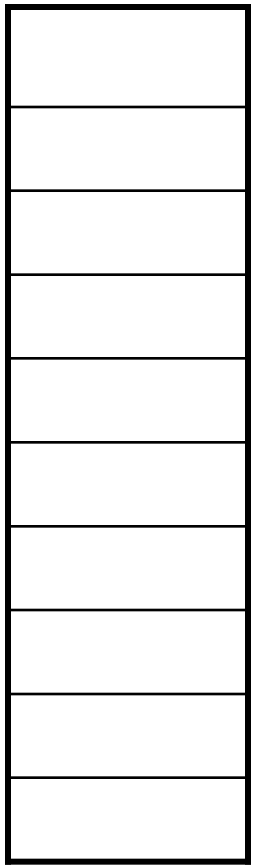
# Lesson 6-SC for FS

- pipe
- dup
- mount, umount
- link, unlink
- FS consistency

# Pipes

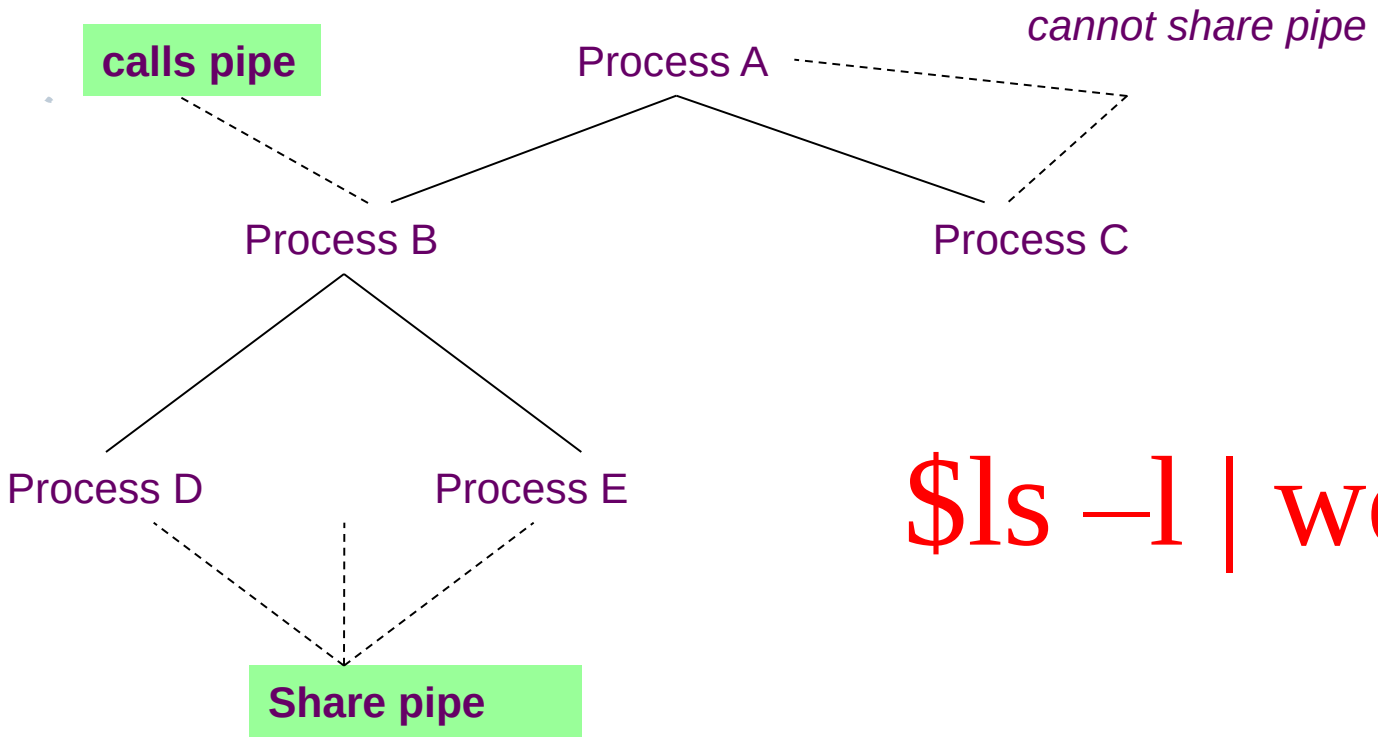


- Pipes dozvoljavaju **prenos podataka**
  - **između procesa na FIFO način**,
  - a takođe omogućavaju sinhronizaciju procesa.

- Postoje 2 vrste pipes:
  - **named pipes**
  - **unamed pipes**

- Obe vrste su identične
- osim pri inicijalnom pristupu pipe, pri čemu se
- za **named pipe** koristi **open SC kao read, write i close**,
- dok se **unamed pipes kreiraju preko pipe SC**
  - Pravila su da samo deca procesa koji je otpočeo pipe SC mogu deliti pristup unamed pipes.

# Pipes




$ls –l | wc -l

- Na ovoj slici **proces B poziva pipe SC**, a **potom kreira 2 nova procesa D i E**, tako da sva **3 procesa B, D i E mogu da dele pipe**, dok **procesi A i C ne mogu**.
- Na drugoj strani, **svi procesi** mogu pristupati **imenovanom pipe** bez obzira na njihove međusobne odnose

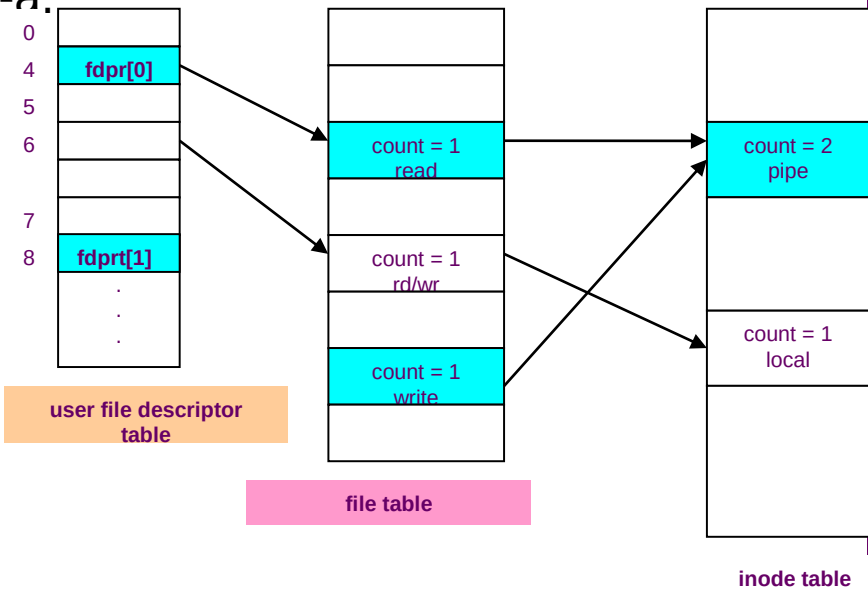
# pipe SC - syntax

- Sintaksa za pipe SC je
- **pipe(fdptr)**
- gde je **fdptr** **ukazivač** na **integer polje** koje će sadržavati 2 file deskriptora za čitanje i upis pipe-a.

- Kako su **pipes u stvari datoteke,** koje ne postoje pre korišćenja, kernel mora dodeliti **jedan inode** prilikom kreiranja pipe-a.

- Kernel dodeljuje **par file deskriptora**
  - **read descriptor** za čitanje iz pipe
  - **write descriptor** za upis u pipe

- Takođe se dodeljuje:
  - **2 FT ulaza za pipe**
  - **kod obične datoteke**, **samo jedan ulaz**

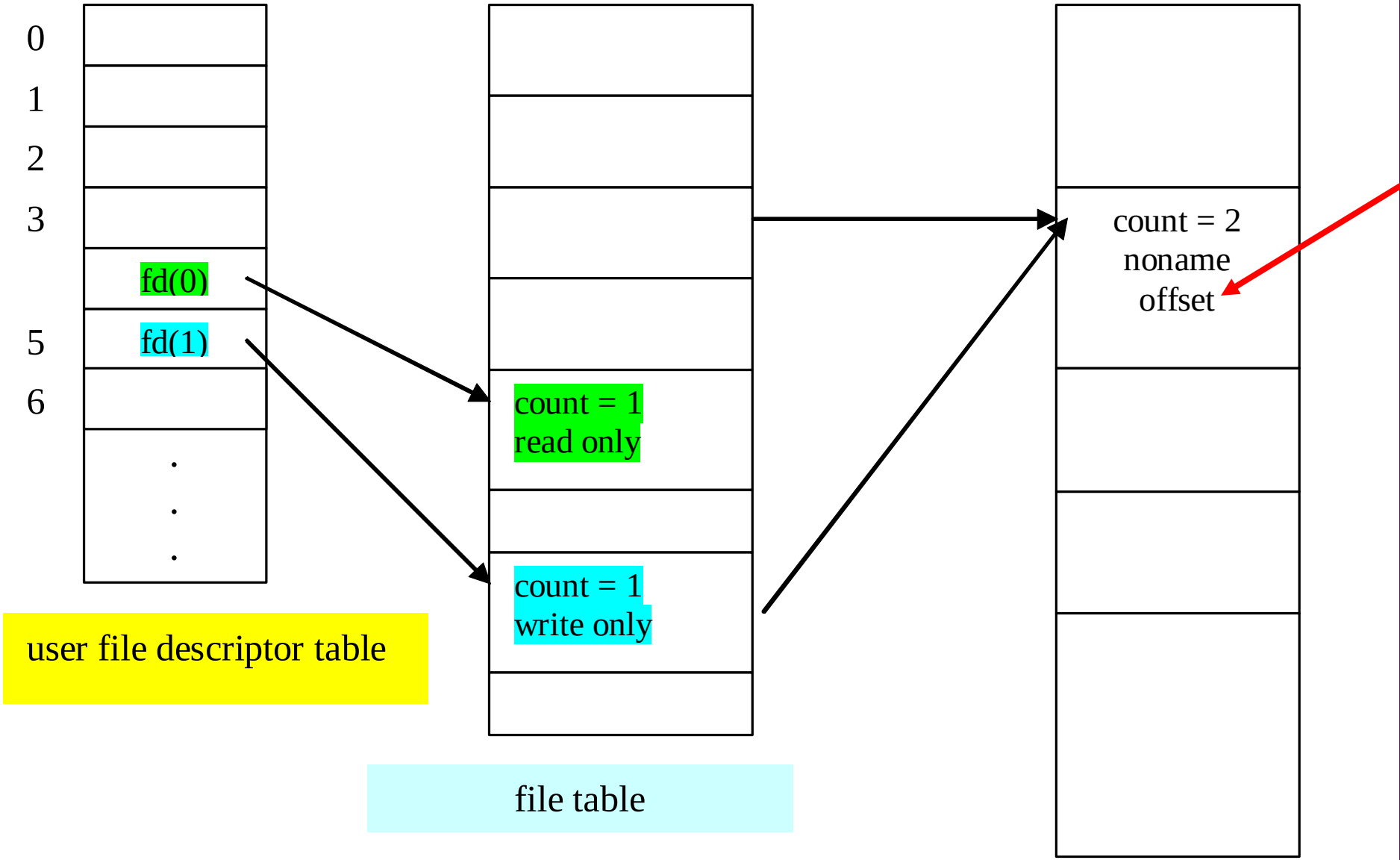



- čim se **kreira pipe**, svi SC (read, write) važe kao i za običnu datoteku.

# algorithm pipe

- **algorithm pipe**
- input: none
- output: **read file descriptor**
- **write file descriptor**
- {
- assign **new inode** for pipe device (algorithm **ialloc**);
- allocate **2 FT**: entry for reading, another for writing;
- initialize FT entries to point to new inode;
- allocate **2 UDFT**:
  - for reading and for writing,
  - initialize to point to respective FT entries;
- set inode **RC** to 2;
- initialize reference count RC of inode readers, writers to 1;
- }

# pipe


0
1
2
3
fd(0)
5
fd(1)
6
count = 1
read only
count = 1
write only
count = 2
noname
offset
user file descriptor table
file table
inode table

# pipe - discussion

- **Kernel dodeljuje inode** za **pipe device** preko algoritma **ialloc**.
- **Pipe device** je onaj FS u kome se dodeljuje inode i data blocks za taj pipe.
- Kada je **pipe aktivan**, kernel **ne obavlja reassign pipe-ovog inoda** i **data blokova**.
- Kernel zatim dodeljuje **2 FT ulaza**: za read i write descriptor i ažurira informacije u in-core inodu.
- Svaki **FT ulaz** čuva informaciju koliko instanci pipe-a je otvoreno za čitanje i upis, a inicijalno se postavljaju na 1.
- U **in-core inodu**, RC pokazuje koliko puta je pipe otvoren a **inicijalno je 2**.
- Takođe **inode** čuva informaciju o **bajt offsetu** gde će sledeći upis ili čitanje da se događa kroz pipe.
- Tu je **osnovna razlika između pipe i obične datoteke,**
  - kod **pipe** je **offset u in-core inodu**,
  - kod **obične datoteke je offset u FT ulazu**.
- **Proces ne može** sam da **podešava offset** kao kod obične datoteke,
  - **lseek** ne važi za **pipe.**
  - **nema radnom pristupa** kroz pipe, isključivo sekvencijalno=FIFO.

# Openining a named pipe

- Imenovani pipe je datoteka čija je semantika potpuna ista kao kod unamed pipes, osim što **ima FCB** u direktorijumu i pristupa joj se po **pathname**.
- **named pipes** se
  - otvaraju kao obične datoteke
  - permanetno postoje u FS
  - brišu se sa unlink SC
- **unamed pipes tranzijentni**,
  - kada svi procesi završe sa korišćenjem pipe-a,
  - kernel oslobađa inode.

- Algoritam za **open** of **named pipe**-a je skoro isti kao kod obične datoteke.
- Međutim, pre završetka open SC, kernel inkremetira read i write count u inodu, ukazujući **koliko procesa** je **otvorilo named pipe** za **čitanje** i **upis**.
- Proces koji otvori named pipe za čitanje ostaje uspavan sve dok drugi proces ne otvori pipe za upis i obrnuto.
- Kernel će probuditi proces koji uspavan na pipe-u, kada se dogodi neki događaj upis ili čitanje.

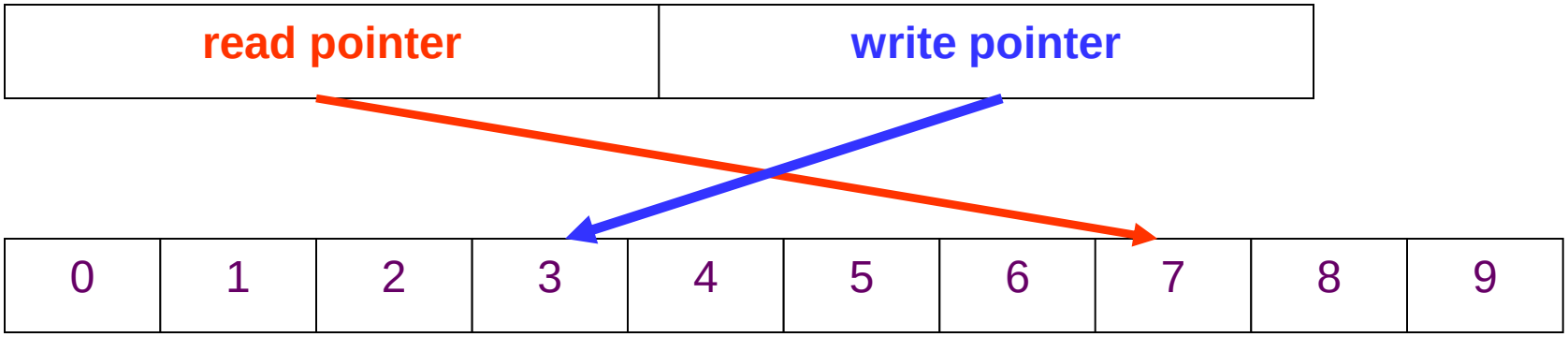
# Reading and Writing Pipes

- Proces pristupa podacima u pipe na FIFO način, koji znači da podaci moraju da se čitaju o **onom poretku** u kome su se **upisali.** Nije neophodno da broj čitaoca bude jednak broju pisaca u pipe, ako je broj pisaca i čitaoca veći od 1, mora se obezbediti sinhronizacija.
- Kernel pristupa podacima u pipe kao kod obične datoteke, blokovi se dodeljuju i pune u toku write SC. Razliku u alokaciji prostora između pipe i obične datoteke što **pipe koristi isključivo direktne pointere**, što povećava performanse ali ograničava veličinu pipes.

- Kernel manipuliše **direktnim blokovima** pipe-a kao sa **kružnim queue**, kontrolišući interne read i write ukazivače da poštuju FIFO poredak.

- Postoje sledeća **4 slučaja čitanja i upisa u pipe**:
  - I **upis** u pipe kada **ima mesta** za upis
  - II **čitanje** iz pipe kada se u pipe **nalaze svi potrebni podaci** koje čitaoc zahteva
  - III **čitanje** iz pipe kada **nema svih traženih podataka**
  - IV **upis** u pipe kada **nema mesta** za upis

# Reading and Writing Pipes



# Reading and Writing Pipes

- **WRITE**
- Prvi slučaj nastupa kada proces upisuje u pipe tako da je suma podataka koji se upisuje + količina podataka koja je već u pipe mora da bude manja od kapaciteta pipe-a.
- **Qnew_data+ Qexisting_data < PIPE_capacity**
- Kernel, tada upisuje podatke kao kod obične datoteke
  - za svaki takav upis **inkremetira se pipe size** i **write pointer**
  - **budi sve read procese** koji čekaju da se **nešto upiše u pipe**.
  - Kada se dođe do **kraj pipe**, iscrpe se **svi direktni pointeri**, ide se **na početak** pipe-a (byte offset 0).
  - Upis je uvek u **rastućem redu**.
- Kod **obične datoteke** veličina ažurira samo ako se upis dešava iza kraja datoteke.
- **READ**
- Kada proces čita pipe, **proverava** se da li je **pipe prazan** ili **nije**.
- Ako nije, read se dešava ali poštujući strogo offset koji je u inodu of pipe.
- **Posle čitanja** svakog bloka,
  - kernel **dekremetira veličinu pipe-a**,
  - **ažurira read pointer** (read offset) i
  - **budi eventualne writer procese** koji čekaju na prazno mesto.

# Reading and Writing Pipes

- Proces koji čita pipe **će se uspavati** ako je **pipe prazan** ili **nema dovoljno podataka**. Postoje i **no delay** opcije koje odmah vraćaju bez čekanja ako **read** ne može da se zadovolji.
- Ako proces želi da nešto upiše u pipe, **a nema mesta**, mora da ide na spavanje dok **reader proces ne isprazni** pipe.
- Ukoliko proces upisuju podatke koji **prevazilaze kapacitet pipe-a**,
  - tada proces ne može da čeka jer **nikada neće dočekati**,
  - već upisuje **samo onaj deo koji može da stane**,
  - u **najboljem slučaju** kapacitet pipe-a,
  - pa čeka da se potpuno ili delimični isprazni.
- Još jedanput, **pipe**
  - **liči na regularnu datoteku**
  - ali **read i write offset** nisu kontrolabilni **od strane procesa**
  - već se **kontrolišu pomoću kernela** i **nalaze se u inode a ne u FT**
  - to su **offseti zajednički za sve procese koji dele** pipe.

# Closing Pipes

- Kada se zatvara (**close**) pipe,
  - proces obavlja sličnu proceduru kao **kod obicne datoteke**,
  - ali kernel mora obaviti **specijalnu proceduru prilikom zatvaranja pipe**, odnosno otpuštanja pipe-ovog inoda.
- Kernel **dekrementira** broj pipe-ovih reader-a i writer-a na svako zatvaranje.
- Ako broj **writera padne na 0**, a **ima uspavanih readera**, kernel in budi i vraća im read SC bez pročitanih podataka.
- Ako **broj readera padne na 0**, a **ima uspavanih writera**, kernel ih budi i salje im signal greške.
- U oba slučaja čekalo bi se na nešto **što nije sigurno da će se dogoditi**,
  - a u slučaju **named pipe** mogao bi se pojaviti novi proces
  - ali ih kernel i named i unamed pipe tretira na isti način.
- Ako **nema ni reader ni writer procesa**,
  - kernel **oslobađa sve data blokove od pipe-a**
  - markira inode da je **pipe prazan**.

# pipe - examples

- Ovaj program ilustruje veštačko korišćenje pipes.
  - Proces kreira pipe i ide u beskonačnu petlju,
  - upisuje string „hello“ u pipe i čita ga iz pipe.
  - Kernel ne kontroliše stanje ako isti proces čita ili piše pipe.

- char string [] = “hello”
- main
- {
- char buf[1024];
- char *cp1, cp2*;
- int fds[2];
- cp1= string;
- cp2=buf;
- while (*cp1) ***cp2++ = *cp1++;  /* popuna bafera "buf" sa stringom hello */**
- **pipe(fds);**
- for (;;)
- {
- **write**(fds[1], buf, 6);
- **read(**fds[0], buf, 6);
- }
- }

# Example for named pipe

```
#include <fcntl.h>
char string [] = “hello”
main(argc, argv)
int argc;
char *argv[]
{
int fd;
char buf[256]
/*create named pipe with read/write permission for all users*/
mknod (“fifo”, 010777,0)

if (argc == 2) fd = open (“fifo”, O_WRONLY);
else           fd = open (“fifo”, O_RDONLY);

for (;;)
  {
        if (argc == 2) write(fd,string,6);
        else           read(fd,buf,6);
  }
}
```

# Example for named pipe

- Proces kreira pipe, a zatim
- ako je pozvan **sa 2 argumenta** onda **kontinualno upisuje string hello** u pipe fifo.

- Ako se pozove bez drugog argumenta on kontinualno čita **named pipe**.

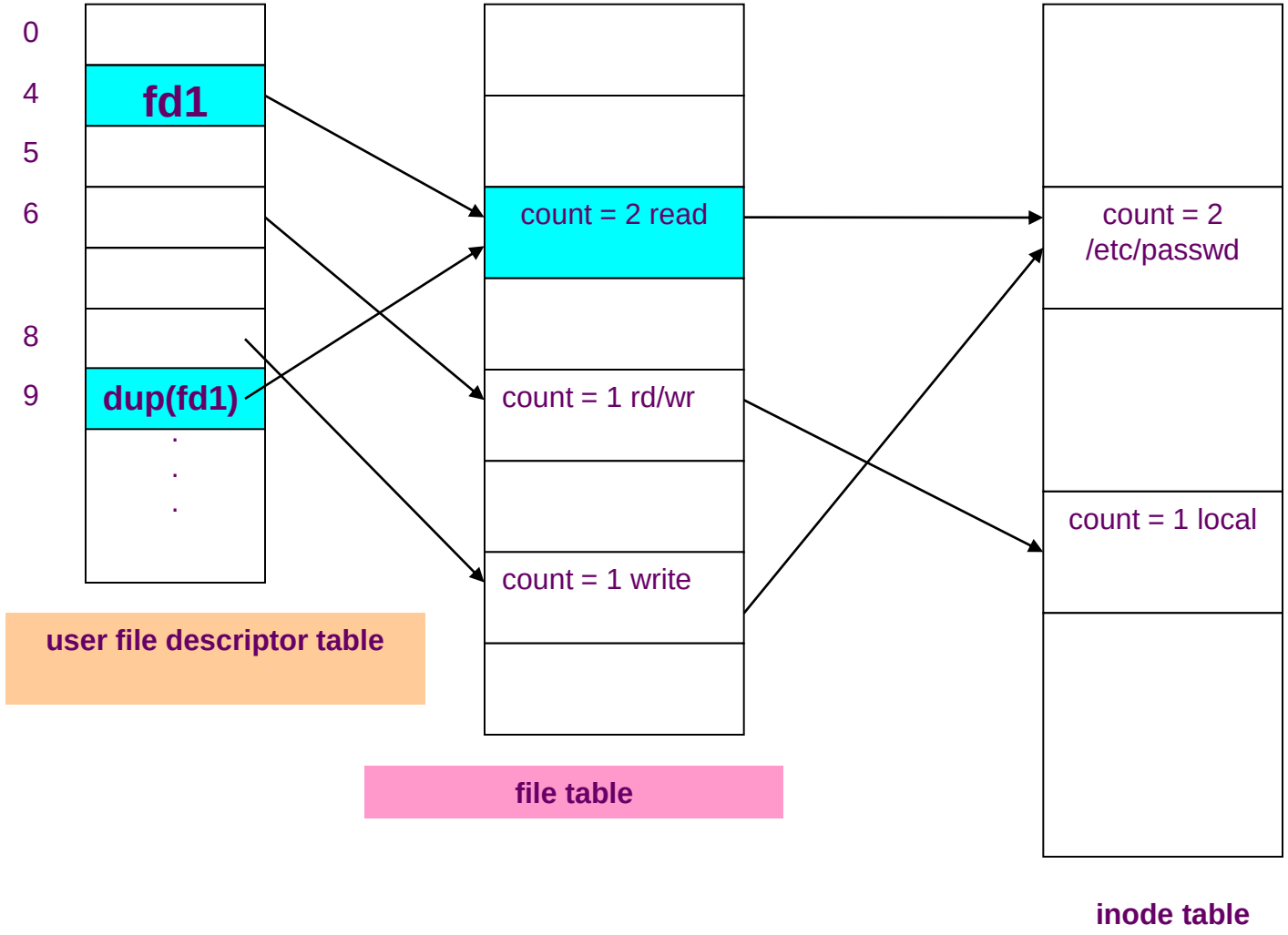
- Ako se od ovog programa kreiraju 2 procesa
- oni će komunicirati kroz isti pipe fifo,
- a može se više procesa uključiti na isti način.

# DUP

- **SC dup**
  - **kopira file descriptor**
  - u **prvi slobodan slot u UFDT**,
  - pri čemu **vraća novi descriptor korisniku**.

- Ovaj SC radi za sve tipove datoteka i ima sledeću sintaksu
- **newfd = dup(fd);**
- pri čemu je **fd descriptor datoteke** koji se **duplicira**,
- **newfd** je **novi descritor datoteke**.

- S obzirom da **dup duplicira file descriptor**,
- **dup** će **inkrementirati count odgovarajućeg FT ulaza**,
- **FT ulaz** sada ima **još jedan UFDT ulaz** koji na taj FT ulaz ukazuje




- Ovde smo imali sitaciju **UFDT(1)** je otvorio datoteku **/etc/passwd fd 1**, **UFDT(3) fd3** je otvorio **local**, **UFDT(5) fd5** je ponovo otvorio **/etc/passwd**. Potom je obavljen **newfd=dup(fd1)** i upisan u **slot 6.**
- SC dup ne izgleda elegantan na prvi pogled, ali je zgodan za razne sofisticirane zadatke u programiranju kao što su shell pipeline

# dup - example

- Posmatrajmo sledeći program
- #include <fcntl.h>
- main()
- {
- int i,j;  char buf1[512], buf2[512];
- **i = open(”/etc/passwd”, O_RDONLY)**
- **j = dup (i);**
- **read** (i, buf1, sizeof(buf1));
- **read** (j, buf2, sizeof(buf2));
- **close(i);**
- **read** (j, buf2, sizeof(buf2));
- }
- Najpre je urađen **open SC** na **/etc/passwd**, dobijen je **fd=i** kreiran jedan ulaz sa njim u UFDT, kreiran ulaz u FT sa offsetom 0.
- Sa dup je kreiran još jedan **fd=j**, još jedan slot u UFDT koji ukazuje na isti FT ulaz to znači sa istim offsetom.
- Prva 2 read SC  će pročitati prva dva disk bloka datoteke, a onda se  zatvori jedan fd a to je **fd=i**, ali se datoteka ne zatvara već samo jedan ulaz u **UFDT nestaje**, ali tu je drugi **fd=j** sa kojim se nastavlja čitanje iz datoteke. Ovo može da se radi i **sa standardnim file descriptorima 0, 1, 2**, mogu se duplicirati, zatvarati itd.

# Mounting and Umounting FS

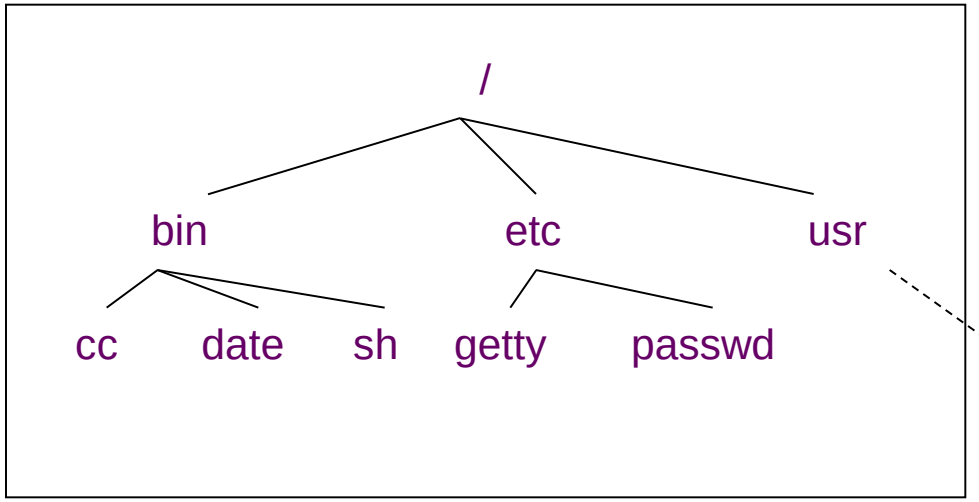
- Fizička disk jedinica se sastoji od više logičkih sekcija, i svaka sekcija ima svoje ime (**device file name**).
- Proces može pristupati podacima u sekciji kao sa datotekom, prvo se otvori device filename, a potom se disk sekcija tretira kao datoteka, po njoj se čita i piše.
- SC mount povezuje FS koji se nalazi u odgovarjućoj sekciji na disku u stablo, dok SC umount izbacuje FS iz stabla.
- SC mount omogućava korisnicima da pristupaju sekciji diska kao FS a ne kao sekvenci disk blokova.

- **Sintaksa** za mount SC je:
- **mount(special pathname, directory pathname, options)**
- pri čemu je:
  - **special pathname** ime specijalne datoteke koja se refencira na deo diska koji sadrži FS koji će biti mountovan
  - **directory pathname** je direktorijum u postojećem stablu gde će FS biti mountovan (**mount-point**)
  - **option** pokazuju kako će FS biti mountovan (read-only će blokirati svaki creat ili write SC u njemu

# Mouinting and Umounting FS

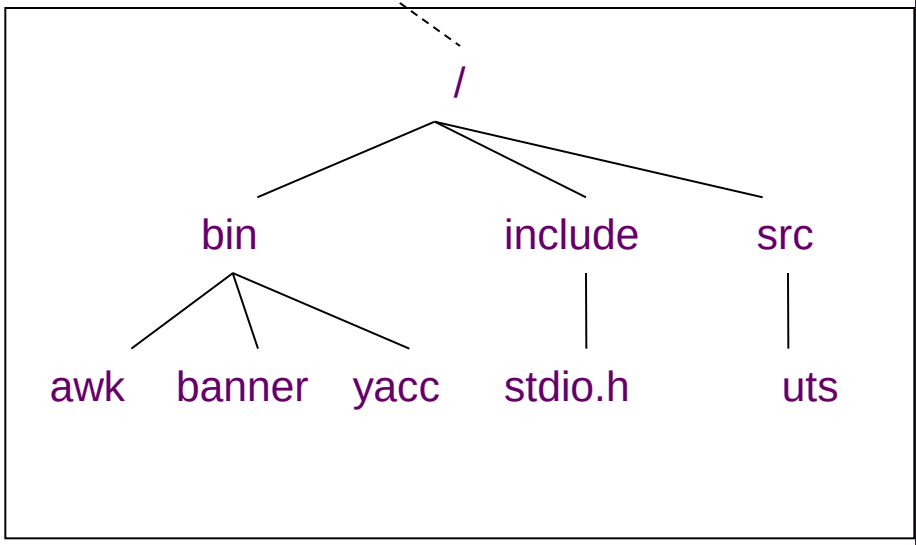
- Na primer ako proces pošalje sledeći SC
- **mount("/dev/dsk1", "/usr", 0)**
- kernel će priključiti FS koji se nalazi u delu diska koji se naziva **/dev/dsk1** ka direktorijum **/usr** u postojećem UNIX stablu.
- Datoteka **/dev/dsk1** je blok specijalna datoteka i po pravilu predstavlja deo diska, a da bi taj deo diska mogao da se mount-uje, on mora da sadrži FS, odnosno da sadrži **superblock**, **inode listu** i **root inode**.
- Kada se **kompletira mount SC**, root direktorijum mountovanog FS je raspoloživ preko **MPD** direktorijuma (**/usr** u ovom slučaju).
- Proces pristupa datotekama na mountovanom FS sasvim normalno bez obzira što su one uklonjive. Jedino **link SC** kontroliše tu činjenicu j**er ne dozvoljava** linkovanje između 2 FS.

# mount("/dev/dsk1", "/usr", 0)




root file system

/dev/dsk1 file system



# Mounting and Umounting FS

- Kernel održava **mount tabelu** (MT) sa ulazom za svaki mountovani FS.

- Svaki ulaz u **mount tabeli** sadrži:
  - **device number** koji identifikuje **mountovani FS**
  - **ukazivač** na **bafer** koji sadrži **superblock** za taj FS
  - **ukazivač** na **root inode mountovanog FS (/** of **/dev/dsk1**)
  - **ukazivač** na **inode of MPD** (usr of root FS)

- Asocijacija **MPD inoda** i **root inoda mountovanog FS**, dozvoljava kernelu da prostiru (traverse) FS vrlo efikasno.

# algorithm mount

- **input**:
  - file name of block special file
  - MPD
  - options
- **output**: none
- {
- if(not superuser) **return (error);**
- **get inode** of **block special file** (algorithm **namei**);
- make legality check;
- **get inode** of **MPD** (algorithm **namei**);
- **if(not directory**, or **reference count > 1**)
- {
- release inodes (algorithm **iput**); **return (error**);
- }
- **find empty slot** in **MT;**
- **invoke** block device **driver open routine**;
- get free buffer from buffer cache;
- read **superblock** into free buffer;
- initialize **superblock** fields;

**in-core superblock**

- get **root inode** of mounted device (algorithm **iget**), save in **mount table**;
- **mark inode** of **MPD** as **mount point**;
- release special file inode (algorithm iput);
- unlock inode of MPD;
- }

**root inode = MPD inode**

# algorithm mount

- Kernel dozvoljava samo **procesu sa superuser privilegijama** da obave mount i umount SC, kako se ne bi dozvoljavao haos u sistemu.
- Kernel pronalazi inode block specijalne datoteke koja predstavlja FS za mounting, izvlači major i minor number, pomoću kojih se identifikuje sekcija na disku koja sadrži taj FS i nalazi inode za MPD.
- **RC za taj MPD ne sme da bude veći od jedan**, jer se ne sme mountovati više FS na isti MPD.
- Kernel zatim alocira slobodan slot u **MT**, markira slot da je zauzet i dodeljuje device number.
- Potom se otvara node, pozivajući **open** proceduru za block-special node, koja proverava da li je node legalan, inicijalizuje driver data strukture i šalje komande hardveru preko disk drajvera.
- Kernel alocira free blok u bafer kešu (**getblk**) u koji će se čitati superblok i preko read algortima čita superblok u njega. Kernel memoriše pionter na inode MPD, čime se praktično obezbeđuje informacija na roditeljsku granu (..).
- Kernel pronalazi root inode FS koji će biti mountovan, i memoriše ga u MT tabeli. Za korisnika, **MPD i root of FS** su **logički ekvivalentni** i kernel uspostavlja njihovu ekvivalenciju u **istom ulazu u MT**.
- Iza mount-a procesi više ne pristupaju inodu of MPD.

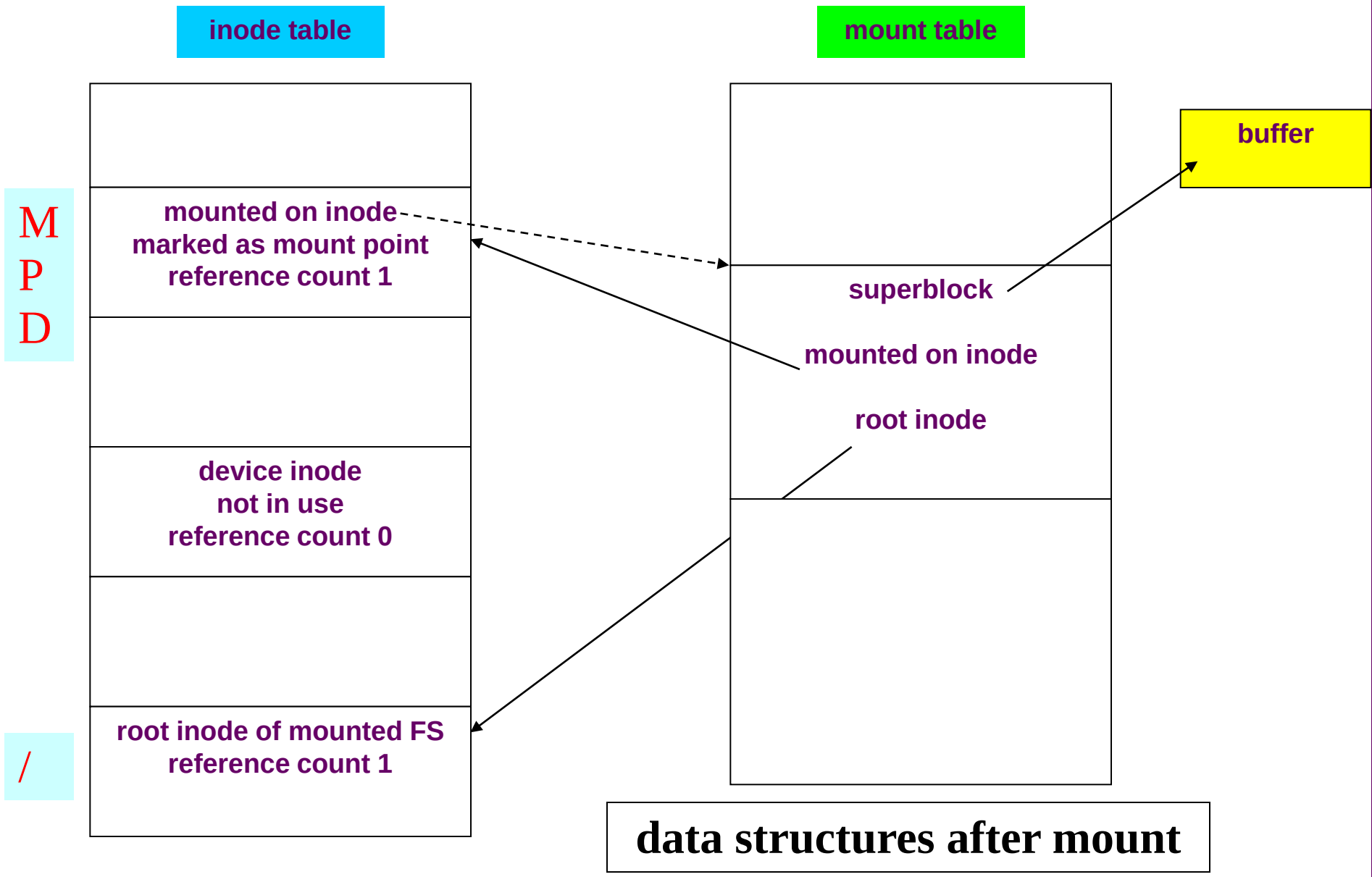
# algorithm mount

- Kernel inicijalizuje polja u superblocku, **brišući lock** za free block listu i free inode listu i postavlja broj free inodova na 0. To se radi da bi se sprečilo oštećenje FS prilikom mountovanja posle sistem crash-a, što će naterati kernel da pretraži slobodne inodove sa ialloc. Nažalost ako su povezane liste free blokova oštećene, kernel ne popravlja, to se postiže sa fsck.

- Ukoliko mount sadrži **read-only opciju**, kernel će postaviti flag u superblock-u.

- Na kraju, kernel će markirati **inode(MPD)** kao **mountpoint**, tako da svi procesi mogu da to identifikuju, što je ilustrovano na sledećoj slici.

# algorithm mount




data structures after mount

# Crossing mount points in file path names

- Razmatrajmo šta se dešava ako pathname prolazi preko MPD, kako se ponašaju **namei** i **iget** algoritmi. Postoje 2 slučaja **prelaska preko mount pointa**:
  - prolazak sa mounted-on na mounted FS (u dubinu)
  - prolazak sa mounted FS na mounted-on (iz dubine)
- To se sledeće 2 sekvence nakon komande
- **mount /dev/dsk1 /usr**
- **cd /usr/src/ust**
- **cd ../../..**
  - **Prva komanda** obavlja mount SC koji mountuje FS(/dev/dsk1) na MPD(/usr).
  - **Prva cd komanda** obavlja chdir SC koji na svom putu prolazi preko mountpoint-a /usr.
  - **Druga cd komanda** ide na 3 roditeljske grane opet prolazeći mountpoint /usr.
- U prvom slučaju, prolazak sa mounted-on na mounted FS (u dubinu), imamo **modifikovani iget algoritam**, koji je sličan kao osnovni algoritam **iget**, osim što proverava **da li je inode mountpoint**, koji se označava kao **mounted-on** ako je to MPD.
- Za svaki takav inode se pronalazi ulaz u MT, iz koga čita device number i root inode, na osnovu kojih kojih može pristupiti root inode mountovanog FS i vratiti sadržaj tog inoda.
- U prvoj cd komandi, kernel pristupa **/usr** inodu u **mounted-on** FS i detektuje da je to **mountpoint**, pronalazi se iz MT **device number** i **root inode**.

# Case #1: modified iget

- **algorithm iget**
  - input: file system and inode number
  - output: locked inode {

```
while(not done)
{
if(inode in inode cache)          /* if #1*/
{
        if(inode locked)          /* if #2*/
        {
                sleep (event inode becomes unlocked);
                continue; /* loop back to while*/
        }/* if #2*/
        /*special processing for mount point directory*/
        if(inode a mount point)
        {
        find mount table entry for mount point;
        get new FS number from MT;
        use root inode number in search;
        continue
        }
        if(inode on inode free list)  remove from free list;
        increment inode reference count;
        return (inode);
  }/*if #1*/
```

# Case #1: modified iget

- /* **inode not in inode cache***/
- if(no inodes on free list) return(error)
- 
- remove new inode from free list;
- 
- reset  inode number in free list;
- remove inode from old hash queue, place on new one;
- read inode from disk(algorithm bread)
- initialize inode (eg. reference count to 1)
- return(inode)

}/*while*/

- }/*main*/ .

# Case #2: modified namei

- Za drugi slučaj posmatajmo modifikovani **namei** algoritam
- **algorithm namei** /* convert path name to inode*/
- input: path name
- output: locked inode
- {
- if (**path name starts from root**)
- **working inode = root inode** (algorithm **iget**);
- else
- **working inode = current directory inode** (algorithm **iget**);

- while (there is more path name)
- {
- read next path name component from input;
- verify that **working inode** is of directory, access permissions OK;
- if (working inode is of root and component is “..”) continue; /*loop back to while*/
- read directory (working inode) by repeated use of algorithms bmap, bread and brelse

# Case #2: modified namei

if (component matches an entry in directory (**working inode**))

- {
- **get inode number for matched component;**
- **if(found inode of root and working inode is root and component name is ..)**
- **/*crossing mount point*/**
- **{**
- **get MT entry for working inode;**
- **release working inode (algorithm iput);**
- **working inode = mounted on inode; (MPD inode)**
- **lock MPD inode;**
- **increment reference count of working inode;**
- **go to component search for ..;**
- **}**
- release working inode (algorithm **iput**);
- working inode = inode of matched component (algorithm **iget**)
- }
- else
- return (no inode);
- }/* while */
- return(working inode);
- }

# Case #2: modified namei

- Kernel za svaki nađeni inode proverava
- da li je root inode i
- ako je working inode root  i
- ako je path komponenta je .., tada je to prolazak kroz mountpoint.
- Tada se working inode zamenjuje sa inode(MPD) koji se čita iz MT.

- Praktično imaš 2 inode za svaki **MPD** direktorijum i kada ga **namei** prolazi, mora doći do zamene **working inoda**.

# Unmounting FS

- Sintaksa za umount SC je:

- **umount(special filename)**

- gde special filename ukazuje na FS koga treba otkačiti.

- Kernel mora prvo proveriti da **nema otvorenih datoteka** i **DW blokova** u kešu da taj u FS koji će se otkačiti.

- Pretražuje se FT za sve potencijalne datoteke koje imaju **FS number**.

# Unmounting FS

- **algorithm umount**
- input: file name of block special file
- output: none
- {

- if(not superuser) **return (error);**

- get **inode** of **block special file** (algorithm **namei**);

- extract major, minor number of FS;

- ger mount table entry, based on major and minor number;

- release special file inode (algorithm iput);

- remove shared text entries from region tables for files belonging FS;

- update superblock, inodes, flush buffer;

# Unmounting FS

- if (files from FS still in use) **return (error);**
- get root inode of mounted FS from MT;
- lock inode;
- release inode (algorithm iput); /*iget was in mount*/
- invoke **close routine** for special device;
- invalidate buffers in pool from unmounted FS;

**release root inode of mounted FS**

- get inode of MPD from MT;
- lock inode;
- clear flag marking it as mountpoint;
- release inode (algorithm iput); /*iget was in mount*/

**release MPD**

- free buffer used for super block;
- free mount table slot;
- }

# Unmounting FS

- Kernel uzima inode za FS,
- ažurira
- superblock,
- inodes i flush buffer,
- oslobađa njegov root inode, zatvara device.

- Potom uzima MPD, briše mu mount point flag, oslobađa ga.

- Na kraju briše ulaz u MT i bafer za superblock.

# link

- Link SC linkuje datoteku sa novim imenom u stablu, tako što
- kreira novi FCB
- sa postojećim inode.

- Sintaksa za link SC je:

- **link(source file name, target file name)**

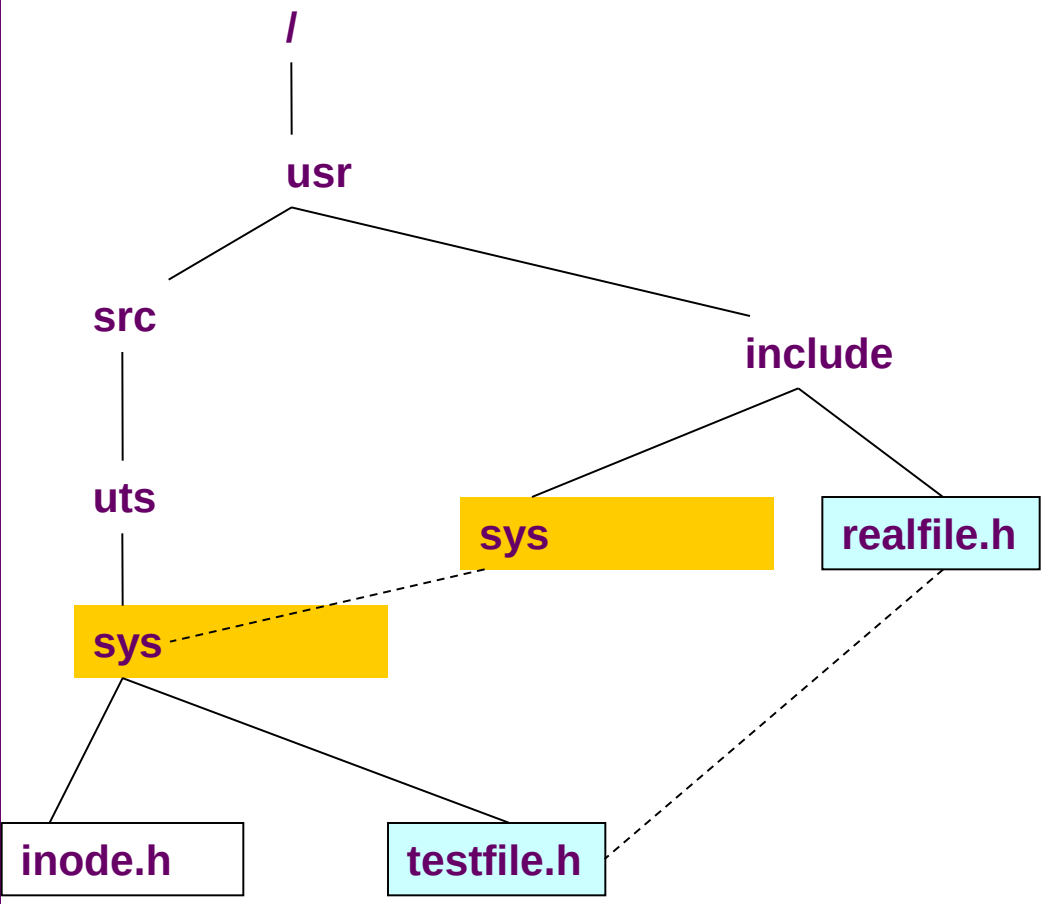
- gde je source file name ime postojeće datoteke, a target file name nova datoteka koja će se kreirati ako link SC uspe.

- FS sadrži path imena za sve linkove koje datoteka ima, i proces može pristupati datoteci po bilo kom **path name**.

- **Kernel ne zna koja je prva odnosno originalna datoteka, tako da su sva imena ravnopravna**.

# link

- Na primer posle izvršenja 2 SC, dogodiće se situacija kao na slici:
- **link** ("**/usr/src/uts/sys**", "**/usr/inlcude/sys**")
- **link** ("**/usr/inlcude/realife.h**","**/usr/src/uts/sys/testfile.h**")




3 path imena ukazuju na istu datoteku:
**/usr/include/realife.h**
**/usr/src/uts/sys/testfile.h**
**/usr/include/sys/testfile.h**

# link

- Kernel dozvoljava samo superuser,
- zbog mogućnosti beskonačnih petlji kada korisnik linkuje direktorijume koji su na istoj grani. a ide se unazad.
- Za superuser-a se pretpostavlja da je oprezniji.

- Potreba za linkovanjem direktorijuma je bila na starim UNIX sistemima,
- zato što mkdir komanda radila preko link SC,
- ali je sve to razrešeno preko uvođenjem mkdir SC,
- čime više **nema potrebe da se linkuju direktorijumi**.

# link - algorithm

- **algorithm link**
- input: {existing file name, new file name}
- output: none
- {
- get inode for existing file name (algorithm **namei**);

- if (too many links on file or linking directory without super user permission)
- {
- release inode (algorithm **iput**);
- **return (error);**
- }
-
- **increment link count on inode;**
- update disk copy of inode;
- unlock inode;

LC++

# link - algorithm

- get **parent inode** for **directory** to contain new file name (algorithm **namei**);
- if (new file name already exists or existing/new file name on different FS)
- {
- **undo update done above**;
- **return (error);**
- }
- **create new FCB-directory entry** in **parent directory** of **new file name**:
- **FCB** include **new file name**, **but inode number** of **existing file**
- 

new file, new FCB, existing inode

- **release parent directory inode** (algorithm **iput**);
- **release inode** of **existing file** (algorithm **iput**);
- }

# link - algorithm

- Kernel prvo **locira inode originalne datoteke**, preko algoritma **namei**,
- inkrementira **link count**,
- ažurira disk kopiju inoda (iz razloga konzistencije),
- a na kraju unlock-uje inode.

- Zatim se traži **target file**,
- **ako target file postoji** link otkazuje, link count se dekrementira jer je prethodno inkremetiran.
- **Ako ne postoji**, dodeli novi FCB slot u target direktorijumu,
- upisuje se filename i inode broj originalne datoteke,
- oslobađa se **inode parent direktorijuma** za target file preko iput.
- Potom se **oslobađa inode postojeće datoteke** čiji link count uvećan za 1.

# deadlock - possibility

- Prilikom linkovanja datoteka **mogući su deadlocks**.
- Obradićemo 2 moguće situacije za deadlock i oba potiču zbog ne-oslobađanja inoda nakon inkrementiranja link count.
- **Pažnja: uvek se lock-uje poslednji inode originala**.
- Ako kernel ne oslobađa inode, 2 procesa bi mogla da uđu u deadlock situaciju, kao na primeru 2 simultana procesa:
- **proces A:** **link("a/b/c/d", "e/f/g");**
- **proces B:** **link("e/f ", "a/b/c/d/ee");**

- Pretpostavimo da **proces A** nađe **inode za "a/b/c/d"** u isto vreme kada **proces B** nađe **inode za "e/f "** i oba postaju locked.
- Kada **proces A pokuša** da nađe inode za "e/f", on mora da čeka da inode za e postane free.
- Na drugoj strani **proces B** mora da čeka da inode za da postane slosoban, i tako su oba procesa uletela u **deadlock**.
- Sve se to sprečava tako što kernel otpusti inode nakon **LC++**;

# Unlink

- **unlink SC** uklanja FCB datoteke iz direktorijuma.

- Sintaksa za unlink SC je:

- **unlink(pathname);**
- gde je pathname ime koje će biti izbačeno iz UNIX stabla.

- Na primer u sledećoj sekvenci, **open SC će otkazati** jer je **datoteka prethodno obrisana:**
- **unlink("myfile");**
- **fd=open("myfile", O_RDONLY)**

- Ako se datoteka unlink-uje kao **poslednji ili jedini link** na datoteku, tada će unlink osloboditi inode i sve blokove datoteke.
- **Ako postoji više linkova**,
  - nestaje samo to ime i LC--;
  - dok je datoteka raspoloživa preko svih ostalih imena.

# unilnk - algorithm

- **algorithm unlink**
- input: file name
- output: none
- {
- **get parent inode** for directory to contain file to be **unlinked** (algorithm **namei**);
- /* **if unlinking the current directory…***/
- if (**last component of file name ".")**
- increment inode reference count;
- else
- get inode of file to be unlinked (algorithm **iget**);

- if(**file is directory but user is not super-user**)
- {
- release inodes (algorithm **iput**);
- **return (error);**
- }
-

# unlink - algorithm

- if (shared text file and link count currently 1) **remove from region table**;
- **write parent directory**: **zero inode number** of unlinked file;
- **release inode parent directory** (algorithm **iput**);
- **decrement** file **link count**;
- **release file inode** (algorithm **iput**);
- /* **iput** checks **if link count is 0**: if so,
- **release file blocks** (algorithm **free**) and
- **frees inode** (algorithm **ifree**)
- */
- }

name removing

file removing

# unilnk - algorithm

- Kernel prvo preko **namei** nalazi
- ali ne datoteku,
- već inode roditeljskog direktorijuma.

- Pristupa se in-core inode datoteke koja će brisati, preko iget.
- Potom se obave testovi,
- da li to shared text i
- da li LC=1, i
- kernel briše FCB iz direktorijuma, tako što na mestu inode broja upiše 0, a to se sinhrono upiše na disk i parent inode se oslobađa preko iput.

- Za datoteku koja se briše LC--; i oslobađa se in-core file inode.

- Ako prilikom otpuštanja in-core inode, LC padne na 0,
- ta datoteka više ne postoji na disku,
- iput tada oslobađa sve data blocks sa free,
  - tako što oslobađa sve direktne i sve indirektne blokove i
  - oslobađa inode sa **ifree** tako što upiše **filetype=0** u inode.

# FS Consistency

- Kernel obavlja svoje upise u **poretku** kako bi **minimizovao šanse** za FS oštećenje u slučaju otkaza sistema.

- Na primer, **kada se briše filename** iz svog direktorijuma (**FCB**),
  - to se se sinhrono upisuje na disk u roditeljski direktorijum,
  - pre nego što se oslobode blokovi datoteke i inode.

- Ako se dogodi **system crash** dok se oslobađaju data blokovi, nije strašno, nema FCB koji ukazuje na taj inode.

- Ako unos ne bi bio sinhron a dogodi se crash, FCB koji nije upisan može ukazivati na potpuno prazan inode ili na pogešan inode što je ozbiljnije oštećenje FS.

- **Mnogo je lakše očistiti inode za koga nema FCB, nego ispraviti pogrešan FCB**.

# FS Consistency

- Evo primera:
- **Pretpostavimo** da imamo **2 linka** sa imenima **a i b** i obavimo **unlink a**.
  - Kernel poštuje poredak write operacija,
  - pa prvo anulira FCB u direktorijumu za link-a i upiše ga na disk.
  - Ako tada sistem pukne, LC=2, a ostao je samo b, tako da će sistem imati problema jedino ako se **briše b**, a svi ostalim slučajevima je sve ispravno.
- Pretpostavimo **drugi poredak write operacija**.
- **Kernel prvo uradi LC--**; i **dogodi se crash** pre nego što je obrisan **FCB(a)**.
  - Posle butiranja sistema, imamo **2 FCB** na isti inode, a **LC=1** i ako se obavi brisanje a ili b, datoteka će fizićki nestati,
  - a ostaje FCB koji postaje neregularan, jer ukazuje ili na prazan inode ili na inode koji se kasnije dodeli za neku drugu datoteku, koja će opet imati LC=1, a dva imena.
- Kernel takođe **oslobađa inode i disk blokove u poretku**,
  - **mogu se prvo oslobađati data blocks** pa **tek onda inode** ili **obrnuto**,
  - ali **uticaj crash-a je sasvim različit**.
- Pretpostavimo da kernel **oslobađa disk blokove** i da se dogodi **crash**. Nakon podizanja sistema **imamo inode koji ukazuje na oslobođene blokove** a to je **prilično neregularna** situacija.
- Ako se **prvo oslobodi inode**, pa sistem pukne, blokovi koji su pripadali datoteci su neupotrebljivi **jer nisu vraćeni u free listu**, ali sve ostalo je sasvim regularno, **nema inode i nema pogrešnog pristupa**. Program fsck lakše oslobađa disk blokove nego što ih čisti.

# Race Condition

- Race condition je prisutan u **unlink SC**, **osobito kada se brišu direktorijumi** sa **rmdir** koji briše direktorijum, samo ako je prazan, a to će konstatovati samo ako svi direktorijumski **ulazi-FCB imaju vrednost inoda = 0**.
  - Kako se rmdir ne obavlja atomski, a to znači da može da se dogodi context switch između čitanja za proveru direktorijumskih blokova i brisanja samog direktorijuma.
  - Na primer može **drugi proces da obavi create**, a ovaj **prvi konstatovao da je već prazan.**
    Jedini način da se ovo spreči je **record locking**. Kada proces izvršava unlink call, nijedan drugi proces ne može da pristupa roditeljskom direktorijumu jer mu je inode locked zbog inoda.

- Podsetimo da algoritam za link SC oslobađa inode pre kraja poziva, ali ako drugi proces pokušava da obriše datoteku, to je moguće jer je link oslobodio inode, ali će brisanje samo **decrementirati LC**, a **datoteka ostaje**.

- **Drugi RC** se dešava kada **jedan proces konvertuje file pathname** u **inode** preko **namei** algoritma, a **drugi proces uklanja granu u toj putanji**.
  - Pretpostavimo da **proces A** razbija **pathname "a/b/c/d"** i odlazi na spavanje dok se ne oslobodi in-core **inode za c**.
  - Proces B na primer želi da obavi **unlink c** i on odlazi na spavanje.
  - Kada se on oslobodi i ako kernel da prednost procesu B, tada će B obrisati granu c, ako je prazna naravno.
  - Kada A **dobije procesor**, **namei** više **ne nalazi in-core c**, jer je obrisan a na njega je čekao i mora da se završi sa porukom o grešci.
  - Može da se dogodi i 3 **situacija da novi proces C**, dobije inode koji je pripadao grani c, napravi novu granu ili datoteku, a da proces a nastavi sa izmenjenom siutacijom, pa će pristupiti pogrešnoj datoteci, što nije dobro.

# brisanje otvorene datoteke

- **Proces može obrisati datoteku** čak i **kada je otvorena od strane nekog drugog procesa.**
- Pošto **open SC**, oslobađa inode na kraju, unlink će biti uspešan.
- Kernel prati unlink agoritam kao da datoteka nije otvorena i uklanja FCB, tako da datoteci više niko ne može da priđe.
- Međutim kako je open SC inkrementirao RC, kernel ne briše datoteku oslobađajući data blocks i inode, u **iput** algoritmu kod **unlink** SC.
- Svi procesi koji su otvorili datoteku rade sa njom (FD). Tek kada se dogodi close SC, tada kernel preko **iput** algoritma briše sadržaj datoteke.
- Ovo se često dešava, proces **kreira tmp file**, pa ga otvorenog **odmah obriše**, ali se pravo brisanje dogodi tek na zadnji close SC.

# brisanje otvorene datoteke

- Evo primera: prvo se otvori datoteka, zatim se briše, III **stat puca** jer napada datoteku koje više nema, ali fstat radi jer ide preko file descriptora
  - #include <sys/types.h>
  - #include <sys/stat.h>
  - #include <fcntl.h>
- **main(argc, argv)**
  - int argc;
  - char *argv[];
- {
  - int fd;
  - char buf[1024];
  - struct stat, statbuf;
- if(argc != 2) exit (); /*need parameter*/
- fd = **open**(argv[1], O_RDONLY)
- if (fd == -1) exit (); /*open fails */
- **if(unlink(argv[1], == -1) exit (); /*unlink just opened */**

# brisanje otvorene datoteke

- if( (**stat(argv[1],&statbuf == -1**)  /*stat the file by name */
  - printf("stat %s fails as it should\n", argv[1]);
  - else
  - printf("stat %s succeded!!!!! \n", argv[1]);

- if( (**fstat(fd,&statbuf == -1**)  /*fstat the file by fd */
  - printf("fstat %s fails !!!!!\n", argv[1]);
  - else
  - printf("fstat %s succeded as it should\n", argv[1]);

- while (**read**(fd, buf, sizeof(buf)>0)  /*read open/unlinked file*/
- printf("%1024s n", buf);  /*prints 1K byte field*/
- )

# FS Abstractions

- Weinberg je **uveo FS tipove-types** da bi obezbedio podršku za njegov network filesystem, što je implemetirano na UNIX System V.
- FS tipovi omogućavaju kernelu da simultano podržava više FS tipova kao što su mrežni FS ili FS različitih operatvnih sistema.
- Procesi koriste **normalan open SC**, a kernel mapira **generički skup I/O** operacija u operacija za specifični tip FS.

- **Inode je interfejs** između **apstraktnog FS** i **specifičnog FS**.

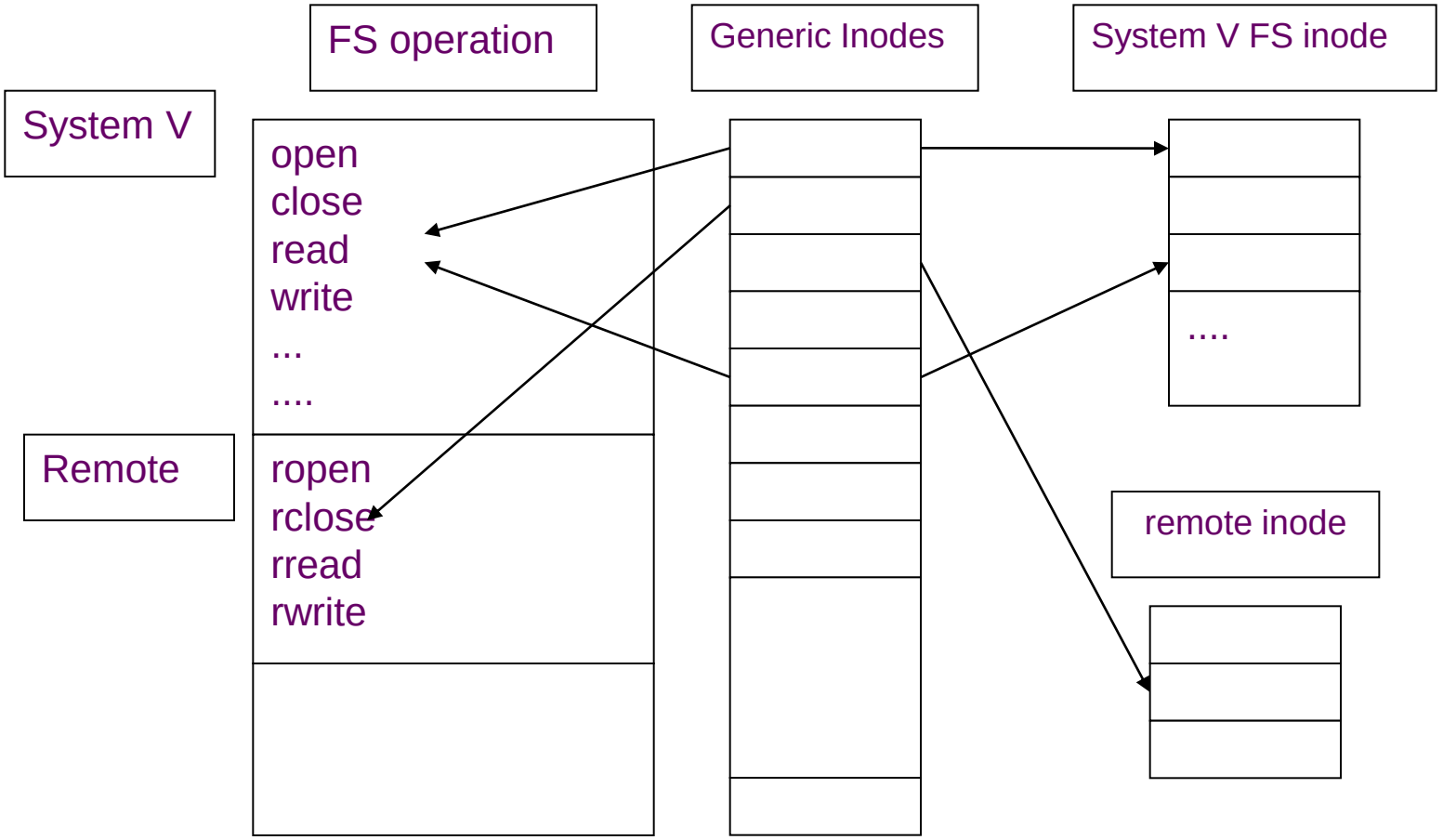
- Generički **in-core inode** sadrži podatke koji su nezavisni od specifičnog FS i ukazuju na file-specifični inode koji sadrži file-specific podatke.

- **File-specific inode** uključuje informacije kao što su ACL, block layout itd.

# FS Abstractions

- **Generic inode** sadrži informacije kao što su
  - device number,
  - inode number,
  - file type,
  - size,
  - vlasnik, i
  - RC.
- Drugi podaci koji su FS-specific sadržani u superblocku i direktorijumskim strukturama.
- Na slici je prikazana generička in-core inode tabela i 2 tabele sa FS-specific inodovima,
  - jedna za UNIX System V FS a
  - druga za remote inode,
  - koja sadrži dovoljno informacija da identifikuju datoteku na udaljenom FS.
- Taj udaljeni FS možda i nema inode baziranu strukturu, međutim sve se to preko generičke inode tabele i FS-specific tabele mapira u inode-like strukturu, tako da se sve radi apstraktno sa open, close, read, write, dobijanje inode za datoteku kao **namei** i **iget**, oslobađanje **inoda** kao **iput**, ažuriranja inoda, provera prava, setovanje prava, **mount** i **umount** FS.

# FS Abstractions



# FS Maintenance-data blocks

- Kernel održava konzistenciju FS za vreme normalne operacije, ali neobične okolnosti kao što je power-fail, mogu ostaviti FS u nekonzistetno stanje.
- Komanda **fsck** proverava **FS** i **ispravlja nepravilnosti**, pristupajući preko block ili character interfejsa, zaobilazeći **regularni file pristup**.

- Za **disk blokove**, pravila su jasna, prvo su svi blokovi slobodni, potom se blok izbaci iz liste a dodeli samo jednom inodu, a da bi se dodelio drugom inodu mora se prvo vratiti listu. Prema tome **disk blok** ili je u **free listi** ili je u **samo jednom inodu**.
- **Tipične neregularnosti** su
  - da **disk blokovi pripadaju više od jednog inoda**
  - ili se istovremeno nalaze u **listi slobodnih blokova** i **inodu.**

- Tipična situacija koja se dešava je kada kernel oslobodi blok iz datoteke, upiše ga in-core free listu superbloka, dodeli ga u **in-core** druge datoteke.
- **Potom treba ažurirati superblock** na **disku i oba inoda**, i ako se pre tih ažuriranja dogodi crash,
  - **može se blok naći u inodovima obe datoteke,**
  - **a takođe može završiti i free listi i u inodu.**
- Postoji **još jedna neregularna situacija** kada blok **nije ni u free listi** ni u **datoteci**, na primer blok se **izbaci iz free liste** upiše u **in-core inode**, ali se ne ažurira a dogodi se crash, pa ga nema nigde.

# FS Maintenance-inodes

- Takođe, **inodovi mogu imati non-0 link count**, a da **taj inode ne postoji u nijednom FCB**, što se dešava kada sistem ako se dogodi crash kada se dodeli inode za pipe ili new-file **a ne upiše se FCB**. Slična sitacija može da se dogodi ako se neregulano obriše grana koja nije prazna.

- I **druga polja u inodu mogu biti neregularna**, na primer **file type**, ako se mountuje nepravilno formatirani **FS**.

- Neregularnosti se pojavljuju u **slobodnim i zauzetim inodovima** jer se zbog **crash inode** može naći i u **FCB** i u **listi slobodnih inodova**.

- Neregularnost je kada **broj slobodnih blokova i broj slobodnih inodova** u **superbloku** ne odgovara realnom stanju na disku, zato što kompletna informacija u superbloku mora biti veoma tačna.