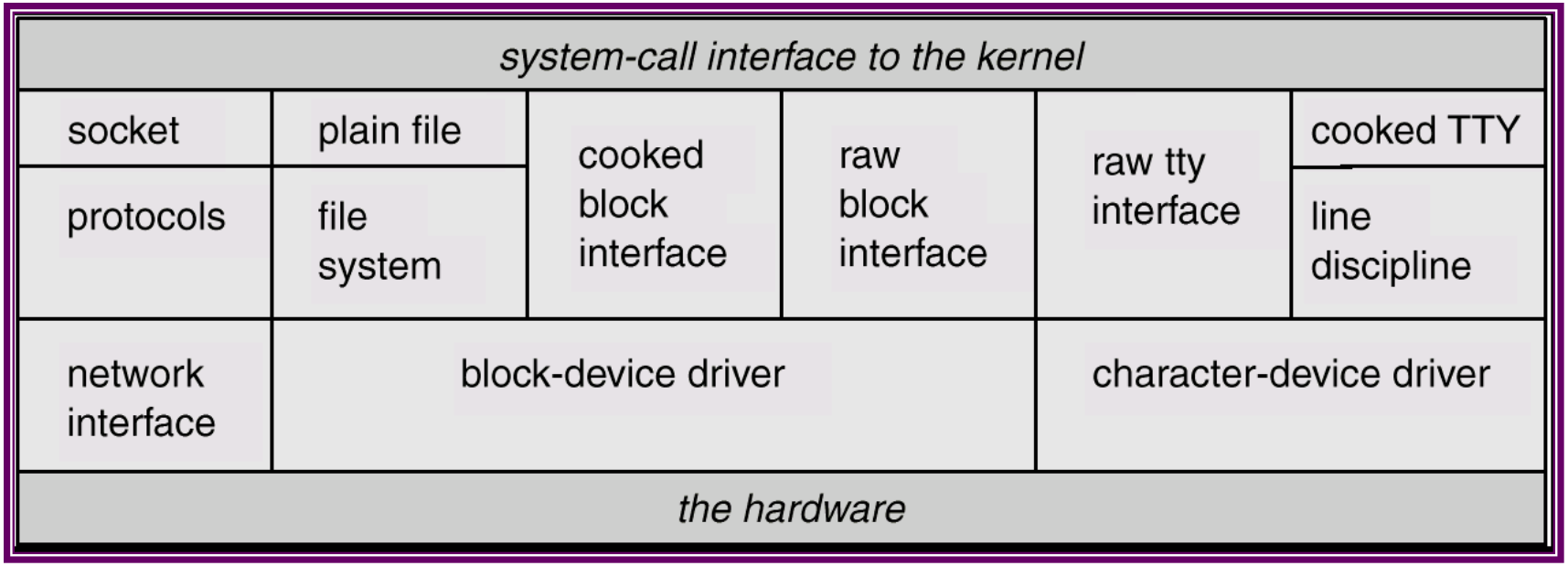
IO Subsystem

- **I/O podsistem** dozvoljava procesu da **komunicira sa periferijskim uređajima**, kao što su diskovi, trake, terminali, štampači, mrežni uređaji.
- **kernelski moduli koji kontrolišu uređaje nazivaju device drajveri**.
  - Obično postoji **one-to-one** korespodencija između drajvera i tipa uređaja:
  - UNIX može sadržavati jedan disk drajver da kontroliše sve diskove u sistemu,
  - jedan terminal drajver da kontroliše sve terminale.
  - Ako imate uređaje iste klase ali od različitih proizvođača drajveri za njih mogu biti različiti, zato što uređaji imaju različiti komandni set.
  - Na drugoj strani drajveri mogu biti prilično univerzalni, da podržavaju različite uređaje koje kontrolišu.
- UNIX podržava "**softverske uređaje**" koji nemaju pridruženi fizički uređaj. Na primer, **memorija** se tretira kao I/O uređaj, iako to nije I/O uređaj. Drajveri upisuju statističke informacije u kernelske data strukture, kao i trace zapise koji su pogodni za **debbuging**.
- Ova lekcija opisuje interfejs između procesa, I/O subsystema i drajvera. Objasnićemo **generalnu strukturu i funkciju drajvera, a detaljno ćemo** objasniti **disk i terminal drajvere**.
- Zaključićemo sa **novom metodom za realizaciju drajvera** koja se **naziva streams**

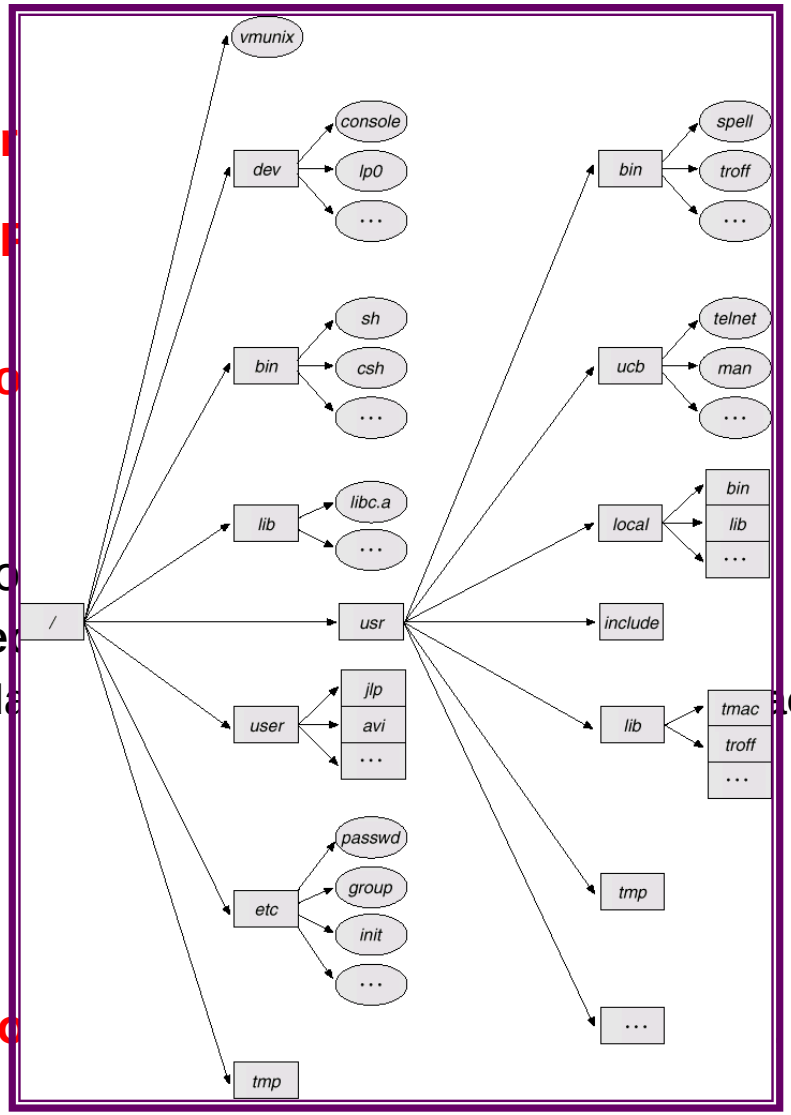
Driver interface

- UNIX sistem **sadrži 2 tipa uređaja**:
  - **block** uređaje
  - **karakter** uređaje (**raw**)
- **Blok** uređaji (**disk, tapes**) izgledaju kao **random-access storage uređaji**,
- **karakter** uređaji su terminali, štampači, mrežni uređaji.
- **Blok uređaji**, pored svog blok interfejsa mogu imati **interfejs za karakter** uređaje, takođe.



Driver interface

- **User interfejs za sve uređaje prolazi kr**
- **Svaki uređaj** ima **posebnu datoteku u F**
- Specijalne datoteke za uređaje imaju **svo** direktorijumu.
- Ove datoteke se razlikuju od običnih dato
  - one su **block specijalne ili karakter spe**
  - Ako uređaj **ima obe vrste interfejsa**, tad**a** **cter**)
- **SC koji važe za obične datoteke**
  - imaju **posebno značenje**
  - kada rade sa **specijalnim datotekama**.
- **SC ioctl** dozvoljava procesima da **kontro** ali se ne primenjuje za regularne datoteke
- brw-rw-rw- 1 root root 17.05.2009 2, 15 ……..hda1



System configuration

- Sistemska konfiguracija je procedura
- pomoću koje administratori **specificiraju parametre** koji su zavisni od instalacije.

- **Neki parametri specificiraju veličine kernelskih tabela**, kao što su:
  - **PT**
  - **inoda tabela**
  - **FT**
  - **broj bafera alociranih za keš bafer pool**

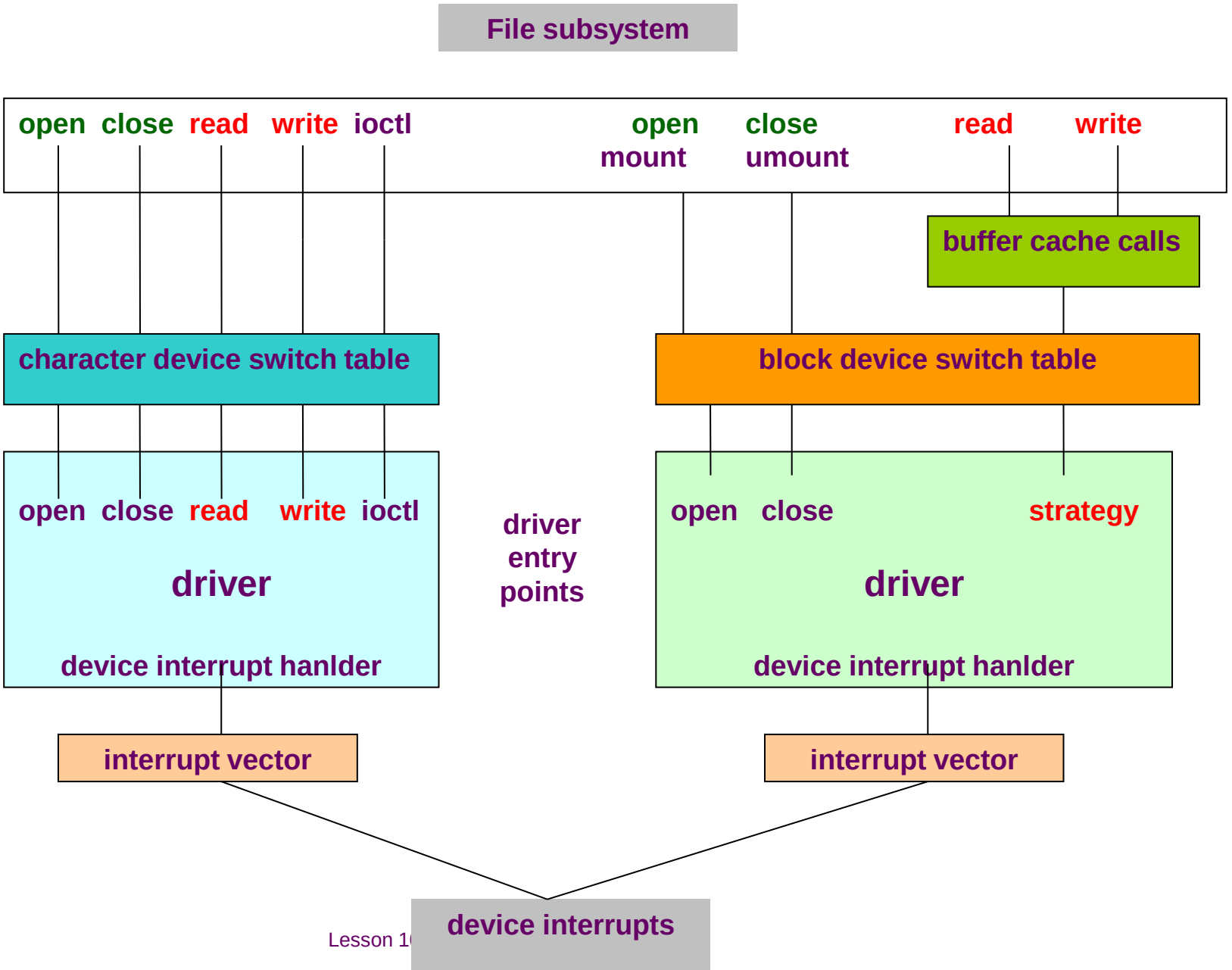
- **Drugi parametri** specificiraju **device konfiguraciju**, objašnjavajući kernelu **koji uređaji su uključeni** u **instalaciju** i **njihove adrese, interrupts, DMA**

- Na primer, konfiguracija može specificirati koja je **terminalska kartica ubodena u server**.

System configuration

- Ima **3 faze** u kojima se **specificira konfiguracija uređaja**.
- **prvo**, administratori mogu da ubace **konfiguracione podatke** koji se **kompajliraju i linkuju** kada je formira **kernel code**. Konfiguracioni podaci se tipično specificiraju u jednostavnom formatu i konfiguracioni program konvertuje ih u datoteku pogodnu za kompajliranje.
- **drugo**, **administratori** mogu da **ubace konfiguracione** podatke **nakon što se UNIX podigne**, a **kernel ažurira svoje interne** konfiguracione tabele **dinamički.**
- **treće**: **postoje "self-identifying"** uređaji koji omogućavaju kernelu da prepozna koji je uređaj instaliran. U tom slučaju, kernel čita hardverske prekidače-informacije i **automatski konfiguriše sam sebe**
- **Kernel drajver interfejs** je opisan preko **dve prekidačke tabele**:
  - **block device switch tabela**
  - **character device switch tabela**

I/O switch table



SC for I/O

- **Svaki tip uređaja** ima **ulaze u SW tabeli**
- koja **direktno upućuje kernel** na **odgovarajući drajver**
- u **saglasnosti** sa **SC**.

- SC **open i close** za **device datoteke** **prolaze** kroz dve **switch tabele**, saglasno **tipu datoteka**.

- SC **read, write i ioctl** za **karakter specijalne datoteke** prolaze kroz odgovarajuće procedure u **karakter switch tabeli**.

- SC **mount i umount** pozivaju **device open** i **device close** procedure **za blok uređaje**.

- **read i write SC** za **block uređaje** i **datoteke** u **mountovanim FS pozivaju** algoritme za **bafer keš**, preko **strategy** procedure

- **Neki drajveri** pozivaju **strategy proceduru**
  - **interno**
  - iz svojih **read** i **write procedura.**

major and minor number

- Administrator postavlja specijalne device datoteke sa **mknod komandom**, čiji ulaz obrazuju **file-type** (block or special) i **major i minor broj**.
- Komanda **mknod** poziva **mknod-SC** da kreira **device datoteku**
- **mknod /dev/tty13 c 2 13**
-
  - **/dev/tty13** je ime karakter datoteka,
  - c predstavlja karakter datoteku
  - sa 2 **kao major brojem** i **13 kao minor brojem**.
- **major broj ukazuje na tip uređaja** koji odgovara ulazu u blok ili karakter switch tabeli, a **minor broj ukazuje na jedinicu uređaja**.
- Ako proces otvara blok datoteku **/dev/dsk1** sa **major brojem=0**, kernel poziva rutinu **gdopen** na ulazu 0 blok karakter switch tabele kao na slici

karakter switch tabele kao na slici

block device switch table			
entry	**open**	**close**	**strategy**
0	gdopen	gdclose	gdstrategy
1	gtopen	gtclose	gtstrategy

major and minor number

- Ako proces čita (read) karakter datoteku **/dev/mem** i njen major broj=3, kernel zove **rutinu mmread** na **3-ćem ulazu karakter switch tabele**. Rutina **nulldev je prazna rutina**, koristi se kada **nema potrebe za posebnim drajverskim** funkcijama**.**

character device switch table					
entry	**Open**	**close**	**read**	**write**	**Ioctl**
0	conopen	conclose	conread	conwrite	conioctl
1	dzbopen	dzbclose	dzbread	dzbwrite	dzbioctl
2	Syopen	nulldev	syread	sywrite	syioctl
3	nulldev	nulldev	mmread	mmwrite	nodev
4	gdopen	gdclose	gdread	gdwite	nodev
5	gdopen	gdclose	gdread	gdwite	nodev

- Mnogi perferijski uređaji imaju **isti major broj**, a **minor broj ih razlikuje** između sebe. Device datoteke ostaju stalno na **/dev**, ne moraju da se kreiraju svaki put kad se UNIX diže, one se menjaju samo ako se promeni hardverska konfiguracija ili se instaliraju novi uređaji.

System call and driver interface

- Ova sekcija opisuje interfejs između **kernela i device drajvera**.
- Za **SC se koriste file-descriptori**, kernel prati pointere iz user file deskirptora u kernelovoj **FT i inodu**, gde se ispituju tip datoteke i pristup blok ili karakter switch tabeli.
  - Izvlače se **major i minor brojevi iz inoda**,
  - koristi se **major broj** za **index u odgovarajuću switch tabelu**
  - **poziva** se odgovarajuća **drajverska funkcija u saglasnosti sa parametrima SC kojima se dodaju major i minor broj dobijeni iz inode.**
- **Važna razlika između SC** za **device i regularne datoteke je da se inode specijalne datoteke ne lock-uje kad kernel izvršava drajver**.
- **Drajveri često spavaju**, čekajući na hardvesrku konekciju ili za nailazak podataka, tako da kernel ne može da odredi koliko će drajver da spava. Ako bi se **inode zaključavao**, **mnogi procesi bi takođe bili uspavani**, jer je prvi proces uspavan u drajverskom kodu.

- **Device drajver** interpretira parametre **SC** kao **parametre za uređaj**. Drajver održava **data strukture** koje **opisuju stanje svake jedinice** koju **kontrolišu**, drajver funkcije i IH izvršavaju se u saglasnosti sa stanjem drajvera i akcijama koje se obavljaju (**data input, data output**). Opisaćemo detaljno svaki interfejs.

open

- Kernel prati **istu proceduru** za **otvaranje uređaja**
- **kao** za **otvaranje regularnih datoteka**:
  - alocira inode u memoriju
  - inkrementira RC
  - dodeljuje FT ulaz i user file deskriptor

- Kernel, vraća **fd** procesu
  - **tako** da **otvoreni uređaj**
  - **liči**
  - **na otvorenu datoteku**.

- Međutim,
  - **kernel poziva**
  - **device-specific open proceduru**
  - pre **povratka u user mod**

algorithm open

- **algorithm open /* for device drivers*/**
  - input: **pathname, openmode**
  - output: **file descriptor**
- {
- convert **pathame to inode**, increment RC, allocate entry in FT, user **fd**,
- kao kod **open regularne datoteke**;
- **get major, minor number from inode**;
- **save context** (algorithm **setjmp**) in case of **long jump from driver**;
- **if(block device)**
- {
- **use major number** as **index** to **block device switch table**;
- **call driver open procedure** for index;
- **pass** minor number, open modes;
- }
- **else**
- {
- **use major number** as **index** to **character device switch table**;
- call driver **open procedure** for index;
- **pass** minor number, open modes;
- }
- **if(open fails in driver)** decrement **file table**, inode count;
- }

open description

- Za **blok uređaje**, **poziva** se **open procedura** kodirana u **blok switch tabeli**
- za **karakter uređaje poziva se open procedura** u **karakter switch tabeli**.
- Ako uređaj ima **obe varijante** (**block & character**) kernel će prozvati **open** proceduru iz **one tabele** zavisno koju je je vrstu datoteke user pozvao.
- **Dve open procedure** mogu biti **čak i identične**, zavisno od drajvera.
- **Device-specific open procedura:**
  - **1. uspostavlja konekciju** između procesa i **otvorenog uređaja**
  - **2. inicijalizuje privatnu drajversku data struktruru**
- **Za terminal**, na primer, **open procedura može uspavati proces** sve dok mašina ne detektuje **hardverski carrier signal** koji ukazuje da user pokušava log-in.
- Zatim se **inicijalizuju drajver data strukture** u saglasnosti sa setovanjem **terminala** (na primer terminal **baud rate**).
- Za **softverske uređaje** kao što je **sistemska memorija**, **open** procedura **nema šta da inicijalizuje**.

open description

- Ako proces mora da spava zbog nekog externog razloga kada se otvara uređaj, moguće je da se događaj koji će probuditi proces **nikada ne desi**. Na primer, ako nema usera da obavi **log-in na terminalu**, **getty proces** koji je otvorio terminal može spavati dugo vremena.
- Kernel mora da bude stanju da **probudi proces** i da prekine njegov open call, tako što pošalje procesu signal: tada se mora resetovati inode, ulaz u PT, user fd za koga je proces dodelio promenjivu ali nije dobio **fd**, jer je **open** otkazao.
- **Kernel** čuva **kontekst** korišćenjem algoritam **setjmp** pre nego što uđe u device-specific open rutinu, pa ako se proces probudi zbog signala, a kernel obnavlja kontekst procesa u stanje pre ulaska u drajver, preko **longjmp** algoritma i **oslobađa data strukture alocirane za open**.
- **Drajver može uhvatiti signal**, i **očistiti svoju privatnu data strukturu**, ako je potrebno. Kernel ponovo podešava FS data strukture kada drajver naiđe na grešku (na primer open za uređaj koji nije konfigurisan, kada **open** otkazuje).

- Procesi mogu **specificirati različite opciju** da kvalifikuju uređaj za **device-open**. Najčešća je opcija je "**no delay**", koja znači da proces **neće na spavanje** za vreme open procedure **ako uređaj nije spreman**. U tom slučaju open se vraća odmah, i tada user proces ne zna da li je kontakt sa hardverom uspostavljen ili ne. Otvaranje uređaja sa "**no delay**" opcijom takođe ima uticaj na semantiku **read call-a**, objasnićemo.

- Ako je **uređaj otvoren više puta**, kernel manipuliše sa user fd, inodom i ulazima u FT, **kao kod običnih datoteka**, stim što se za svaki open-uređaja poziva device-specific open procedura. Drajver može brojati koliko puta je uređaj otvoren, i ako je broj neodgovarajući, open SC se može otkazati. Na primer, to ima smisla, ako se dozvoli da više procesa da otvore terminal za upis, tako a useri mogu da razmenuju poruke. Ali **nema smisla** dozvoliti da više procesa **otvori printer za simultani upis**, pošto oni mogu da **prepisuju jedan drugome podatke**.

close

- Proces zatvara otvoreni uređaj, preko close procedure.
  - Kernel poziva **device-specific close proceduru**
  - **samo ako** je **zadnji close za uređaj**
  - što znači da nema više procesa koji imaju taj **device open**,
  - zato što device close završava hardversku konekciju, što znači da mora da čeka da nema procesa koji pristupaju uređaju.
- Zato što kernel poziva **device-open proceduru** za **vreme svakog open SC**, a **device-close samo jedanput**, drajver nije nikada nije siguran koliko procesa koristi uređaj.
- Drajveri lako mogu da izgube kontrolu ako nisu pažljivo napisani: ako spavaju u close proceduri, a drugi proces otvori uređaj pre nego što se **device-close kompletira**, uređaj će može postati neupotrebljiv ako kombinacija **open i close** procedura **dovede drajver u nedefinisano stanje**.
- Algoritam za closing uređaja je **sličan algoritmu za close regularne datoteke**

algorithm close

- **algorithm close /* for device*/**
- input: file descriptor
- output: none
- {
- do regular close algorithm;

- if (**file table RC not 0**) go to finish;

- if (there is another open file and its major, minor numbers are the same as device being closed) go to finish; /*not last close after all*/

- **if(character device)**
- {
- **use major number** as **index to character device switch table**;
- **call driver close procedure**: parameter minor number;
- }

algorithm close

- **if(block device)**
- {
- if(device mounted) goto finish;
- write device blocks in buffer cache **to device**; /*flushing*/
- **use major number as index** to block device switch table;
- **call driver close procedure**: parameter minor number;
- **invalidate** device blocks **still in buffer cache**;
- }
- finish:
- **release inode;**
- }

read/write (buffering)

- Kernel algoritam za read i write urađaja, **sličan je kao za regularnu datoteku**.
- Ako **proces čita ili piše karakter uređaj**, kernel poziva drajversku **read/wite** proceduru.
- Postoje slučajevi kada kernel prebacuje podatke **direktno između korisničkog adresnog prostora i uređaja,**
- **1. drajver po pravilu baferiše podatke, interno**. Na primer, **terminal drajveri** koriste **clist** da baferišu podatke. U takvim slučajevima, drajver **alocira bafer**, kopira podatke iz user adresnog prostora za vreme **write** procedure, i šalje podatke iz svog bafera na uređaj.
- **2. drajver usklađuje brzine**: ako user proces generiše podatke brže nego što uređaj može da upisuje, **write** procedura mora da uspava proces dok uređaj ne bude spreman da primi nove podatke.
- Za **read**, drajver prima podatke u svoj **bafer**, a onda ih iz bafera prosleđuje u
- user adresni prostor.
- Metod po kome drajver kominicira sa uređajem zavisi od hardvera, pri čemu se koristi **separatni I/O** ili **memorijski mapirani I/O**, kao na VAX-u.
- Neke mašine koriste **programirani I/O**, koji znači da mašina sadrži instrukcije koje kontrolišu uređaje (IO instructions, not memory).

read/write

- Pošto je interface između drajvera i hardvera mašinski zavistan, ne postoji standardni interfejs na ovom nivou.
- Za obe vrste I/O i memory-mapped i programmed I/O, drajver može koristiti DMA, između uređaja i memorije, pri čemu DMA to radi ceo transfer paralelno sa CPU, a na kraju uređaj postavlja prekid koji ukazuje da je transfer završen. Drajver postavlja fizičke adrese u DMA na pazi virtulenih adresa i paging mape.

- **Brzi uređaji mogu direktno kopirati** podatke
  - **između uređaja** i **user adresnog prostora**,
  - **bez upotrebe kernel bafera**,
  - a to će **ubrzati transfer**,
  - jer **imamo jednu operaciju kopiranja manje** (**device to kernel buffer**),
  - a **veličina trasfera** nije limitirana veličinom **bloka kernel bafera**.

- Takav transfer se naziva "**raw**" IO, obično se poziva iz **karakter read** i **write** procedura koje pozovu **block strategy** interface.

Strategy interface

- **Kernel koristi strategy interfejs** za
  - prenos podataka između **baferskog keša** i **uređaja**
  - mada **read** i **write procedure karakter uređaja**
  - ponekad koriste njihove **strategy procedure** za **transfer podataka**
  - direktno, između uređaja i korisničkog adresnog prostora
- **Strategy procedura** može **I/O poslove** da stavi u **queue za uređaj**,
- pa posle toga se startuje **sofisticirani I/O scheduling**.

- Kernel prosleđuje adresu **bafer headera** u **strategy proceduru**,
- a header sadrži listu adresa (pages) i veličinu za prenos podataka sa ili u uređaj.

strategy interface

- **kernel identifikuje svaku datoteku** u FS preko:
  - **device broja** za **FS**
  - **inoda datoteke**.

- **Device broj** se **koduje** kao **major i minor number**. Kada kernel pristupa bloku iz datoteke, on kopira **device broj** i **broj bloka** na **bafer header**

- Kada bafer keš algoritmi (**bread ili bwrite**) pristupaju disku, oni pozovu **strategy proceduru** na bazi **major broja uređaja**.

- **Strategy procedura** koristi polja u **bafer headeru:**
  - za **minor broj** i **broj bloka**
  - da identifikuju gde da nađe podatke na uređaju
  - i **bafer adresu** u **koju će prenositi podatke**

- Slično, ako proces pristupa **blok uređaju direktno**, (ako proces otvara blok uređaj i čita ga i piše ga), on koristi **bafer keš algoritme** koji rade na upravo opisani način.

Ioctl

- **SC ioctl** je **nastao kao generalizacija terminal-specific:**
  - **1. stty SC** (set terminal settings)
  - **2. gtty SC** (get terminal settings),
  - a to su SC raspoloživi na ranim verzijama UNIX-a.

- **ioctl** obezbeđuje generalnu **ulaznu tačku** za **device-specific komande**, dozvoljavajući procesu **da postavlja:**
  - **hardverske opcije** za **uređaj**
  - **softverske opcije** za **drajver**

- **Akcije** koje omogućava **ioctl zavise** od **uređaja**, a definisane su od strane drajvera.

- Program koji koristi **ioctl** mora da zna **tip datoteke** sa kojom će da se bavi, zato što su one **device-specific**. Ovo je jedini izuzetak generalnog pravila da UNIX ne pravi razliku između tipova datoteka.

ioctl

- Videćemo detaljni korišćenje ioctl za terminale, a sada da objasnimo sintaksu:
- **ioctl(fd, command, arg)**
- gde je
  - **fd file descriptor** koji je vratio open SC,
  - **command** je **zahtev drajveru da obavi neku akciju**, a
  - **arg** je **parametar** za **command** (moguće pointer na strukturu) ,

- **Komande** su **driver-specific**,
  - međutim svaki drajver interpretira komande prema internim specifikacijama,
  - a format data strukture **arg** zavisi od komande.

- **Drajveri mogu čitati arg** iz **user-prostora** na **bazi unapred definisanog formata** ili mogu **upisati device setovanje** u **arg**.

- Na primer, **ioctl omogućava korisniku** da:
  - postavi **baud-rate za terminal**
  - da **premotaju traku**
  - da **postave mrežnu adresu** na **mrežne kartice**

Other FS related calls

- SC za FS, kao što su **stat** i **chmod**
- rade sa uređajima kao da su regularne datoteke,
- oni manipulišu **inodom bez pristupanja drajveru**.

- **lseek SC** radi sa uređajima,
  - na primer ako proces obavi **lseek na traku**,
  - **kernel ažurira FT offset**
  - ali **ne obavlja driver-specific operaciju**.

- **Kada proces kasnije čita** ili **piše**
  - kernel pomera FT offset u **u-area**
  - kao za **običnu datoteku**
  -
  - a **uređaj se fizički pomera** na **korektan offset** na koji ukazuje **u-area**.

Disk drivers

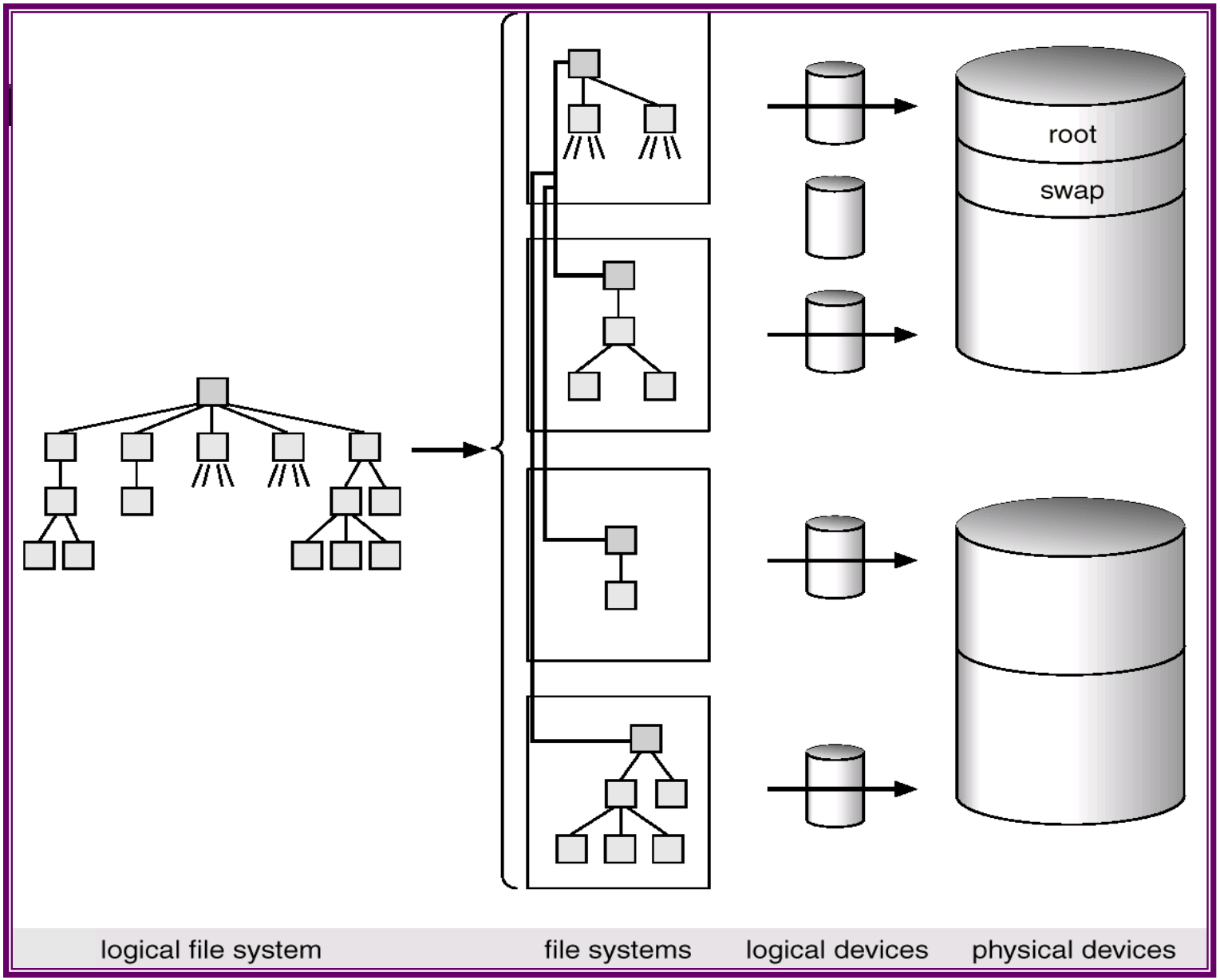
- Istorijski, UNIX je uvek dozvoljavao da se disk deli na više delova koji sadrže FS. Na primer, ako disk sadrži 4 FS, administrator može ostaviti jedan FS da bude umounted, da jedan mountuje kao read-only, a da 2 mountuje kao r/w. Mada su svi FS, delovi istog diska, korisnik ne može pristupati datotekama u umount FS niti može pisati u read-only FS.

- Disk drajver **translira FS adresu** koja se sastoji od:
  - **1. broja uređaja (device number)**
  - **2. broja bloka (block number)**
  - u partikularni sektor na disku.

- **Drajver dobija adresu** na jedan od **2 načina**:
  - ili **strategy** procedura **koristi bafer iz bafer pool-a**, a **bafer header** sadrži **device broj** i **blok broj**
  - ili **read** i **write** procedurama se prosleđuje **device broj kao parametar**, a procedure konvertuju **byte offset** (sačuvan u u-area) u odgovarajuću **disk blok adresu**.

- **Disk drajver** koristi **device broj** da identifikuje **fizički disk i FS**, preko interne tabele u kojoj je označen početak svakog FS. Na bazi te tabele i offseta u FS, doda se **offset na početak FS** i dobije se **traženi sektor**.

primer podele diska



primera podele diska

- Pretpostavimo da su disk blok datoteke **/dev/dsk0, /dev/dsk1, /dev/dsk2 i /dev/dsk3**, prve 4 sekcije na disku sa major broj =0 i minor brojevima od 0 do 3. Pretpostavimo da je **sistem_blok = disk_blok = 512 bajtova**.
- Ako sistem pokušava sa se obrati bloku **940** na FS **/dev/dsk3**, disk drajver će konvertovati tu adresu u adresu bloka na disku **336940 = 336000 + 940**.
- Veličina FS na disku varira, a određuju je administratori sistema.
- Za **ceo disk** postoji posebna datoteka, u ovom slučaju **/dev/dsk7**.
- Korišćenjem fiksnih sekcija na disku smanjuje se fleksibilnost.
- Zato se informacija o disk particijama ne čuvaju u **disk drajveru**, nego u konfigurabilnim tabelama na **samom disku**, mada je iz razloga kompatibilnosti teško definisati univerzalnu poziciju za tu tabelu na disku.
- Na primer **boot block** je **početak svakog diska** i obično je u njemu **master volume tabela**, a **superblock je početak svakog FS**. Bez, obzira na sve, disk drajver zna gde je master volume tabela za dati disk.
- Mnogo pažnje je posvećeno disk performansama, disk sheduling, data transfer, sve je prebačeno iz drajvera u sam disk kontroler itd.

block v raw access

- Sistemski programi za obradu diska, mogu koristiti ili raw ili block interfejs da pristupe disku direktno bez FS poziva.
- Dva značajna program za diskove su **mkfs i fsck**.
- 1. Program **mkfs** kreira FS u particiji, **kreira supeblock, inode listu, linkovanu listu free disk blokova, root directorijum na novom FS**.
- 2. Program **fsck** proverava konzistenciju postojećeg FS i ispravlja moguće greške.

- Analizirajmo program na slici. Prethodno pogledajmo blok i karakter datoteku za /dev/dsk15 dobijenu sa
- **#ls –l /dev/dsk15 /dev/rdsk15**
- **br-------- 2 root root 0, 21 Feb 12 15:40 /dev/dsk15**
- **crw-rw---- 2 root root 7, 21 Mar 7 09:29 /dev/rdsk15**

- Datoteka /dev/dsk15, vlasnik je root, za blok datoteku samo root ima pravo čitanja, i ima major broj =0, minor broj =21. Datoteka /dev/rdsk15, vlasnik je root, za ovu datoteku i vlasnik i grupa imaju pravi read i write i ima major broj =7, a minor broj =21.

block v raw access

```
#include "fcntl.h"
main()
{
char buf1[4096], buf2[4096];
int fd1, fd2, i;
if(( fd1=open("/dev/dsk15", O_RDONLY)) = = -1)
|| ((fd2=open("/dev/rdsk15", O_RDONLY)) = = -1))
    {  printf("failure on open");    exit();    }

lseek (fd1, 8192L,0)
lseek (fd2, 8192L,0)

if((read(fd1, buf1, sizeof(buf1)) == -1) || (read(fd2, buf2, sizeof(buf2)) = = -1))
    {    printf("failure on read");   exit();    }

for(i=0; i< sizeoff(buf1); i++)
{
if( buf1[i] != buf2[i])
    {  printf("different at offset %d",i);          exit();    }
printf("reads match");
} }
```

block v raw access

- Proces otvara datoteku **/dev/dsk15**, pristupa disku preko **blok switch tabele**, a datoteku **/dev/rdsk15** preko **karakter switch tabele**.
- Kako su **minor brojevi** isti za obe datoteke,
  - obe datoteke ukazuju na istu **particiju na disku**,
  - pa će proces izvršiti open proceduru istog drajvera 2 puta (preko različitih switch tabela),
  - obave lseek na isti offset i čitaju podatke sa istih blokova diska,
  - što bi trebalo da dobije **isti rezultat**.
- Programi koji čitaju i pišu direktno su opasni jer mogu da **prepišu osetljive podatke**. Administratori moraju da **zaštite i blok i raw pristup disku**, kao na u primeru i blok i karakter datoteke su **vlasništvo root-a** i samo on može da ih **čita ili piše**.
- **Programi koji čitaju i pišu direktno** mogu da **oštete konzistenciju FS** data jer mogu da **prepišu direktorijume i inode blokove** itd.

block v raw access

- **Razlika između blok i karakter disk interfejsa** je u tome da li se bave sa **bafer keš-om**.
- Kada proces **otvara blok specijalnu datoteku** (**block interface**),
  - **sve je isto** kao kod **obične datoteke**,
  - osim što posle konvertovanja **bajt offseta** u blok offset (bmap),
  - kernel tretira **logički blok offset** kao **fizički broj bloka** u **FS.**
  - **Zatim pristupa podacima preko bafer keša preko strategy procedure**.
- Kada proces **otvara karakter specijalnu datoteku** (**raw interface**),
  - kernel ne konvertuje byte offset u **file offset**
  - nego prosleđuje offset direktno drajveru u **u-area**,
  - a onda drajver r/w rutine obavljaju konvertovanje **bajt offseta** u blok **offset**, i kopiraju podatke direkno na/u disk (**bypass cache**).

block v raw access

- **Ako jedan proces piše u blok datoteku**,
- a **drugi proces to čita** isti blok ali kroz **karakter datoteku**,
  - raw proces možda ne pročita ono što je block proces upisao,
  - zato što su mu podaci u kešu a ne na disku (**delayed write**),
  - a da je čitao kroz **blok interfejs**, dobio bi iz keša sveže podatke.

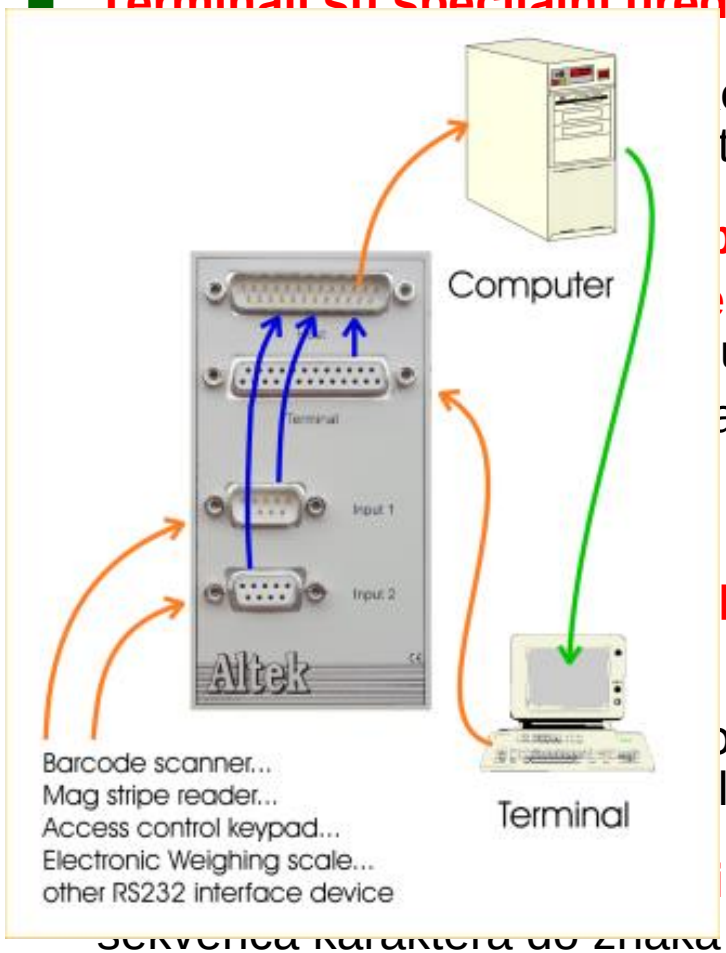
- **Korišćenje raw interfejsa** može napraviti **čudno ponašanje**. Ako proces čita ili piše na raw device u jedinicama manjim od bloka na disku, rezultati zavise od samog drajvera. Na primer ako pošaljete 1-no bajtne upise na tape, svaki bajt može se pojaviti na različit tape blok.

- **Prednost korišćenja raw interfejsa je brzina**, ako nema ponovnog čitanja, kada bi keš imao veću prednost. Procesi koji pristupaju po blok interfejsu, su ograničeni na **sistem blok size transfere**, jedna disk operacija prenosi 1 sistemski blok podataka (**1K na primer**).

- Ako se koristi **raw disk interfejs**, veličina bloka je **ograničena disk kontrolerom**. Funkcionalno, **čitanje će biti isto**, stim što je **raw interfejs mnogo brži.**
- U prethodnom programu proces će **4K** (**1K system block**) da pročita **u 4 iteracije**, dok će **čitanje** kroz **disk interfejs** da se zadovolji u **1-oj disk operaciji**, **nema ni duplog transfera sa diska u keš**, pa iz **keša u user prostor.**

Terminal drivers

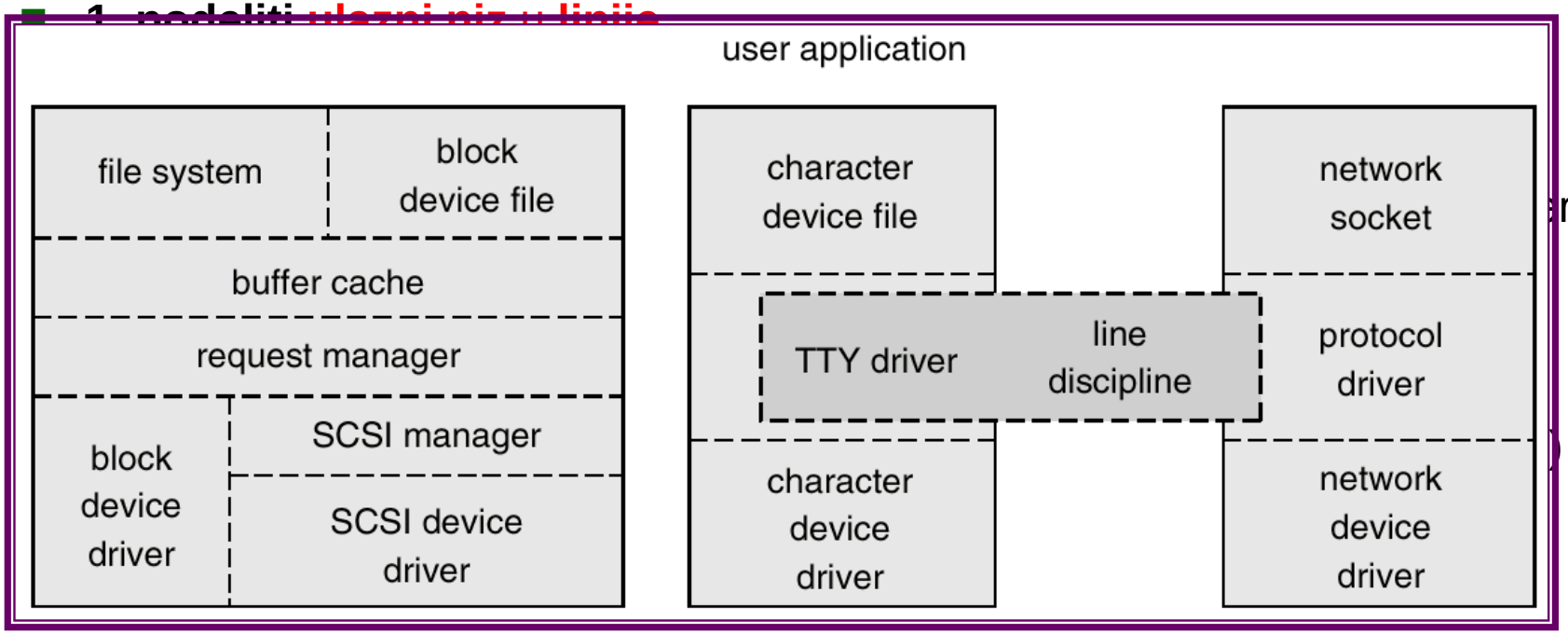
- **Teminal drajveri** imaju funkciju da kontrolišu **prenos podataka** sa/u terminal.
- **Terminali su specijalni uređaj**



na linije baferiše podatke u linije (linija je
za novi red) i procesira **erase** karaktere interno
pre nego što pošalje **promenjenu sekvencu procesu.**

line discipline

- Funkcije **discipline linije** su:
- **1. podeliti ulazni niz u linije**

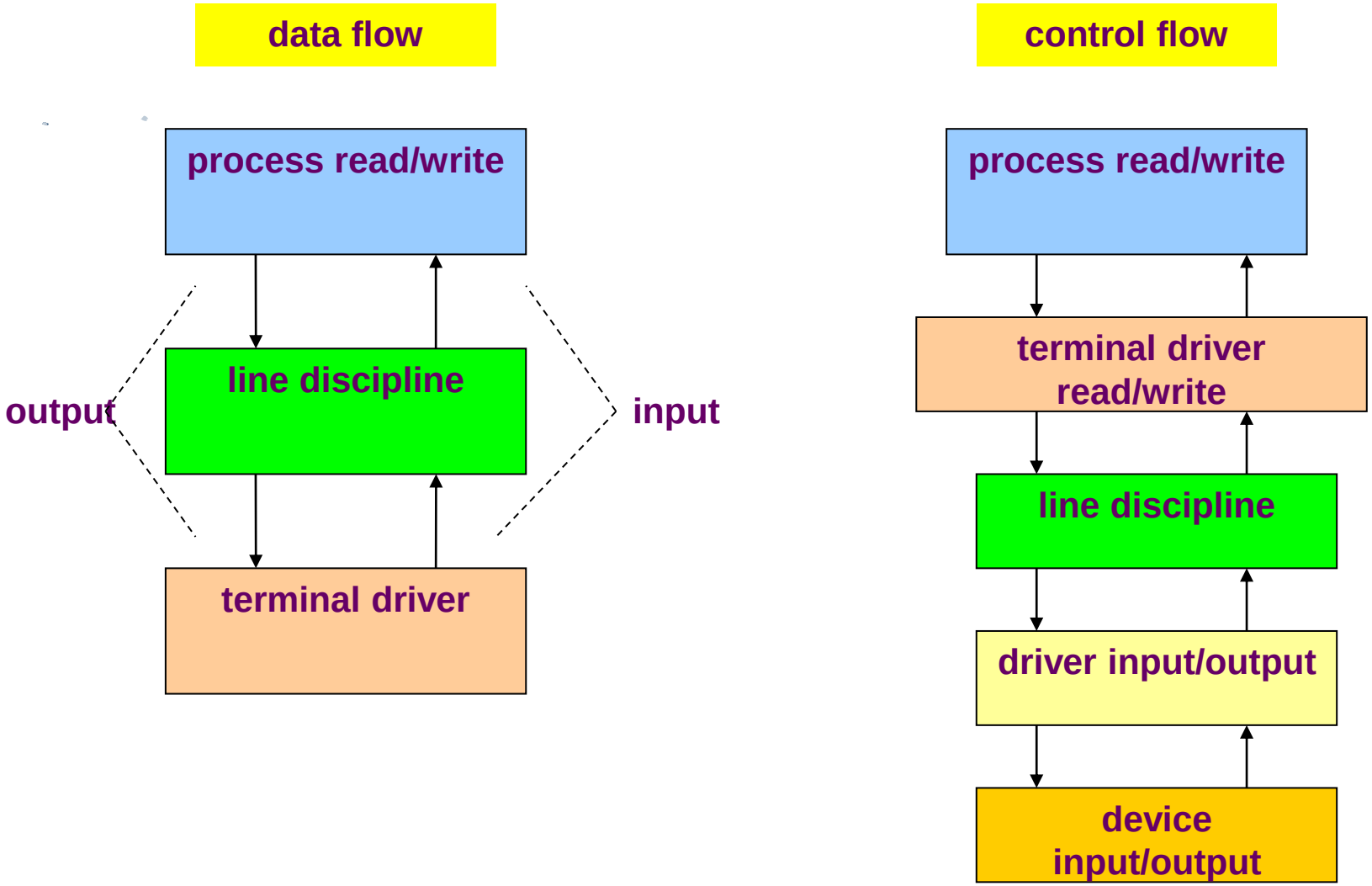


- **7. dozvoliti raw mode** koji ne interpretira specijalne karaktere kao erase, kill ili CR

raw mode

- Da bi se podržavao **raw mode**,
  - podrazumeva se upotreba **asinhronog terminala**,
  - tako da proces može čitati svaki karakter koji se otkuca,
  - umesto da proces čeka da **user otkuca CR**
- Prvi UNIX-i nisu imali **disciplinu linije** u kernelu što je kasnije urađeno, jer je disciplina linije potrebna za mnoge programe a ne samo za shell i editor, pa je **ugrađena u kernel**.
- Mada disciplina linije izvršava funkcije
  - koje su smeštene između terminal drajvera i ostatka kernela,
  - kernel ne poziva **LD direktno**,
  - nego kroz terminal drajver.
- Na slici je prikazan:
  - **logički tok podataka kroz terminal drajver** i LD
  - **tok kontrole kroz terminal drajver**.
- Korisnici mogu da specificiraju **koju disciplinu linije** hoće da koriste preko **ioctl SC**, ali je **teško realizovati da jedan uređaj** korsiti **više LD simultano**, gde **svaki LD modul** poziva **sledeći modul** da procesira obrađene podatke.

data flow + control flow

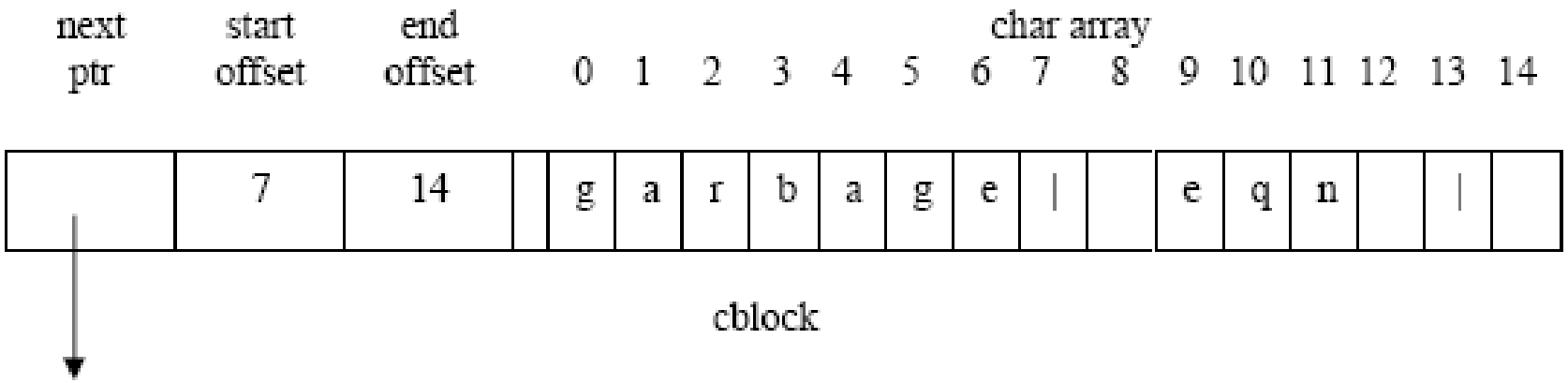


Clists

- **C-liste** obezbeđuju prosti **bafer mehanizam**,
- **zgodan** za **male količine podataka**
- **spore uređaje** kao što je terminal.
- One dozvoljavaju manipulaciju podataka **jedan po jedan karaker** u grupi **c-blokova**.

Clists

- Line disciplina manipuliše podacima u clist-ama.
- **Clist (character list),**
  - je linkovana lista **cblock-ova** promenljive dužine
  - sa promenljivim brojem karaktera u listi
- **Cblock** sadrži:
  - pointer na sledeći **cblok** u **linkovanoj listi**,
  - polje podataka (obično malo) i
  - **dva offseta** (start i end) koji ukazuju na **validne podatke** u **polju**, kao na slici

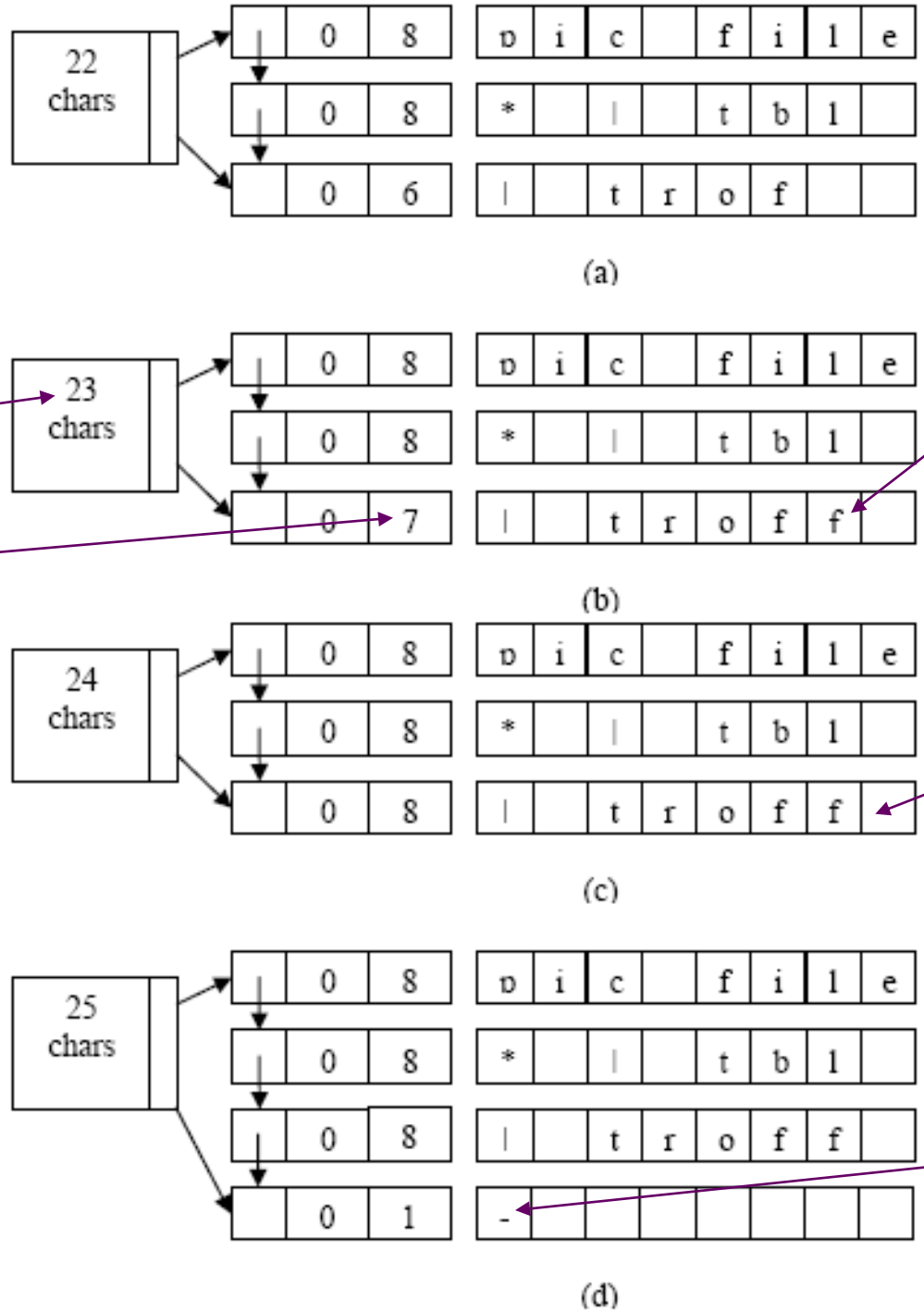


C-list operations

- Kernel podržava linkovanu listu slobodnih cblokova i ima **6 operacija** nad **clistama** i **cblokovima**:
- 1. **operacija dodele cblocka** iz **free liste u drajver**
- 2. **operacija povratka cblocka** u **free listu**
- 3. **kernel može proslediti prvi karakter iz cliste**:
  - on **uklanja prvi karakter** iz **prvog cbloka** i **podešava clist** broj karaktera
  - ukazuje u cbloku da sledeća operacija neće biti prosleđivanje istog karaktera.
  - ako se prosleđivanje odnosi na zadnji karakter u **cbloku**, kernel plasira **prazan cblok** na **free listu** i podešava **clist pointere**. Ako clista ne sadrži kakatere kada se prosleđivanje dogodi, kernel vraća **null karakter**.
- 4. **Kernel može postaviti karakter na kraju cliste**,
  - nalazeći zadnji cblok, stavljući karakter u njega i podešavajući **offset** vrednosti.
  - Ako je **cblock** pun, alocira se **novi cblok** iz free liste, ubacuje se u clistu a u njega se apenduje karakter.
- 5. **Kernel može da ukloni** grupu karaktera sa početka **cliste** i to jedan po **jedan cblock** u **jednom trenutku**, što je **ekvivalentno ukljanjanju** svih karaktera u **cbloku jedan** po **jedan**.
- 6. **Kernel može plasirati popunjeni cblock** na **kraj cliste**

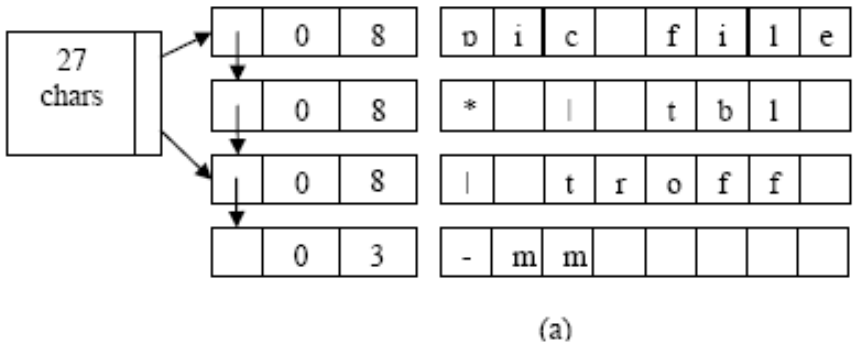
C-list example (adding character)

- slika prikazuje dodavanje karaktera u clistu
- (cblock sadrži 8 karaktera) i
- kada nema više mesta, kernel linkuje novi cblock (d).


22 chars
0 8
0 8
0 6
p i c f i l e
* | t b l
| t r o f
(a)
23 chars
0 7
f
(b)
24 chars
char blank
(c)
25 chars
char -
(d)

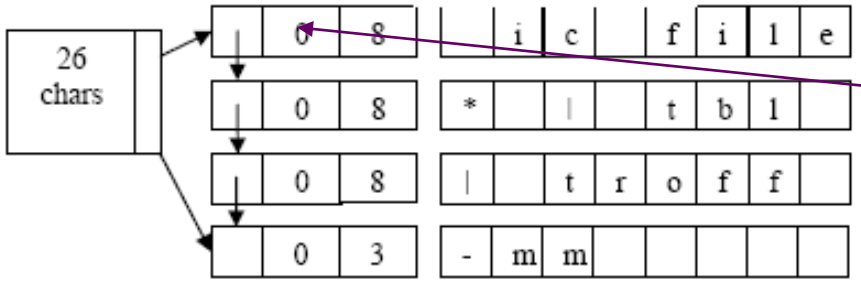
C-list example (removing character)

- Na primer slika (c) prikazuje ukljanjanje karaktera iz cliste
- kernel ukljanja jedan po jedan karakter iz prvog cbloka (a)-(c)
- sve dok **nema više karaktera** u cbloku (d)
- **kada se cblock izbacuje**



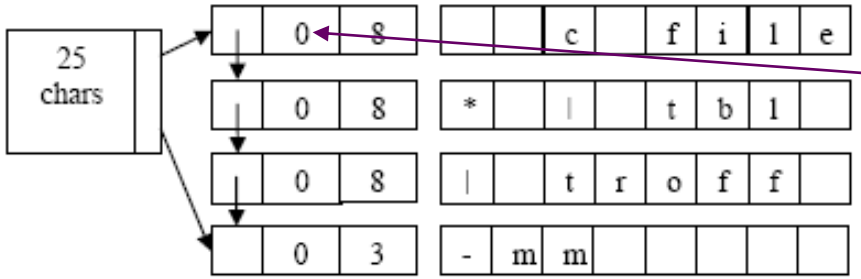
(a)

removing p



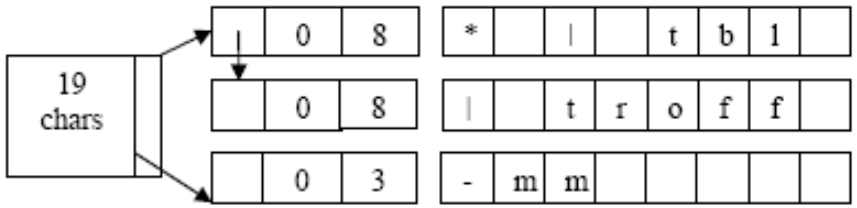
(b)

removing i



(c)

removing pic file



(d)

Terminal driver in canonnical mode

- **Data strukture** za **terminal drajvere imaju**
- **3 vrste clist-a**

- **1. output clist**
  - **clist** koja čuva podatke za **output terminal**

- **2. raw input clist**
  - **clist** koja čuva **"raw" input** podatke koje obezbeđuje IH kada user kuca na tastaturi

- **3. cooked input clist**
  - **clist** koja čuva "**cooked**" **ulazne podatke**,
  - pošto disciplina linije konvertuje specijalne karaktere u "**raw**" listi,
  - kao što je **erase** ili **kill** karakter.

algorithm terminal_write

- Kada proces upisuje na terminal, terminal drajver poziva **LD modul**.
- **LD ima petlju**,
  - **čita output karaktere from user adresnog prostora**
  - ubacuje ih u **output-clist**
  - sve dok ima mesta.
- **LD procesira izlazne podatke**,
- **proširuje tab** karaktere u seriju **space karaktera**, itd.
- Ako broj karaktera u **output clisti** postane veći od **high-water mark**,
  - LD poziva **drajversku proceduru**
  - za prenos podataka iz **output-cliste** na **terminal**
  - postavlja **writing proces na spavanje**.
- Kada **količina podataka** u **output-clisti padne ispod low-water mark**, **IH budi sve procese uspavane** na događaju, da **terminal može da prihvati još podataka**.
- **LD završava svoju petlju**,
  - **kopira sve podatke iz user adresnog prostora** u **output-clistu**
  - **poziva drajverske procedure** da **prebace podatke** na **terminal**

algorithm terminal_write

- **algorithm terminal_write** (čitaj objašnjenje ispod algoritma)
- {
- **while(more data to be copied from user space**)
- {
- **if(tty flooded with output data**) /*high WM*/
- {
- **start write operation to hardware with data on output clist;**
- sleep (event: tty can accept more data);
- continue; /*back to while loop*/
- }
- **copy cblock size** of **data from user space** to **output clist**;
- **line discipline convert tab characters**, etc;
- }
- **start write operation to hardware with data on output clist;**
- }

algorithm terminal_write

- Ako **više procesa pokušava upis na terminal**,
- oni prate opisanu proceduru nezavisno.

- **Izlaz može biti čudan**, podaci raznih procesa **mogu se preklapati** na **terminalu**.
- Ovo se može desiti zato što proces **može pisati na terminal** koristeći **više write SCs.**

- Kernel može obaviti **CSw** dok je proces u **user modu**, između **sukcesivnih write SC**, i novi aktivni proces može pisati na terminal dok početni proces spava.

- **Izlazni podaci mogu biti izmešani** na terminalu, **zato što proces koji piše** može **zaspati u sredini write SC**, dok čeka da se **upišu prethodni podaci**.

- **Kernel može izabrati drugi proces** koji **piše na terminal dok je prvi uspavan**. Kernel ne garantuje da će sadržina data bafera za **output na terminal da bude kontinualna na terminalu**.

Terminal write example

```
char form[] = "this is a sample output string from child"
main
{
char output[128];
int i;
for (i=0, i<18; i++)
{
switch (fork())
{
        case -1: exit(); /* error = = = hit max processes*/
        default: /*parent process*/
           break;
        case 0; /*child process*/
        /*format output string in variable output*/
        sprintf(output, "%s%d\n%s%d\n", form, i, form, i);
for(;;)
     write(1, output, sizeof(output));
}
}
}
```

Terminal write example

- Proces roditelj kreira do **18 dece**,
  - svako dete-proces formatira niz (preko funckije spintf) u polju **output**, sprintf uključuje poruku i vrednost i koja je redni broj **fork()**,
  - a samim tim redni broj deteta,
  - a **zatim dete u petlji upisuje string na terminal** (**fd = 1**).

- **Namerno** je podešeno da **output string bude veći od 64 bajta**, da ne može da stane u jedan **cblock** koji je na UNIX System V = **64 bajta**.

- Kada terminal drajver traži više od 64 bajta za upis na terminal,
- output može postati i**zmešan**, kao
- **this is a sample output string from child 1**
- **this is a sample outthis is a sample output string from child 0**
-

Canonical read

- Čitanje podataka sa terminala u **kanoničnoj formi** je **mnogo kompleksnija** operacija.
- **SC read** specificira **broj bajtova koje proces želi** da **pročita**,
- ali **LD će zadovoljiti SC read**, **uvek kada user otkuca CR tj <enter>**,
- bez obzira što **bajt count nije zadovoljen**.
- To je praktično, s obzirom da proces ne može da predikuje koliko će karaktera user otkucati na tastaturi, pa nije bitno koliko ima karaktera.
- Na **primer nije bitno za usera** da li je otkucana komanda prosta kao **date** ili ls, ili je složena kao
- **pic file* | tbl | eqn | troff – mm –Taps | apsend**
- Terminal drajver ne zna ko čita sa terminala, da li je to shell, editor ili nešto drugo,
  - **LD reaguje na svaki CR**
  - zadovoljava **SC read odmah posle CR**

algorithm terminal_read

- Pretpostavimo da je teminal u **kanoničkom modu**.
- **Ako nema podataka** na **ulaznoj clisti**, **reading proces spava** sve dok ne **dođu podaci** sa **terminala**.
- **Kada se podaci unesu**, terminal IH poziva IH iz LD,
  - koji plasira podatke na raw clistu za ulaz reading procesa i
  - na **output-clistu** za prikazivanje **echo-back** na terminalu.
- Ako **input string sadrži** CR, **IH će probuditi** sve uspavane **reading** procese.
- Kada se **reading proces izvršava**,
  - **drajver izbacuje** karaktere iz **raw cliste**,
  - obavlja **erase** i **kill procesiranje**,
  - **plasira karaktere** u **kanoničnu clistu**.
- Zatim se **kopiraju karakteri** u **user adresni space** **sve dok** se **ne pojavi CR** ili se **ne zadovolji count** u **terminal_read** zahtevu (zavisno koji je broj manji).
- Međutim, **probuđeni proces** može da **detektuje da podaci zbog kojih** se **probudio više ne postoje**, zato što **drugi procesi mogu pročitati terminal** i ukloniti podatke iz **raw cliste** pre nego što probuđeni proces **dobije kontrolu**.
- **Slična sitacija** se **dešava kada više procesa čita** iz **pipe**.

algorithm terminal_read

- Karakter procesiranje u ulaznom i izlaznom smeru je **asimetrično**,
- praćeno sa **2 ulazne cliste** i jednom **izlaznom clistom**.
- za terminal_write LD
  - izbacuje podatke iz user prostrora,
  - procesira je i
  - ubacuje u output clistu.
- Da bi bilo simetrično, trebalo bi da bude **samo jedna ulazna clista**.

- Ali to bi značilo da IH u procesiranju erase i kill karaktera bude kompleksniji i da blokira druge prekide u kritičnoj sekciji.

- Korišćenjem **2 ulazne cliste**, znači da
- IH može da upisuje karaktere prosto u **raw clistu**, i
- probudi **uspavane reading procese**.

- **IH takođe gura ulazni karakter** neposredno na **output clistu**,
- tako da **posle minimalnog kašnjenja user vidi otkucani karakter** na **terminalu**.

algorithm terminal_read

- **algorithm terminal_read**
- {
- **if(no data on canonical clist)**
- {
- **while(no data on raw clist)**
- {
- **if(tty opened** with **no delay option**) return;
- {
- if(tty in raw mode based on timer and timer not active)
- arrange for timer wakeup (callout table);
- **sleep**(event: **data arrives from terminal**);
- }
- /* there is data on raw clist*/
- **if(tty in raw mode**) **copy all data** from **raw clist** to **cannonical clist**;
- else

algorithm terminal_read

- else
- {
- **while(characters on raw clist)**
- {
- **copy one character** at time from **raw clist** to **canonical clist**:
- do erase, kill processing;

- if(char is CR or end-of-file) break; /*out of while loop*/
- }
- }/*else*/

- }/* if*/

- **while(characters on canonical list** and **read count not satisfied**)

- **copy** from **cblocks** on **canonical list** to user **address space**;
- }

terminal_read example

- Slika prikazuje program gde se kreira puno dece18 koja čitaju svoj standarni ulaz, a to je terminal i **koji je prespor** da zadovolji sve reading procese, tako da će procesi provesti mnogo vremena **spavajući u terminal-read algoritmu.**
- char input[256]
- main
- {
- register int i;
- for (i=0, i<18; i++)
- {
- switch (fork())
- { case -1: /* error = = = hit max processes*/ printf("error cannot fork"); exit();
- **default: /*parent process*/ break**;
- **case 0; /*child process*/**
- for(;;) {
- **read(0, input, 256); /*read line*/**
- **printf("%d read %s \n", i, input);**
- }
- } } }.

terminal_read example

- **Kada korisnik unese liniju podataka**, **terminal IH budi sve reading procese**, **pošto** oni **svi spavaju** na **tom istom prioritetnom nivou,** svi su poželjni.

- **User ne može da predvidi** koji će **proces da se izvršava**, i čita liniju, to će program da ispiše i **proces-dete (i)** i liniju koju je pročitalo.

- **Ostali procesi** će **dobiti CPU**, ali **verovatno neće** imati **input-data** na **input-clistu** i **idu opet na spavanje**. Procedura se ponavlja na unos svake nove linije, **ne može se garantovati** da jedan proces **ne potroši sve ulazne podatke**.

- Situacija sa **višestrukim čitanjem terminala je komlikovana**, ali kernel mora dozvoliti višestrukim procesima da čitaju terminal simultano, jer proces koga je startovao shell, a očekuje da input neće nikada raditi, zato što shell pristupa standarnom ulazu takođe.
- Zato procesi moraju sinhronizovati svoj **pristup terminalu na user nivou**.

terminal settings

- User setuje terminalske parametre kao što je **erase** i **kill karakter** i dobija **setovane vrednosti** preko **ioctl SC**.

- Slično, **oni kontrolišu**
- da li terminal izbacuje echo za svoj ulaz,
- setuje terminal baud rate,
- prazni U/I request,
- ručno postavlja start i stop karakter.

- **Terminal drajver** ima **data strukture** koje **čuvaju različito terminalsko setovanje**, a **LD prima parametre preko ioctl SC**, i postavlja ili čita polja iz terminalskih data struktura.

- **Kada proces setuje terminalske parametre**, **to važi za sve procese koji koriste terminal**. Setovanje se ne ukida automatski, kada proces obavi exit.

terminal driver in raw mode

- Procesi mogu da postave terminal u **raw mode**,
- gde LD prebacuje karaktere onako kako ih je **user otkucao**,
- **nikakvo input procesiranje** se ne obavlja.

- **Kernel mora da zna** kako da **zadovolji read SC**,
- u slučaju da se **CR** tretira kao **običan karakter**.
- **To se radi tako što** se **read zadovolji:**
  - **posle minimalnog broja ulaznih karaktera** ili
  - **posle nekog fiksnog vremena** od **prijema prvog karaktara** sa **terminala**,
  - a to vreme kernel namešta u **callout tabeli**.

- Oba kriterijuma se setuju preko ioctl SC.
- Kada se **neki kriterijum ispuni**, **IH of LD budi uspavane procese**.

- **Drajver prebacuje sve karaktere** iz **raw liste** u **kanoničku listu** i **radi sve** kao u algoritmu **za kanonički mod**.

- **Raw mode** je **značajan za screen orijentisane aplikacije**, kao što je **vi editor**, koji ima mnoge komande koje se **završavaju sa CR**, koji je SC zahtevao user.

ioctl example

- Na slici je dat program koji obavlja **ioctl**
- **da sačuva tekući terminalni settings** za **fd 0**, (**standardni ulaz**).

- Komanda **ioctl TCGETA** daje **nalog drajveru** da **dobije setovanje** za **terminal** i **sačuvan** je u **strukturi savetty** u **user** prostoru.
- Ova komada se koristi da odredi da li je datoteka terminal ili ne, a ne menja ništa u sistemu: ako otkaže to nije terminal.

- Ovde proces obavi **drugi ioctl SC** koji **postavlja terminal u raw mode**.
  - **Ukida se echo** i
  - aranžira **da se zadovolji terminal read**
  - posle **minimalno 5 karaktera primljenih sa terminala**, ili
  - kad prođe **10 sec od prijema prvog**.

- Kada **dobije prekidni signal**, proces resetuje terminal u **originalno setovanje** i završava.

ioctl example

- inlcude <signal.h> #inlcude <termio.h>
- **struct** termio **savetty**;
- main()
- {
- **extern sigcatch();**
- **struct termio newtty;**
- int nrd;
- char buf[32];
- **signal(SIGINT, sigcatch)**
- **if(ioctl(0, TCGETA, &savetty) == -1)**
- { printf("ioctl failed: **not a tty**")exit(); }

- **newtty**=**savetty**;
- newtty.c_lflag &= ~ICANON /* turn off canonical mode*/
- newtty.c_lflag &= ~ECHO /* turn off echo */
- newtty.c_cc[VMIN] = 5 /* minimum 5 characters */
- newtty.c_cc[VTIME] = 100 /* 10 sec interval */

ioctl example

```
if(ioctl(0, TCSETAF, &newtty) == -1)
   {
printf("cannot put tty into raw mode")
exit();
    }

for (;;)
{
nrd = read(0, buf, sizeoff(buf));
buf[nrd] =0;
printf("read %d chars %s",nrd, buf)
    }
}
```

Terminal polling

- Ponekad je zgodno **polirati uređaj**, što znači čitati ako su podaci prisutni, a ako nisu nastaviti regularno procesiranje. Program prikazuje taj slučaj:

```
#include <fctl.h>
main
{
register int i, n;  int fd; char buf[256];
/*open terminal read-only with no-delay option*/
if((fd = open("/dev/tty", O_RDONLY | O_NDELAY)) == -1) exit();
n=1;
for(;;) /*forever*/
  {
      for(i=0; i<n; i++)
        ;
if(read(fd,buf, sizeof(buf) >0)
        {
printf("read at n%d",n)
n--;
       }
       else /*no data to read; returns due to no-delay*/
       n++;
 }
}
```

Indirect terminal driver

- Procesi često imaju potrebu da **čitaju** i **da pišu direktno** u **kontrolni terminal**, čak i kada kada su **standarni ulaz i izlaz redirektovani** u **druge datoteke**.

- Na primer, **shell script** može poslati **urgentnu poruku direktno na terminal**, mada su **fd (1 i 2) redirektovani negde drugde**.

- UNIX obezbeđuje **indirektni terminal pristup** preko **device datoteke /dev/tty** koja je označena kao kontrolni terminal za svaki proces koji ga ima.

- Useri logovani na različitim terminalima mogu pristupati **/dev/tty** ali će pristupati različitim terminalima.

Indirect terminal driver

- Postoje **2 implementacije** da **kernel nađe kontrolni terminal** **preko /dev/tty**.

- Prvo, **kernel može definisati specijalni device** broj za **indirektni terminal** file u **specijalnom ulazu** u **switch karakter tabeli**.

- Kada se **prozove indirektni terminal**, **driver uzima major i minor broj kontrolnog terminala** iz **u-area** i **poziva realni terminal drajver** preko **swith tabele.**

- **Drugo način**, **kontrolni terminal** se **nalazi tako što** se **testira** ime **/dev/tty** pre **open procedure**, a **taj test** treba da vrati **ime realnog terminala**.

logging in

- Proces **init** se izvršava u **beskonačnoj petlju**, čita /etc/inittab i izvršava linije prevodeći sistem u iz jednog stanja u drugo.
- U **multiuser modu**, init mora da omogući **userima login proceduru**.
- **Init** izvršava proces **getty** (**get terminal**) i **čuva track** koji je **getty proces otvara koji terminal.**
- **Svaki getty proces**
  - **postavlja svoju procesnu grupu preko setgrp SC**,
  - **radi open za terminalsku liniju**
  - obično **spava u open SC**,
  - dok mašina hardverski ne konektuje terminal.
  - Kada se open završi, **getty uradi exec za login proces** koji zahteva od usera da **se identifikuje sa login name i password.**
- Ako se user loguje uspešno, login obavi exec SC za shell u kome korisnik počinje da radi. Ovaj poziva shell koji se zove **login shell**.
- **Shell proces ima isti PID kao i getty** i zato je on **process leader**. Ako se user ne konektuje uspešno, login pravi exit, zatvara (close) terminalsku liniju,a **init ponovo izvršava novi getty**. Init pauzira dok ne primi "**death of child**" signal, onda se probudi i izvršava **novi getty** koji ponovo otvara isti terminal.

algorithm login

- **algorithm login /* procedure fo logging in*/**
- {
- **getty process executes**;
- **set process group** (setgrp SC);
- **open tty line**; /*sleep until opened*/
- **if(open successful)**
- {
- **exec login program**;
- **prompt** for **user name**;
- **turn off echo**, **prompt** for **password**;
- if successful)
- {
- **put tty in canonical mode (ioctl)**;
- **exec shell**
- }
- else
- count **login attempts**, try again up to a point;
- } }

Streams

- **Razne šeme za realizaciju drajvera pate** od **raznih nedostataka**. Različiti drajveri teže da **dupliciraju funkcionalnost**, **posebno drajveri koji realizuju mrežne protokole**, koji uključuju **komponentu za hardver** i komponentu za protokol. Takođe, cliste su korisne za baferske funkcije, ali su preskupe **zbog manipulacije karakter po karakter**. Učinjen je **pokušaj za povećanjem** performansi **putem uvođenja modularnosti u I/O drajvere**. Nedostaju zajedničke procedura na user-level, gde više komandi mogu da obavljaju zajedničku funkciju, ali za različite uređaje.

- **Ritchie** je impementirao šemu koja se **naziva streams** i
- koja **obezbeđuje veću modularnost** i **fleksibilnost za I/O sisteme**.

- **Stream** je **full-duplex konekcija** između **procesa** i **device drajvera**.
  - **Sastoji** se od **skupa linearno povezanih queue parova**,
  - a u **paru imamo deo za ulaz** i **deo za izlaz**.
  - Kada **proces upisuje podatke u stream**, **kernel šalje podatke** u **output queues**,
  - **kada drajver prima ulazne podatke** on ih šalje po **input queues** do **reading procesa.**

- **Queues prosleđuju poruke** sa **susednim queues** na **bazi precizno definisanog interfejsa.**

Queue pair

- **Svaki queue par** ima **pridružen kernelski modul**, kao što je:
  - **drajver**
  - **LD**
  - **protokol**
- a moduli manipulišu podacima prosleđujući ih kroz queue parove.
- **Svaki queue** je **data struktura** koja **sadrži sledeće elemente**:
  - **open proceduru** (poziva se za vreme open SC)
  - **close proceduru** (poziva se za vreme close SC)
  - **put proceduru**, (poziva se da prosledi poruku u queue)
  - **service proceduru**, (poziva se kad se queue izabere za izvršenje)
  - **pointer** na **sledeći queue u stream-u**
  - **pointer** na **listu poruka koje čekaju servis**
  - **flagove, high-water mark i low-water mark**, koriste se za **flow-contr**ol, **schedulling, održavanje stanja queue-a**
- Kernel alocira **queue parove** koji su **susedni u memoriji** tako da jedan član **queue** može **lako** da **nađe svog člana** para

streams je namenjen za karakter uređaje

- **postoji specijalno polje** u **karakter switch tabeli**
- **koje ukazuje** na **stream inicijalizacionu strukturu koja sadrži**
  - adresu rutine
  - high-water mark i low-water mark.

- Kada **kernel izvršava open SC** i **otkrije da je to karakter specijalna datoteka**, ispituje se **karakter switch tabela**, **da li je reč** o **stream drajveru ili ne**, na bazi polja za taj ulaz.

- Ako nije **stream drajver**, **kernel prati običnu proceduru** za **karakter uređaje**.

streams je namenjen za karakter uređaje

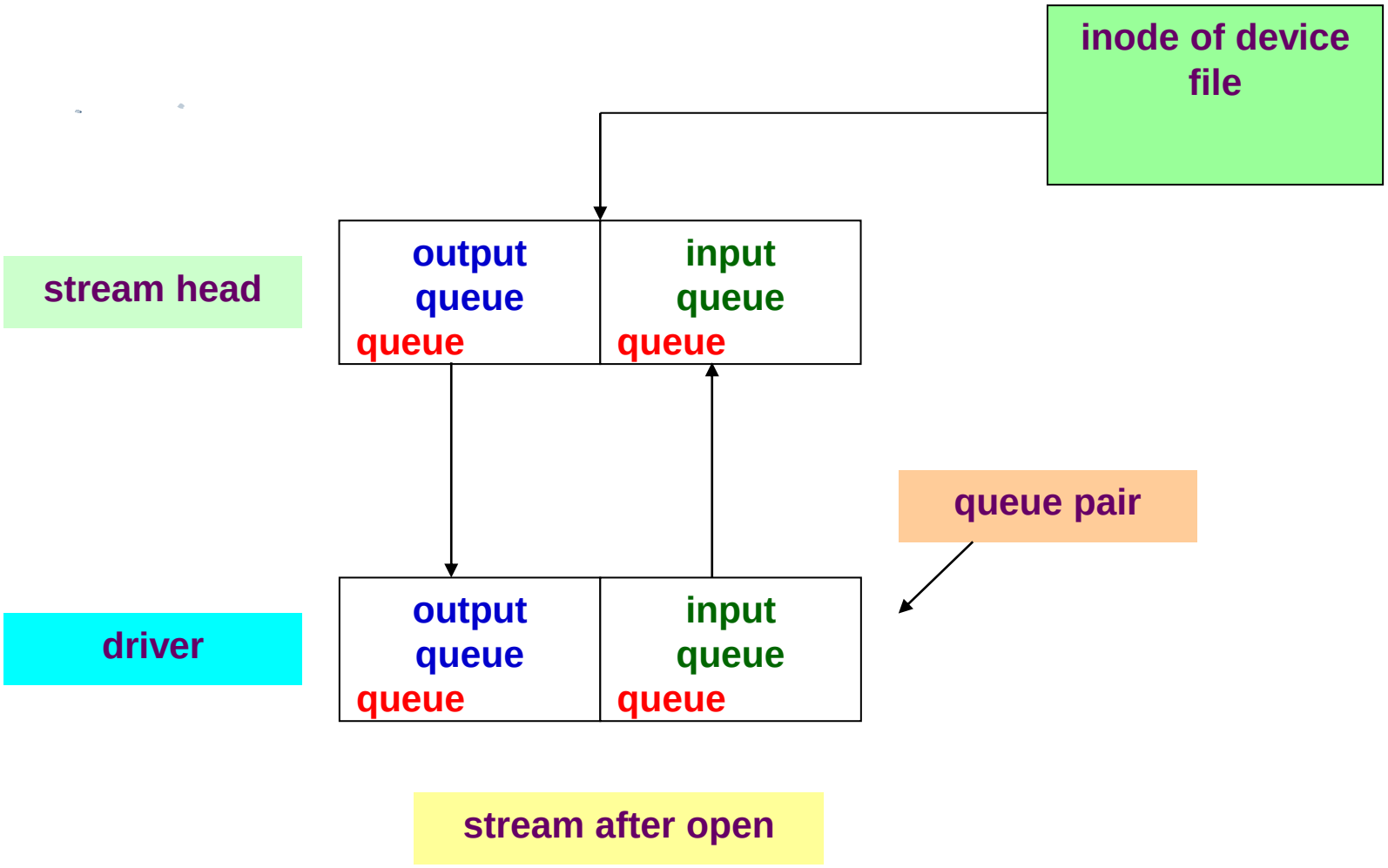
- Pri **prvom open-u za stream drajver**, **kernel alocira par queues**,
  - **prvi za stream-head**
  - **drugi za drajver**

- **Stream-head modul** je **identičan za sve streamove**:
  - imaju **put** i **service proceduru**
  - **interfejse** ka high-level **kernelskim modulima**
  - koji **implementira read, write i ioctl SC.**

- **Kernel inicijalizuje drajver queue strukturu**, **dodeljujući queue pointere**
- i **kopirajući adrese drajverskih rutuna** iz per-drajver inicijalizacionih struktura
- Potom, **poziva drajver open proceduru**. Drajver open procedura obavlja običnu inicijalizaciju ali čuva informaciju da pozove queue sa kojim je pridružen.

- **Na kraju kernel** dodeljuje specijalan pointer na **in-core inode** da ukazuje na **stream-head**. Kad drugi proces otvara uređaj, **kernel nalazi prethodno alocirani stream** preko **inode pointera** i poziva **open proceduru svih modula** u **streamu**

stream after open



stream after open

messages in the stream

- **Moduli komuniciraju prosleđivanjem poruka sa susednim modulima**.
  - Poruka se sastoji linkovanih lista MBH (**message block headers**),
  - svaki **MBH** ukazuje na **početnu i krajnu lokaciju blokova podataka**.

- Ima **2 tipa poruka**:
  - **kontrolne**
  - **data poruke**
- tip se nalazi u **identifikatoru MBH**.

- **Kontrolne poruke** su rezultat
  - **ioctl SC** ili
  - **specijalnih uslova**, kao što je teminal hang-up

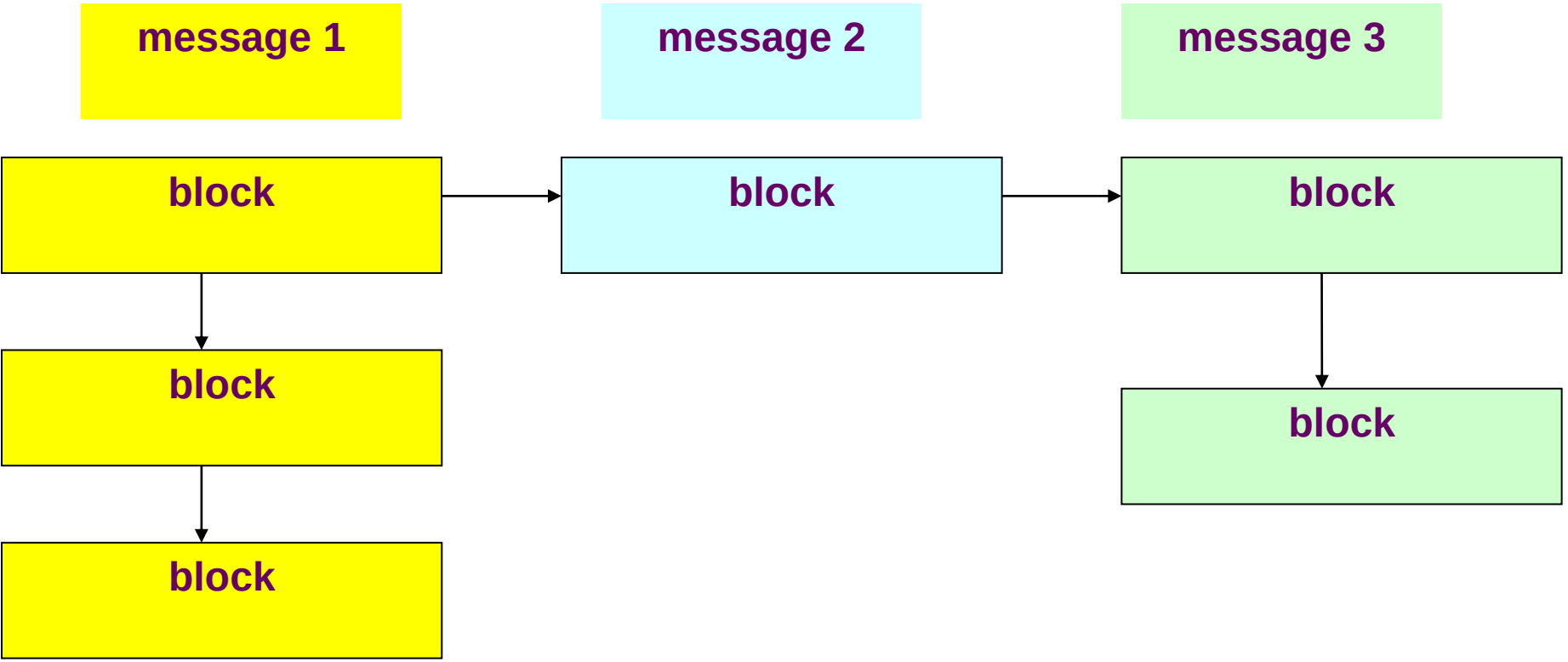
- **data poruke** su posledica **write SC** ili read (dolazak **podataka sa uređaja)**.

messages in the stream

- **Kada proces upisuje** u **stream**,
  - **kernel kopira data iz user prostora**
  - u **blokove poruke** koje su **alocirani** za **stream-head**.

- **Stream-head** modul poziva "**put**" **proceduru** od **sledećeg queue modula**, **koji može:**
  - **procesirati poruku**
  - **proslediti je direktno u sledeći queue**
  - moze je **izbaciti iz queue za kasnije procesiranje,**
    - **kada modul linkuje MBH** na **linkovanu listu**,
    - **na taj način** se **formira two-way linkovana lista**.

Stream messages



stream messages

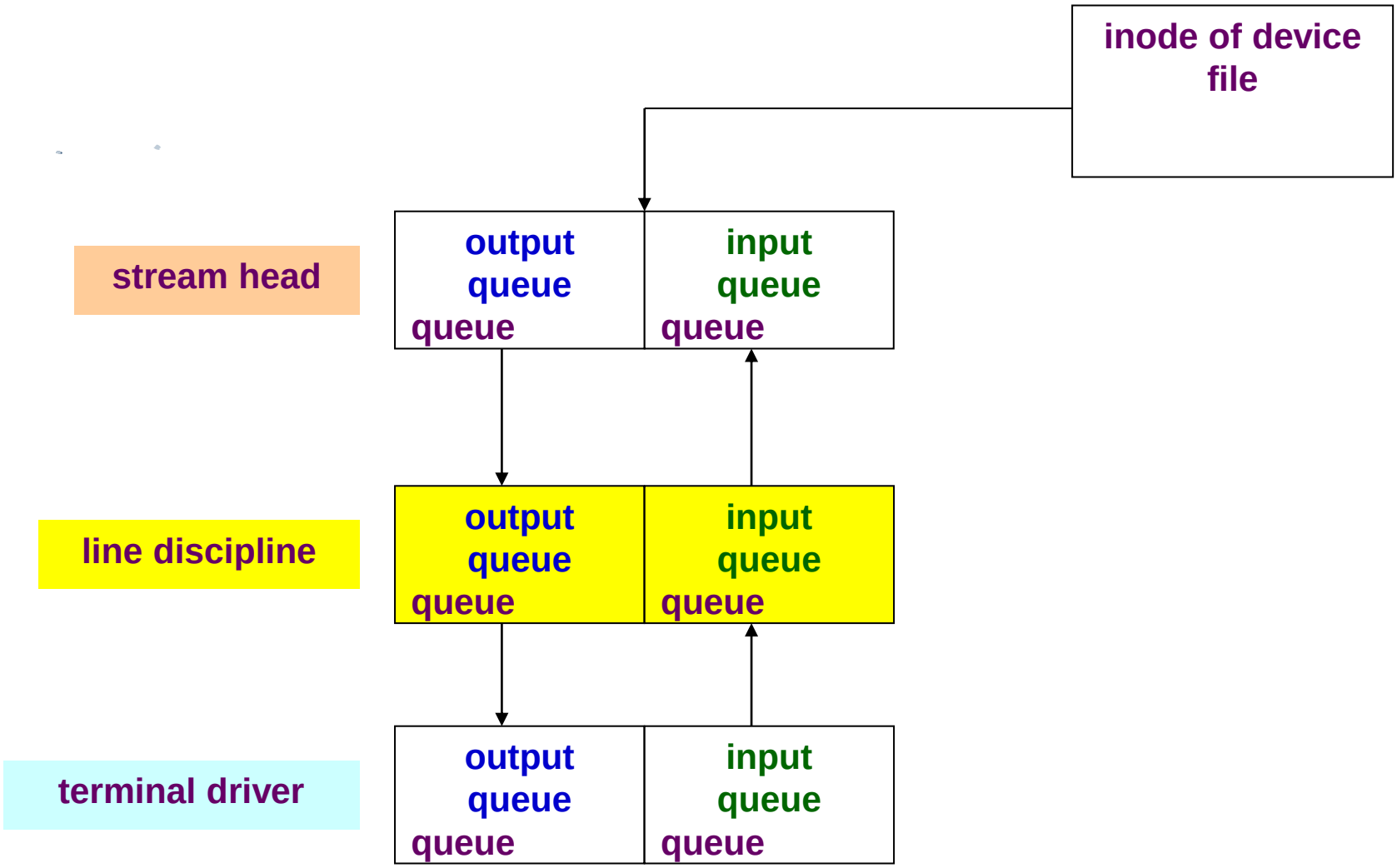
modules

- Zatim se setuje flag u **svojoj queue data strukturi** da ukazuje da ima **podatke za obradu i servisiranje**.
- **Modul plasira queue**
  - na **linkovanu listu queue-ova koji zahtevaju servis** i
  - **poziva scheduling mehanizam**,
  - taj scheduller poziva servisne procedure za svaki svaki queue u listi.
- Kernel može **izabrati modul preko softverskog prekida**, slično kao što **poziva callout tabelu**, a softverski prekidni handler poziva individualne servisne procedure.

- Proces može **gurnuti modul na otvoreni stream**, preko **ioctl SC**.
- Kernel ubacuje modul **ispod stream-head** i konektuje queue pointere da čuvaju strukturu **duple linkovane liste**.
- **Niži moduli stream-a ne obraćaju pažnju** da li **komuniciraju sa headom ili umenutim modulom** zato što je interfejs **"put" procedura sledećeg queue** na streamu, a sledeći queue postaje ubačeni modul.

- **Na primer**, proces može **ubaciti LD modul** u **terminal stream drajver** kao na slici, da obrađuje **kill** i **erase** karakter**.**

LD as module


inode of device file
stream head
output queue
queue
input queue
queue
line discipline
output queue
queue
input queue
queue
terminal driver
output queue
queue
input queue
queue

LD as module

- Na ovakan način LD nema isti interfejs kao prethodno opisana LD, ali joj je funkcija ista.
- **Bez LD modula terminal drajver ne procesira ulazne karaktere** tako da oni neizmenjeni stižu do **stream-heada**.

- Program koji **otvara terminal** i **ubacuje LD** modul može da **izgleda**:
- **fd=open ("/dev/ttyxy", O_RDWR);**
- **ioctl(fd, PUSH, TTYLD);**

- **Nema limita koliko se modula** može **ubaciti (pushed) u stream.**

- Proces **može izbaciti modul iz streama** (pop) u LIFO poretku, koristeći drugi ioctl SC
- **ioctl(fd, POP, 0);**

- LD modul se može primenjivati i na **mrežne interfejse**, umesto na terminale, tako da modul ispod nje može da bude **mrežni drajver**.

Analysis of streams

- **Moduli za steams** se realizuju kao **softverski prekidi,**
  - **ne kao posebni procesi**
  - pa ne **prolaze kroz CPU schedulling**.
- Oni su uvek u **kontekstu svog procesa**, ne mogu spavati sami za sebe, nego **mogu uspavati svoj proces**.
- **Moduli čuvaju svoje stanje interno**.

- **Neke anomalije** se mogu pojaviti i u **realizaciji streams-ova**:
  - **process accounting je dosta težak sa streams**
  - **bez streams** , **user može lako prebaciti terminal** u **raw mod** i **vratiti se brzo ako nema raspoloživih podataka**, **što nije lako realizovati** u **steams drajveru**
  - **stream su linearne konekcije** i nije lako **dozvoliti multipkesiranje u kernelu** (**mpx je mulipleksiranje na user-level**)

- Bez obzira na anomalije, **streams su moderan i kvalitetan koncept u realizaciju device drajvera.**