Оперативни системи 2

Факултет техничких наука Косовска Митровица

Драгиша Миљковић

Датотеке и директоријуми

Увод

До сада смо видели две кључне апстракције оперативних система:

- процес, који је виртуализација CPU, и адресни простор, који је виртуализација меморије

У овој и неколико следећих презентација додајемо још једну јако важну ставку у виртуализацији: **перзистентно смештање података** који омогућују дуготрајно чување информација.

За почетак, погледаћемо како изгледа API: тј. интерфејси који се користе на Unix системима датотека.

Шта је то датотека?

Датотека је просто низ бајтова који су смештени на перзистентан начин и који могу бити читани и уписивани.

Датотеке морају имати своја имена како би им се могло приступати.

- Свака датотека има јединствено име ниског нивоа у облику ***inode* броја**.
  - (*inode* = ***index node***)
- Поред њега, постоје још два типа имена: путање (***path***) и дескриптори датотека (***file descriptor***).

Систем датотека се састоји из много датотека. Када се каже „систем датотека“, мисли се на колекцију датотека, али и на део оперативног система који управља датотекама.

Систем датотека обично и не зна много о структури датотеке (да ли је то слика, текст…), он само има одговорност да чува такве податке перзистентно.

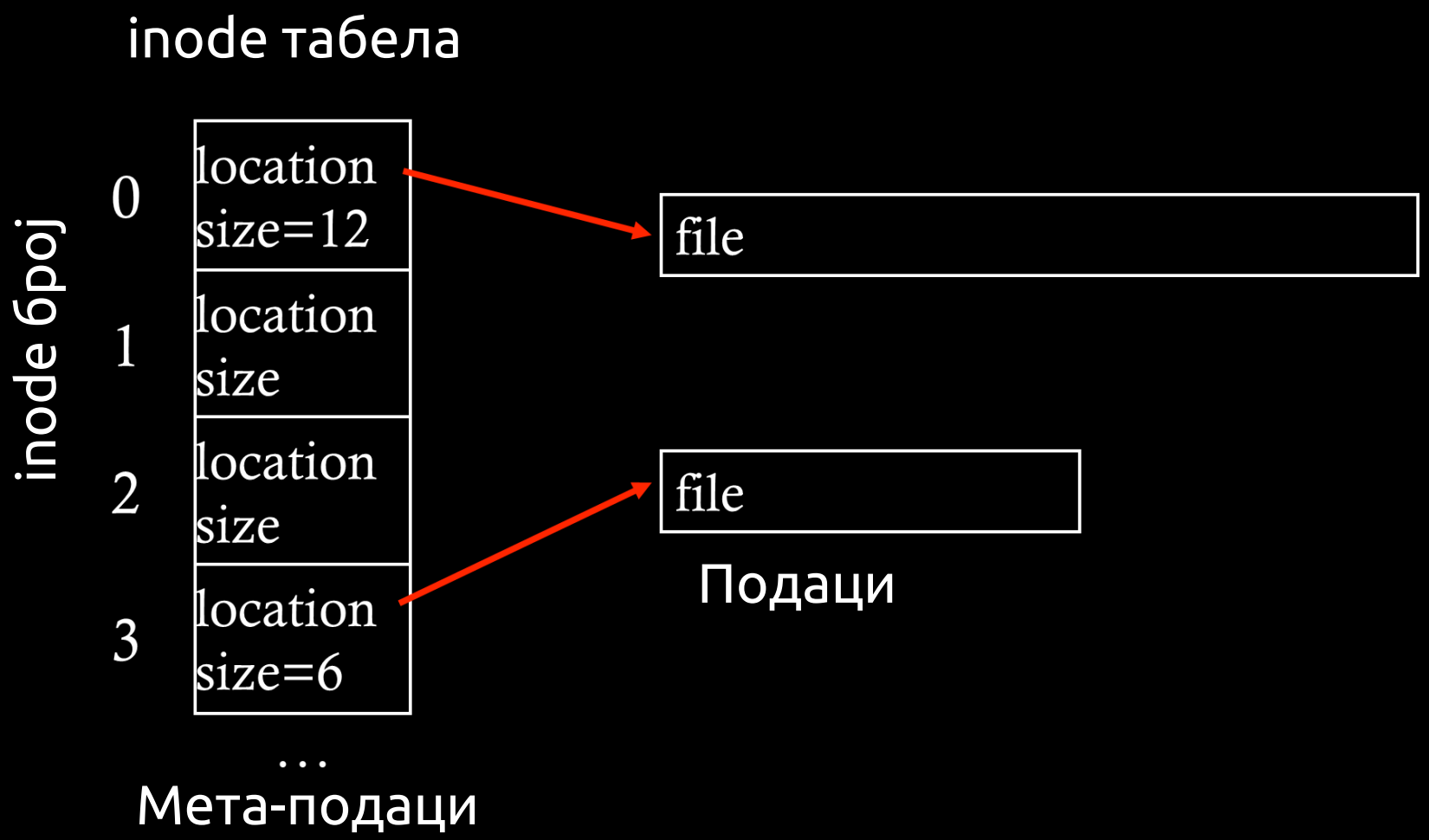
Inode број

Свака датотека има тачно један индексни чвор.

У сваком тренутку ови бројеви у систему датотека морају бити јединствени.

Различити системи датотека могу да користе исте бројеве.

Након брисања неке датотеке, њен *inode* постаје опет доступан за коришћење.


inode табела
inode број
0
1
2
3
location
size=12
location
size
location
size
location
size=6
...
Мета-подаци
file
file
Подаци

Интерфејс система датотека – први покушај

Погледајмо детаљније интерфејс система датотека

- Креирање, приступ и брисање података.

```
read(  int inode, void *buf, size_t nbyte)
write( int inode, void *buf, size_t nbyte)
seek(  int inode, off_t offset)
```

seek се извршава тек код читања/уписивања

Мане овог приступа?

- Теже је памтити *inode* него симболичка имена; индексни чворови се не додељују на неки организован начин, нити њихове вредности имају неки виши смисао;
- Како нпр. омогућити да више процеса истовремено приступа? (Са различитим офсетима.)

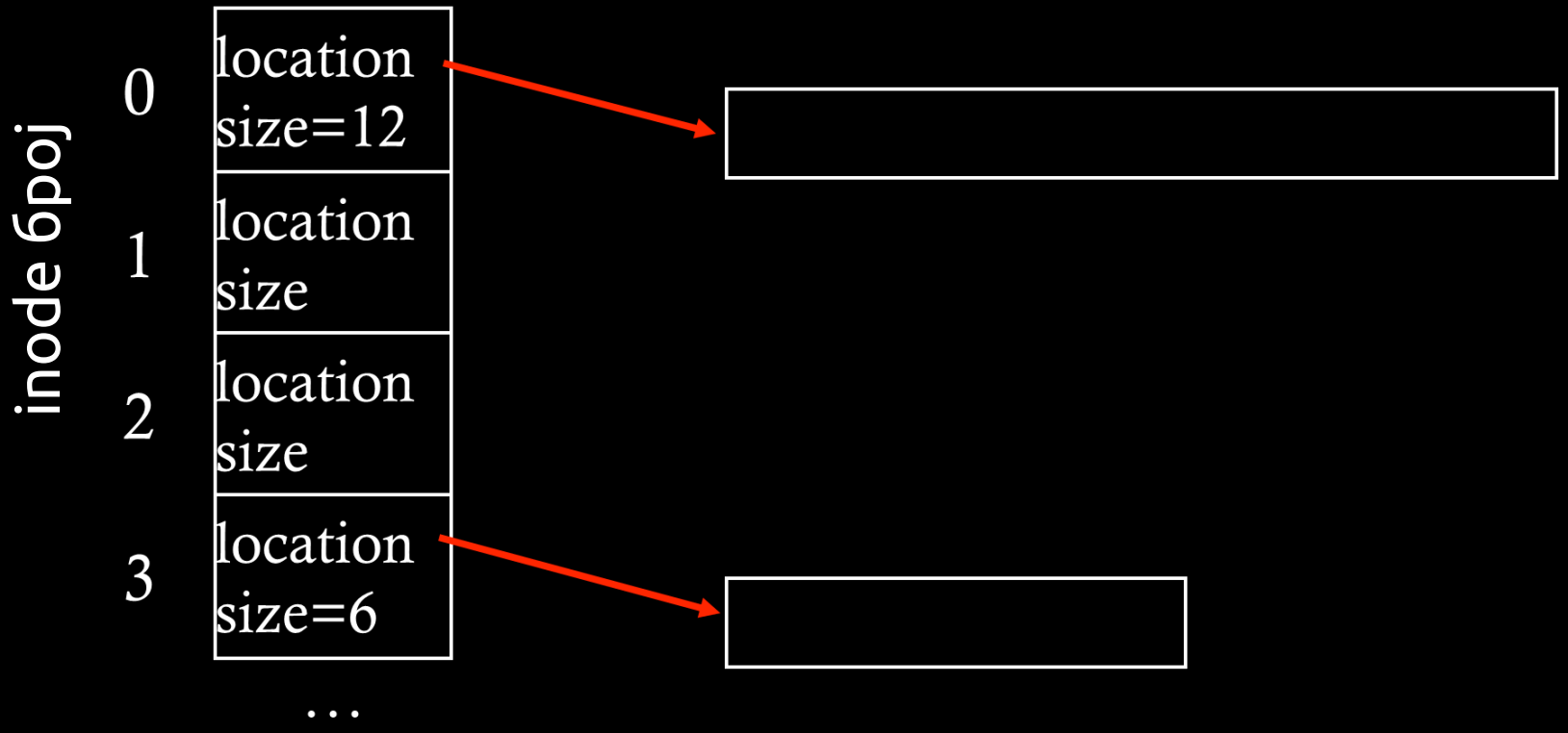
Путање (*Paths*)

Текстуална (симболичка) имена су много „ближа“ људима него што су им то нумеричка.

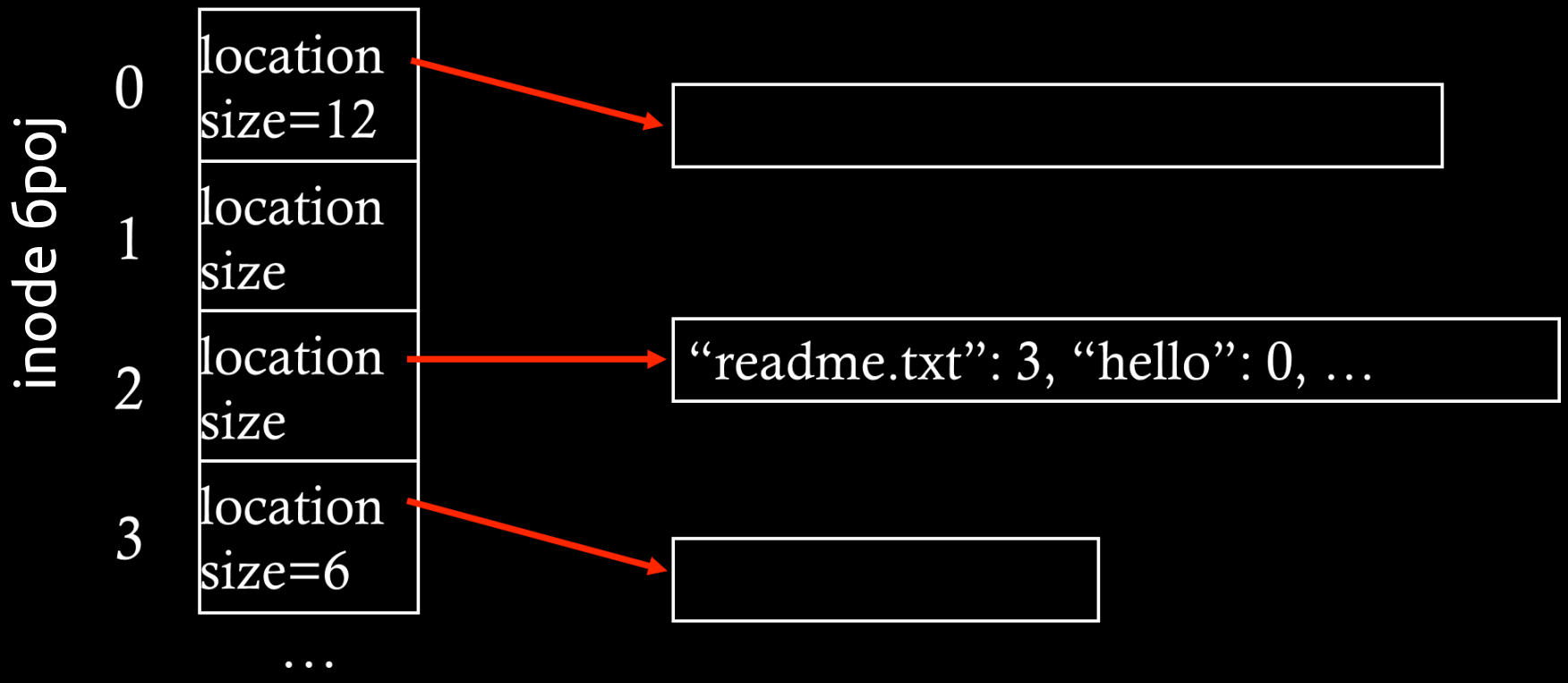
Системи датотека свакако и даље раде са *inode* чворовима, али се чувају и *path-inode* мапирања у предодређеној root датотеци (за root директоријум се обично додељује **inode 2**).

- Inode 1 на ext4 системима датотека се користи за лоше блокове хард-диска.

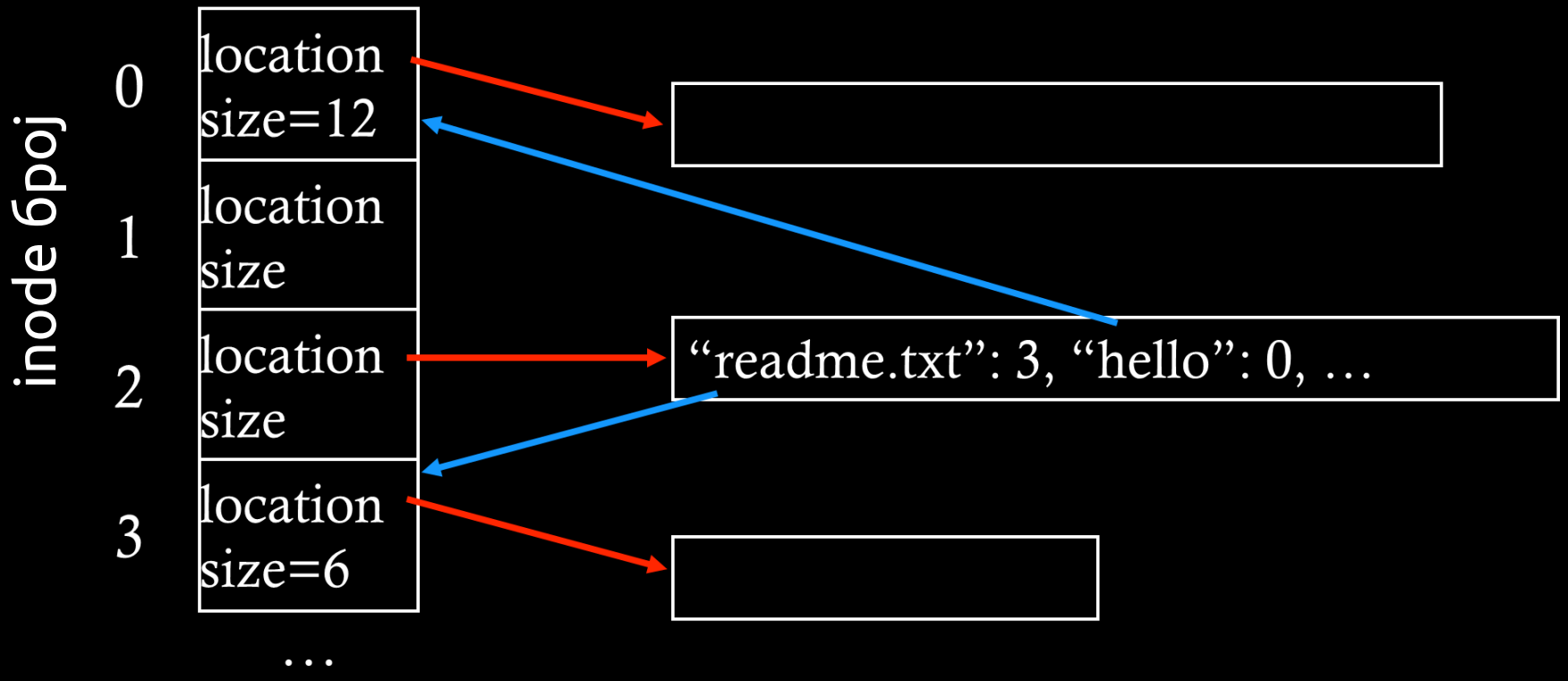
inode табела



inode табела



inode табела



Путање (*Paths*)

Генерализација!

Имамо **стабло директоријума** (*Directory Tree*) уместо једног *root* директоријума.

Директоријуми и датотеке могу да имају иста имена све док се налазе на различитим локацијама у оквиру стабла система датотека, па је ово дозвољено:

- /usr/d/nekoime.txt
- /tmp/nekoime.txt

У сваком директоријуму се чувају мапирања симболичких имена према *inode* бројевима.

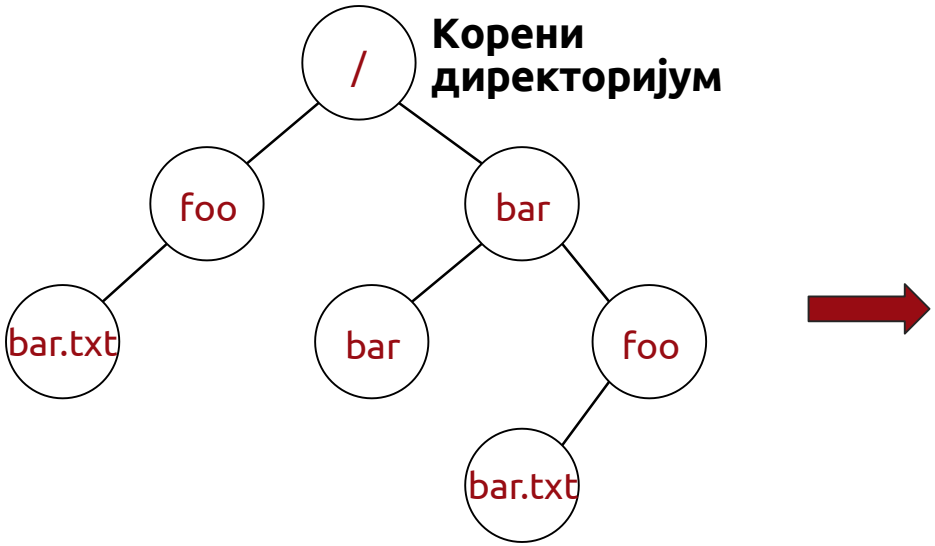
Директоријум

Директоријум је попут датотеке. И он има и *inode* име. Али његов садржај је другачији:

- Он чува листу парова:

  име ниског нивоа ⇆ **име читљиво кориснику**

- Свака ставка у директоријуму се односи или на неку датотеку или на неких под-директоријум.

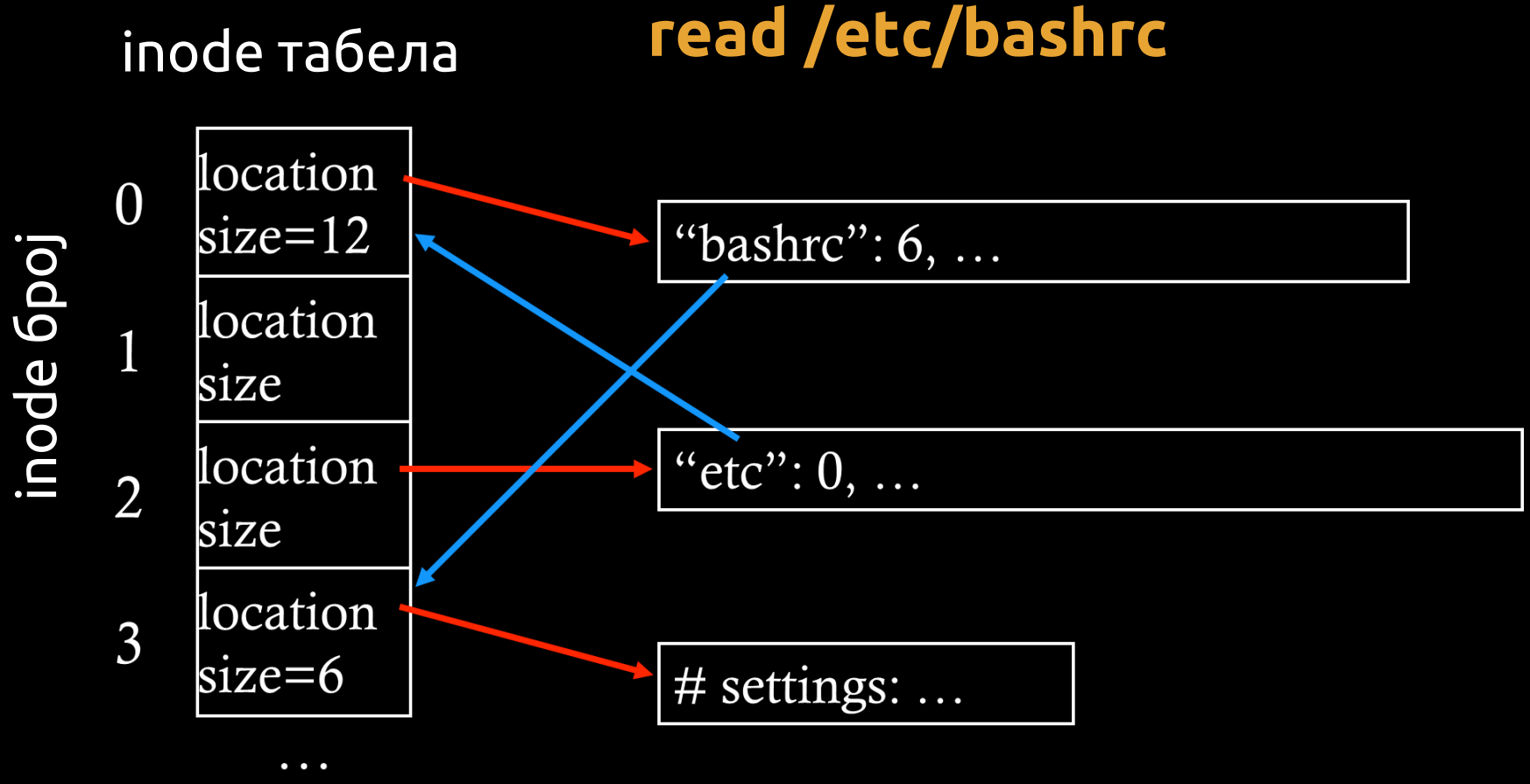


Пример стабла директоријума

ВАЛИДНИ ФАЈЛОВИ:
/foo/bar.txt
/bar/foo/bar.txt

ВАЛИДНИ ДИРЕКТОРИЈУМИ:
/
/foo
/bar
/bar/bar
/bar/foo/

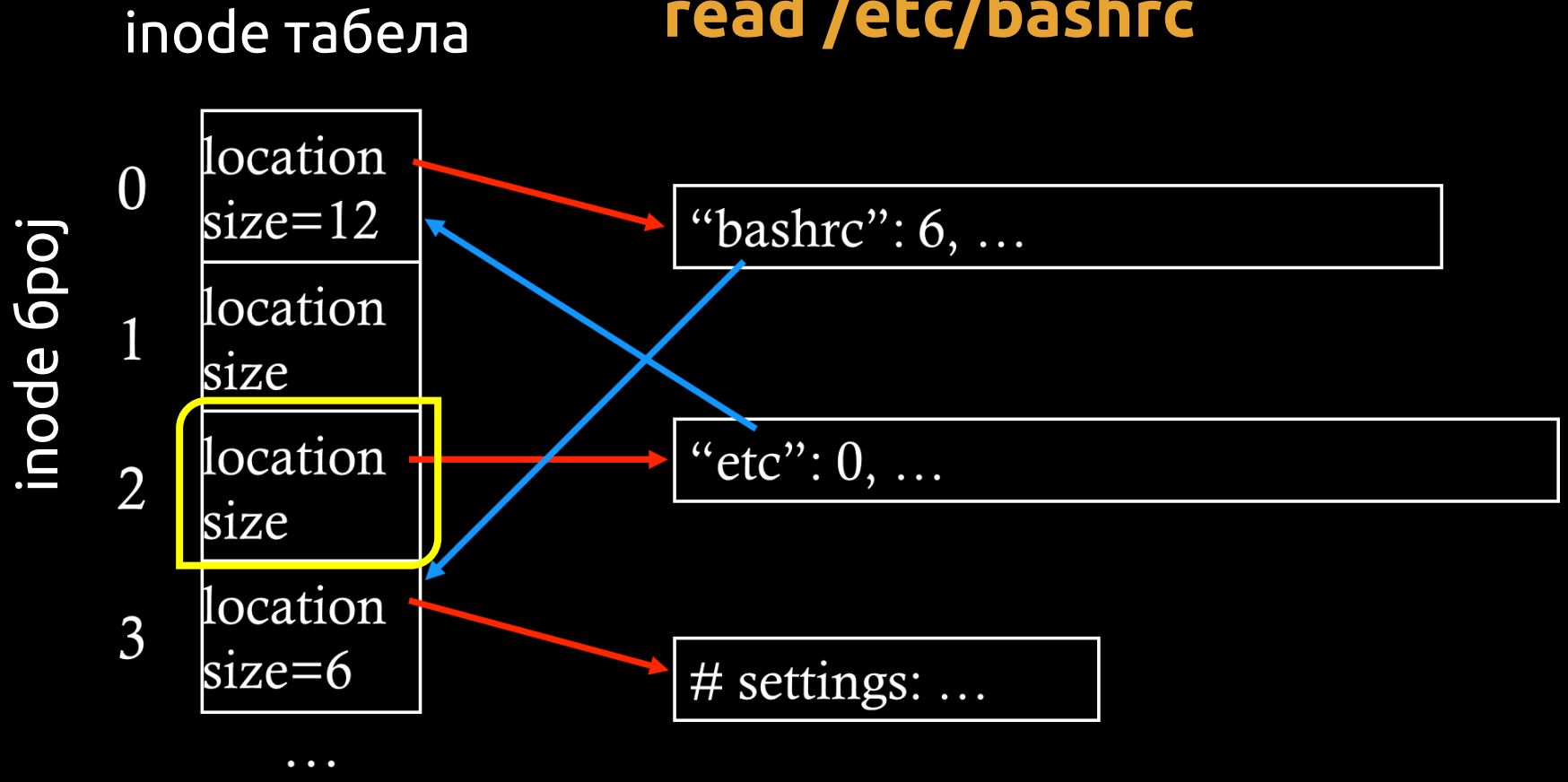
Пример читања



број_читања: 0

Пример читања – наставак

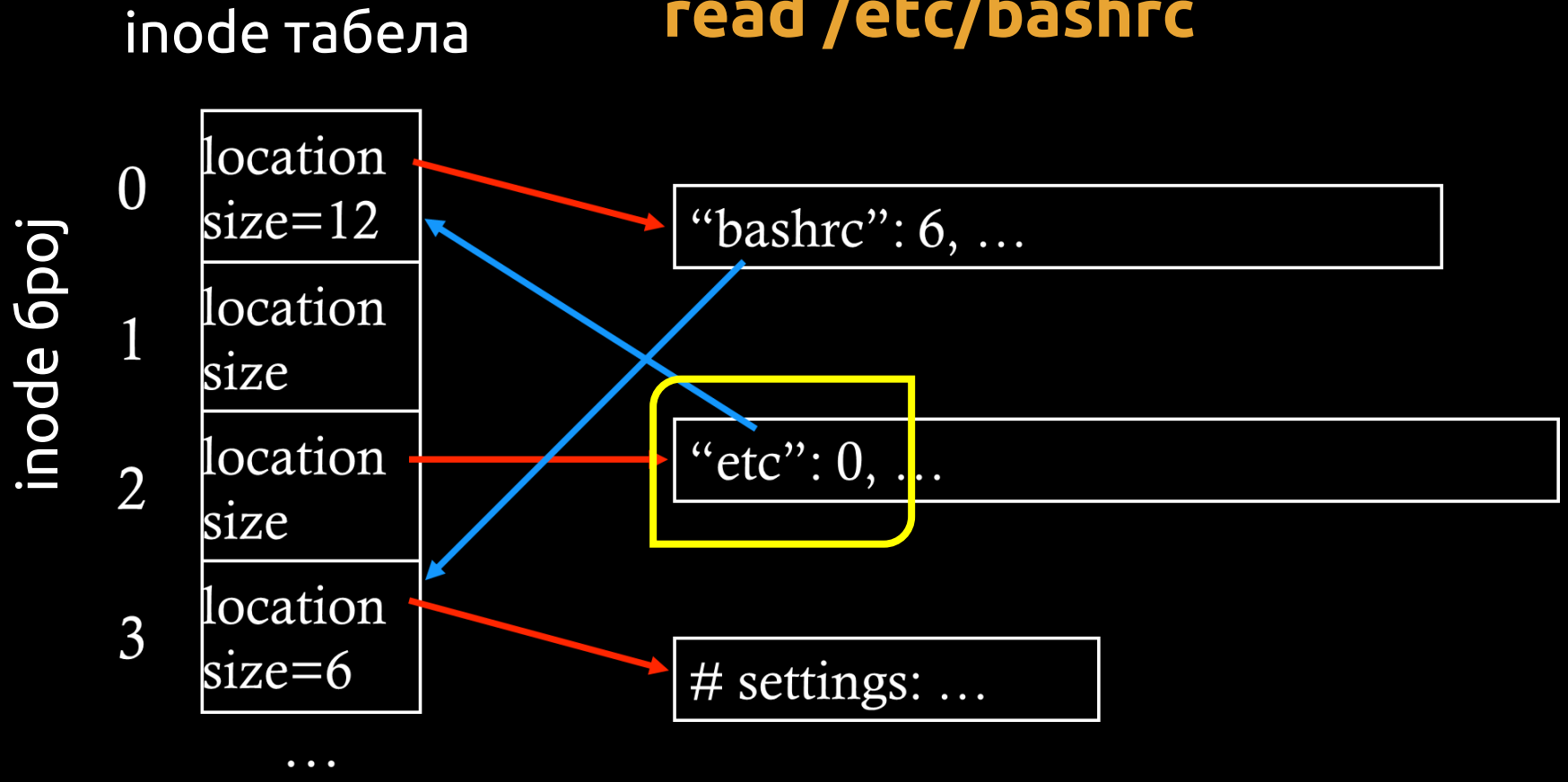
read /etc/bashrc



број_читања: 1

Пример читања – наставак

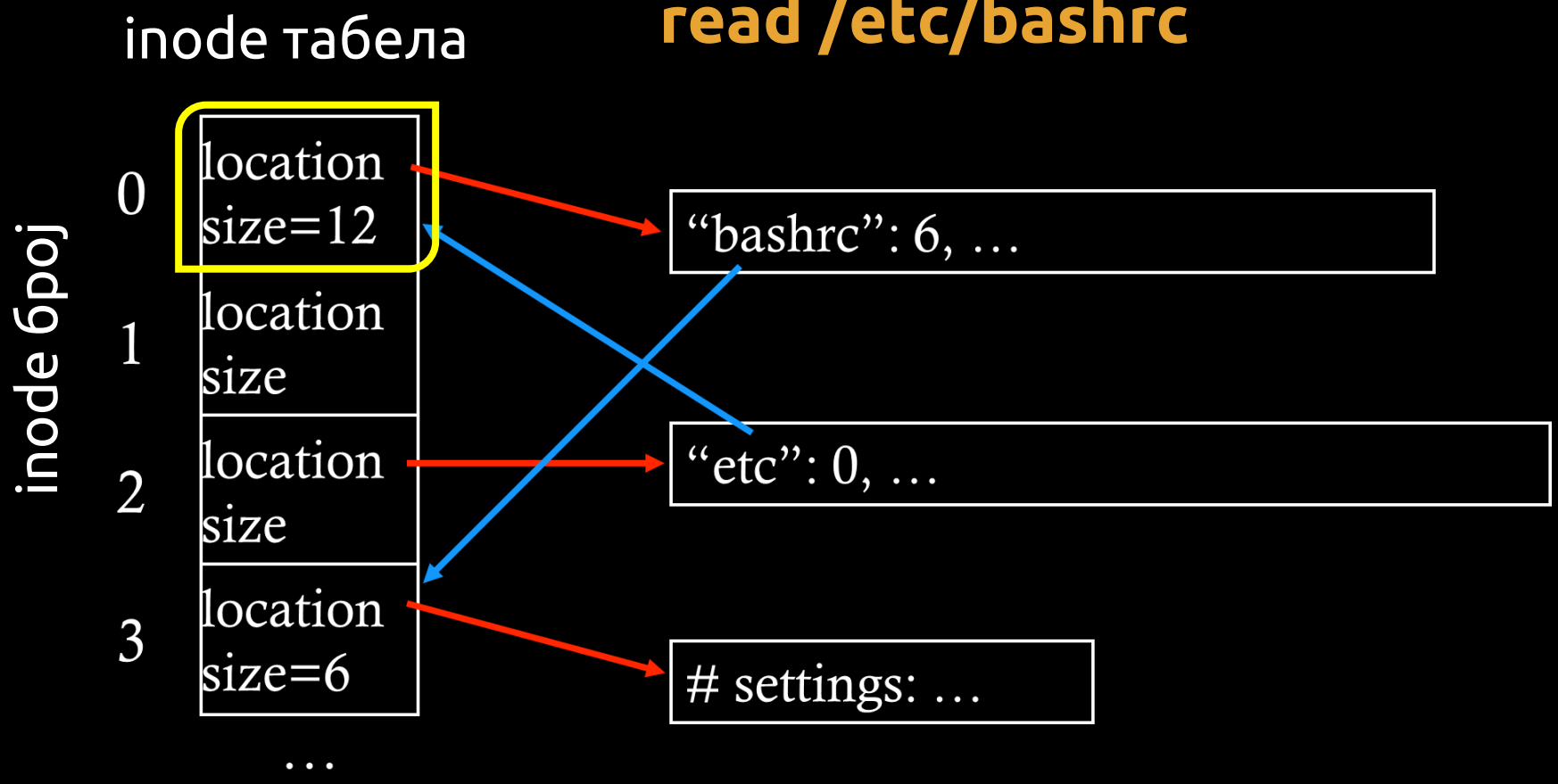
read /etc/bashrc



број_читања: 2

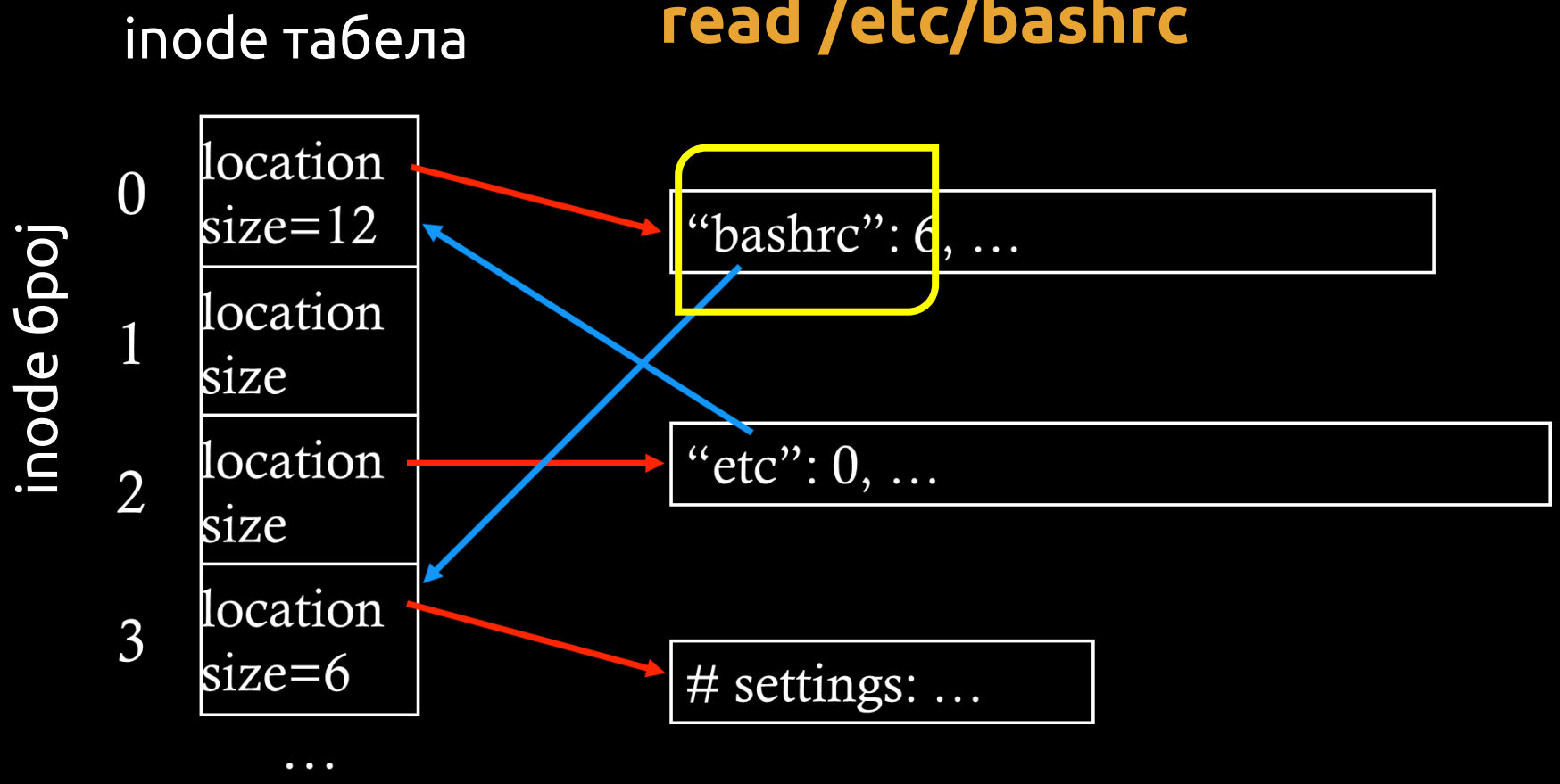
Пример читања – наставак

read /etc/bashrc



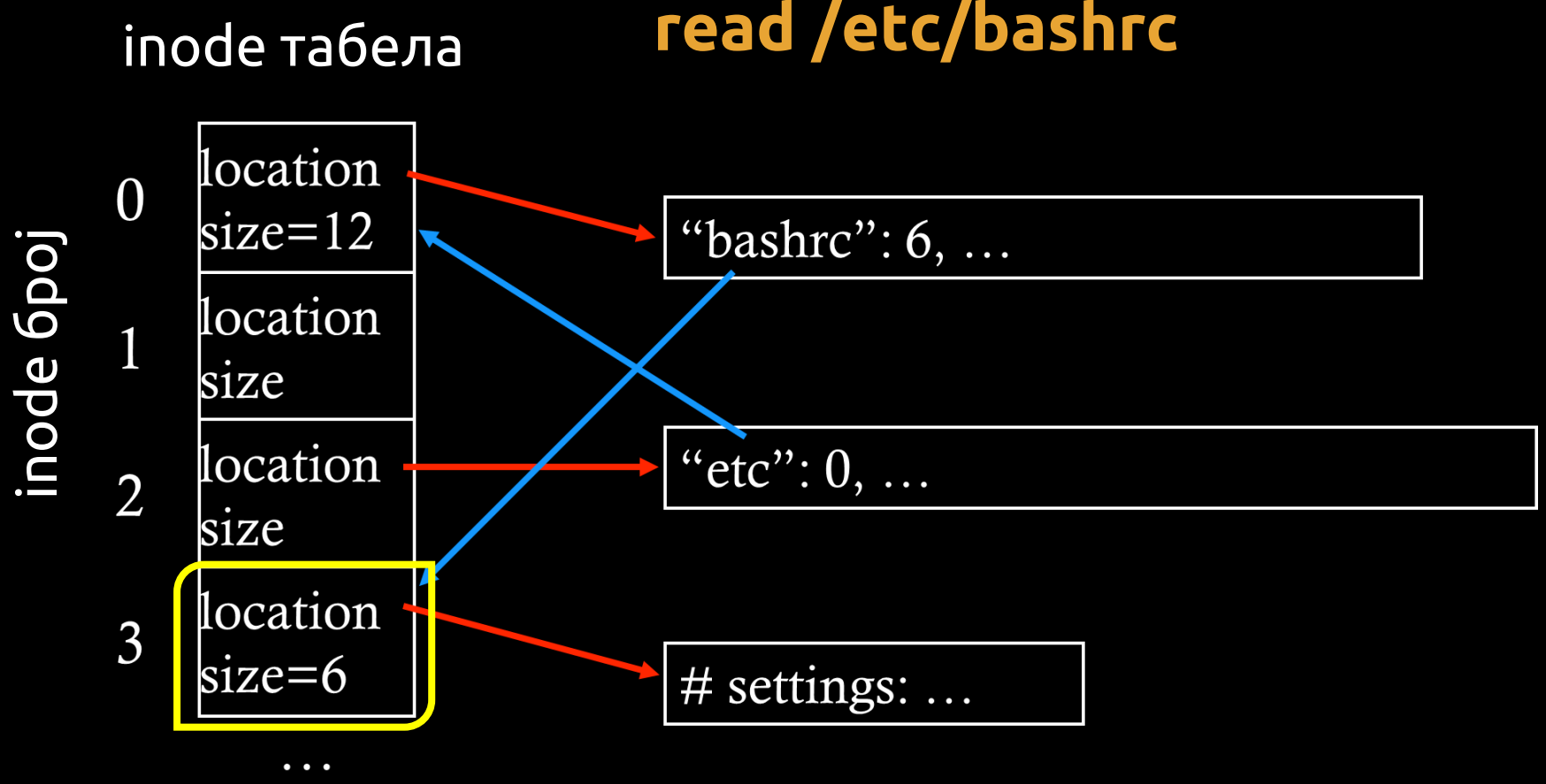
број_читања: 3

Пример читања – наставак



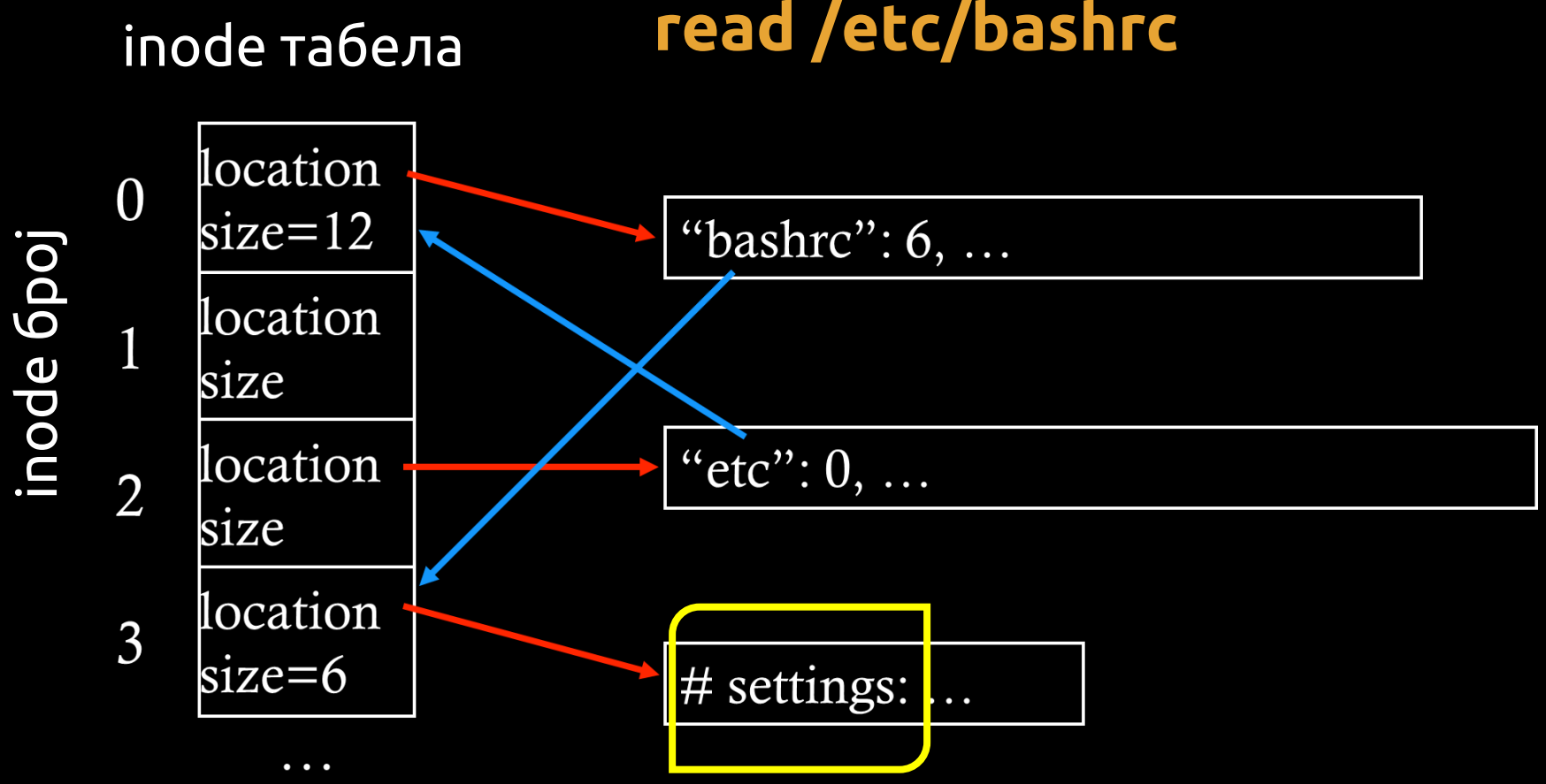
број_читања: 4

Пример читања – наставак



број_читања: 5

Пример читања – наставак



читај root дир. *(inode и податке)*
читај etc дир. *(inode и податке)*
читај bashrc дат. *(inode и податке)*

број_читања: 6

Интерфејс система датотека – други покушај

Отварање датотека преко њихових путања:

```
pread(char *path, void *buf, off_t offset, size_t nbyte)
pwrite(char *path, void *buf, off_t offset, size_t nbyte)
```

Мана – много читања док се дође до датотеке.

Како ово да решимо? Наћи датотеку у првом читању, а онда кеширати њену путању.

Имена датотека

Присетимо се, датотеке (и директоријуми) имају три врсте имена:

- ***inode***
- ***path***
- ***file descriptor***

Дескриптор датотека

Идеја – процес може да чува у себи кеширане путање путање до датотека:

- Уместо скупих вишеструких читања, желимо да, након што отворимо датотеку, сачувамо њен дескриптор (чувамо га у меморији).
- Читање и уписивање надаље иде преко дескриптора, он памти и колики је померај у оквиру датотеке.

Сваки процес који се извршава чува своју табелу дескриптора која садржи показиваче према отвореним датотекама.

И три стандардна тока података (преко којих обављамо I/O) имају своје место у овој табели:

- stdin: 0,
- stdout: 1,
- stderr: 2

File descriptor табела (xv6)

```
struct file {
  ...
  struct inode *ip;
  uint off;
};

// За сваки процес који се извршава се прави структура
struct proc {
  ...
  struct file *ofile[NOFILE];  // Отворене датотеке...
}
```

Интерфејс система датотека – трећи покушај

Предности овог приступа:

- Имена су симболичка (текстуелна),
- Хијерархијска организација,
- Кеширање превођења дескриптора датотека,
- Различити помераји су тачно дефинисани.

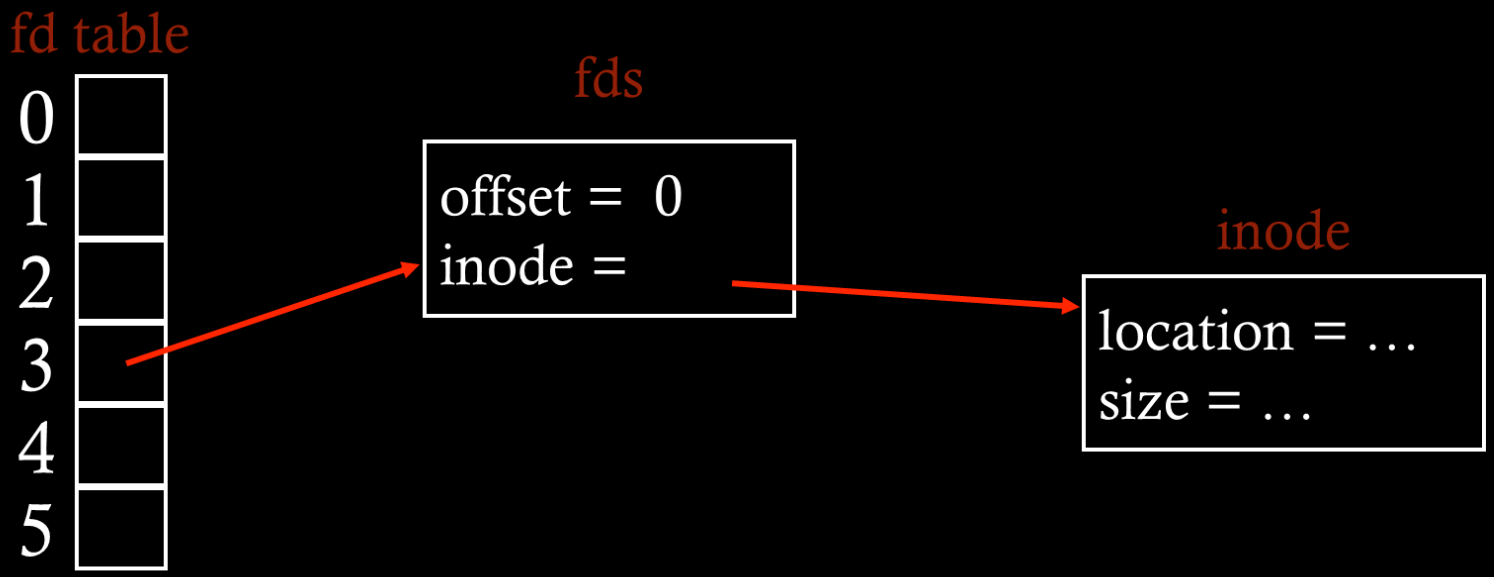
```
int fd = open(char *path, int flag, mode_t mode)
read(int fd, void *buf, size_t nbyte)
write(int fd, void *buf, size_t nbyte)
close(int fd)
```

Пример

```
// stdin = 0, stdout = 1, stderr =2
int fd1 = open("datoteka.txt");          // fd1 = 3
read(fd1, buf, 12);
int fd2 = open("datoteka.txt");          // fd2 = 4
int fd3 = dup(fd2);                      // fd3 = 5
```

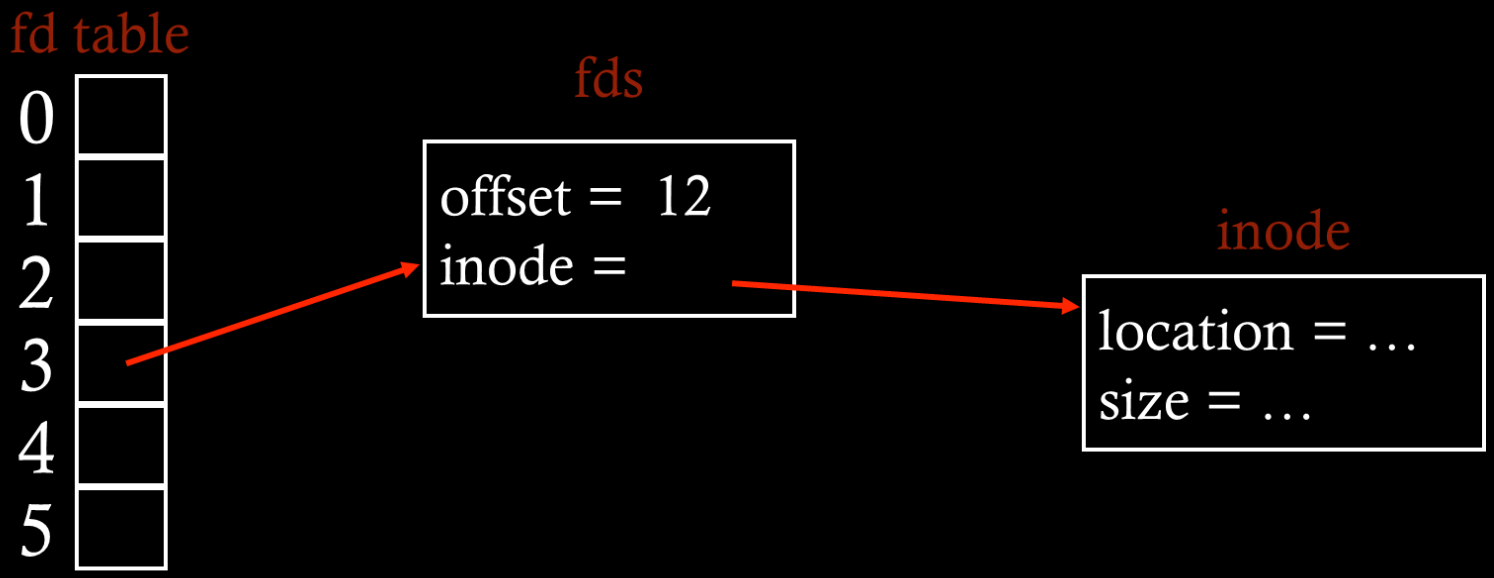
dup() позив дозвољава процесу да креира нови дескриптор датотек који референцира неку већ отворену датотеку са додељеним дескриптором.

Пример



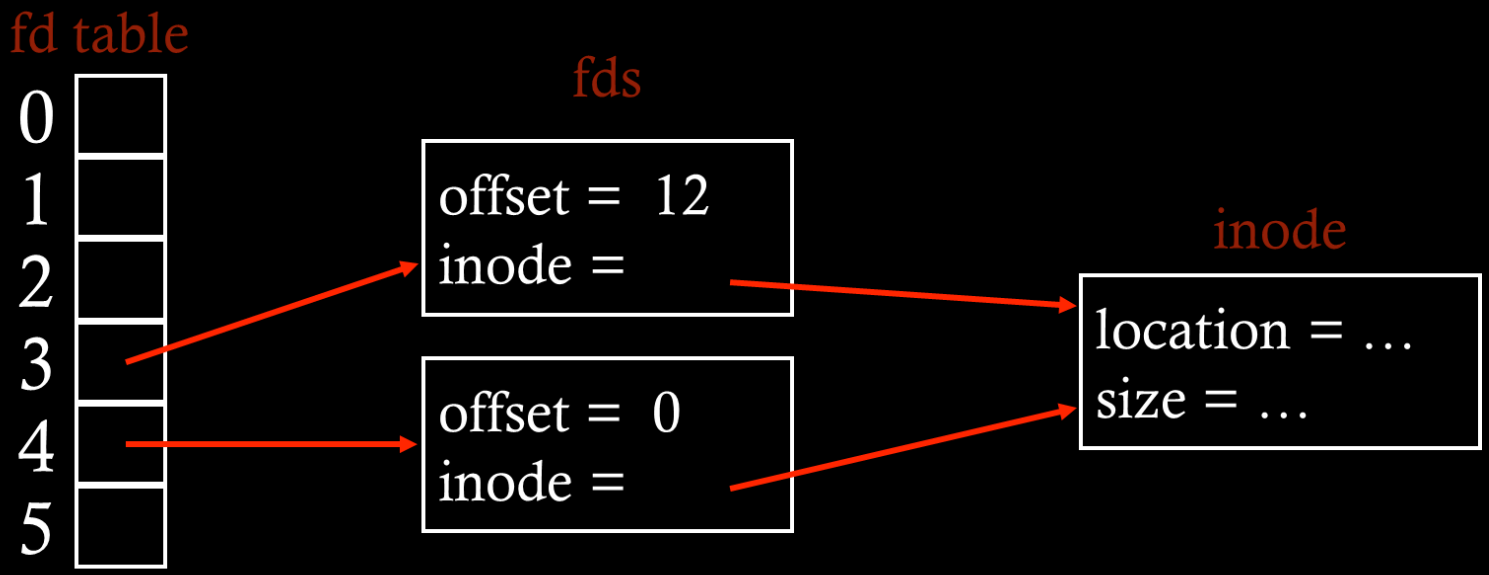

```
int fd1 = open("datoteka.txt");                    // fd1 = 3
```

Пример



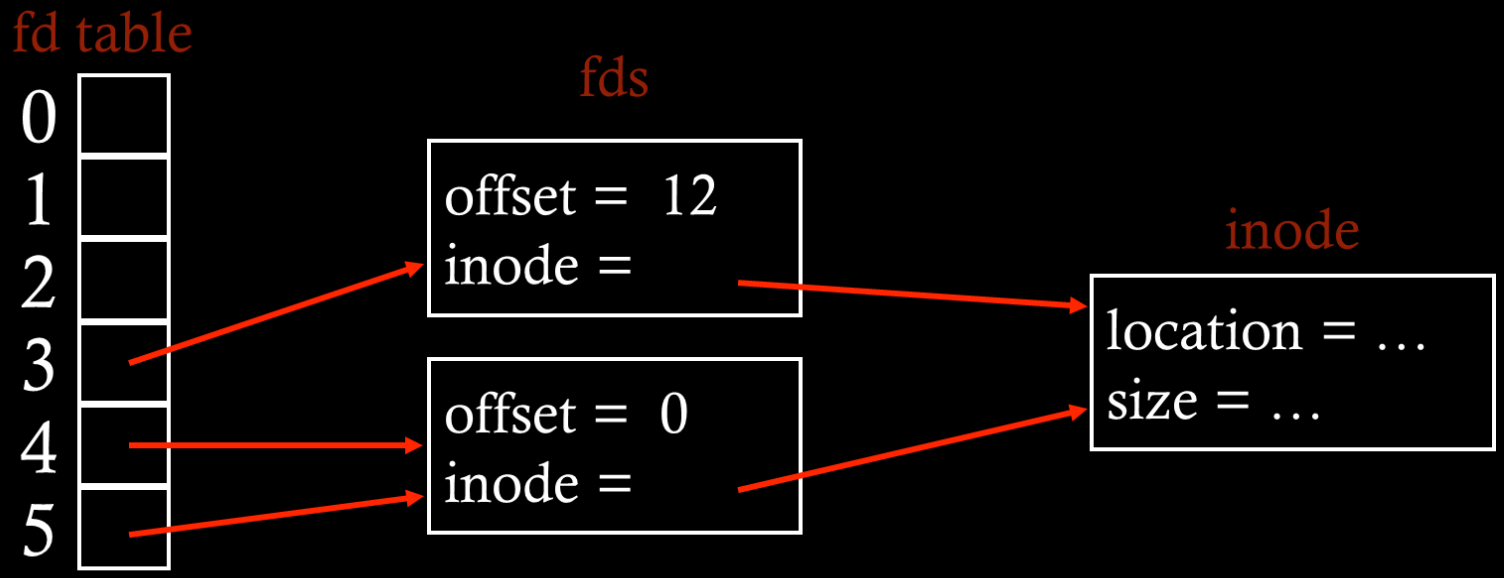

```
int fd1 = open("datoteka.txt");                    // fd1 = 3
read(fd1, buf, 12);
```

Пример




```
int fd1 = open("datoteka.txt");          // fd1 = 3
read(fd1, buf, 12);
int fd2 = open("datoteka.txt");          // fd2 = 4
```

Пример




```
int fd1 = open("datoteka.txt");          // fd1 = 3
read(fd1, buf, 12);
int fd2 = open("datoteka.txt");          // fd2 = 4
int fd3 = dup(fd2);                      // fd3 = 5
```

Отварање/креирање датотека

Коришћење системског позива **open()** уз флег **O_CREAT**.

- **O_CREAT** – ако се зада овај флег, ОС ће датотеку креирати – уколико она већ не постоји (без њега би само пробао да је отвори);
- **O_RDONLY, O_WRONLY, или O_RDWR** – флегови који отварају датотеку за читање, уписивање, или оба;
- **O_TRUNC** – постављање величине датотеке на нулу (постојећи садржај се уклања). Постоји и флег **O_APPEND** – он свако ново уписивање позиционира на крај датотеке.

Системски позив **open враћа назад дескриптор датотеке** (Ово је цео број, и увек је то најмања ненегативна слободна вредност која ће бити коришћена за приступ тој датотеци.

- Нпр. `read (file_descriptor)`

За више информација погледајте open

Читање и уписивање датотека

Пример читања и уписивања датотеке *foo*.

```
prompt> echo hello > foo  // сачувај излаз у датотеку foo
prompt> cat foo           // избаци садржај на екран
hello
prompt>
```

Како програм ***cat*** приступа датотеци *foo*?
Искористићемо ***strace*** да бисмо прегледали системске позиве које програм позива да би извршио ***cat***.

Strace је алат који омогућава преглед свих системских позива које ће програм извршити, њихове аргументе и повратне вредности. Има бројне опције, нпр.

- Уколико се зада са флагом –f пратиће и сву децу-процесе
- Са флагом –t ће исписивати и време када су позиви обављени
- -e trace=open,close,read,write ће извештавати само о системским позивима који су наведени након флага –e

Читање и уписивање датотека (наставак)

Резултат извршавања програма *strace* да би се видело шта ради програм *cat*.

```
prompt> strace cat foo              // strace ће показати шта то ради наредба cat
…
open("foo", O_RDONLY)               = 3
read(3, "hello\n", 4096)            = 6   // 6 је бр. прочитаних бајтова
write(1, "hello\n", 6)              = 6
hello
read(3, "", 4096)                   = 0
close(3)                            = 0
…
```

- **open()** – отвара датотеку за читање преко флага O_RDOLY. Враћа дескриптор датотеке **3** (0, 1 и 2 су за стандардни улаз, излаз и грешку – они су подразумевани за све процесе)
- **read() –** чита бајтове из датотеке 3 (наша датотека *foo*).
- **write() –** исписује бафер на стандардни излаз.
  - Приметите да **write** исписује у датотеку 1 (а то је стандардни излаз – тј. терминал).

Читање и уписивање датотека (наставак)

Како прочитати или уписати податке са одређеним померајем у оквиру датотеке? Ово што смо видели пре је све секвенцијално.

Користи се системски позив **lseek()**

```
off_t lseek(int fd, off_t offset, int odakle);
```

- Трећи аргумент означава како се извршава *seek*, он може бити:
  - SEEK_SET: до броја бајтова које чува **offset**.
  - SEEK_CUR: до тренутне позиције + бајтови **offset**.
  - SEEK_END: до величине фајла + бајтови **offset**.

lseek() мења вредност промењиве у оквиру кернела.
Ово није исто што и операција *seek* над диском!

File descriptor табела (xv6) – наставак

```
struct file {
  int ref;
  char readable; // Да ли може да се чита
  char writable; // Да ли може да се уписује
  struct inode *ip; // Којој датотеци одговара ова структура
  uint off; // Памти тренутни померај у оквиру датотеке...
};
```

Пример приступа датотеци

Процес отвара датотеку (величине 300 бајтова) и чита је узастопним позивима системског позива read(), при чему сваки пут чита по 100 бајтова. Ево како би изгледали ти позиви, њихове повратне вредности, као и вредност тренутног офсета у оквиру табеле датотеке.

Системски позиви	Повратна вредност	Тренутни померај
`fd = open("file", O_RDONLY);`	3	0
`read(fd, buffer, 100);`	100	100
`read(fd, buffer, 100);`	100	200
`read(fd, buffer, 100);`	100	300
`read(fd, buffer, 100);`	0	300
`close(fd);`	0	–

Пример уписивања у датотеку

Процес отвара исту датотеку два пута и врши уписивање у њу помоћу оба дескриптора. Приметите да се помераји ажурирају независно.

Системски позиви	Повратна вредност	fd1 – тренутни померај	fd2 – тренутни померај
`fd1 = open("file", O RDONLY);`	3	0	–
`fd2 = open("file", O RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

Пример репозиционирања у оквиру датотеке

Процес користи позив lseek() да промени позицију тренутног помераја (на 200) пре него што изврши читање.

Системски позиви	Повратна вредност	Тренутни померај
`fd = open("file", O RDONLY);`	3	0
`lseek(fd, 200, SEEK SET);`	200	200
`read(fd, buffer, 50);`	50	250
`close(fd);`	0	–

Дељена табела датотека – fork()

У већини случајева, као на претходним слајдовима, уколико два процеса отворе исте датотеке – сваки има своју табелу и мапирање.

У неким случајевима су табеле датотека дељене – нпр. код креирања детета процеса позивом fork().

Нпр. у коду (програм `fork-seek.c`) на следећем слајду родитељ креира дете и чека да се оно изврши. Дете мења свој офсет у оквиру датотеке и онда се завршава. Након тога, родитељ штампа своји офсет (који ће бити једнак ономе код детета).

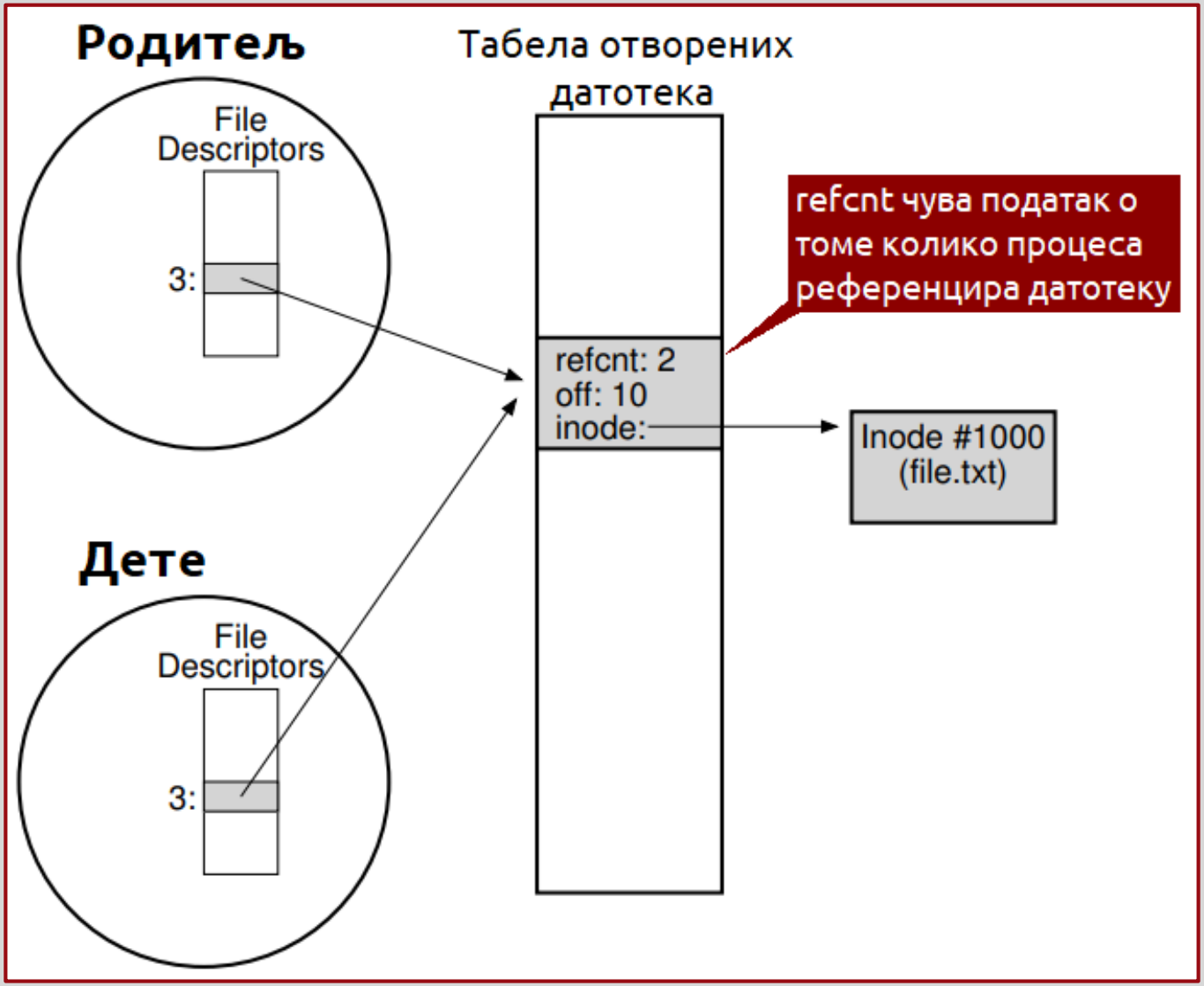
- Могућност да родитељ и дете деле датотеку је повремено веома корисна, нпр. омогућава да креирате више процеса који ће моћи да заједнички уписују у неку датотеку без да морате размишљати о координацији.

Дељена табела датотека – fork() – наставак

```c
int main(int argc, char *argv[]) {
  int fd = open("tmp.txt", O_CREAT|O_RDONLY);
  assert(fd >= 0);
  int rc = fork();
  if (rc == 0) {
    rc = lseek(fd, 10, SEEK_SET);
    printf("DETE: offset %d\n", rc);
  } else if (rc > 0) {
    (void) wait(NULL);
    printf("RODITELJ: offset %d\n", (int)lseek(fd, 0, SEEK_CUR));
  }
  return 0;
}
```

```
prompt> ./fork-seek
DETE: offset 10
RODITELJ: offset 10
prompt>
```

Дељена табела датотека – fork() – наставак



Неодложно уписивање преко fsync()

write() уписује податке на перзистентни медиј за чување података. Али он то не ради одмах, већ (из разлога бољих перформанси) их баферује и уписује тек након неког времена (нпр. након 5 секунди).

- На који начин баферовање уписивања поправља перформансе?

Ипак, некима апликацијама је потребно више од гаранције да ће подаци бити уписани кроз неко време. Уколико би се у међувремену десио квар, ово би могло имати опасне последице.

- Нпр. DBMS с времена на време захтева уписивања на диск без одлагања.

Када се користи **fsync()** за одређени дескриптор датотеке, систем датотека на то реагује тако што присилно уписује све измењене (**dirty**) податке на диск.

Неодложно уписивање преко fsync() – наставак

fsync() сва уписивања се извршавају без одлагања.

```
off_t fsync(int fd)
```

Пример коришћења fsync():

```
1   int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
2   int rc = write(fd, buffer, size);
3   rc = fsync(fd);
```

- Уколико је датотека направљена, она постаје део директоријума.
- Уколико желите бити апсолутно сигурни да неће бити грешке, треба позовати fsync() и над директоријумом. (Честа је грешка да се ово заборави.)

Преименовање датотека

rename() служи да се датотека преименује.

- Имплементиран је као атомични позив. Шта то значи?
- Нпр. Промена *foo* у *bar*.

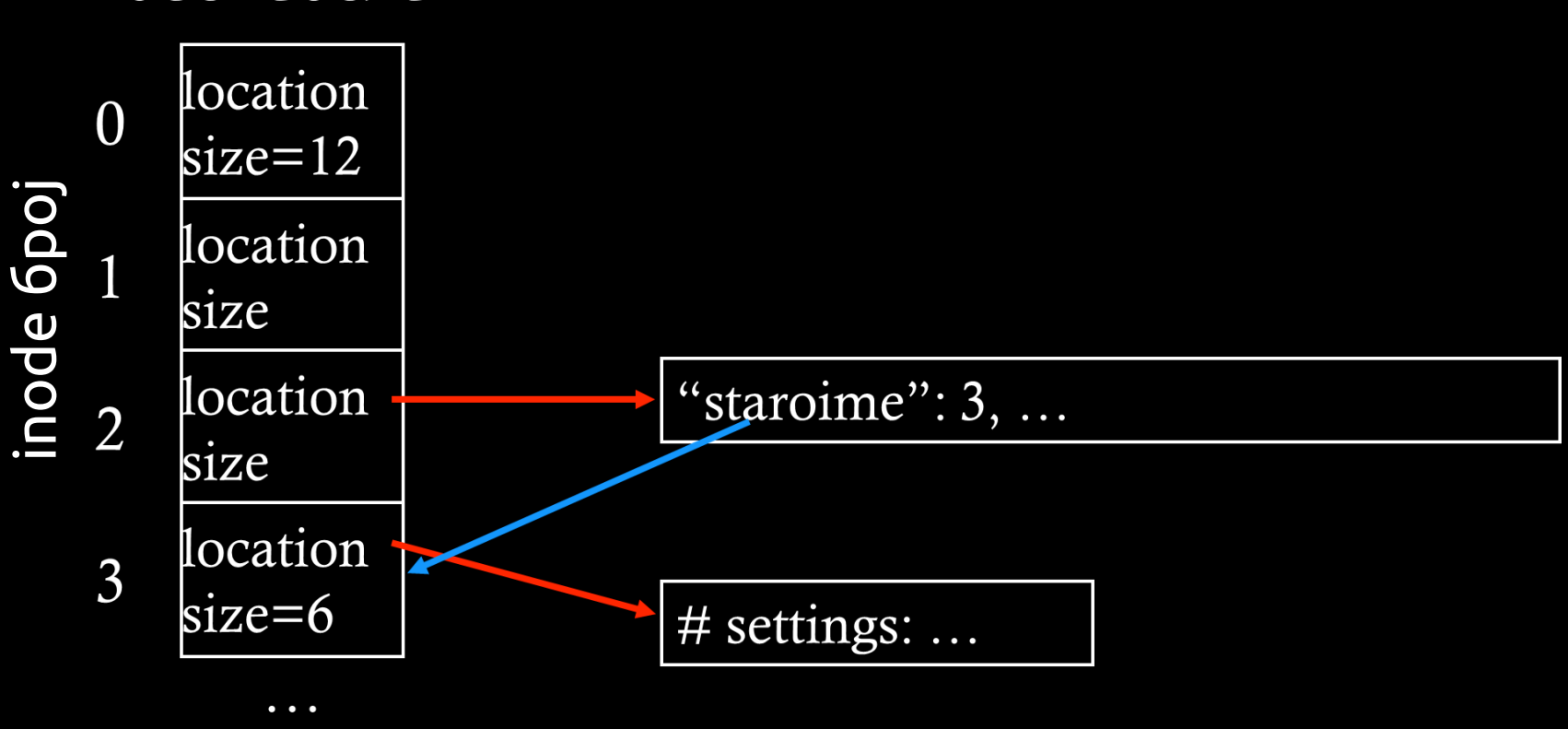
```
promt > mv foo bar
```

Резултат извршавања **strace** да се прикаже шта ради наредба **mv**:

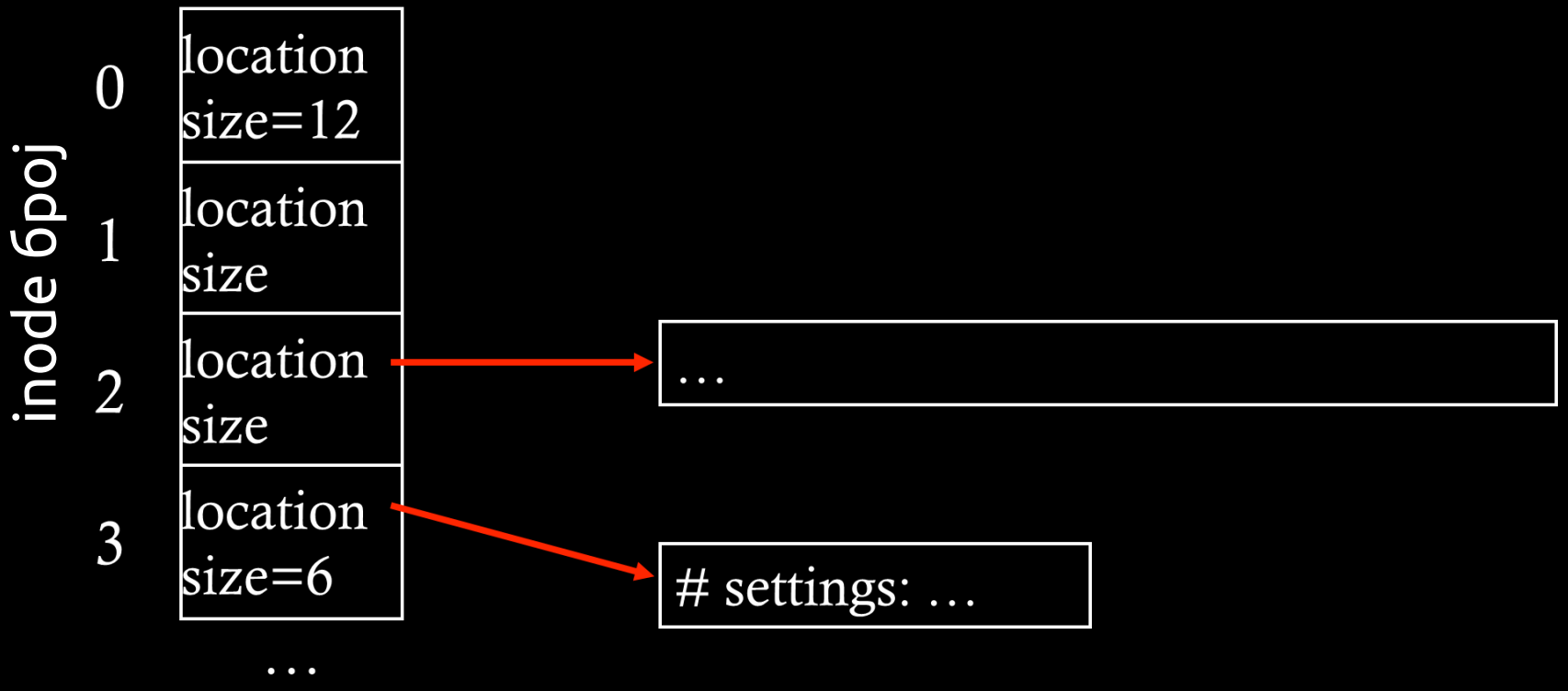
```
lstat("foo", {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
lstat("bar", 0x7ffcd2ea0f50)       = - ENOENT (No such file or directory)
rename("foo", "bar")               = 0
```

- rename() прима два аргумента, изворно име и ново име датотеке. Као што видите, да би ово успело, систем мора и да провери и да ли изворна датотека постоји (уколико не – јавља грешку), као и да ли постоји одредишна датотека (у том случају ће је пребрисати).

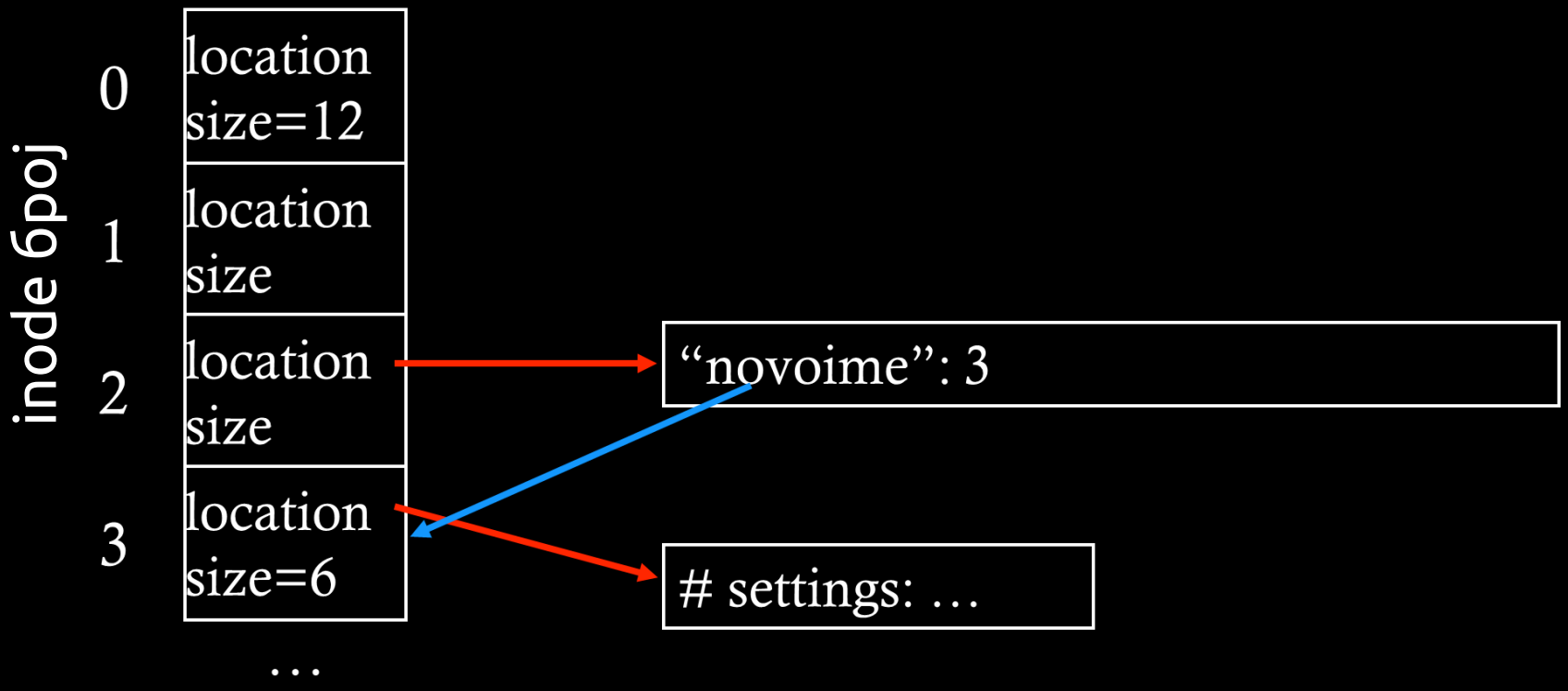
Пример – rename()



Пример – rename()



Пример – rename()



Шта би било да rename није атомично?

```
rename(char *old, char *new)
```

- брише стару везу према датотеци
- ствара нову везу према датотеци

Шта уколико се деси отказ?

Систем датотека ово предупређује тако што гарантује атомичност помоћу позива rename()! Следи пример.

Атомично ажурирање датотеке

Рецимо да апликација жели да ажурира датотеку foo.txt

Ово мора бити атомично.

Мора се обезбедити да, уколико систем откаже, буде могуће видети или стару датотеку или нову.

1. подаци се уписују у датотеку **foo.txt.tmp**
2. та датотека се упише на диск **fsync foo.txt.tmp**
3. потом се **foo.txt.tmp** преименује у **foo.txt**, тако да нова датотека замењује стару

Коришћење преименовања датотека – пример

Рецимо да користите текстуални едитор и желите да измените неку отворену датотеку (нпр. foo.txt). Како би едитор могао да гарантује да ће те промене бити успешно уписане?

```
1   int fint fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC);
2   write(fd, buffer, size); // Упиши нову верзију датотеке
3   fsync(fd);
4   close(fd);
5   rename("foo.txt.tmp", "foo.txt");
6   d = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
7   int rc = write(fd, buffer, size);
8   rc = fsync(fd);
```

Едитор може да упише нову верзију датотеке са привременим именом (foo.txt.tmp), затим преко fsync() натера ОС да упише датотеку на диск, а онда атомичним позивом rename() замени стару датотеку новом (при чему се исотвремено брише стара верзија).

Добијање информација о датотекама

stat() приказује **метаподатке** датотеке. Метаподаци су информације о неком појединачном фајлу, нпр. величина, права приступа, …

- Структура коју позив **stat()** враћа је дата испод:

```
1    struct stat {
2        dev_t st_dev;           /* ID uređaja koji sadrži fajl */
3        ino_t st_ino;           /* inode broj */
4        mode_t st_mode;         /* zaštita */
5        nlink_t st_nlink;       /* broj tvrdih veza */
6        uid_t st_uid;           /* "user ID" vlasnika */
7        gid_t st_gid;           /* "group ID" vlasnika */
8        dev_t st_rdev;          /* "device ID" (if special file) */
9        off_t st_size;          /* ukupna veličina, u bajtovima */
10       blksize_t st_blksize;   /* veličina bloka sistem datoteka I/O */
11       blkcnt_t st_blocks;     /* broj alociranih blokova */
12       time_t st_atime;        /* vreme poslednjeg pristupa */
13       time_t st_mtime;        /* vreme poslednje izmene */
14       time_t st_ctime;        /* vreme poslednje promene statusa */
15   };
```


Добијање информација о датотекама (наставак)

Пример извршавања **stat()**

- Све информације су у оквиру индексног чвора:

```
prompt> echo hello > file
prompt> stat file
  File: ‘file’
  Size: 6  Blocks: 8   IO Block: 4096   regular file
Device: 811h/2065d   Inode: 67158084   Links: 1
Access: (0640/-rw-r-----) Uid: (30686/ root) Gid: (30686/ despot)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

Уклањање датотека

Резултат извршавања *strace* како би се утврдило шта ради *rm*.

- *rm* је команда Линукса за брисање датотеке.
- *rm* позива ***unlink()*** како би уклонио датотеку.

```
1  prompt> strace rm foo
2  ...
3  unlink("foo")
4  ...
5  prompt>
```

Зашто се позива *unlink*(), а не *remove* или *delete*?
Ово ће бити јасније након што видимо тврде везе.

Уклањање датотека

Нема системског позива за брисање датотека.

Inode (и њему додељену датотеку) брише ***garbage collector*** када „види“ да нема више референци. Овај приступ вам је познат из Јаве.

Путање се бришу када се позове ***unlink*()**

Дескриптори се бришу када се позове ***close*()**

Прављење директоријума

***mkdir*()** прави нови директоријум.

- Овај системски позив има исто име као команда терминала mkdir
- Не може се ништа директно уписивати у директоријум, већ се он може ажурирати једино индиректно: кроз нпр. креирање датотека, директоријума у оквиру тог директоријума.

```
1  prompt> strace mkdir foo
2  . . .
3  mkdir("foo", 0777) = 0
4  . . .
5  prompt>
```

Прављење директоријума – наставак

Новонаправљени директоријум се сматра „празним“, међутим – он у најмању руку има барем две ставке.

1. Показивач на самог себе: **.** (тачка)
2. Показивач на родитељски директоријум: **..** (тачка-тачка)

```
1  prompt> ls -al
2  total 8
3  drwxr-x---  2 root root    6 Oct 30 16:17 ./
4  drwxr-x--- 26 root root 4096 Oct 30 16:17 ../
```

Читање директоријума

Једноставан код (програм listdir.c) за читање ставки директоријума. Овај код ради исти посао као наредба терминала **ls**.

Програм у ову сврху користи три позива: opendir(), readdir(), closedir()

- Да је било потребно приказати више информација о садржају директоријума, могао је бити позван позив stat() над сваком ставкокм у директоријуму. Овако ради наредба ls –l

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");  // otvaranje trenutnog direktorijuma
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) { // čitanje direktorijuma
        printf("%d %s\n", (int) d->d_ino, d->d_name);
    }
    closedir(dp);   // zatvaranje trenutnog direktorijuma
    return 0;
}
```

Садржај dirent структуре

Ова декларација испод приказује информације које су доступне у оквиру сваког директоријума.

```
struct dirent {
    char d_name[256]; // ime
    ino_t d_ino; // inode broj
    off_t d_off; // pomeraj do sledeceg dirent
    unsigned short d_reclen; // duzina ovog sloga
    unsigned char d_type; // tip datoteke
};
```

Брисање директоријума

rmdir() брише директоријум.

- rmdir() захтева да директоријум буде празан пре брисања.
  - Ово је уведено због заштите од нежељеног брисања, директоријуми могу у себи чувати много података.
- Позив rmdir() над непразним директоријумом ће бити неуспешан.
- У случају да је директоријум непразан, мора се рекурзивно позвати наредба rm: `rm -r imedir`

Тврде везе

link() системски позив повезује стару датотеку и нову датотеку.

- Прављење тврде везе под именом *file2*:

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2 // pravljenje tvrde veze, povezivanje file i file2
prompt> cat file2
hello
```

Резултат позива **link()**

- Две датотеке имају исти inode број, али два различита симболичка имена (*file*, *file2*).

```
prompt> ls -i file file2
67158084 file      // inode vrednost je 67158074
67158084 file2     // inode vrednost je 67158074
prompt>
```

Тврде везе (наставак)

Шта се дешава сваки пут када направите датотеку?

- Корак 1: Прави се **inode**, чувају се све информације о датотеци.
- Корак 2: Повезује се (**link**) симболичко име са датотеком.
- Корак 3: Датотека тврде везе се чува у тренутном директоријуму.

Након креирања тврде везе према некој датотеци, нема разлике између ње и старе датотеке. Оне су обе само веза према садржају датотеке.

Сада када ово знате, можда је јасније зашто се да би се уклонила датотека позива **unlink()**.

Раскидање тврдих веза – unlink

Шта ради **unlink()**?

- Проверава бројач референци (*link count*) у оквиру *inode* броја.
- Уклања везу између симболичког имена и *inode* броја.
- Умањује бројач референци
  - Уколико је стигао до нуле, датотека се брише (а *inode* и повезани блокови се ослобађају).

Раскидање тврдих веза – unlink (наставак)

Резултат извршавања unlink()

```
prompt> echo hello > file          /* pravi se fajl */
prompt> stat file
... Inode: 67158084 Links: 1 ...   /* Broj linkova je 1 */
prompt> ln file file2              /* Tvrda veza file2 */
prompt> stat file
... Inode: 67158084 Links: 2 ...   /* Broj linkova je 2 */
prompt> stat file2
... Inode: 67158084 Links: 2 ...   /* Broj linkova je 2 */
prompt> ln file2 file3             /* Tvrda veza file3 */
prompt> stat file
... Inode: 67158084 Links: 3 ...   /* Broj linkova je 3 */
prompt> rm file                    /* Brisemo file */
prompt> stat file2
... Inode: 67158084 Links: 2 ...   /* Broj linkova je 2 */
prompt> rm file2                   /* Brisemo file2 */
prompt> stat file3
... Inode: 67158084 Links: 1 ...   /* Broj linkova je 1 */
prompt> rm file3
```

Симболичке (меке) везе

Симболичке везе су корисније од тврдих веза.

- Тврде везе не могу да се направе за директоријум.
- Тврде везе се не могу направити према датотеци која се налази на другој партицији (јер су *inode* бројеви јединствени само у оквиру једног система датотека).

Пример симболичке везе:

```
prompt> echo hello > file
prompt> ln -s file file2  // опција -s прави симболичку везу
prompt> cat file2
hello
```

Симболичке везе (наставак)

Која је разлика између симболичких веза и тврдих веза?

- Симболичке везе су посебна врста датотека (разликују се од датотека и директоријума). Таква веза само указује на оригиналну датотеку. Величина меке везе је једнака величини путање до оригиналне датотеке.

Пример симболичке везе :

```
prompt> ls -al
drwxr-x--- 2  pera pera 29   May 3 19:10 ./
drwxr-x--- 27 pera pera 4096 May 3 15:14 ../     // директоријум
-rw-r----- 1  pera pera 6    May 3 19:10 file    // обична датотека
lrwxrwxrwx 1  pera pera 4    May 3 19:10 file2 -> file // симбол. веза
```

```
prompt> stat file
... regular file ...
prompt> stat file2
... symbolic link ...
```

Симболичке везе (наставак)

Када се уклони оригинална датотека, симболичке везе не указују ни на шта.

Ово је тзв. висећа референца (*dangling reference*):

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

Битови дозвола

Као што код CPU и меморије сваки процес има илузију да је то све само његово (а у стварности ОС мора да користи различите технике како би осигурао да се све дешава на безбедан и сигуран начин), тако се и код система датотека пружа виртуалан поглед на диск.

Међутим, овде се јавља велика разлика, код дискова су датотеке дељене између различитих корисника и процеса, и нису (барем не увек) приватне. Због овог постоји скуп механизама који осигуравају да постоје различити степени дељења датотека.

Један такав механизам су битови дозвола које користе UNIX системи.

Битови дозвола – наставак

```
prompt> ls -l foo.txt
-rw-r--r-- 1 despot d 0 Aug 24 16:29 foo.txt
```

Шта представља овај излаз?

Први карактер означава тип датотеке (овде је – јер се ради о обичној датотеци, за директоријуме је d, а за симболичке везе је l)

Након тога следе битови дозвола, подељени су у три групе од по три бита: права власника, права групе којој припада власник, и права свих осталих.

Погледајте материјал из директоријума „**линукс**“ да бисте додатно разјаснили ово.

Битови дозвола – пример

Бит дозволе извршавања је нарочито занимљив. Испод је пример скрипте која може да се извршава.

```
prompt> ./zdravo.sh
zdravo, shell svete.
```

Уколико променимо дозволу тако да више ни власник нема право извршавања, нећемо моћи да је покрећемо.

```
prompt> chmod 600 zdravo.sh
prompt> ./ zdravo.sh
./hello.sh: Permission denied.
```

Много система датотека

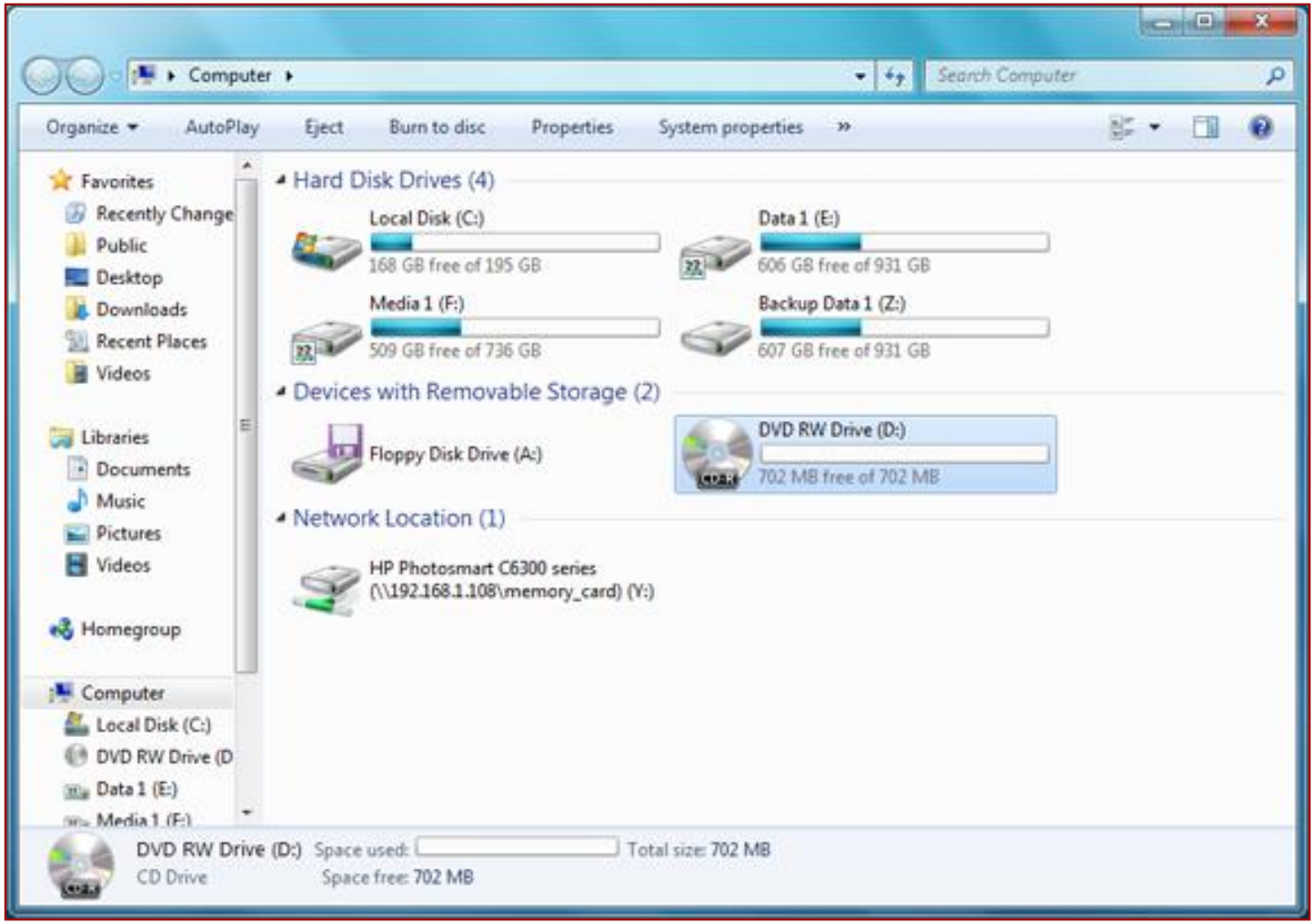
Корисници често желе да користе много различитих система датотека истовремено.

Нпр.

- Главни диск
- Backup диск
- AFS
- Уређаји са флеш меморијом
- …

Који је најпаметнији начин да се ово постигне?

Много система датотека – први приступ



Много система датотека – други приступ

Идеја: убацити све системе датотека у један велики корени систем датотека.

```
prompt> mount
/dev/sda1 on / type ext4 (rw)
/dev/sdb1 on /backups type ext4 (rw)
AFS on /home type afs (rw)
```

Пример

/dev/sda1 је у **/**

/dev/sdb1 је у **/backups**

AFS је у **/home**