi predmet prevođenja je `modules` (drugi je `clean`)
 - predmet `modules` rekurzivno pokreće početni Makefile izvorna kôda jezgre Linuxa (`make -C $(KERNELDIR)` s predmetom `modules` i varijablom `M=$(PWD)`)
 - ovaj predmet nalaže izgradnju modula za jezgru iz direktorija `M=$(PWD)`
 - tada se ovaj prikazani Makefile ponovno učitava i izvodi se "if" dio, tj. `obj-m := shofer.o` koji definira koje module treba izgraditi (samo jedan, `shofer.o`)

6.3. Jednostavna virtualna znakovna naprava

Izvorni kôd korišten u ovom odjeljku je iz direktorija `02-simple-device`

6.3.1. Izgradnja naprave

1. Identifikacijski broj naprave

- Naprave u Linuxu su identificirane brojem (tip `dev_t`) koji se sastoji od dva dijela (broja):
 - a) glavnog (engl. *major*)
 - b) pomoćnog (engl. *minor*)
- Pri stvaranju nove naprave (tj. učitavanju njenog upravljačkog programa) potrebno je prvo rezervirati jedan takav identifikator
- Ima nekoliko sličnih načina kako to napraviti, jedan je preko sučelja:

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor,
    unsigned count, const char *name);
```

- Funkcija od OS-a traži `count` brojeva za naprave
- Početni pomoćni broj se definira s `baseminor`, uobičajeno nula
- Ime s kojim su naprave povezane je zadano s `name`

- Prvi identifikator vraća se u dev
- Ako je traženo više brojeva, oni su uzastopni dodijeljenom

Isječak kôda 6.3. Rezervacija broja za napravu

```
/* u funkciji shofer_module_init */
dev_t dev_no = 0;
[...]
retval = alloc_chrdev_region(&dev_no, 0, 1, DRIVER_NAME);
```

2. Registracija naprave (i njenih operacija)

- Registracija se može napraviti na nekoliko načina; u nastavku je jedan od njih (“na višoj razini”)
- Nakon dobivanja broja treba definirati što naprava radi
- Treba inicijalizirati strukturu `struct cdev` (`cdev_init`) s operacijama naprave, opisane u strukturi `struct file_operations`
- Struktura `struct file_operations` može sadržavati kazaljke na puno funkcija, ali treba popuniti samo one koje naprava ostvaruje, npr. *otvori*, *čitaj*, *piši*, *zatvori*
- Takvu strukturu se predaje funkciji `cdev_add` koja ju dodaje u sustav (“registrira”)

Isječak kôda 6.4. Registracija naprave

```
/* u config.h */
struct shofer_dev {
    dev_t dev_no;          /* device number */
    struct cdev cdev;     /* Char device structure */
    struct buffer *buffer; /* Pointer to buffer */
};
[...]
/* globalna varijabla u shofer.c */
static struct file_operations shofer_fops = {
    .owner = THIS_MODULE,
    .open = shofer_open, /* funkcije postoje */
    .release = shofer_release,
    .read = shofer_read,
    .write = shofer_write
};
[...]
/* u funkciji shofer_module_init */
shofer = shofer_create(dev_no, &shofer_fops, buffer, &retval);
[...]
static struct shofer_dev *shofer_create(dev_t dev_no,
    struct file_operations *fops, struct buffer *buffer, int *retval)
{
    struct shofer_dev *shofer = kmalloc(sizeof(struct shofer_dev), GFP_KERNEL);
    [...]
    cdev_init(&shofer->cdev, fops);
    shofer->cdev.owner = THIS_MODULE;
    shofer->cdev.ops = fops;
    *retval = cdev_add (&shofer->cdev, dev_no, 1);
    [...]
}
```

3. Korištenje naprave preko datotečnog sustava

- Opisanim sučeljima se stvara naprava, ali ona još nije vidljiva u datotečnom sustavu

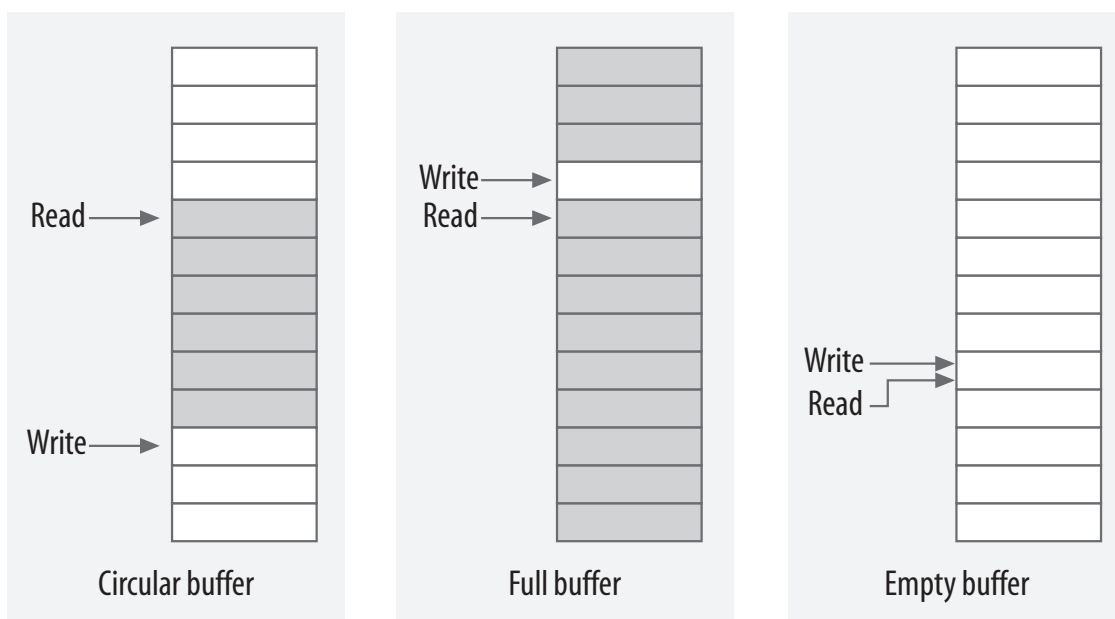
- Dodavanje naprave u datotečni sustav može se obaviti naredbom `mknod` (u ljsuci)


```
sudo mknod /dev/ime-naprave c glavni-broj pomoćni-broj
```
- Nakon toga naprava se iz programa može otvoriti i koristiti kao datoteka


```
open("/dev/ime-naprave", zastavice)
```

6.3.2. Kružni međuspremnik

- U ovom primjeru se stvara jedna naprava koja koristi jedan kružni međuspremnik u koji upisuje znakove poslane operacijom `write` te ih vraća pri operaciji `read`.
- Kružni međuspremnik je vrlo učinkovit način korištenja međuspremnika i često se koristi pri komunikaciji.
- Za praćenje popunjenih i praznih mjesta potrebne su dvije kazaljke: kazaljka na prvo prazno mjesto (ulaz/write) i kazaljka na prvo nepročitano (izlaz/read).



Slika 6.1. Primjer kružnog međuspremnika (izvor [Corbet, 2005])

- U primjeru je korišten kružni međuspremnik (engl. *circular buffer*) podržan sučeljem (kodom) jezgre definiranim u `linux/kfifo.h`
- Sučelje radi samo za međuspremnike čija je veličina potencija broja dva $N=2^P$ radi izbjegavanja operacija dijeljenja; ionako su takve veličine međuspremnika uobičajene u jezgri
- povećanje kazaljke u međuspremniku kapaciteta N mjesta (u C -u):
 - s modulo aritmetikom: $i = (i+1) \% N$
 - brže, bez dijeljenja: $i = (i+1) \& (N-1)$
- broj nepročitanih elemenata: $(ulaz - izlaz) \& (N-1)$
- broj praznih mjesta: $(izlaz - ulaz) \& (N-1)$
- navedene operacije su “skriveno”, tj. međuspremnik se koristi isključivo preko sučelja
- primjer korištenja sučelja uz kôd jezgre Linuxa: [samples/kfifo/bytestream-example.c](https://samples.kfifo.org/bytestream-example.c)
- ukratko o najosnovnijem sučelju u sljedećem isječku kôda

Isječak kôda 6.5. Primjer korištenja sučelja kfifo međuspremnika

```
#include <linux/kfifo.h>

struct kfifo fifo; /* 'opisnik' međuspremnika */

[...] /* inicijalizacija */
    /* buffer - adresa već rezerviranog bloka memorije veličine 'size' */
    ret = kfifo_init(&fifo, buffer, size);
/* ili */
    ret = kfifo_alloc(&fifo, size, GFP_KERNEL); /* on rezervira memoriju */

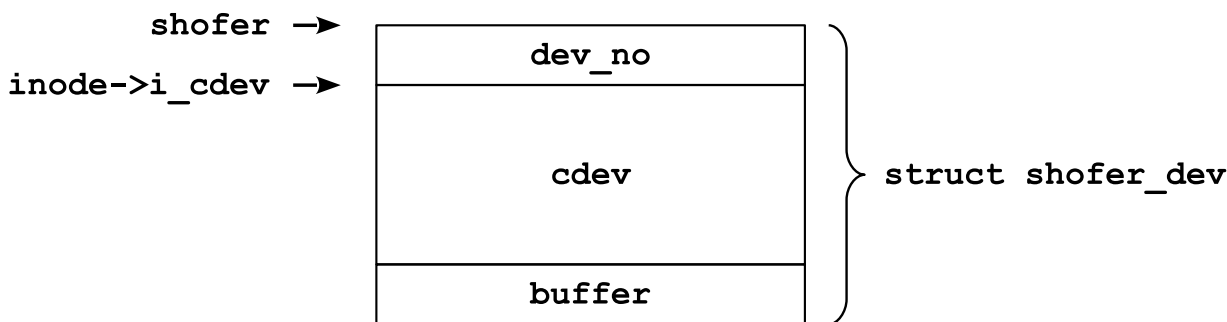
[...] /* stavljanje/uzimanje iz međuspremnika */
    /* copied - broj stavljenih/uzetih elemenata */
    copied = kfifo_put(&fifo, i); /* dodaj jedan element */
    copied = kfifo_get(&fifo, &i); /* pročitaj jedan element */
    copied = kfifo_in(&fifo, buf, size); /* dodaj 'size' elemenata */
    copied = kfifo_out(&fifo, buf, size); /* pročitaj 'size' elemenata */

    /* dohvaćanje elemenata, bez njihova micanja iz međuspremnika */
    copied = kfifo_peek(&fifo, &i); /* dohvati prvi element */
    copied = kfifo_out_peek(&fifo, buf, size); /* dohvati 'size' elemenata */

[...] /* kopiranje iz međuspremnika izravno u prostor korisnika i obratno */
    ret = kfifo_to_user(&fifo, (char __user *) ubuf, size, &copied);
    ret = kfifo_from_user(&fifo, (char __user *) ubuf, size, &copied);
    /* ret je 0 kad je sve OK ili -EFAULT u slučaju greške */
[...]
    kfifo_free(&fifo); /* oslobađanje memorije, samo uz kfifo_alloc */
```

6.3.3. Opis operacija upravljačkog programa znakovne naprave

- Ostvarene su funkcije za otvaranje naprave, čitanje i pisanje te zatvaranje naprave; sve se ove operacije pozivaju samo kad ih neki proces zatraži
- Kroz slojeve OS-a odgovarajući pozivi iz procesa (npr. `open`) dođu do povezanih operacija (npr. `shofer_open`), uz proširenje/zamjenu parametara (slika 5.4.).
 - poziv iz programa: `int open(const char *pathname, int flags);`
 - operacija naprave: `int open(struct inode *inode, struct file *filp);`
 - preko dobivenih parametara, u funkciji naprave može se doći do njena opisnika:
 - * `shofer = container_of(inode->i_cdev, struct shofer_dev, cdev);`
 - * u `inode->i_cdev` nalazi se kazaljka koja je predana sa `cdev_add`, kazaljka na element `struct cdev cdev` unutar strukture `struct shofer_dev`
 - * makro `container_of` pomoću te adrese pronalazi početak strukture `shofer_dev` koja sadrži sve elemente upravljačkog programa naprave (slika 6.2.)
 - samo drugi element (`struct file *filp`) šalje se u ostale funkcije (`read/write/*`), pa je uobičajeno da se on iskoristi za spremanje kazaljke na podatke upravljačkog programa
 - * `filp->private_data = shofer;`



Slika 6.2. Dohvat adrese objekta uz poznavanje adrese njegova elementa

- operacije čitanja i pisanja su sličnije običnim

```

ssize_t read(struct file *filp, char __user *ubuf,
             size_t count, loff_t *f_pos);
ssize_t write(struct file *filp, const char __user *ubuf,
             size_t count, loff_t *f_pos);

```

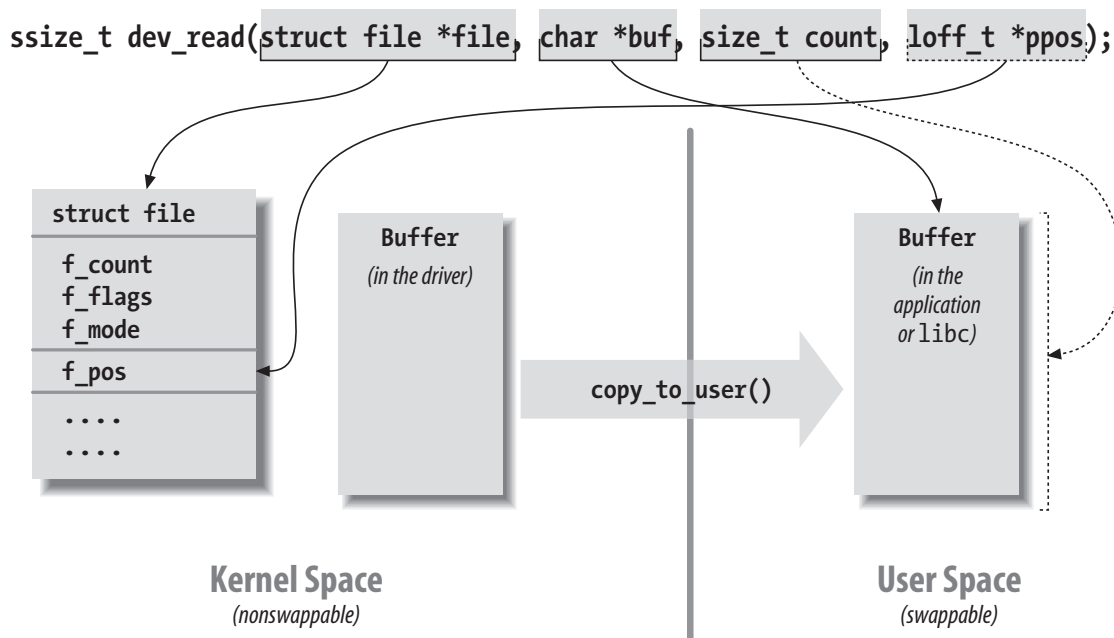
- oznaka `__user` označava da je kazaljka na adresni prostor u procesu, te da se ne smije izravno koristiti izvan jezgre
- kopiranje u/iz procesa u memoriju jezgre obavljati sa:

```

unsigned long copy_to_user(void __user *to, const void *from,
                          unsigned long n);
unsigned long copy_from_user(void *to, const void __user *from,
                          unsigned long n)

```

- slika 6.3. ilustrira funkciju `read`



Slika 6.3. Primjer prijenosa parametara u funkciju za čitanje (izvor [Corbet, 2005])

- Za zauzimanje dinamičke memorije u jezgri koristiti `kmalloc`
 - `kazaljka = kmalloc(veličina, zastavice);`
 - neke zastavice (prefix GFP – *get-free-pages*):
 - * GFP_ATOMIC – dohvat memorije za jezgru, izvan konteksta dretvi, u atomarnim okruženjima – ne blokira
 - * GFP_KERNEL – dohvat memorije za jezgru, unutar konteksta dretvi – može blokirati
 - * GFP_USER – dohvat memorije unutar adresnog prostora procesa (dohvatljive jezgri, ali i procesu izvan jezgrinih funkcija) – može blokirati
 - * `__GFP_DMA` – dostupna za DMA; `__GFP_COLD` – koja nije u priručnim spremnicima
 - * druge mogućnosti za zastavicu – [pogledati upute](#)
 - druga sučelja:
 - * uvijek ista veličina zahtjeva: `kmem_cache_create/destroy/alloc/free`
 - * hitni zahtevi koji moraju biti posluženi: `mempool_create/destroy/alloc/free`
 - * alokacija stranica (kad treba više memorije): `kmem_getpages/freepages/*`
 - * alokacija virtualne memorije: `vmalloc/vfree/*`
- Objekti koji bi mogli biti u situaciji da se paralelno koriste trebaju biti zaštićeni
 - u primjeru se objekt `struct buffer` štiti zaključavanjem preko `mutex`, `struct mutex lock` koji je dio te strukture
 - npr. dva različita procesa mogu istovremeno htjeti obaviti operacije čitanja/pisanja nad tom napravom – tj. mijenjati taj međuspremnik
 - pošto se kôd naprave (barem `open/close/read/write`) obavlja u jezgrinom kontekstu pozivajuća procesa taj se kôd može blokirati (`mutex_lock`)
 - u kôdu se koristi `mutex_lock_interruptible`, funkcija koja će zaključati `mutex` ako je slobodan ili blokirati dretvu ako nije; međutim, za razliku od obične `mutex_lock` ova dozvoljava da ju prekine signal; ako je dretva blokirana i dođe joj signal čekanje se prekida i funkcija `mutex_lock_interruptible` vraća grešku – najčešće se u tom slučaju izlazi iz jezgrine funkcije s odgovarajućom greškom
 - ključ se u ovom primjeru mogao staviti u objekt naprave umjesto međuspremnika; ponekad to ima smisla; međutim, u idućim primjerima više naprava može koristiti isti međuspremnik, pa ima više smisla ključ povezati s međuspremnikom

Isječak kôda 6.6. Zaključavanje

```

/* pri početku funkcija shofer_read i shofer_write */
if (mutex_lock_interruptible(&buffer->lock))
    return -ERESTARTSYS; /* neka ponovno pozove read nakon obrade signala */
[...]
/* pri kraju obiju funkcija */
mutex_unlock(&buffer->lock);

```

Slijedi dio kôda za otvaranje naprave, čitanje i pisanje

Isječak kôda 6.7. Operacije open, read i write iz 02-simple-device/shofer.c

```
/* Called when a process calls "open" on this device */
static int shofer_open(struct inode *inode, struct file *filp)
{
    struct shofer_dev *shofer; /* device information */
    shofer = container_of(inode->i_cdev, struct shofer_dev, cdev);
    filp->private_data = shofer; /* for other methods */
    return 0;
}

/* Read count bytes from buffer to user space ubuf */
static ssize_t shofer_read(struct file *filp, char __user *ubuf, size_t count,
    loff_t *f_pos /* ignoring f_pos */)
{
    ssize_t retval = 0;
    struct shofer_dev *shofer = filp->private_data;
    struct buffer *buffer = shofer->buffer;
    struct kfifo *fifo = &buffer->fifo;
    unsigned int copied;

    if (mutex_lock_interruptible(&buffer->lock))
        return -ERESTARTSYS;

    retval = kfifo_to_user(fifo, (char __user *) ubuf, count, &copied);
    if (retval)
        printk(KERN_NOTICE "shofer:kfifo_to_user failed\n");
    else
        retval = copied;

    mutex_unlock(&buffer->lock);

    return retval;
}

/* Write count bytes from user space ubuf to buffer */
static ssize_t shofer_write(struct file *filp, const char __user *ubuf,
    size_t count, loff_t *f_pos /* ignoring f_pos */)
{
    ssize_t retval = 0;
    struct shofer_dev *shofer = filp->private_data;
    struct buffer *buffer = shofer->buffer;
    struct kfifo *fifo = &buffer->fifo;
    unsigned int copied;

    if (mutex_lock_interruptible(&buffer->lock))
        return -ERESTARTSYS;

    retval = kfifo_from_user(fifo, (char __user *) ubuf, count, &copied);
    if (retval)
        printk(KERN_NOTICE "shofer:kfifo_from_user failed\n");
    else
        retval = copied;

    mutex_unlock(&buffer->lock);

    return retval;
}
```

6.3.4. Paralelno pozivanje funkcija upravljačkog programa

- Funkcije upravljačkog programa, one definirane u strukturi `struct file_operations`, moguće je pozivati paralelno, od strane različitih dretvi (*reentrant* funkcije)
- Stoga ovakve funkcije treba dobro analizirati i po potrebi zaštititi
- Ako se koriste zajedničke varijable onda ih zaključati, barem u dijelovima kôda
- Odabrali prikladan mehanizam: ako se koriste samo iz dretvi, onda može mutex, inače, ako se koriste i iz atomarnih okruženja, onda koristiti zaključavanje radnim čekanjem (*spinlock*)
- Ako su samo neke varijable u pitanju, možda je moguće koristiti samo atomarne operacije koje ne trebaju zaključavanja (npr. Gcc Built-in Functions for Memory Model Aware Atomic Operations; podrška u Linuxu preko `linux/atomic.h` i sličnih sučelja); više o tome u poglavlju 8.1.

6.4. Logiranje, parametri, liste, odgoda

Izvorni kôd korišten u ovom odjeljku je iz direktorija `03-lists-delay`

Idući, prošireni primjer korištenja naprava demonstrira nekoliko mehanizama:

1. jednostavnije pisanje u dnevnik, s više “automatski generiranih” informacija
2. korištenje parametara prilikom učitavanja modula
3. korištenje listi preko mehanizama Linuxa
4. odgoda izvođenja jezgrinih dretvi

6.4.1. Logiranje

- funkcija `printk` nije složena, ali može se bolje i jednostavnije
- pri debugiranju dobro je znati s koje linije je ispis
- automatsko dodavanje imena modula olakšava pretraživanje dnevnika (log datoteka)
- ispis je potreban pri debugiranju, ali suvišan nakon toga – umjesto dodavanja/brisanja ispisa, on može ostati u kôdu, ali “neaktivan” kad nije potreban
- preprocesorskim direktivama se može upravljati ponašanjem makroa
 - `ccflags-y += -DSHOFER_DEBUG` u Makefile-u (obično je `CFLAGS`, ali ...)
 - kad se taj makro postavi, `LOG` ispisuje
 - kad bi se on maknuo, `LOG` makro se zamjenjuje “ničim”
- kad bi bilo više datoteka s izvornim kôdom, trebalo bi dodati i ime datoteke

Isječak kôda 6.8. Logiranje preko makroa

```
#define klog(LEVEL, format, ...) \
printf ( LEVEL "[shofer] %d: " format "\n", __LINE__, ##__VA_ARGS__ )

#ifdef SHOFER_DEBUG
#define LOG(format, ...)      klog(KERN_DEBUG, format,  ##__VA_ARGS__)
#else /* !SHOFER_DEBUG */
#define LOG(format, ...)
#endif /* SHOFER_DEBUG */
```

6.4.2. Parametri pri učitavanju modula

- Pri učitavanju modula s `insmod` mogu se zadati parametri koji se mogu iskoristiti unutar modula za prilagodbu ponašanja
 - Npr. `sudo insmod shofer driver_num=5`
 - varijabla `driver_num` mora biti definirana u kôdu
 - preko makroa `module_param(driver_num, int, S_IRUGO)`; se povezuje vrijednost zadana u naredbenoj liniji s varijablom
 - varijabli treba statički pridijeliti vrijednost koju će ona imati, ako nije zadana preko naredbenog retka
 - opis parametra se daje s `MODULE_PARM_DESC`, npr.
`MODULE_PARM_DESC(driver_num, "Number of devices to create");`
- Opis modula, koji uključuje i popis parametara, može se dobiti naredbom `modinfo`

```
$ modinfo shofer.ko
[...]
parm:          buffer_size:Buffer size in bytes (int)
parm:          buffer_num:Number of buffers to create (int)
parm:          driver_num:Number of devices to create (int)
```

6.4.3. Liste

- Liste se koriste za povezivanje mnoštva različitih elemenata
- Radi jednostavnosti korištenja i čitljivosti kôda, u Linuxu se liste koriste kroz sučelje definirano u [linux/list.h](#)
- Lista ostvarena navedenim sučeljima je dvostruko povezana – svaki element ima kazaljku na prethodni i sljedeći element
- Element u kojem su te kazaljke jest `struct list_head` s kazaljkama `next` i `prev` (`struct list_head { struct list_head *next, *prev; }; linux/types.h`)
- Ovaj element ujedno služi i kao zaglavlje liste i kao element koji se ugrađuje u druge strukture koje žele biti u listi.

Isječak kôda 6.9. Primjer zaglavlja liste i umetanje u strukturu

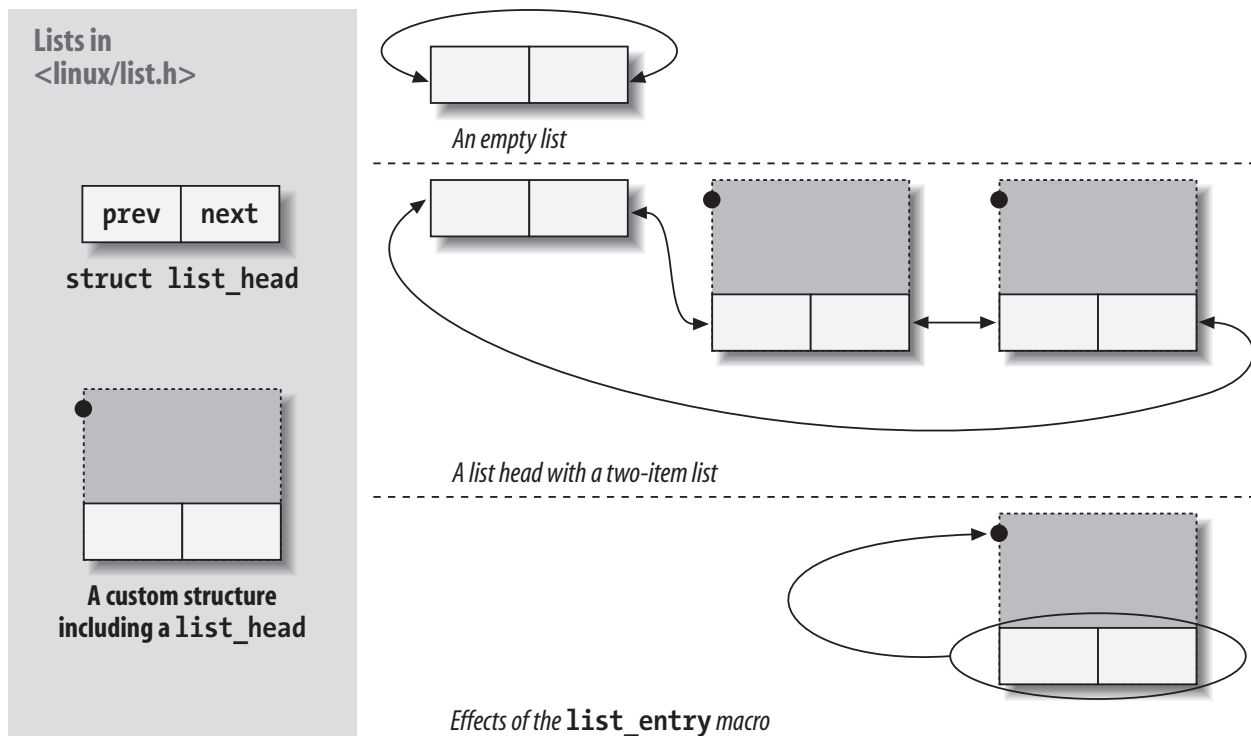
```
/* moglo bi ovako: */
struct list_head buffers_list;
/* ili ovako za statički napravljene i inicijalizirane liste */
struct list_head buffers_list = {.prev = &buffers_list, .next = &buffers_list};

/* ali najbolje je sve raditi preko makroa i funkcija: */
LIST_HEAD(buffers_list);

/* inicijalizacija dinamički stvorene liste bila bi: */
INIT_LIST_HEAD(buffers_list); /* ovdje je buffers_list kazaljka */

/* u strukturama koje trebamo stavljati u listu treba dodati struct list_head: */
struct buffer {
    struct kfifo fifo;
    [...]
    struct list_head list; /* ovo ime elementa se koristi u funkcijama/makroima */
};
```

- Slika 6.4. ilustrira strukturu podataka i primjere lista



Slika 6.4. Primjer liste i struktura podataka (izvor [Corbet, 2005])

- Rad s listama ide preko makroa/funkcija
- Većina sučelja očekuje/vraća kazaljku na element liste (`struct list_head`), a ne objekt
- npr. dodavanje na kraj: `list_add_tail(&buffer->list, &buffers_list);`
- Sučelja koja vraćaju kazaljku na objekt imaju "entry" u svom nazivu
- Dohvat kazaljke na objekt, ako je poznata kazaljka na element `struct list_head` unutar objekta, dobiva se s `list_entry`
- prvi element liste:
`buffer = list_first_entry(&buffers_list, struct buffer, list);`
druga dva argumenta su potrebna da se dođe do adrese objekta koji je u listi
- idući element: `buffer = list_next_entry(buffer, list);`
- iteracija po listi – u petlji se elementu pristupa preko prve kazaljke (`shofer` u primjeru):
`list_for_each_entry(shofer, &shofers_list, list){`
- micanje iz liste: `void list_del(struct list_head *entry)`
- slika 6.4. ilustrira strukturu podataka i primjere lista

6.4.4. Odgoda izvođenja unutar jezgre

- Odgoda je moguća samo za kôd koji se izvodi u kontekstu neke dretve, npr. za jezgrinu funkciju pozvanu od strane procesa, za poslove iz reda poslova (*workqueue*)
- Odgoda nije moguća unutar obrade prekida (*top half*), unutar *taskleta* niti alarma
- Odgodu koristiti vrlo oprezno, jer je moguće da će ona i druge stvari odgoditi. Npr. ne ju koristiti nakon zaključavanja mutex-a ili slično; u kontekstu dretvi koje izvode redom poslove (i drugih procesa), ...

- Mehanizam odgode `wait_event_interruptible_timeout` (primjer u nastavku) očekuje i opisnik reda u kojem će dretva čekati; neki drugi oblici čekanja (npr. mutex) već imaju takvu strukturu u sebi

Isječak kôda 6.10. Primjer odgode izvođenja u jezgri

```
int retval;
wait_queue_head_t wait;
init_waitqueue_head (&wait);
retval = wait_event_interruptible_timeout(wait, 0, msecs_to_jiffies(delay_ms));
```

- Funkcija (makro) `wait_event_interruptible_timeout` omogućuje čekanje i na uvjet, ali je on u prethodnom primjeru postavljen na nula i nikada neće biti ispunjen, pa je blokiranje definirano vremenom – brojem prekida sata – jiffies
- većina internih jezgrinih funkcija koristi vrijeme preko broja otkucaja sata koji se ažuriraju kroz varijablu `jiffies`
- iako je broj otkucaja sata zapisan u konstanti HZ, bolje je koristiti makroe za pretvorbu vremena, kao u prikazanim primjerima
- zbog `interruptible` u imenu čekanje može biti prekinuto i prije, zbog signala (kada funkcija vraća `-ERESTARTSYS`)
- kada bi uvjet bio postavljen, npr. $x > y$, i u nekom trenutku zadovoljen – onaj koji ga zadovoljava treba pozvati `wake_up(&wait)`; tada funkcija vraća neprospavano vrijeme
- ako (kada) je zadani interval istekao, dretva se odblokira i funkcija vraća nulu

6.5. Alarm, red poslova

Izvorni kôd korišten u ovom odjeljku je iz direktorija `04-timers-workqueue`

6.5.1. Alarmi

- Unutar jezgre mogu se definirati i događaji koji trebaju biti pokrenuti u budućnosti – tzv. alarmi (engl. *timer*)
- Objekt alarma se definira sa: `struct timer_list timer;`
- inicijalizira sa: `timer_setup(&timer, timer_function, 0);`
- postavlja vrijeme aktiviranja u `timer.expires`
- te aktivira sa: `add_timer(&timer);`
- Funkcija `timer_function` mora postojati – ona definira operaciju alarma

Isječak kôda 6.11. Primjer korištenja alarma

```
static struct timer_list timer;
[...]  
timer_setup(&timer, timer_function, 0);  
timer.expires = jiffies + msecs_to_jiffies(TIMER_PERIOD);  
add_timer(&timer);  
[...]  
del_timer(&timer);  
[...]
```

```

static void timer_function(struct timer_list *t)
{
    [...]
    /* promijeni alarm tako da se (opet) aktivira u budućnosti */
    mod_timer(t, jiffies + msecs_to_jiffies(TIMER_PERIOD));
}

```

- Izvorni kôd u fukciji alarma mora biti atomaran – ne smiju se koristiti blokirajući pozivi (a mnogi su takvi, npr. `kmalloc`)
- Može se koristiti `spin_lock` – radno čekanje, koje se koristi za vrlo kratka zaključavanja
- Kad alarm više nije potreban obrisati ga s `del_timer`

6.5.2. Red poslova

- Poslovi jezgre koji mogu potrajati, a ne izvode se u kontekstu procesa, mogu se ostvariti mehanizmom reda poslova (`workqueue`)
- Redovi poslova jedan su od načina kako napraviti duži dio posla pri obradi prekida
- Posao (u takvom sustavu) može koristiti i čekanje jer se izvodi u kontekstu neka jezgrina procesa
- Za poslove se može stvoriti vlastiti red ili koristiti globalni
- Poslovi u istom redu se izvode jedan za drugim! Stoga je potrebno paziti “što se radi” u pojedinom poslu, jer odgoda u nekom poslu ne odgađa samo njega već i sve ostale u tom redu.
- Posao se definira funkcijom koja se poziva pri aktivaciji posla.
- Stvaranje reda za poslove može se napraviti sa:
 - `create_workqueue` i `create_singlethread_workqueue`
- Razlika između ova dva je što se s prvim stvara po jedna dretva za svaki procesor, a u drugom samo jedna dretva – te će dretve izvoditi poslove koji dolaze u red
- Pojedini posao se:
 1. definira preko strukture `struct work_struct work;`
 2. inicijalizira s `INIT_WORK(&work, work_op);`
 - `work_op` je ime funkcije koja će se pozvati za obradu tog posla
 3. dodaje u red s `queue_work(work_queue, &work)`
 - `work_queue` je kazaljka na red poslova dobivena s `create_workqueue`
- Funkcija za obradu posla prima kazaljku na sam posao `struct work_struct`
- U obradi su obično potrebni i podaci. Zato se `struct work_struct` umeće u veću strukturu koja ima te podatke, a do nje se dođe korištenjem kazaljke na taj posao:
 - neka postoji struktura čiji je dio i opisnik posla:


```
struct nesto { [...]; struct work_struct mywork; [...] }
```
 - kazaljka na `mywork` nekog objekta se daje opisniku posla i to će biti argument funkcije
 - u funkcija posla `void work_op(struct work_struct *work)` do kazaljke na objekt dolazi se sa:

```
struct nesto *x = container_of(work, struct nesto, mywork);
```

- Ako je potrebno čekati da se neki posao obavi to se može napraviti na nekoliko (puno) načina: mehanizam događaja (engl. *events*) i mehanizam završetaka (engl. *completion*) su prikazani u nastavku

6.5.3. Sinkronizacija preko mehanizma događaja

- Čekanje na događaj već je dijelom pokazano prilikom ostvarenja odgode
- U ovom dijelu će se prikazati čekanje na ispunjenje uvjeta
- Kao i kod alarma, i ovdje prvo treba definirati i inicijalizirati red za blokirane dretve (`struct wait_queue_head wq; init_waitqueue_head(&wq)`)
- Čekanje na događaj ispunjenja uvjeta obavlja se sa: `wait_event(wq, uvjet);`
- `wait_event` je makro u kojem se nalazi petlja; svaki puta kad se dretva odblokira provjerava se uvjet; ako on nije ispunjen dretva se opet blokira
- Odblokiranje se može napraviti s `wake_up` ili sličnom funkcijom – ove se funkcije mogu pozvati i iz obrade prekida, alarma i sličnih atomarnih operacija

Isječak kôda 6.12. Primjer sinkronizacije događajima

```
/* u definiciji strukture */
struct shofer_dev {
    [...]
    struct wait_queue_head wqueue;
};
[...]
struct wq_data {
    struct work_struct work;
    struct buffer *buffer;
    char *buf;
    size_t len;
    unsigned int copied;
    int op; /* 0 - read, 1- write */
    union {
        struct wait_queue_head *queue;
        struct completion *completion;
    } wakeup;
};
[...]
/* pri inicijalizaciji upravljačkog programa */
init_waitqueue_head(&shofer->wqueue);
[...]
/* u funkciji shofer_write */
struct wq_data wqd; /* reserved on stack, since here we wait */
[...]
wqd.copied = 0;
wqd.wakeup.queue = &shofer->wqueue;
wait_event(shofer->wqueue, wqd.copied > 0);
[...]
/* u funkciji posla, workqueue_operations */
wqd->copied = kfifo_out(fifo, wqd->buf, wqd->len);
wake_up_all(wqd->wakeup.queue);
```

6.5.4. Sinkronizacija preko mehanizma završetaka

- Završeci (*engl. completion*) su mehanizmi sinkronizacije koji omogućuju da jedna strana (dretva) bude blokirana na upitu o završetku nečega, što neka druga strana dojavljuje
- Potreban objekt: `struct completion cmpl;`
- Inicijalizacija: `init_completion(&cmpl);`
- Čekanje da nešto bude gotovo: `wait_for_completion(&cmpl);`
- Označavanje da nešto je gotovo: `complete(&cmpl);`
- Oznaka da je nešto gotovo može se napraviti i prije nego netko pozove `wait*` funkciju
- Operacija `complete()` može se pozvati i iz obrade prekida, alarma i sličnih atomarnih operacija

6.6. Višedretvena obrada prekida

- Ponekad upravljački program naprave treba moći obraditi prekid naprave (ponekad to ide kroz sustav, ovisno o sučelju)
- Da bi se pozvala neka funkcija upravljačkog programa na prekid naprave, potrebno je registrirati funkciju za taj prekid
- Sučelje takve funkcije za registraciju je:

```
int request_irq ( unsigned int irq,
                 irq_handler_t handler,
                 unsigned long flags,
                 const char *name,
                 void *dev );
```

- Osnovni parametri su identifikator prekida (`irq`) i funkciju koju treba pozvati (`handler`)
- Ukoliko posao koji funkcija treba obaviti traje duže, preporuke je da se u toj funkciji napravi samo neophodni dio posla (*top half*), a ostatak (*bottom half*) delegira na neki drugi mehanizam (*softirq, tasklet, workqueue, threaded IRQ*). Na ovaj način sustav ostaje kraće u načinu rada sa zabranjenim prekidanjem – omogućuje brži odziv na nove prekide.
- Ukoliko je unaprijed poznato da će taj dodatni dio trebati, preporuka je koristiti novo sučelje koje stvara dretvu koja će obaviti taj dodatni posao:

```
int request_threaded_irq ( unsigned int irq,
                          irq_handler_t handler,
                          irq_handler_t thread_fn,
                          unsigned long flags,
                          const char *name,
                          void *dev );
```

- U ovom drugom slučaju, pri pojavi prekida prvo se poziva prekidna funkcija (`handler`), a ako ona kao povratnu vrijednost vrati `IRQ_WAKE_THREAD` onda se aktivira i prekidna dretva koja izvodi funkciju (`thread_fn`)
- Prekidna dretva se može obavljati i duže, može biti odgođena, čekati na nešto i slično; može se obavljati i paralelno s drugim dretvama koje obrađuju druge jezgrine aktivnosti.
- Kad se prekid obradi do kraja (bilo u početnoj funkciji ili dretvi) funkcija treba vratiti

- Primjer u `05-irq-info` prikazuje primjer korištenje tog sučelja.
 - Primjer je naveden samo radi ilustracije, on NE RADI, ruši sustav!
 - Problem je što se prekid izaziva programski (simulira) te ga sustav iz nekog razloga ne povezuje s registriranom funkcijom (koju je on registrirao za prekid naprave)

6.7. Sučelja `poll` i `ioctl`

6.7.1. Rad s više datoteka preko `poll`

- Ponekad treba u programu čekati da se nešto dogodi nad skupom otvorenih “datoteka” (datoteka, naprava, mrežnih konekcija, ...)
 - Primjerice, u nekom poslužitelju koji je istovremeno spojen s puno klijenata, upravljački program treba reagirati kad se na bilo kojoj vezi nešto dogodi.
 - Iako se ovakav problem često rješava dretvama – svaka upravlja zasebnom vezom, to se može riješiti i s jednom dretvom koja ovako poslužuje sve.
- Sučelja koja to omogućuju su: `select`, `poll`, `epoll` i druga
- U nastavku je opisan `poll` i kako ga ostvariti u upravljačkom programu naprave
- Funkcija `poll` koja se poziva iz korisničkog programa je:
 - `int poll(struct pollfd *fds, nfd_t nfd, int timeout);`
 - `fds` – popis opisnika koje treba promatrati, preko strukture `struct pollfd`

```

struct pollfd {
    int    fd;           /* file descriptor */
    short events;       /* requested events */
    short revents;      /* returned events */
};

```
 - `nfd` – broj opisnika u polju `fds`
 - `timeout` – koliko treba čekati u milisekundama da se nešto dogodi (negativne vrijednosti za beskonačno, 0 bez čekanja)
- Npr. ako treba čekati da se pojavi novi podatak na nekoj od pet otvorenih datoteka (beskonačno dugo), potrebno je inicijalizirati polje od pet struktura `struct pollfd`, postaviti identifikatore otvorenih datoteka u element `fd`, koje događaje promatramo u `events` (npr. `POLLIN`) te pozvati:


```
ready = poll(pfds, 5, -1);
```
- Da bi navedeno bilo moguće i s napravama, upravljački program treba proširiti podrškom za `poll`

1. Treba ostvariti funkciju `poll`:

```
unsigned int poll(struct file *filp, poll_table *wait);
```

2. i dodati ju u `struct file_operations` naprave

```
static struct file_operations fops = {
    .owner = THIS_MODULE,
```

```

    [...]
    .poll =      poll
};

```

- Obzirom na mehanizam koji Linux koristi za ostvarenje te operacije dodatno je potrebno stvoriti redove `struct wait_queue_head` preko kojih će se proslijediti obavijest o događajima
- Ako se ostvaruje više događaja (npr. naprava je spremna za čitanje, pisanje, ...) za svaki takav događaj trebalo bi stvoriti zaseban red
- U funkciji `poll` treba označiti sve takve redove sa:

```

void poll_wait(struct file *filp, wait_queue_head_t *wait_address,
               poll_table *p)

```
- Prvi i zadnji parametar su oni proslijeđeni u `poll`.
- `wait_address` je kazaljka na red
- Svaki puta kad se dogodi neki događaj/promjena nad napravom koja mijenja neko od navedenih stanja, treba nakon toga pozvati i `wake_up` ili `wake_up_all` nad tim redom.

Isječak kôda 6.13. Isječci kôda iz 1ab2a za poll

```

/* u config.h */
struct shofer_dev {
    [...]
    struct wait_queue_head rq, wq; /* for poll */
};
/* u shofer.c */
static struct file_operations shofer_fops = {
    [...]
    .poll =      shofer_poll
};
[...]
static unsigned int shofer_poll(struct file *filp, poll_table *wait)
{
    struct shofer_dev *shofer = filp->private_data;
    struct buffer *buffer = shofer->buffer;
    struct kfifo *fifo = &buffer->fifo;
    unsigned int len = kfifo_len(fifo);
    unsigned int avail = kfifo_avail(fifo);
    unsigned int mask = 0;

    poll_wait(filp, &shofer->rq, wait);
    poll_wait(filp, &shofer->wq, wait);

    if (len)
        mask |= POLLIN | POLLRDNORM; /* readable */
    if (avail)
        mask |= POLLOUT | POLLWRNORM; /* writable */

    return mask;
}
[...]
static ssize_t shofer_read(struct file *filp, char __user *ubuf, size_t count,
                           loff_t *f_pos)
{
    [...]
    wake_up_all(&shofer->rq); /* for poll */
}

```

```

    return retval;
}
/* slično s shofer_write: */
wake_up_all(&shofer->wq); /* for poll */

```

6.7.2. Slanje naredbi/upita upravljačkom programu preko `ioctl`

- Ponekad je dozvoljeno korisničkim programima da šalju naredbe i zahtjeve “izravno” upravljačkom programu (naprave) preko sučelja `ioctl`

- sučelje za programe, u zaglavlju `sys/ioctl.h`:

```
int ioctl(int fd, unsigned long request, ...);
```

- Da bi naprava mogla reagirati na takav zahtjev treba u `struct file_operations` dodati sučelje `unlocked_ioctl`

```
long unlocked_ioctl (struct file *, unsigned int, unsigned long);
```

- Prefix `unlocked` naglašava da se radi o novom sučelju koje, za razliku od starog ne koristi zaključavanje jezgre (engl. *big kernel lock*); zaključavanje jezgre negativno utječe na učinkovitost i odziva sustava te je staro sučelje bez tog prefiksa izbačeno

- Drugi i treći argument su oni koji su i predani pri pozivu `ioctl` iz programa

- Što će ta funkcija naprave napraviti ovisi o napravi

- Povratna vrijednost se može koristiti za vraćanje uspješnosti (0) ili nečeg drugog

- Argumenti funkcija `ioctl/unlocked_ioctl` su:

- `fd` definira o kojem se opisniku radi (prethodno otvorenom)

- `request` je kombinacija bitova koji definiraju zahtjev

- često se koristi i treći argument `unsigned long arg` koji može biti kazaljka na neku strukturu podataka u procesu koji poziva `ioctl` – preko njega se može prenijeti dodatna struktura podataka ili u njega učitati vrijednosti

- Detaljnije o argumentu `request`

- teoretski može biti bilo kakav broj (kombinacija bitova)

- ipak je uobičajeno da je ta kombinacija bitova nešto predstavlja

- neka pravila i makroi su definirani na: [Documentation/ioctl/ioctl-number.txt](#) i [asm-generic/ioctl.h](#)

- ukratko:

- * bitovi 31-30: način pristupa (čitaj/piši)

- * bitovi 29-16: veličina trećeg argumenta u bajtovima, tj. veličina argumenta u memoriji na koji pokazuje treći argument

- * bitovi 15-8: tip naprave (pogledati dokument [ioctl-number.txt](#))

- * bitovi 7-0: serijski broj

- primjer stvaranja tog broja i njegova korištenja je u predlošku za `lab2b`

Dodatak u config.h

```
#ifndef SHOFER_C
... sve što je prije bilo u ovoj datoteci
#endif /* SHOFER_C */

/* for ioctl */
#define SHOFER_IOCTL_TYPE    0x8A    /* type, unused */
#define SHOFER_IOCTL_NR     1        /* serial number */
#define SHOFER_IOCTL_COPY   1        /* command */

struct shofer_ioctl {
    unsigned int command;
    unsigned int count;
};
```

Dodatak u test/ioctl.c

```
...
#include "../config.h"
...
    unsigned long request, num;
    struct shofer_ioctl cmd;
    ...
    num = atol(argv[2]);
    ...
    /* create request */
    request = _IOC(_IOC_WRITE, SHOFER_IOCTL_TYPE, SHOFER_IOCTL_NR,
                  sizeof(struct shofer_ioctl));

    /* command (COPY) and count are passed with struct_ioctl as
       third argument to ioctl */
    cmd.command = SHOFER_IOCTL_COPY;
    cmd.count = num;

    count = ioctl(fd, request, (unsigned long) &cmd);
```

Dodatak u shofer.c

```
#define SHOFER_C /* neposredno prije #include "config.h" */
...
long control_ioctl (struct file *filp, unsigned int request, unsigned long arg)
{
    ...
    struct shofer_ioctl cmd;

    if (_IOC_TYPE(request) != SHOFER_IOCTL_TYPE ||
        _IOC_NR(request) != SHOFER_IOCTL_NR)
    { ... }
    if (_IOC_SIZE(request) != sizeof(struct shofer_ioctl))
    { ... }
    retval = copy_from_user(&cmd, (const void __user *) arg,
                           sizeof(struct shofer_ioctl));
    /* sada je sve u cmd */
```

6.8. Dodatne operacije s napravama

- Rad naprava se često može pratiti i kroz datotečni sustav kroz /proc i /sys, ako naprava ostvaruje podršku kroz njih
- U prikazanim primjerima te mogućnosti nisu korištene

- Ostvarenje je slično već prikazanim sučeljima – koriste se druga sučelja i funkcije za registraciju
- za /sys pogledati sučelje: `sysfs_create_file`
- za /proc pogledati sučelje: `create_proc_entry`, `create_proc_read_entry`

6.9. Kako spriječiti i popraviti “oops”

6.9.1. Kopije, kopije, ...

- Napraviti kopiju radna okruženja (npr. cijelog virtualnog računala)
- Često kopirati kôd (npr. na svoj git repozitorij)

6.9.2. Praćenje sustava kroz dnevnike, procese, ...

- npr. `/var/log/kern.log`
- `tail -f /var/log/kern.log` – u drugom terminalu (ili u pozadini s `&`)
- `grep riječ-pretrage < /var/log/kern.log`
- `less /var/log/kern.log pa /riječ-pretrage`
- `ls -l /dev | grep ime-naprave`
- `ps -A | grep ime-programa` (ne ime modula)
- `df` – slobodni prostor na disku (pogledati za /)
- `top` – koji proces troši procesorsko vrijeme

6.9.3. “Pokušaji popravka”

1. Brisanje prevelikih datoteka dnevnika (zbog greške u modulu)

- `ls -lS /var/log | head`
- `sudo truncate -s 0 /var/log/ime-datoteke` (brisanje sadržaja datoteke)

2. Prekidanje programa

- `Ctrl+C` – pokušaj prekida pokrenuta programa
- `ps -A | grep ime-programa` – ispis svih procesa radi pronalaska željenog
- `kill -9 PID`
- `killall -9 ime-programa`

3. Brisanje modula

- `sudo rmmod -f ime-modula`
- ako prethodno ne obavi zadano, najprije probati ugasiti proces koji koristi zadani modul (koji ste pokrenuli)

4. Ponovno pokretanje računala – isprobavati redom dok se ne uspije

- kroz sučelje računala
- ako je računalo virtualno, kroz hipervizor (npr. VMware Workstation)

- restartati stvarno računalo, normalnim putem ili prekidačem (ako normalno zapne)
- ako se nakon ponovna pokretanja računalo ne uspije u potpunosti pokrenuti (dobiti grafičko sučelje), probati doći do konzole s Alt+F1-F6, te tada u konzoli pokušati popraviti sustav te ponovno restartati

Pitanja za vježbu 6

1. Koji su sve koraci potrebni za ostvarenje naprave da bi se ona mogla koristiti iz programa?

Odgovor:

1. ostvarenje upravljačkog programa kao modula, s osnovnim operacijama
2. učitavanje modula u jezgru

2. Koja su uobičajena sučelja koja znakovna naprava mora ostvariti, a da bi se uklopila u Linux?

Odgovor:

- učitaj-modul, oslobodi-modul*
- otvori/zatvori*
- čitaj/piši*

3. Prilikom uključivanja upravljačkog programa u Linux, osnovne funkcije su:

```
alloc_chrdev_region(&dev_no, 0, 3, DRIVER_NAME);  
cdev_add (&cdev, dev_no, 1);
```

Koju zadaću radi prva, a koju druga funkcija? Ne treba opisivati argumente.

Odgovor:

- alloc_chrdev_region rezervira ime za upravljački program, povezuje ga s internim brojevima (rezervira te brojeve)*
- cdev_add dodaje upravljački program u jezgru*

4. Upravljački programi se učitavaju u jezgru Linuxa kao moduli. Koje sve funkcije mora modul imati ako se radi o ulaznoj napravi koja daje podatke procesima?

Odgovor:

1. početnu funkciju za inicijalizaciju modula
2. funkcije *open, read*

5. Da bi se uklopio u jezgru operacijskog sustava (Linux), tj. njegove podsustave, upravljački program treba ostvariti samo dio operacija koje već nisu ostvarene kroz jezgru. Navesti četiri najbitnije takve operacije za znakovnu napravi.

Odgovor:

- open, close, read, write*

6. U ostvarenju kružnog međuspremnik "kfifo" veličina međuspremnik mora biti potencija broja 2 ($N = 2^X$). Zašto? Što omogućava to ograničenje?

Odgovor:

izbjegavanje operacije MOD pri radu s međuspremnikom

umjesto: $ULAZ = (ULAZ + 1) \bmod N$

ide: $ULAZ = (ULAZ + 1) \& (2^n - 1)$

7. Može li se u operaciji `write` koju ostvaruje naprava koristiti `mutex_lock`? Obrazložiti zašto da/ne.

Odgovor:

može jer se izvodi u jezgrinom kontekstu procesa – takva dretva može se blokirati

8. Koje su prednosti korištenja makroa za ostvarenje operacija zapisa u dnevnik (LOG vs `printf`)?

Odgovor:

*jednostavniji dohvat (i ispisi) pozicije s koje se funkcija zove
jednostavnije uklanjanje ispisa nakon što je ispitivanje koda gotovo*

9. Neka struktura nešto ima element `element`. Ako u varijabli `x` imamo adresu dijela `element` nekog objekta, kako doći do adrese objekta koji sadrži taj element? Pokazati kodom.

Odgovor:

`objekt = container_of(x, struct nešto, element)`

10. Koje su prednosti korištenja liste preko postojećeg sučelja, npr. preko sučelja `linux/list.h`?

Odgovor:

*Linux svugdje koristi te liste – svi ih znaju
puno makroa koji olakšavaju korištenje listi*

11. Što se nalazi u varijabli `jiffies`? Za što se ona koristi?

Odgovor:

za praćenje sata u "otkucajima" posebnog brojila koje se za to koristi

12. Opisati način korištenja mehanizma reda poslova (workqueues).

Odgovor:

*za svaki red postoji dretva koja obrađuje poslove koji se tamo pošalju
takva dretva (jezgrina) može se i blokirati, stoga je takav mehanizam namjenjen obradi duljih poslova*

13. Opisati sinkronizaciju preko mehanizma događaja (events).

Odgovor:

*netko može čekati na neke događaje a drugi ih objavljuju
obično se uz čekanje vežu i uvjeti koji trebaju biti zadovoljeni prije nastavka rada – dretva koja čeka nakon odblokiranja ponovno provjerava uvjet, ako nije ispunjen ponovno čeka*

14. Opisati sinkronizaciju preko mehanizma završetaka (completions).

Odgovor:

netko čeka završetak a netko ga objavljuje

15. Koje se od navedenih funkcija smiju koristiti u atomarnim kontekstima: `mutex_lock`, `mutex_unlock`, `wait_event`, `wake_up`, `wait_for_completion`, `complete`, `spin_lock`, `spin_unlock`.

Odgovor:

wake_up, complete, spin_lock, spin_unlock

16. Opisati što rade sljedeće funkcije (u jezgri Linuxa), po čemu se međusobno razlikuju i kada ih koristiti.

```
mutex_lock(&lock);  
mutex_lock_interruptible(&lock);  
spin_lock(&key);
```

Odgovor:

mutex_lock - zaključava ključ lock ili se blokira čekajući da se on može zaključati (da ga netko otključa)

mutex_lock_interruptible - zaključava ključ lock ili se blokira čekajući da se on može zaključati (da ga netko otključa), uz mogućnost da se to čekanje prekida ako dretva dobije signal

spin_lock - zaključava ključ key ili radno čeka da ga netko otključa

Funkcije mutex_lock/mutex_lock_interruptible se mogu koristiti samo iz koda koji se izvodi u kontekstu neke dretve, ne i atomarnog koda. Funkcija spin_lock se može koristiti i u atomarnom kodu.

17. Čemu služe sučelja `request_irq` i `request_threaded_irq`? Kako se ona međusobno razlikuju?

Odgovor:

request_irq - za registraciju funkcije za obradu prekida

request_threaded_irq - isto kao i prva uz dodatnu mogućnost da ako obrada nije bila gotova u funkciji za obradu prekida, da se obrada nastavi u posebnoj dretvi (s dozvoljenim prekidanjem)

18. Tko i kad vraća vrijednosti `IRQ_HANDLED`, `IRQ_WAKE_THREAD`?

Odgovor:

funkcija za obradu prekida može vratiti obje vrijednosti, ako vrati `IRQ_WAKE_THREAD` onda se aktivira i dretva koja završava obradu (radi onaj dulji dio posla)

19. Opisati čemu služi sučelje `poll`. Što se njime može postići?

Odgovor:

provjerava se (i čeka) da se na nakom opisniku datoteke (i/ili naprave, mrežne priključnice i sl.) nešto dogodi - pojavi podatak ili može se poslati novi podatak (ovisno o postavljenim zastavicama danim uz poll)

20. Opisati čemu služi sučelje `ioctl`. Što se njime može postići?

Odgovor:

šalje se naredba upravljačkom programu (naprave)

Dio II

Višeprocessorski sustavi

Sažetak

Drugi dio skripte opisuje svojstva višeprocessorskih sustava s perspektive operacijskog sustava i njegove interne građe te njegova korištenja od strane programa. Najprije je opisan razlog korištenja višeprocessorskih sustava te tipične arhitekture takvih sustava (tipovi procesora i načina njihova spajanja u višeprocessorskim računalima). Slijedi prikaz nekih problema i mogućih rješenja pri implementaciji operacijskog sustava za višeprocessorska računala. Jedna od najvećih promjena u odnosu na jednoprocessorsko računalo je raspoređivanje dretvi te je ono prikazano u idućem poglavlju. Zadnje poglavlje prikazuje raspoređivanje dretvi u Linuxu, ukratko opisuje strukture podataka koje se pri tome koriste i koje mogućnosti pružaju.

7. Svojstva višeprocessorskih sustava

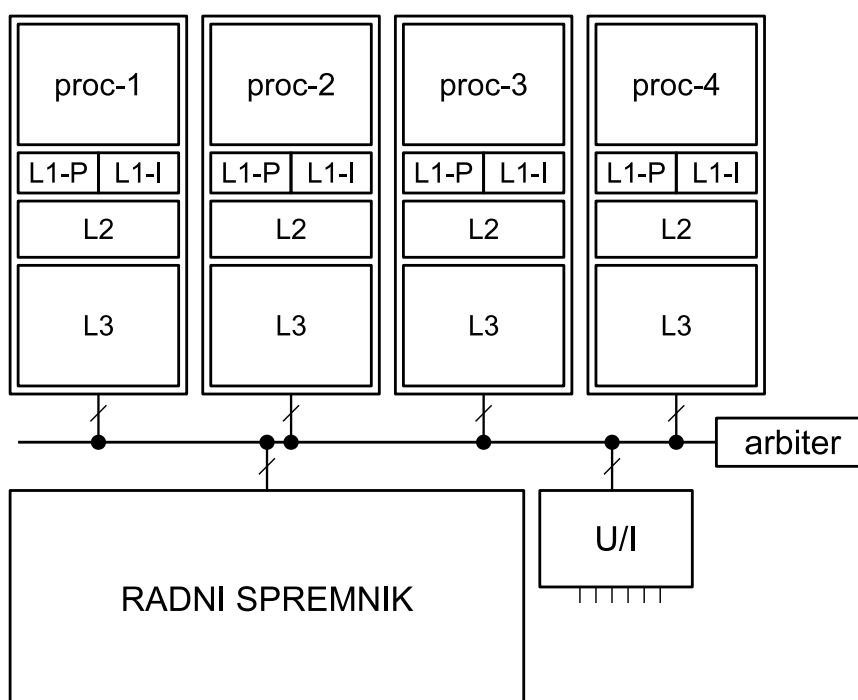
7.1. Zašto su višeprocessorski sustavi danas svugdje?

- u prošlosti (prije 2000. godine) višeprocessorski sustavi su uglavnom bili samo za poslužitelje jer su jednoprocessorski bili dovoljni za normalna (osobna) računala
 - razvoj poluvodičke industrije omogućavao je da se kroz otprilike svake dvije godine dvostruko poveća frekvencija procesora (i ostalih povezanih komponenti)
- od 2000. gotovo da i nema povećanja frekvencije procesora – sadašnja tehnologija ne omogućava efikasno korištenje veće frekvencije (a da se ne troši jako puno energije na hlađenje)
- povećanje snage (uglavnom) ide kroz veći broj “jezgri” na istom procesoru
- svaka jezgra može izvoditi dretvu – efektivno je ona zaseban procesor
- (neke) jezgre dijele dijelove priručnog spremnika procesora
- u nastavku se i za jezgru koristi termin procesor, tj. sustav s jednim višejezgrenim procesorom se proglašava višeprocessorskim sustavom, jer on to i jest sa stanovišta OS-a
- da bi iskoristili ovakvu processorsku snagu treba paralelizirati posao, podijeliti ga na dijelove koji bi se mogli paralelno izvoditi, ali je onda potrebna i sinkronizacija

7.2. Osnovne vrste višeprocessorskih sustava

Višeprocessorske sustave možemo prema građi podijeliti u nekoliko skupina:

1. simetrični višeprocessorski sustavi (slika 7.1.)
 - svaki procesor je zaseban čip (s jednom jezgrom)

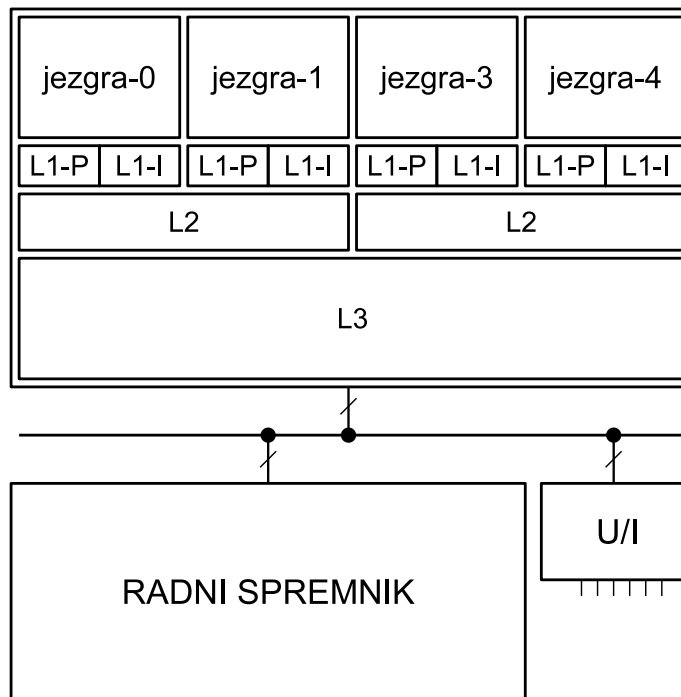


Slika 7.1. Primjer simetričnog višeprocessorskog sustava

- svaki procesor ima svoj priručni spremnik (često u razinama L1-L2-L3)
- zajednička memorija je odvojena i pristupa joj se preko dijeljene sabirnice
- “stari” višeprocorski sustavi spadaju u ovu kategoriju

2. višejezgreni procesori (slika 7.2.)

- jedan čip s više procesorskih jedinki – jezgri
- priručni spremnik podijeljen je u razine (L1-L2-L3); neke se dijele među jezgrama
- zajednička memorija je odvojena
- većina današnjih sustava spada u ovu kategoriju



Slika 7.2. Primjer višejezgrenog procesora

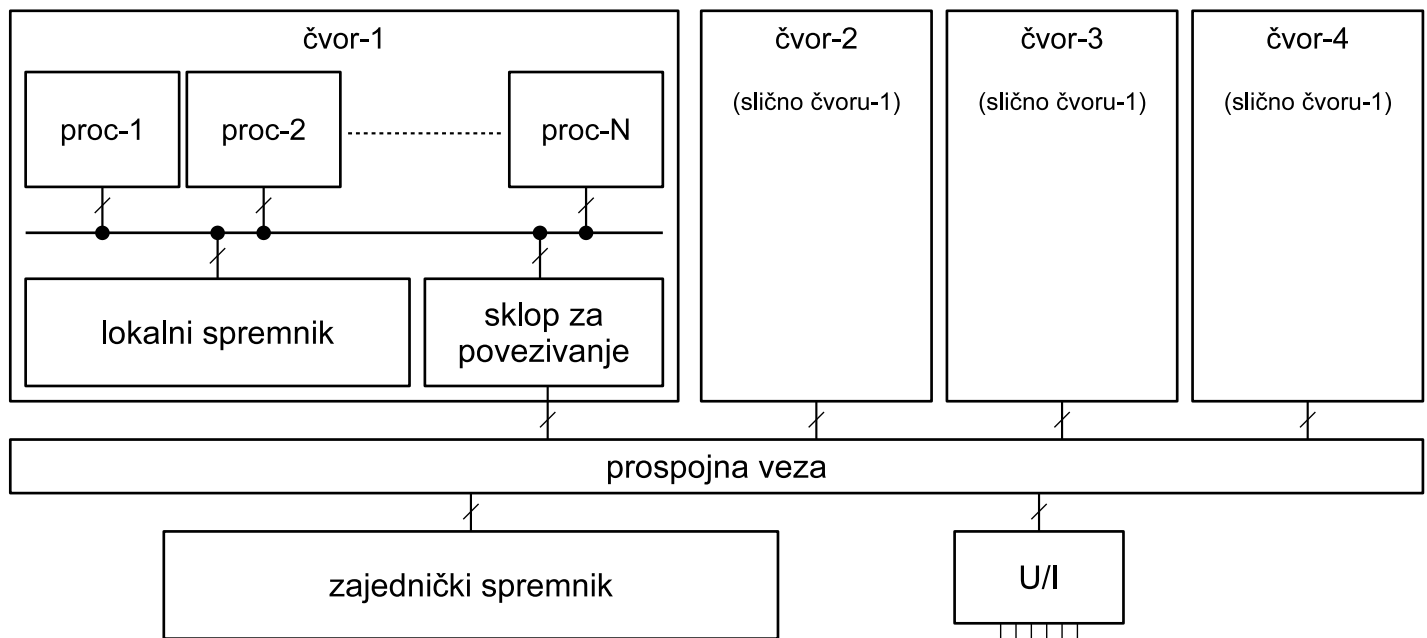
3. simetrični višeprocorski sustavi s višejezgrenim procesorima

- simetrični višeprocorski sustavi kod kojih je svaki procesor višejezgreni
- uglavnom se koriste u poslužiteljima

4. raspodijeljeni višeprocorski sustavi (slika 7.3.)

- više procesorskih kartica, na svakoj više procesora (vjerojatno višejezgrenih) i lokalna memorija
- odvojena zajednička memorija
- NUMA sustavi (engl. *Non-uniform memory access*)
- koristi se kod (jačih) poslužitelja

Iako se često za računanje mogu koristiti i posebni sklopovi (npr. grafičke kartice), njihovo korištenje se neće razmatrati jer je ono samo za specifičnu uporabu, skoro kao i naprave. Ovakvi sustavi se u literaturi nazivaju heterogenima (engl. *heterogeneous multiprocessor system*).



Slika 7.3. Primjer NUMA sustava

7.2.1. Simetrični/asimetrični, homogeni/nehomogeni višeprosesorski sustavi

- Simetričnost obilježava pristup zajedničkom spremniku, a homogenost jednakost procesora

1. Simetrični i asimetrični višeprosesorski sustavi

a) Simetrični sustavi: jednaka svojstva bez obzira na procesor koji traži operaciju

- uobičajena arhitektura sa zajedničkom memorijom kojoj se pristupa preko zajedničke sabirnice
- izvorno: symmetric multiprocessing / shared-memory multiprocessing – SMP

b) Asimetrični sustavi: pristup je brži/sporiji ovisno o procesoru koji pristupa pojedinom dijelu memorije

- memoriji bliže procesoru se brže pristupa nego udaljenoj memoriji (npr. na drugoj procesorskoj kartici)
- tipično za NUMA sustave s raspodijeljenom memorijom

2. Homogeni i nehomogeni višeprosesorski sustavi

a) Većina procesora podržava dinamičko određivanje frekvencije rada (po jezgri)

- Iako identične, jezgre jednog procesora mogu u pojedinim trenucima raditi na različitim frekvencijama
- Time se formalno narušava homogenost, ali ipak se takve procesore ne svrstava u nehomogene

b) Moderniji višejezgreni procesori, pogotovo oni namijenjeni za baterijom pogonjena računala imaju različite jezgre (pored dinamičkog upravljanja frekvencijom)

- brze jezgre (snažnije, P – performance)
- spore jezgre (štedljivije, E – efficiency)
- OS koristi jedne i/ili druge u različitim situacijama: kad ne treba performanse samo spore, kad treba performanse onda brze (i možda spore)

- ovdje se već govori o jednom obliku nehomogenosti

7.3. Dodatne metode povećanja učinkovitosti

Općenito gledano, povećanje učinkovitosti procesora postiže se na nekoliko načina. Neki od njih su navedeni u nastavku.

1. Optimiranje protočne strukture

- manje ili više elemenata u protočnoj strukturi
- bolje predviđanje grananja i oportuno pripremanje i izvođenje instrukcija
- paralelno izvođenje susjednih nepovezanih instrukcija iste dretve

2. Optimiranje korištenja priručnog spremnika

- povećanje njegova kapaciteta i brzine
- prikladno dijeljenje dijelova priručnog spremnika među jezgrama procesora
- algoritmi upravljanja podacima u priručnom spremniku (što van/unutra)
- ažuriranje sadržaja u priručnim spremnicima jezgri/procesora uslijed promjene na drugoj jezgri/procesoru (engl. *cache coherence problem*)
 - primjeri upravljanja kopijama podataka u priručnim spremnicima:
 - a) poništavanje (engl. *invalidation*) – promjena varijable kod jednog procesora izaziva poništavanje kopija u drugim procesorima
 - b) prislušivanje (engl. *snooping*) – svi procesori prislušuju što se mijenja i ažuriraju svoje kopije pri promjenama

3. Sklopovska višedretvenost (engl. *simultaneous multithreading, hyper-threading*)

- izvođenjem instrukcija jedne dretve često se događaju situacije kad procesor stane jer mora pričekati nešto, npr.:
 - dohvat podataka iz memorije (procesor je značajno brži od memorije)
 - dovršetak složene operacije (kroz koprocesor)
- mnogi dijelovi procesora tada ne rade
- ideja je imati u pripremi i druge instrukcije neke druge dretve koja tada može “uletjeti” sa svojim instrukcijama
- procesor se sam prebacuje na drugu dretvu – izvodi instrukcije te dretve
- prema van se takav procesor prikazuje kao dva procesora (dvije jezgre)
- dobitak na performansama je daleko manji nego da imamo dvije jezgre, ali je ipak osjetan jer je bolja iskorištenost sklopovlja
- npr. Intelovi procesori kroz hyper-threading mogu povećati učinkovitost do 30%
 - da bi to postigli takvi procesori najčešće imaju i dodatne jedinice za računanje (za cjelobrojne i realne operacije)
- ovi mehanizmi se mogu i isključiti kroz postavke (npr. preko BIOSa)
 - ovakav način rada unosi nedeterminizam – dretve ne napreduju očekivanim tempom s obzirom na logički broj procesora

- u nekim situacijama može se dogoditi i degradacija performansi (npr. zbog prevelikih zahtjeva za priručnim spremnikom prevelikog broja paralelno izvođenih dretvi)

7.4. Okruženja primjene višeprocorskih sustava

Osnovna okruženja:

1. poslužitelji
2. osobna računala (stolno računalo, prijenosnik)
3. pametni telefoni, tableti
4. ugrađena računala

Različita svojstva tih sustava uzrokovala su i stvaranje različitih procesora, s različitim ciljevima:

1. optimiranje obrade paralelnih nezavisnih poslova (npr. poslužitelji)
2. optimiranje rada jedne aplikacije (višedretvene; poslužitelji ali i ostala računala)
3. smanjena potrošnja radi dužeg rada na bateriji
4. determinističko ponašanje za upravljačka djelovanja

Procesori izrađeni za poslužitelje (npr. procesori iz serije Intel Xeon) imaju veći broj jezgri i osjetno veći priručni spremnik što doprinosi osjetno većom učinkovitosti kod poslužiteljskih poslova (ali su zato i osjetno skuplji).

Iskorištenje mogućnosti procesora ostvaruje se kroz podršku operacijskog sustava. On postavlja potrebne postavke procesorima te raspoređuje dretve na njih. Osnovna promjena operacijskog sustava koji je napravljen za jednoprocorsko računalo u odnosu na onaj koji je napravljen za višeprocorsko je u načinu upravljanja dretvama – potrebno je odabrati dretvu za svaki procesor zasebno. Stoga su u ovim materijalima prikazani postupci raspoređivanja i kako bi se oni trebali ponašati na višeprocorskim računalima. Prije toga su prikazani problemi ostvarenja operacijskog sustava za ovakva računala, obzirom da se jezgrine funkcije mogu izvoditi paralelno različitim procesorima.

Pitanja za vježbu 7

1. Zašto su višeprocessorski sustavi danas svugdje?

Odgovor:

tehnologija ne omogućava povećanje frekvencije ali omogućava povećanje gustoće tranzistora na procesorskom čipu – zato jedan čip ima više procesorskih jezgri koje mogu donijeti povećanje performansi (ali samo ako se uposle s višedretvenim programima)

2. Navesti osnovne vrste višeprocessorskih sustava i gdje se one koriste.

Odgovor:

*simetrični jednojezgreni višeprocessorski sustavi – zastarjeli poslužitelji
višejezgreni procesori – osobna računala, telefoni, ugrađeni sustavi
simetrični višejezgreni višeprocessorski sustavi – poslužitelji
raspodijeljeni višeprocessorski sustavi – jači poslužitelji*

3. Navesti prednosti i nedostatke simetričnih i asimetričnih višeprocessorskih sustava.

Odgovor:

*simetrični-prednost: dohvat podataka jednako traje jer je samo jedna zajednička memorija
simetrični-nedostatak: sabirnica preko koje se komunicira sa zajedničkom memorijom može postati usko grlo sustava
asimetrični-prednost: ako su podaci dobro raspodijeljeni, bliže su procesorima koji ih koriste
asimetrični-nedostatak: dohvat podataka iz različitih memorija različito traje jer je zajednička memorija raspodijeljena*

4. Koji principi i mehanizmi se koriste u današnjim višejezgrenim procesorima koji im daju "nehomogenost" (različitost)?

Odgovor:

*dinamičko upravljanje frekvencijom
različiti procesori (performance, efficiency)
sklopovska višedretvenost*

5. Što je to sklopovska višedretvenost (npr. hyperthreading)?

Odgovor:

procesor, tj. neke jezgre se predstavljaju kao dvije iako ne mogu cijelo vrijeme paralelno izvoditi dvije dretve; umjesto da procesor čeka na završetak izvođenja neke složene instrukcije jedne dretve, procesor se prebacuje na drugu dretvu s kojom nastavlja izvođenje

8. Problemi pri ostvarenju operacijskog sustava za više-procesorske sustave

Ostvarenje operacijskog sustava na višeprocorskom računalu ima nekoliko problema koje treba riješiti:

- definirati prikladnu strukturu podataka i očuvati njenu konzistentnost obzirom na moguće paralelno korištenje
- maksimalno iskoristiti potencijal svih elemenata sustava, pogotovo mogućnosti paralelnog rada i korištenja priručnih spremnika procesora
- omogućiti pravednu podjelu procesorskog vremena (kod raspoređivanja dretvi)
- omogućiti rezervaciju sredstava (npr. procesorskog vremena)
- ostvariti determinističko ponašanje za sustave kod koji je to vrlo bitno

Glavni dijelovi sustava kojima OS upravlja su procesi/dretve te ulazno izlazne naprave.

- Da li dozvoliti da se dretve mogu izvoditi na bilo kojem procesoru?
 - uglavnom nema razloga da to nije tako
 - iznimke su kada bi se neki procesor rezervirao za druge stvari, npr. upravljanje UI napravama
 - OS programima nudi mogućnost da definiraju procesore na kojima se žele izvoditi (tzv. afinitet)
- Da li upravljanje napravama (obrade prekida istih) izvoditi na nekom određenom procesoru ili na bilo kojem?
 - nije neophodno da se odredi procesor za to, ali možda pojednostavljuje sustav; npr. neki upravljač prekidima se programira da sve zahtjeve šalje jednom procesoru
 - ako se na nekom procesoru ne izvode druge dretve, pa ni jezgrine funkcije koje one pozivaju, taj procesor može puno prije reagirati na vanjske zahtjeve naprava (prekide)

U nastavku se najprije razmatraju problemi ostvarenja konzistentnosti u korištenju strukture podataka jezgre pri paralelnom izvođenju jezgrinih funkcija.

8.1. Očuvanje konzistentnosti podataka kroz paralelni rad

- zajedničku strukturu podataka jezgre treba zaštititi od istovremenog korištenja, ili barem od istovremene promjene koja narušava konzistentnost podataka
- konzistentnost se može ostvariti kroz:
 1. zaključavanja
 2. atomarne operacije
 3. oprezno korištenje

8.1.1. Zaključavanje podataka

- u jezgru se ulazi mehanizmom prekida, ali to nije dostatno za ostvarenje međusobnog isključivanja na višeprocorskim sustavima
- različite jezgrine funkcije možda koriste različite podatke – nije potrebno sve jezgrine funkcije ostvariti kao jedan kritični odsječak, da se izvode jedna po jedna (kao što je to pretpostavljeno u jednostavnom modelu jezgre prikazanom u [Budin, 2010])
- neke jezgrine funkcije mogu se izvoditi paralelno
- zato se uglavnom ne koristi jedno zaključavanje (“big kernel lock”) već puno manjih
- gotovo svaka netrivialna struktura podataka ima svoj ključ
- zaključavanje može biti:
 1. blokirajuće – dretva se zaustavlja i miče u stranu, ako se jezgrina funkcija izvodi u kontekstu neke (jezgrine) dretve ili
 2. radno čekanje (spin-lock) – dretva u petlji provjerava neki uvjet dok se ne ispuni (i troši procesorsko vrijeme), tj. dok ga neka druga dretva na nekom drugom procesoru ispuni (koja paralelno radi)
- što će se odabrati ovisi i o strukturi podataka koja se time štiti i gdje se ona sve koristi (ako ju se koristi i iz atomarnog okruženja koje se ne može blokirati)
- radno čekanje je efikasno samo ako se zaključavaju kratki dijelovi koda – tada će radno čekanje biti kratko
- ipak, treba uzeti u obzir da i blokiranje ima svoj trošak u kućanskim poslovima (spremanje konteksta, promjena dretve i prateće operacije s priručnim spremnikom!), pa je za postizanje veće učinkovitosti potrebno dobro analizirati što zaključati na jedan ili drugi način
- radno čekanje u kojem se cijelo vrijeme dohvaća i provjerava jedna vrijednost može imati loše djelovanje na priručni spremnik – sustav (sklopovski) koji to mora dohvaćati s odgovarajućeg mjesta (radni spremnik, priručni spremnik nekog drugog procesora)
 - postoje (teoretski) algoritmi koji to pokušavaju izbjeći tako da se u slučaju već zaključanog ključa koristi pomoćni ključ, zasebni za dretvu, kojoj se onda to mora signalizirati kad se početni ključ odključa
- za kratke operacije možda se zaključavanje može izbjeći korištenjem atomarnih operacija

8.1.2. Atomarne i slične operacije

- atomarne operacije koriste se posebnim mogućnostima procesora koje prevoditelj mora uključiti u odgovarajućim situacijama (kad mu se to u kodu navede)
- primjeri operacija: `atomic_load`, `atomic_store`, `atomic_exchange`, `atomic_compare_exchange`, `atomic_add_fetch`, `atomic_fetch_add`, `atomic_test_and_set`
- osim obavljanja navedenih operacija kao atomarnih, ovakvi pozivi mogu utjecati i na susjedne operacije i njihov redoslijed obavljanja
- problem: zbog paralelnog (spekulativnog/oportunog) izvođenja susjednih instrukcija s predviđanjem vrijednosti svakakve “anomalije” se mogu dogoditi, pogotovo kad se u instrukcijama koriste kazaljke (zato su potrebne te dodatne radnje uz atomarne instrukcije)

- prevoditelj i procesor optimiraju izvođenje jedne dretve
 - zato oni imaju slobodu reorganizirati izvođenje susjednih instrukcija tako da one što prije završe
 - uzimaju u obzir ovisnosti među tim instrukcijama
 - međutim, oni nisu svjesni drugih dretvi i da ovakva reorganizacija može loše utjecati na njih
 - prevoditelju se može ukazati na to posebnim načinima tako da on pažljivo slaže instrukcije, uz dodatak novih, a da bi se očuvao redoslijed naveden u programu
- postoji nekoliko mogućnosti (razina) “atomarnosti” koje se mogu koristiti (navesti oznaku uz makroe atomarnih operacije) – u nastavku su razmotreni problemi i kako ih neke mogućnosti rješavaju ili ne
- izvori:
 - [GCC: Built-in Functions for Memory Model Aware Atomic Operations](#)
 - [Memory model synchronization modes](#)

8.1.2.1. Primjer problema u paralelnom radu

Primjer 1. Paralelno izvođenje dretvi bez zaštite

```
x = y = 0
```

```
Dretva 1 {
  x = 1
  y = 2
}
```

```
Dretva 2 {
  ispiši(y)
  ispiši(x)
}
```

- ako dretve rade paralelno sve kombinacije ispisa su moguće!
 - 0 0 – kada dretva 2 napravi sve prije dretve 1
 - 2 1 – kada dretva 1 napravi sve prije dretve 2
 - 0 1 – kada dretva 1 zapiše 1 u x, a dretva 2 ispiše y prije njegove promjene u dretvi 1
 - 2 0 – kada procesor u izvođenju dretve 1 promijeni redoslijed obavljanja operacija, najprije u y zapiše 2 a potom u x 1; dretva 2 prvi ispis obavi nakon y=2 a drugi prije x=1
 - da bi dretva 2 vidjela promjene koje radi dretva 1, te promjene moraju biti upisane u zajedničku memoriju (x i y mogu privremeno biti u registrima pri izvođenju dretve 1)
 - pretpostavka je da ispis ide slijedno, najprije ispiši(y), pa ispiši(x) – to procesor ne okreće!

8.1.3. Konzistencija nad slijedom operacija (*total ordering*)

- kod ove razine traži se da se poštuje redoslijed svih označenih operacija (spremi/dohvati) te da operacije koje su navedene prije budu gotove iako nisu označene, a operacije koje su kasnije da se kasnije i obavave
- na taj način su promjene odmah vidljive svima
- ovo je “najstroža” inačica održavanja konzistencije i najzahtjevnija je za ostvarenje – potrebne su dodatne instrukcije za njeno ostvarenje; traži najviše vremena pri radu

Primjer 2. Korištenje atomarnih operacija

```
//početne vrijednosti, prije pokretanja dretvi
a = b = x = y = 0

Dretva 1 {
    a = 5
    b = 7
    x.atomarno_spremi(1)
    y.atomarno_spremi(2)
}

Dretva 2 {
    ispiši(y.atomarno_dohvati())
    ispiši(x.atomarno_dohvati())
    ispiši(b)
    ispiši(a)
}
```

- pretpostavljeno ponašanje funkcija `atomarno_*`, kad drukčije nije navedeno, jest očuvanje konzistentnosti nad slijedom operacija
- sama operacija `atomarno_spremi`, osim što će to napraviti atomarno, osigurava da su sve prethodne operacije obavljene, bez obzira jesu li označene kao atomarne ili ne
- u navedenom primjeru, ako je za `y` ispisano 2 za `x`, `b` i `a` će sigurno biti ispisane vrijednosti koje je dretva 1 upisala, tj. ispis će biti {2 1 7 5}
- isto vrijedi i za `x`: ako je ispisano 1 onda će se za `b` i `a` ispisati 7 i 5
- `a` i `b` nisu zaštićene pa redoslijed upisivanja vrijednosti u njih od strane dretve 1 nije definiran (može biti jednak navedenom, ali i obratni)
- sve mogućnosti su: {0 0 0 0}, {0 0 0 5}, {0 0 7 0}, {0 0 7 5}, {0 1 7 5}, {2 1 7 5},

8.1.4. Konzistencija nad jednom operacijom (relaksirana)

- "Stroga" atomarnost iz prethodnog primjera može tražiti dosta skupih operacija sinkronizacije (priručnih spremnika u višeprocorskim sustavima), a ponekad to nije neophodno.
- Stoga postoje i blaže varijante "atomarnih" operacija i njihova utjecaja na susjedne operacije
- Jedna od njih je "relaxed" koja održava konzistentnost samo nad navedenim podacima
- Uz dodatak argumenta `RELAXED` (`__ATOMIC_RELAXED`) osigurava se da ako je "spremi" operacija napravljena prije "dohvati", da je dohvaćena ta spremljena vrijednost
- To se inače ne mora dogoditi kad bi se koristile obične operacije (`x=1`) jer bi procesor to mogao držati u registru neko vrijeme (ili u priručnom spremniku, ako nema sklopovske podrške održavanja njihove konzistencije u višeprocorskom sustavu).

Primjer 3. Relaksirana atomarnost

```
a = x = y = 0

Dretva 1 {
    a = 5
    x.atomarno_spremi(1, RELAXED)
    y.atomarno_spremi(2, RELAXED)
}

Dretva 2 {
    ispiši(y.atomarno_dohvati(RELAXED))
    ispiši(x.atomarno_dohvati(RELAXED))
    ispiši(a)
}
```

- u navedenom primjeru redoslijed operacija dretve 1 može biti proizvoljan (`x` prije `a`, `y` prije `x`), tj. moguće su sve kombinacije ispisa

Primjer 3. Relaksirana atomarnost (2)

```
x = 0

Dretva 1 {
    a = 1
    x.atomarno_spremi(a, RELAXED)
    b = 2
    x.atomarno_spremi(b, RELAXED)
}

Dretva 2 {
    ispiši(x.atomarno_dohvati(RELAXED))
    ispiši(x.atomarno_dohvati(RELAXED))
}
```

- u ovom primjeru poštivati će se redoslijed spremanja i dohvata jer se radi o istoj varijabli (najprije će se pohraniti 1 a potom 2)
- ispisi mogu biti: {0 0}, {1 1}, {1 2} i {2 2}, ali nikada {1 0}, {2 0} ili {2 1}

8.1.5. Konzistentija nad povezanim varijablama (*acquire/release*)

- pod “povezane varijable” misli se na korištenje globalnih varijabli koje nisu u dohvati/pohrani već neposredno prije/poslije
- ovakve atomarne operacije osiguravaju i da varijable koje se koriste/mijenjaju prije/poslije budu pohranjene/pročitane
- pri spremanju koristiti “release” – osigurava da se ova operacija obavi nakon svih prethodnih operacija pohrane u memoriju (da procesor ne bi izmijenio redoslijed i neku instrukciju za pohranu koja je u kodu prije napravio nakon ove operacije)
- pri dohvatu koristiti “acquire” – osigurava da se ova operacija obavi prije svih slijedećih operacija čitanja
- na neki način stvara se zavisnost (uređenost) operacija među dretvama koje koriste ovakve operacije
- ova atomarnost je slična prvoj, nad slijedom instrukcija (najstrožoj), jedino što se ne garantira “globalna” sinkronizacija sadržaja među svim dretvama, ako i one ne koriste ovakve operacije (kao što je to kod prve, najstrože atomarnosti)

Primjer 3. Atomarnost acquire/release

```
a = b = x = y = 0

Dretva 1 {
    a = 5
    x.atomarno_spremi(1, RELEASE)
    y.atomarno_spremi(2, RELEASE)
    b = 7
}

Dretva 2 {
    ispiši(y.atomarno_dohvati(ACQUIRE))
    ispiši(x.atomarno_dohvati(ACQUIRE))
    ispiši(b)
    ispiši(a)
}
```

- pohrani varijable x mora prethoditi pohrana varijable a ; pohrani varijabli y mora prethoditi pohrana varijabli x i a
- čitanju varijabli x mora prethoditi čitanje varijable y ; čitanju varijabli a mora prethoditi čitanje varijabli x i y
- “ b ” nije uređen, može prije/poslije ostatka koda (kod Dretve 1)

8.1.6. Konzistencija nad nepovezanim varijablama (consume/release)

- Slično kao i *acquire*, ali se gledaju samo operacije koje su povezane s ovom – koriste iste podatke/memoriju
- za razliku od relaksirane, ovdje se poštuje redoslijed operacija dohvati/pohrani

Primjer 3. Atomarnost *consume*

```
a = b = x = y = 0
z = NULL
```

```
Dretva 1 {
    a = 5
    b = 7
    x.atomarno_spremi(1, RELEASE)
    y.atomarno_spremi(2, RELEASE)
    z.atomarno_spremi(&b, RELEASE)
}

Dretva 2 {
    ispiši(y.atomarno_dohvati(CONSUME))
    ispiši(x.atomarno_dohvati(CONSUME))
    w = z.atomarno_dohvati(CONSUME)
    ispiši(*w)
    ispiši(a)
}
```

- redoslijed spremlj/dohvati je očuvan ($x \rightarrow y \rightarrow z$)
- korištenje varijable *w* (*ispiši(*w)*) je sinkronizirano s dohvatom te varijable, tj. obavlja se nakon dohvata
- korištenje varijable *a* u drugoj dretvi nije sinkronizirano, može se dohvatiti i stara vrijednost varijable (u dretvi 2 ne radi se zaštita redoslijeda varijable *a* u odnosu na *x*, *y*, i *z*)

8.1.7. Atomarnost pohrane i čitanja

- problem paralelnog korištenja podataka (u jezgri, ali i običnim višedretvenim programima) je vrlo velik i težak za ispravno ostvarenje
- prethodno navedene atomarne operacije pomažu pri tome, ali unose dosta popratnih radnji (nisu baš toliko jeftine) – stoga se i ne koriste baš svugdje
- kada se ne koriste, prevoditelj ima “punu slobodu” nad optimiranjem koda i svašta se još može dogoditi što u paralelnom radu može prouzrokovati neočekivane probleme
- npr. ponekad spremanje i samo jedne vrijednosti (ili čitanje) može biti odgođeno ili čak i razlomljeno na više sabirničkih ciklusa
- da bi se spriječilo neke neželjene posljedice optimiranja i pristupa memoriji, ali i ukazalo na korištenje zajedničkih varijabli u kodu (izvan zaključanog dijela koda) preporuka je koristiti posebne makroe (npr. u kodu Linuxa `READ_ONCE`, `WRITE_ONCE`), koji se na nekim arhitekturama mogu i prevesti u korištenje nekih korisnih mogućnosti
- najčešće se u implementaciji (u C-u) koristi ključna riječ `volatile` koja forsira (gotovo) svako korištenje varijable njenim dohvatom iz memorije (ili pohranom) i smanjuje mogućnosti prevoditelja u promjeni redoslijeda izvođenja operacija
- izvori:
 - [Why kernel code should use READ_ONCE and WRITE_ONCE for shared memory accesses](#)
 - [GCC: When is a Volatile Object Accessed?](#)
 - [Linux kod: READ_ONCE i WRITE_ONCE](#)

- [Linux kernel memory barriers](#)

8.1.8. Ostale mogućnosti

- Jedna od ostalih mogućnosti optimiranja korištenja zajedničkih podataka jest mehanizam Read-copy update (RCU) koja je korisna ako se podaci većinom čitaju a rjeđe mijenjaju
 - više na: [What is RCU? – "Read, Copy, Update"](#)

8.2. Strukture podataka

8.2.1. Raspoređivanje dretvi

- očita promjena prema jednoprocorskim operacijskim sustavima: više aktivnih dretvi
- što s redom pripravnih dretvi?
 - uobičajeno je da se za svaki procesor izradi zasebni red pripravnih
 - osnovni razlog za to je korištenje priručnog spremnika procesora (hot cache)
 - ideja je da dretve koje se izvode na tom procesoru, iako se povremeno ne izvode (zbog drugih dretvi na tom procesoru) ipak će pri povratku izvođenja u tom priručnom spremniku naći neke svoje podatke koje neće trebati ponovno dohvaćati iz radnog spremnika – tako se povećavaju performanse
- OS treba povremeno provjeriti treba li prebaciti koje dretve iz jednog reda u drugi (radi balansiranja opterećenja ili sličnih problema)
- organizacija tih redova pripravnih ovisi o načinu raspoređivanja koji se koristi
- uzeti u obzir i hijerarhiju procesora/jezgri – koje jezgre dijele koje razine priručnog spremnika (možda neko stablo koje reflektira procesore, višejezgrene, hyper-treading, dijeljeni priručni spremnici)
 - raspoređivač koji to uzima u obzir može osjetno povećati performanse sustava (ili pojedinih dretvi/aplikacija)

8.2.2. Ostale strukture jezgre

- ostale strukture su zajedničke za cijeli sustav (sve procesore)
- najčešće svaka ima svoj ključ za zaštitu
- strukture su definirane tamo gdje su potrebne
 - ideja je da se izravno ne koriste iz drugih dijelova koda (modula), već ako je potrebno samo preko sučelja tog modula
 - na taj način se interne strukture mogu mijenjati bez promjena o ostatku koda
- izuzeci su neke vrlo često korištene “varijable” (`jiffies`, `current`), a koje zapravo i same mogu biti funkcije (preko makroa)

Pitanja za vježbu 8

1. Navesti načine za očuvanje konzistentnosti strukture podataka jezgre u višeprocorskom sustavu.

Odgovor:

*zaključavanja (blokirajuće ili radno)
atomarne operacije
drugi algoritmi*

2. Koje dvije osnovne vrste zaključavanja postoje? Koja su njihova dobra i loša svojstva?

Odgovor:

*blokirajuće ili radno
blokirajuće-prednosti: dretva koja ne može dalje ne troši procesorsko vrijeme
blokirajuće-nedostatci: dosta kućanskih poslova (zamjena konteksta dretvi)
radno-prednosti: nema zamjene dretvi, ako druga dretva brzo otključa, ova ne troši previše procesorskog vremena
radno-nedostatci: neproduktivno trošenje procesorskog vremena (ako ona druga dretva dulje drži ključ)*

3. Kada ima smisla koristiti atomarne operacije?

Odgovor:

za jednostavnije operacije $a = a + b$ i slično

4. Zašto samo korištenje atomarnih operacija (koje jednu operaciju, npr. $a = b + c$, naprave atomarno) ponekad nije dovoljno? Što još najčešće treba napraviti?

Odgovor:

*operacije koje prethode atomarnoj moraju se dovršiti
operacije koje slijede atomarnu ne smiju početi prije nego atomarna završi*

5. Koja od navedenih "razina atomarnosti" je najzahtjevnija pri radu?

- a) konzistencija nad slijedom operacija (*total ordering*)
- b) konzistencija nad jednom operacijom (*relaksirana*)
- c) konzistencija nad povezanim varijablama (*acquire/release*)
- d) konzistencija nad nepovezanim varijablama (*acquire/consume*)?

Odgovor:

konzistencija nad slijedom operacija (total ordering)

6. Zašto ponekad i samo spremanje/čitanje jedne varijable može biti problem (zašto se koriste `READ_ONCE` i `WRITE_ONCE`)?

Odgovor:

ako je varijabla struktura onda je moguće da je pri čitanju/pisanju elemenata neka druga dretva s drugog procesora uletjela i čitala/pisala po istoj strukturi ponekad to vrijedi i za obične podatke (npr. `double`)

9. Raspoređivanje dretvi u višeprocessorskim sustavima

Teoretski bi na višeprocessorskom sustavu paralelno mogli imati više operacijskih sustava, primjerice, na svakom procesoru svoji. Međutim, ako je i potrebno tako nešto, to se danas ostvaruje kroz mehanizam virtualizacije. Tim mehanizmom OS zapravo vidi samo što mu je dodijeljeno – kao da je na takvom sklopovlju. Zato se u nastavku neće razmatrati takvi sustavi već samo oni koji imaju jedan operacijski sustav koji upravlja svim procesorima.

9.1. Ukratko o raspoređivačima

- Za višeprocessorske sustave raspoređivači koriste iste postupke/ideje kao i za jednoprocessorske sustave
- Razlikujemo raspoređivanje nekritičnih (običnih) dretvi i raspoređivanje vremenski kritičnih dretvi (real-time)
- Kratki opis najčešće korištenih raspoređivača slijedi u nastavku
- [Više o raspoređivanju dretvi u okviru predmeta *Sustavi za rad u stvarnom vremenu*](#)

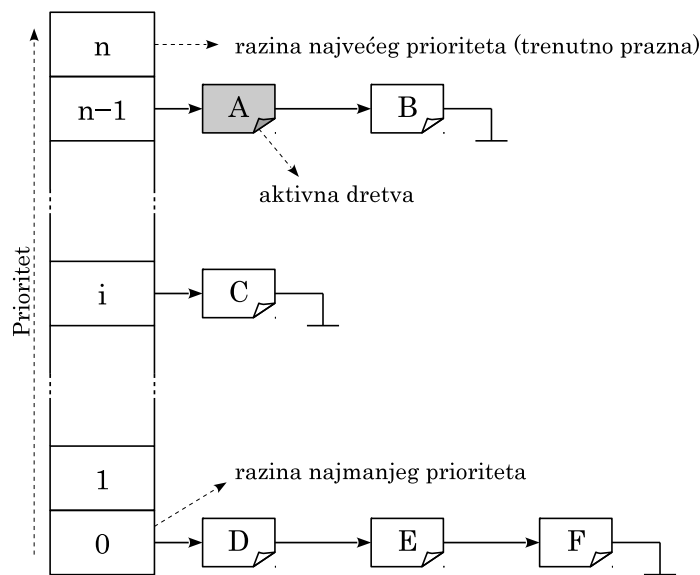
9.1.1. Raspoređivanje vremenski kritičnih poslova (dretvi koje ih obavljaju)

Postoji nekoliko uobičajenih raspoređivača za ovakve poslove.

1. SCHED_FIFO – prioritetno raspoređivanje, dretve ista prioriteta jedna za drugom
 2. SCHED_RR – prioritetno raspoređivanje, dretve ista prioriteta dijele procesorsko vrijeme (kružno posluživanje takvih dretvi)
 3. SCHED_SPORADIC – prioritetno raspoređivanje, gdje se prioritet može privremeno smanjiti (rijetko implementiran način raspoređivanja)
 4. SCHED_DEADLINE – raspoređivanje prema krajnjem trenutku završetka – dretva koja ima najbliži rok prva se odabire (Linux)
- prioritet se dodjeljuje prema važnosti poslova
 - ako su poslovi periodički, može se koristiti metoda mjere ponavljanja – poslovima koji se češće javljaju dati veći prioritet (engl. *rate monotonic priority assignment* – RMPA)
 - slika 9.1. prikazuje uobičajenu strukturu podataka za red pripravnih kada se koristi prioritetno raspoređivanje – za svaki prioritet postoji zasebna lista tako da operacije uzimanja najprioritetnije dretve kako i ubacivanje neke dretve u red bude složenosti $O(1)$

9.1.2. Raspoređivanje nekritičnih poslova

- osnovna ideja je pravedno podijeliti procesorsko vrijeme te kratkim poslovima dati prednost – to su uglavnom poslovi koji komuniciraju s napravama, poslovi koji upravljaju grafičkim sučeljem
- teorijski algoritam sa zadanim svojstvima: višerazinsko raspoređivanje s povratnom vezom (engl. *multilevel feedback queue* – MFQ)



Slika 9.1. Red pripravnih za prioritno raspoređivanje

1. više FIFO redova, poredanih po prioritetu (slično kao na slici 9.1.)
 2. kad nova dretva postane pripravna ulazi u najprioritetniji red, na kraj tog reda
 3. aktivna dretva je prva iz prvog nepraznog najprioritetnijeg reda
 4. aktivna dretva dobije kvant vremena
 - ako ta dretva završi prije isteka kvanta, onda izlazi iz sustava raspoređivača (nije više pripravna)
 - inače, kada potroši cijeli kvant, OS ju prekida, smanjuje prioritet za jedan te ju ubacuje u red pripravnih (na kraj prvog idućeg reda manjeg prioriteta)
 5. dretve u redu najmanjeg prioriteta se poslužuju kružno (po trošenju kvanta vremena stavljaju se na kraj tog reda)
- različiti OSevi na razne načine (raznim algoritmima) nastoje postići ovakva svojstva
 - na Windowskima se koristi varijanta SCHED_RR, ali uz iznimke tako da i manje prioritetniji poslovi dobivaju nešto procesorskog vremena
 - na Linuxu se za nekritične poslove (SCHED_OTHER) koristi *Potpuno pravedan raspoređivač* (engl. *Completely Fair Scheduler – CFS*) – raspoređivač prati koliko je koja dretva trebala dobiti vremena u odnosu na njenu razinu dobrote – nice level, “prioritet”, koliko je zapravo dobila, te na temelju razlike među njima odabire onu s najvećom razlikom

9.2. Posebnosti višeprocorskih sustava

- za svaki procesor zaseban red pripravnih dretvi
 - zbog performansi – boljem korištenju priručnog spremnika (“hot cache”)
 - upravljanje priručnim spremnikom se odrađuje sklopovski, ali ipak to utječe na performanse
 - vrijedi općenito za većinu raspoređivača, za kritične i nekritične zadatke
- uzimanje u obzir različitosti procesora
 - utjecaj dijeljenja (nekih razina) priručnog spremnika jezgri u višejezgrenom procesoru

- sklopovska višedretvenost (hyper-threading) povećava broj logičkih procesora s kojima OS upravlja, ali uzima li OS to u obzir, barem pri raspoređivanju kritičnih poslova?
- spore jezgre, brze jezgre
- NUMA sustavi

9.2.1. Raspoređivanje kritičnih poslova

1. problem: ostvariti da se najprioritetnije dretve uvijek izvode prve (povlačenje i guranje dretvi)
 - Aktivne dretve u svakom trenutku moraju biti najprioritetnije dretve – ne smije u nekom redu pripravnih biti dretva većeg prioriteta od neke aktivne na nekom procesoru
 - Kad neka dretva na nekom procesoru završi s radom (ili nije više pripravna), koju dretvu odabrati za iduću aktivnu na tom procesoru? Što ako prva u redu pripravnih na tom procesoru ima manji prioritet od prve u redu pripravnih nekog drugog procesora?
 - U tom slučaju treba takvu dretvu većeg prioriteta *povući* iz tog reda i nju izvoditi
 - Kad se jedna dretva ili više njih odblokira na jednom procesoru, kamo s njima? Ako imaju veći prioritet od drugih trenutno aktivnih dretvi na tom ili drugim procesorima?
 - Po potrebi dretve treba *gurnuti* odgovarajućim procesorima, tako da u svakom trenutku među svim pripravnim dretvama, aktivne dretve budu one najvećeg prioriteta.
 - Npr. neka u sustavu postoje četiri procesora s pripadnom aktivnom dretvom i redovima pripravnih (indeks dretve označava njen prioritet):
 - P1:{6;3}, P2:{8;5,2}, P3:{9;4}, P4:{12;1} (oznake: proces:{aktivna; pripravne})
 - ako sad procesor P4 završi s dretvom 12, on treba od P2 uzeti dretvu 5 jer je ona najprioritetnija pripravna dretva (koja se ne izvodi) i ima veći prioritet od dretve 1 koja je u redu pripravnih procesora P4, tj. novo stanje bi bilo:
 - P1:{6;3}, P2:{8;2}, P3:{9;4}, P4:{5;1}
 - ako bi sada P3 odblokirao dretve 7 i 10 koje bi inače spadale u red pripravnih tog procesora on bi ih trebao gurnuti procesorima P1 i P4:
 - P1:{7;6,3}, P2:{8;2}, P3:{9;4}, P4:{10;5,1}
2. problem: utjecaj manje prioritetnih dretvi (priručni spremnik, sabirnica, ...)
 - manje prioritetne dretve (uključujući i nekritične dretve) mogu svojim radom utjecati na istodobno aktivne prioritetnije dretve
 - obzirom da jezgre procesora mogu dijeliti priručni spremnik neke manje prioritetne dretve agresivnim korištenjem tih spremnika mogu smanjiti performanse prioritetnijima
 - slično je i s korištenjem zajedničkog spremnika preko sabirnice ili korištenje nekih naprava
 - OS uglavnom ne intervenira u ovakvim slučajevima!
 - aplikacija bi se trebala sama pobrinuti za takve situacije; npr. doznati koji su procesori na raspolaganju te stvoriti možda i dodatne dretve koje ne rade koristan posao, ali će u kritičnim trenucima maknuti dretve manjeg prioriteta i time spriječiti probleme koje one mogu izazivati dretvama većeg prioriteta (najkritičnije dretve)

9.2.2. Raspoređivanje nekritičnih (običnih) dretvi

- osnovna načela su pravedna podjela vremena te što veća efikasnost
- obzirom da prioritet nije primarni kriterij, nema potrebe za povlačenjem ili guranjem dretvi na različite procesore kao kod raspoređivanja kritičnih dretvi
- ipak je potrebno povremeno uravnotežiti opterećenja procesora – balansirati redove pripravnih dretvi, da sve ravnopravno dobiju procesorsko vrijeme
- raspoređivanje dretvi ili procesa?
 - kako gledati na dretve različitih procesa?
 1. zanemariti pripadnost procesu i sve dretve smatrati ravnopravnima?
 2. procesorsko vrijeme pravedno raspodijeliti procesima (tako da se gleda ukupno vrijeme svih dretvi pojedinog procesa)?
 - oba navedena pristupa imaju svoje prednosti i nedostatke: prvi je jednostavnije za ostvariti, a drugi je pravedniji
- optimiranje velikog posla
 - neki procesi mogu stvoriti više dretvi koje paralelno rade na problemu
 - međutim, takve dretve mogu imati povećanu potrebu za sinkronizacijom
 - ako se istovremeno paralelno izvode na različitim procesorima mogu biti učinkovitije nego ako ne rade istovremeno – onda bi se mogle češće blokirati
 - treba li OS to uzeti u obzir i pokušati takve dretve paralelno izvoditi na više procesora?
 - * npr. “grupno raspoređivanje” (engl. *gang scheduling*)
 - nedostaci: složenost raspoređivača koji bi morao sinkronizirati sve procesore
 - ovakav problem je uglavnom prisutan na poslužiteljima koji rade na zadacima različitih korisnika; na osobnim računalima (i sličnim sustavima) je najčešće samo jedan takav posao – sve ostale dretve sustava ukupno traže vrlo malo procesorskog vremena

Pitanja za vježbu 9

1. Koji se kriterij raspoređivanja najčešće koristi za vremenski kritične dretve (primjerice SCHED_FIFO, SCHED_RR) a koji kriterij za nekritične (obične, SCHED_OTHER)?

Odgovor:

za vremenski kritične najčešće se koristi prioritet, a za nekritične pravedna podjela procesorskog vremena

2. Zašto se za “red pripravnih dretvi” na višeprocessorskom operacijskom sustavu zapravo koristi više redova, po jedan za svaki procesor?

Odgovor:

*bolje iskorištenje priručnog spremnika
sitnije zaključavanja pri raspoređivanju*

3. Kada se koristi *guranje* a kada *povlačenje* dretvi kod raspoređivanja kritičnih dretvi?

Odgovor:

kod raspoređivanja vremenski kritičnih dretvi prema prioritetu: među svim pripravnim/aktivnim dretvama, aktivne moraju biti one najvećeg prioriteta

guranje: kad se na jednom procesoru odblokira dretva koja ima veći prioritet od neke aktivne dretve na drugom procesoru, ona se gurne tom procesoru

povlačenje: kad neki procesor završi s jednom dretvom, onda provjerava da li u redovima pripravnih drugih procesora ima dretva većeg prioriteta od prve u svom redu; ako ima onda takvu dretvu povuče k sebi

4. Kada se neke dretve prebacuju iz jednog reda pripravnih u drugi (drugog procesora) kod raspoređivanja nekritičnih dretvi?

Odgovor:

radi ujednačavanja (balansiranja) redova; sve nekritične dretve neovisno o tome gdje su bi trebale dobiti podjednako procesorskog vremena

10. Primjeri iz implementacije raspoređivanja u Linuxu

U ovom poglavlju su navedene neke specifičnosti iz implementacije jezgre Linuxa vezane uz raspoređivanje na višeprocorskim sustavima.

U ovom tekstu se ponekad koristi pojam *zadatak* umjesto *dretva*, ali je istog značenja (u Linuxu se koristi pojam *task*, ali označava dretvu – *thread*).

10.1. Raspoređivači

- Linux sadrži nekoliko klasa raspoređivača:
 1. STOP – interni, koristi se pri micanju svih zadataka s nekog procesora
 2. DEADLINE – raspoređivanje vremenski kritičnih zadataka, prema kriteriju najbližeg krajnjeg trenutka završetka (klasa SCHED_DEADLINE)
 3. REALTIME – prioritarno raspoređivanje vremenski kritičnih zadataka (klase SCHED_FIFO i SCHED_RR)
 4. CFS – raspoređivanje normalnih zadataka podjelom vremena (klase SCHED_OTHER, SCHED_BATCH, SCHED_IDLE)
 - SCHED_OTHER je predviđen za normalne zadatke
 - SCHED_BATCH je predviđen za dugotrajnije zadatke koji nisu interaktivni
 - * ideja je da dobiju procesorsko vrijeme koje im pripada, ali da se mogu prekidati/odgađati kada je to potrebno za interaktivne dretve (koje upravljaju sučeljem i ostalim napravama)
 - SCHED_IDLE je klasa za najmanje bitne zadatke (ali ipak korisne), koji će se izvoditi tek kad nema drugih zadataka – svaki zadatak iz SCHED_OTHER i SCHED_BATCH čak i s najmanjom dobrotom (+19) će istisnuti zadatke iz klase SCHED_IDLE
 5. IDLE – kad nema drugih zadataka, onda se koriste ovi jezgrini
- svaka klasa ima opisnik preko strukture `sched_class` koja sadrži implementaciju sučelja raspoređivača
- navedene klase su povezane u listu navedenim redosljedom (STOP → ... → IDLE)
- kad treba odabrati novi zadatak na nekom procesoru, onda se navedeni raspoređivači gore navedenim redosljedom redom pitaju (preko `pick_next_task`) da daju zadatak (`task_struct`)
 - npr. ako raspoređivač STOP nema zadataka, pita se raspoređivača DEADLINE; ako on nema pita se idućeg REALITME; ako on nema pita se CFS; ako on nema, onda nema korisna posla i uzima se zadatak iz raspoređivača IDLE
- u višeprocorskom sustavu za svaki procesor postoji zasebna struktura podataka za svaki raspoređivač (radi optimiranje korištenja priručnog spremnika, 8.2.1.)
- po potrebi se zadaci s jednog procesora prebacuju na drugi
 - radi balansiranja opterećenja (svi podjednako)
 - povlačenje/guranje prioritetnijih zadataka

- gašenja nekih procesora radi uštede energije
- ...
- svaki raspoređivač ima svoje načine raspoređivanje te svoju strukturu podataka
- npr. STOP i IDLE imaju samo po jedan zadatak (po procesoru)
- DEADLINE ima strukturu (crveno-crno stablo) u kojoj su pripremni zadaci posloženi prema trenutku kada moraju biti gotovi
- REALTIME ima za svaki prioritet zasebnu listu zadataka (po procesoru)
- CFS koristi crveno-crna stabla
- u nastavku će se razmatrati samo CFS jer je Linux prvenstveno namijenjen za obične zadatke i najveća pažnja je posvećena upravo njima

10.2. Raspoređivač CFS

- CFS – Completely Fair Scheduler – potpuno pravedan raspoređivač
- CFS se koristi za raspoređivanje normalnih dretvi, onih označenih sa SCHED_OTHER
- umjesto prioriteta koristi se “razina dobrote” (engl. *nice level*): od -20 do +19
 - manji broj veća dobrota (ekvivalent prioritetu)
 - negativne vrijednosti može postavljati samo povlašteni korisnik (root)
 - 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 dobroti
 - npr. dretva dobrote 0 treba dobiti više procesorskog vremena od dretve čija je dobrota 5, oko $1,15^{5-0} \approx 2$ puta više vremena
- CFS se koristi od jezgre inačice 2.6.23 (od 2007. godine)
- za sortiranje dretvi koristi razliku (1)-(2):
 1. izračunatog “virtualnog vremena” koje pripada pojedinoj dretvi s obzirom na njenu dobrotu i ostale dretve u sustavu
 2. 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)
- u nastavku se koristi pojam *red pripremljenih dretvi* koji označava strukturu podataka koja sadrži sve pripremljene dretve u sustavu
 - "red" nije ostvaren kao struktura "red", ali svejedno se koristi to ime
 - ekvivalentan engleski termin je *run queue*, kratica *rq* u imenima varijabli/strukture
- osnovna struktura podataka koja se koristi za red pripremljenih zadataka jest struktura – crveno-crno stablo `cfs_rq`
- za svaki procesor postoji zasebna takva struktura, ali njih može biti i više s obzirom na

grupiranja zadataka

- element raspoređivanja – ono što se kao čvor nalazi u stablu `cfs_rq` je `sched_entity`
- `sched_entity` može biti:
 1. jedan zadatak – koji je opisan s `task_struct`
 2. grupa zadataka – koja je opisana s `task_group`
- `sched_entity` je dio tih struktura (`task_struct`, `task_group`) koje reprezentira u odgovarajućem redu (`cfs_rq`)
- `task_struct` opisuje jedan zadatak
 - kada ne bi grupirali zadatke u grupe, onda bi se stablo sastojalo samo od zadataka
 - položaj zadatka u stablu definira kad će on doći na red
 - iz stabla se uzima prvi element (koji je skroz lijevo)

10.2.1. Grupe zadataka

- mehanizam grupe zadataka omogućava grupiranje zadataka ili drugih grupa u nekakvu hijerarhijsku strukturu koja je ponekad potrebna
 - zadaci s posebnim svojstvima, nekako povezani (npr. zadaci istog procesa)
 - radi optimiranja u nekom višejezgrenom, višeprosorskom ili NUMA sustavu
- `task_group` opisuje skup zadataka na sljedeći način:
 - `task_group` je reprezentiran s po jednim `sched_entity` za svaki procesor u sustavu (svi oni reprezentiraju ovu grupu u tim redovima)
 - svaki takav `sched_entity` ima zaseban red `cfs_rq` (element `my_q`) koji sadrži zadatke te grupe (ili podgrupe) – namijenjene izvođenju na procesoru s kojim je ovaj objekt povezan (u čijem je redu, izravno ili neizravno preko drugih objekata)
 - * iako su elementi `cfs_rq` zapravo `sched_entity`, oni svi smiju biti samo zadaci ili nove grupe, nije dozvoljeno miješanje različitih tipova čvorova u `cfs_rq`
 - * iznimka je početni `cfs_rq` procesora koji može imati i zadatke koji predstavljaju jezgrine zadatke (dretve) – da se za njihovo izvođenje smanji vrijeme pretraživanja
- izvadak iz navedenih struktura te primjer stanja sustava prikazani su u nastavku

Isječak kôda 10.1. Izvadak iz struktura za raspoređivanje s kratkim opisom

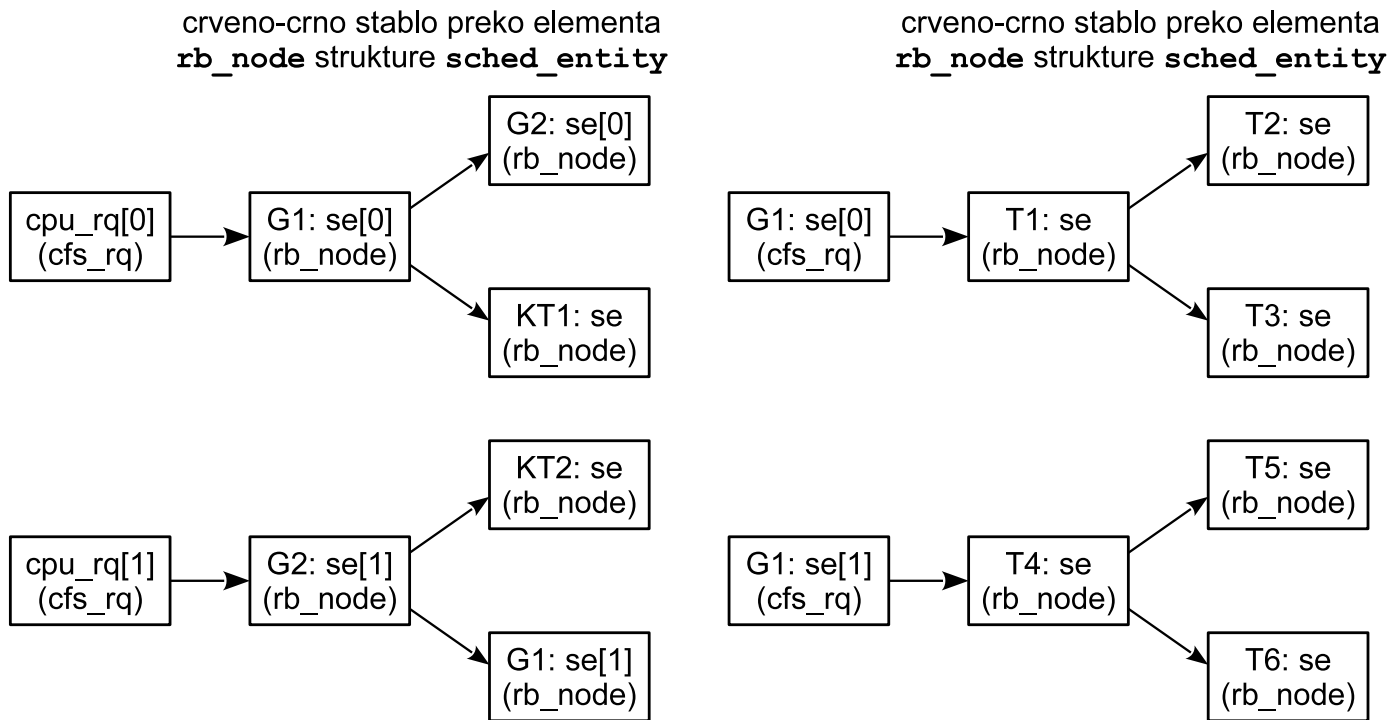
```
struct cfs_rq - sadrži crveno-crno stablo
    struct rb_root_cached tasks_timeline; - pokazuje na početni čvor u tom stablu

struct sched_entity - jedan čvor u 'tasks_timeline'
    struct rb_node run_node; - za ostvarenje stabla
    struct cfs_rq *my_q; - novi red, ako opisuje grupu

struct task_struct - opisuje zadatak, sadrži (pored ostalog)
    struct sched_entity se;

struct task_group -- opisuje grupu zadataka ili podgrupa, sadrži (pored ostalog)
    struct sched_entity **se; - za svaki procesor jedan se (polje kazaljki)
    komentar u kodu: schedulable entities of this group on each CPU
```

```
struct cfs_rq **cfs_rq; -- za svaki procesor jedan (my_q od sched_entity)
komentar u kodu: runqueue "owned" by this group on each CPU
```



Oznake: G-grupa, KT-jezgrin zadatak, T-zadatak, se-element `sched_entity`

Slika 10.1. Primjer struktura za red pripravnih dretvi

- raspoređivanje procesorskog vremena obavlja se na temelju grupa
- npr. u primjeru na slici 10.1. procesorsko vrijeme na procesoru 0 će se ravnopravno podijeliti među grupama G1, G2 i zadatku KT1: svi zadaci od G1 će sumarno dobiti slično vrijeme kao svi zadaci u grupi G2, tj. zadatak KT1
- na ovaj način mogli bi za svaki proces stvoriti zasebnu grupu te pravedno rasporediti procesorsko vrijeme po procesima, bez obzira na broj dretvi koji oni imaju; primjerice ako proces P1 ima 10 dretvi a P2 samo dvije, onda bi svaki od njih mogao dobiti jednako procesorskog vremena (tj. tada bi svaka dretva od P2 dobila pet puta više vremena od pojedine dretve iz procesa P1)
- stvaranje grupa može biti ručno, ali i automatsko
- ako je automatsko grupiranje omogućeno (a vjerojatno je) onda se za svaku novu grupu procesa (session) stvara nova grupa

10.2.2. Procesorske domene

- pokušaj da se uzme u obzir heterogenost procesora ostvaren je i kroz procesorske domene (izvorno *scheduling domain*)
- svaka domena `struct sched_domain` sadrži skup procesora nekih zajedničkih svojstava
- domene mogu biti hijerarhijski povezane
- svaka domena ima jednu ili više grupa zadataka (prethodno opisan mehanizam) koje se raspoređuju nad pripravnim procesorima
- za svaku domenu definira se skup pravila kojima se nastoji iskoristiti svojstva tih procesora

- kako često raditi balansiranje među redovima procesora (npr. za sklopovsku višedretvenost vrlo često, višejezgrene procesore nešto rjeđe, NUMA sustave još rjeđe)
- koliko dugo vremena se neki zadatak može ne izvoditi na nekom procesoru, a da ga kasnije i dalje ima smisla vratiti na isti procesor zbog mogućih podataka u priručnom spremniku
- dijeljenje napajanja – može li se samo jedan ugasiti ili slično?
- ...

Pitanja za vježbu 10

1. Navesti klase raspoređivača u Linuxu namijenjene za korisničke dretve.

Odgovor:

DEADLINE (SCHED_DEADLINE)
REALTIME (SCHED_FIFO, SCHED_RR)
CFS (SCHED_OTHER, ...)

2. Kod raspoređivanja nekritičnih dretvi u Linuxu se koristi CFS.

- a) Koju podatkovnu strukturu koristi taj raspoređivač?

Odgovor:

crveno crna stabla

- b) Što sve može biti element te strukture? Zašto (koja korist od toga)?

Odgovor:

opisnik dretve ili opisnik grupe (elementi grupe su ili druge grupe ili dretve)

3. Što je to grupa zadataka (`task_group`) u kontekstu CFS-a? Čemu grupe služe?

Odgovor:

služe za grupiranje dretvi po nekim kriterijima

npr. dretve istog korisnika mogu biti u jednoj grupi; dretve istog procesa mogu biti u jednoj (pod)grupi

raspoređivač može raspoređivati vrijeme prema grupi, npr. da sve grupe dobe jednako (npr. svi procesi jednako, bez obzira na broj dretvi u tim procesima)

4. Koja je osnovna zamisao u korištenju procesorskih domena (engl. *scheduling domains*)?

Odgovor:

bolje iskoristiti sustav poznavajući organizaciju procesora (sklopovska višedretvenost, dijeljenje priručnog spremnika)

radi segmentacije sustava – rezervacije procesora za različite namjene

Literatura

Općenito o operacijskim sustavima

- [Budin, 2010] Leo Budin, Marin Golub, Domagoj Jakobović, Leonardo Jelenković, *Operacijski sustavi*, Element, 2010.
- [Silberschatz, 2018] Abraham Silberschatz, Greg Gagne, Peter Baer Galvin, *Operating System Concepts, Tenth Edition*, Wiley, 2018.
- [Tanenbaum, 2015] Andrew S. Tanenbaum, Herbert Bos, *Modern Operating Systems, Fourth Edition*, Prentice-Hall, 2015.

Upravljački programi u Linuxu

- [Corbet, 2005] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, *Linux Device Drivers, Third Edition*, O'Reilly, 2005, <https://www.oreilly.com/library/view/linux-device-drivers/0596005903/>, <https://lwn.net/Kernel/LDD3/>.
- [Linux, API] *The Linux driver implementer's API guide*, <https://www.kernel.org/doc/html/latest/driver-api/>.
- [Shofer] Leonardo Jelenković, *Primjeri naprava (za NOS)*, <https://github.com/ljelenkovic/nos-shofer>.

Raspoređivanje dretvi

- [SRSV, 2021] Leonardo Jelenković, *Sustavi za rad u stvarnom vremenu*, skripta za predavanje, 2021. <http://www.zemris.fer.hr/~leonardo/srsv>
- [Kukolja, 2020] Antun Kukolja, *Novi raspoređivač poslova za Linux*, diplomski rad, 2020. https://www.bib.irb.hr/1073617/download/1073617.Final_0036489410_55.pdf
- [Linux, 2021] *Opisi raznih mehanizama u raspoređivačima Linuxa*, 2021. <https://www.kernel.org/doc/Documentation/scheduler/>
- [Corbet, 2004] Jonathan Corbet, *Scheduling domains*, 2021. <https://lwn.net/Articles/80911/>