Lesson 5 – SC for FS

- SC **za postojeće datoteke:**
  - **open**
  - **read**
  - **write**
  - **lseek**
  - **close**
- SC za **kreiranje novih datoteka** kao što su:
  - **creat**
  - **mknod**
- SC koji manipulišu inodovima i FS kao celinom kao što su:
  - **chdir, chroot, chown, chmod, stat, fsstat**
- **advanced SC**:
  - **pipe** and **dup** koji su značajni za implementaciju pipeline komandi u shell-u.
  - **mount** i **umount** SC
  - **link** i **unlink** SC
- Uvešćemo **kernelske strukture** podataka:
  - **FT** **(file table)**
  - **UFDT** **(user file descriptor table)**
  - **MT** **(mount table)**

Na ovom casu

- SC **za postojeće datoteke**
  - **open**
  - **read**
  - **write**
  - **lseek**
  - **close**
- SC za **kreiranje novih datoteka** kao što su:
  - **creat**
  - **mknod**
- SC koji manipulišu inodovima i FS kao celinom kao što su:
  - **chdir, chroot, chown, chmod, stat, fstat**

table of SC

FS calls

Return File Descriptor	Use of namei	Assign inodes	File Attributes	File I/O	File Sys Structures	Tree Manipulation
open stat dup pipe close	open stat creat link chdir unlink chroot mknod chown mount chmod umount	creat mknod link unlink	chown chmod stat	read write lseek	mount umount	chdir chown
	Lower Level FS algorithm					
	namei iget iput		ialloc ifree		alloc free bmap	
	buffer allocation algorithm					
	getblk brelse bread breada bwrite					

SC kategorije

- SC se mogu klasifikovati u sledeće kategorije:
  - SC koji vraćaju **fajl deskriptore** koje mogu da koriste drugi SC
  - SC koji koriste **namei** algoritam da **razbija path name** do **inode**
  - SC koji **dodeljuju i oslobađaju inodove** preko algoritama **ialloc** i **ifree**
  - SC koji **setuju** ili **menjaju atribute datoteke**
  - SC koji **obavljaju I/O operacije preko algoritama alloc, free** i **baferskih** alokacionih algoritama
  - SC koji **menjaju strukturu FS**
  - SC koji omogućavaju procesu da **promeni svoj izgled stabla**

open SC - syntax

- Open je SC je **prvi korak** koji proces mora obaviti da bi pristupio datoteci.
- Sintaksa za open SC je:
- **fd = open(pathname, flags, modes)**
  - **pathname** ime datoteke,
  - **flags** ukazuju na tipove otvaranja (za čitanje ili upis)
  - **modes** daje ACL ako se datoteka kreira

- SC open daje jedan integer **fd** koji se naziva **user file descriptor** i koji se kasnije koristi za druge file operacije kao što su:
  - reading
  - writing
  - seeking
  - duplicating of file descriptors
  - setting a file I/O parameters
  - određivanje file statusa
  - zatvaranje datoteke

open SC - algorithm

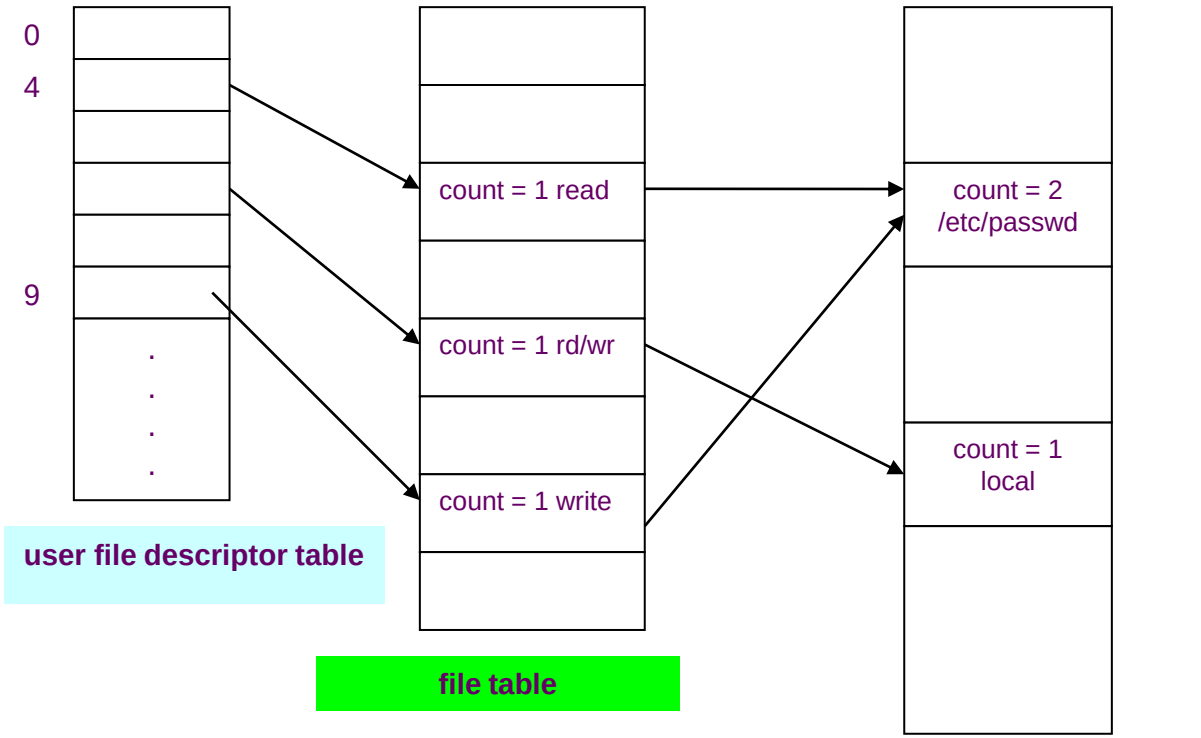
- **algorithm open**
  - inputs: {filename, type of open, file permissions (for creation type of open)}
  - output: file descriptor
- {
- **convert file name to inode** (algorithm **namei**) #iget produce in-core inode
-
- **if(file does not exist** or **not permitted access**) return(**error**);
- **allocate file table entry** for **inode**, initialize count, offset; **#FT**
- **allocate user file descriptor entry**, set pointer to file table entry; **#UFTD**
-
- **if** (**type of open** specifies **truncate** file) **free all file blocks** (algorithm **free**)
-
- **unlock(inode);**
- **return(user file descriptor);**
- }

open SC - description

- Kernel pretražuje FS za zadatu datoteku preko algoritma **namei**,
  - koji ako nađe datoteku kroz sve path grane
  - proverava ACL
  - otvara inode u memoriju, kreira in-core inode
- open SC otvara jedan ulaz u **FT** za tu otvorenu datoteku.
- Taj ulaz u **FT** sadrži sledeće informacije:
  - **pointer** na **in-core inode**
  - **offset** gde kernel očekuje da bude sledeći upis ili čitanje iz datoteke.
  - Ovo polje kernel prilikom **open** SC postavlja na 0, a kasnije akcije sa datotekom ga modifikuju.
  - Proces može otvoriti datoteku u **write-append** modu u kom slučaju kernel inicijalizuje offset na kraj datoteke **= filesize**.
- Pored globalne FT, kernel preko **u-area** inicijalizuje i **UFDT**, a pointer na **UFDT** postavlja u **u-area**. Naravno da **UFDT** mora ukazivati na datoteku u **FT**.

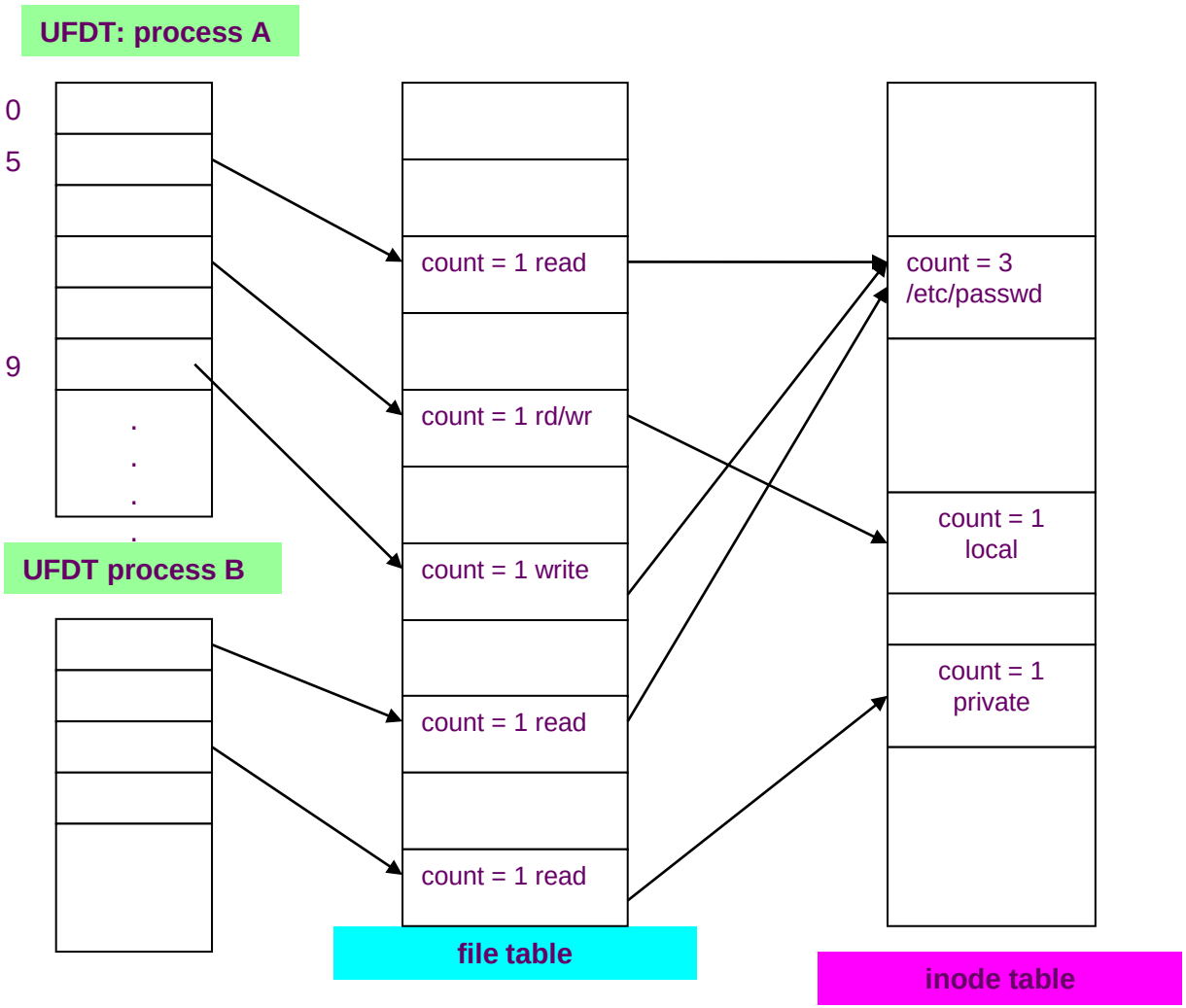
open example

- Pretpostavimo da se dogode 3 open SC:
  - **fd1= open("/etc/passwd", O_RDONLY)**
  - **fd2 = open("/etc/local", O_RDWR)**
  - **fd3 = open("/etc/passwd", O_WRONLY)**
- **Svaki openSC** vraća poseban **FD** procesu u odgovarajući ulaz u **UFDT** koji ukazuje na odgovarajući ulaz u **FT** i u slučaju da je datoteka otvorena više puta. **Sve otvorene instance** za istu datoteku **ukazuju** na **jedinstveni in-core inode.**



open example

- Pretpostavimo da **drugi proces B** izvršava sledeći kod:
  - fd1= open("/etc/passwd", O_RDONLY)
  - fd2 = open("private", O_RDONLY)



FT v UFDT

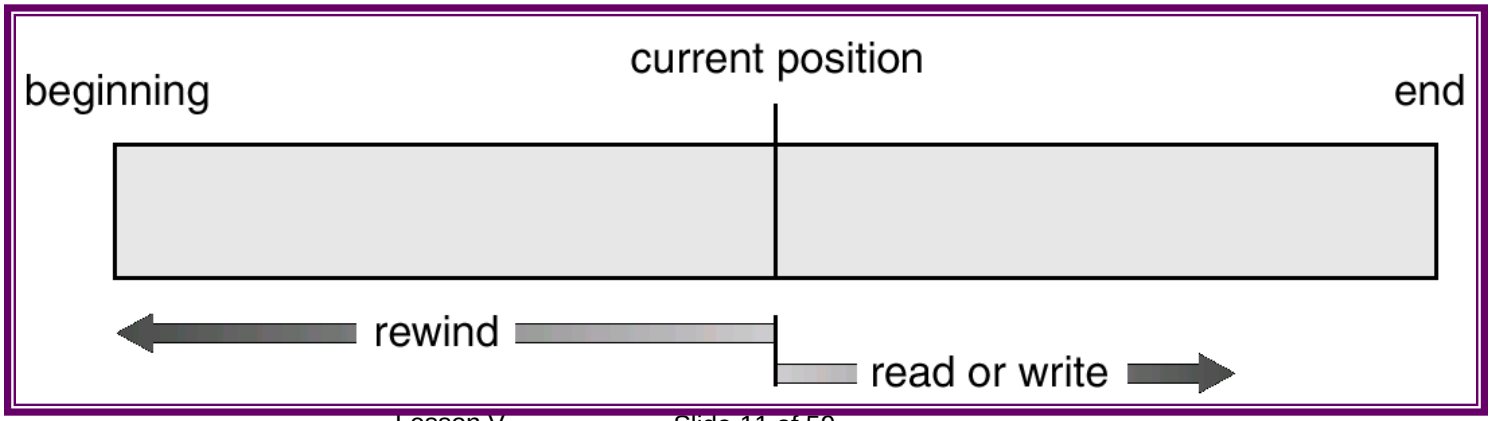
- Opet a slike vidimo da svaki **open** ima
  - jedinstveni ulaz u **UFDT** za proces
  - jedinstveni ulaz u kernelovoj **FT**
  - a samo jedan **in-core inode** za svaku otvorenu datoteku.
- **UFDT** ulazi bi mogli da sadrže
  - **offset** za sledeće **file I/O operacije** i
  - **pointer** na **in-core** inode,
  - tako da FT ne bi trebalo da postoji,
- međutimFT prisustvo je opravdano jer omogućava
  - **deljenje file descriptora fd**
  - **deljenje pointera**
  - što se koristi u **dup** i **fork** SC**.**
- Prva **3 korisnička fd** su (0, 1 i 2) i predstavljaju:
  - **standarni ulaz** **stdin**
  - **standarni izlaz** **stdout**
  - **izlaz za greške** **stderr**

read SC - syntax

- Sintaksa za **read SC** je:

- **number = read(fd, buffer, count)**

- pri čemu je:
  - **fd file descriptor** dobijen od open SC
  - **buffer** je adresa memorijskog bafera koje je proces dobio za čitanje
  - **count** je broj bajtova koji se čita iz datoteke
  - **number** je broj koji se upravo pročitao preko ovog read SC



read SC - algorithm

- **algorirthm read**
- **inputs:**
- {user file descriptor, address of buffer in user process, number of byte to be read }
- **output:** count of bytes copied into user space

- {
- get **file table** entry from **user file descriptor**;
- check **file accessibility**;
- **set parameters** in **u area** for **user address**, **byte count**, **I/O to user**;
- **get inode** from FT; **#in-core inode**
- **lock inode**;
- **set byte offset** in **u-area** from **file table offset**;
-

read SC - algorithm

- while (**count not satisfied**)
- {
- convert file offset to disk block (algorithm **bmap**)
- calculate offset into block, number of byte to read;
- if (**number of bytes to read is 0**) break; /* **end of file***/
-
- read block (algorithm **breada** if with read ahead, **bread** otherwise);
- copy data from **system buffer** to **user address**;
- update **u area fields** for file
  - **byte offset**,
  - **read count**,
  - **address to write into user space**
- **release buffer** /* locked in bread*/
- }
- **unlock inode**;
- **update FT offset** for next read;
- **return** (total number of bytes read);
- }

loop for reading:
reading block by block
through cache

reading done

read SC - description

- Kernel prvo na bazi **fd** uzima **ulaz UFTD**. Tada se setuju I/O parametri u **u-arei**, konkretno kod read SC poziva, se postavlja da je **read I/O**, **count** i postavlja se **user bafer adresa**, a iz njega čita ulaz u FT, iz koga se dobija **offset** u file koji se takođe setuje u u-arei.
- Polja koja se postavljaju u **u-arei** su:

mode	indicates read or write
count	count of bytes to be read or write
offset	byte offset in file
address	target address to copy data in user or kernle memory
flag	indicates if address is in user or kernel memory

- Pored **offseta**, kernel iz FT čita ukazivač na **in-core** inode, pronalazi taj inode, **lock-uje** ga i počinje **kroz keš** da **čita datoteku**.
- **Čitanje** se odvija u **petlji** na sledeći način:
  - na bazi offseta se određuje broj bloka preko algoritma **bmap**
  - blok se čita u keš preko **bread** algoritma
  - modifikuju se polja u u-arei, count i offset
  - petlja se ponavlja sve dok se ne pročita zahtevani broj bajtova
- Na kraju se setuje **novi offset** za datoteku u FT, koji se može sa **lseek()** SC modifikovati

read - example

```
#include <fcntl.h>
main()
{
int fd;
char litlbuf[20], bigbuf[1024];
fd =  open("/etc/passwd", O_RDONLY);
read (fd, litlbuf, 20);
read (fd, bigbuf, 1024);
read (fd, litlbuf, 20);
}
```

- Prvi open će otvoriti datoteku i postavti offset na 0.
- **Prvi read** će pročitati 20 bajtova i postaviti ofset na 20. Naravno u keš ide 1K, a iz keša 20 bajtova. U u-arei se setuje count=0 (read je zadovoljen), offset 20, a potom se ukupan broj pročitanih bajtova setuje na 20.
- **Drugi read** čita 1K ali u offsetu 20, i skoro sve je u kešu (ako je malo vremena proteklo između prvog i drugog read-a) ali tu ima samo 1004 potrebna podatka, opet se ide na inode na sledeći direktni blok, pa se sa diska čita sledeći blok datoteke i u kešu se nalaze 2 bloka datoteke. U bigbuf se u 2 iteracije prebacuju 1024 bajta, offset = 1044.
- **Treći read** čita 20 bajta na ofsetu 1044 koji je u kešu (najverovatnije) i postavlja finalni offset na 1064.

read SC - discussion

- Kernel će na bazi read count-a da proceni da li da radi **bread** ili **breada**.
- Ukoliko vrednost pointera inodu = 0, tada nema čitanja sa diska već se korisnički bafer puni nulama (**sparse file**)
- Inode je **locked** dok traje read, jer bi moglo doći do inkonzistenicje podataka.
- Pored toga uvodi se pojam **lock-ovanja file/record**, da vi obezbedila konzistentnost podataka
- kao u primeru gde su 2 procesa, koja su istovremeno otvorila istu datoteku, a jedan ima 2 read-a a drugi 2 write.
- Šta će da pročitati prvi proces, zavisi od redosleda rd1, rd2, wr1, wr2 ili rd1, wr1, rd2, wr2, ili rd, wr1, wr2, rd2. Zato se pored otvaranja datoteke uvodi lock na record, tako da jedan proces ne može **otvoriti locked file**.

2 processes: 2 x read, 2 x write

```
#include <fcntl.h>
/*process A*/
main()
{
int fd; char buf[512];
fd = open("/etc/passwd", O_RDONLY)
read (fd, buf, sizeof(buf))          /*read 1*/
read (fd, buf, sizeof(buf))          /*read 2*/
}
/*process B*/
main()
{
int fd,i; char buf[512];
for (i=0; i<sizeof(buf); i++) buf[i] = ‘a’;
fd = open("/etc/passwd", O_WRONLY)
write (fd, buf, sizeof(buf))         /*write 1*/
write (fd, buf, sizeof(buf))         /*write 2*/
}
```

one process two independent reads

- Na sledećem primeru proces može otvoriti datoteku 2 puta i čitati je preko 2 različita deskriptora sa nezavisnim file offsetima.

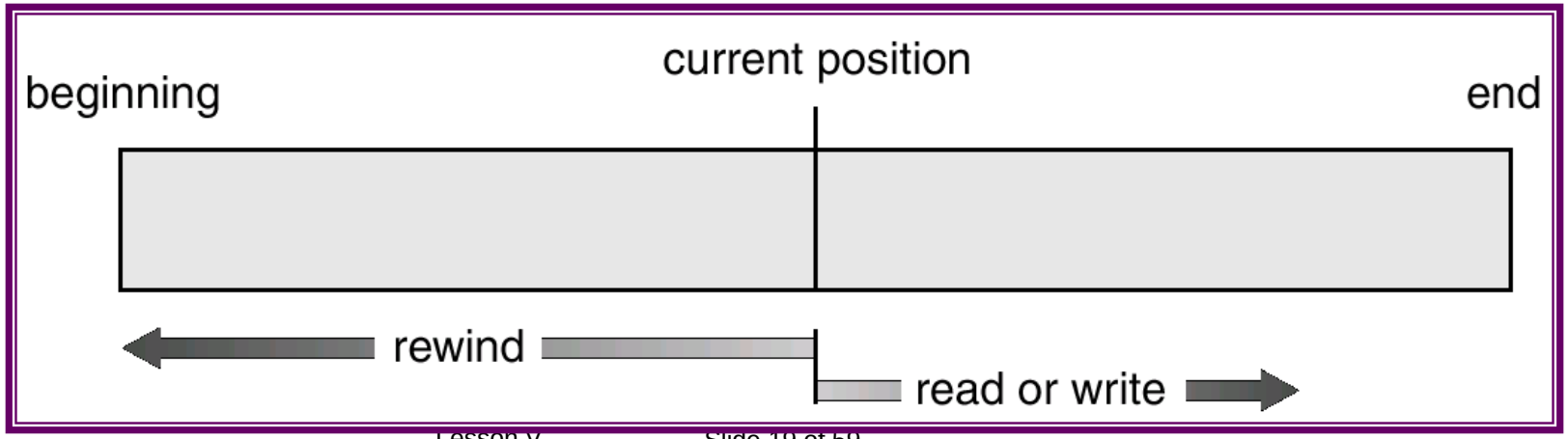
```
#include <fcntl.h>
main()
{
int fd1, fd2;
char buf1[512], buf2[512];

fd1 = open("/etc/passwd", O_RDONLY)
fd2 = open("/etc/passwd", O_RDONLY)

read (fd1, buf1, sizeof(buf1))          /*read 1*/
read (fd2, buf2, sizeof(buf2))          /*read 2*/
}
```

write SC - syntax

- Sintaksa za **write SC** je:

- **number = write(fd, buffer, count)**

- pri čemu su parametri isti ako za read SC
  - **fd file descriptor** dobijen od open SC
  - **buffer** je adresa memorijskog bafera koje je proces dobio za upis
  - **count** je broj bajtova koji se upisuje u datoteku
  - **number** je broj koji se upravo upisan preko ovog read SC



write SC - syntax

- Algoritam za **write** SC je **sličan** algoritmu za **read SC**,
  - izuzev ako datoteka ne sadrži blok koji odgovara file offsetu za upis,
  - tada kernel alocira novi blok preko algoritma **alloc** i
  - podešava inode.
  - Ako je file offset odgovara indirektnom bloku, kernel mora dodeliti više blokova koji će se koristiti za indirektne i data blokove.
- **Inode is locked** za vreme write(SC), zato što kernel može promeniti inode u toku write SC. Kada se upis završi, kernel ažurira inode pointere i filesize polje ako je došlo do promene veličine.
- **Kada kernel obavlja upis**, dešava se kao kod read ciklusa pri čemu su moguće sledeće situacije.
  - **Jedna potencijalna situacija** je da se **upisuje ceo blok u jednoj u iteraciji**.
  - **Druga situacija** je da se u **nekoj iteraciji upisuje deo bloka**, tada prvo blok mora da se učita u memoriju u keš blok, pa se prepiše deo bloka, a onda ceo blok se upisuje.
  - **Treća situacija** je kada mora **prvo da se alocira indirektni blok** pa tek onda upis.
- Koriste se **delayed write** kroz kernelski keš **koji je efikasan za pipe datoteke** i i **datoteke koje se privremeno kreiraju** i čitaju se veoma skoro posle toga, kao što su **privremene datoteke koje kreira editor teksta i briše ih**. Dakle **najveći uspeh se postiže** za upis koji će **živeti samo u kešu** i nikada neće dostići disk.

File i record locking

- Originalni UNIX od Thomsona i Ritchija nije imao interni mehanizam koji obezbeđuje eksluzivni pristup datoteci. Locking mehamizam se nije koristio, jer kako kaže Ritchie, nisu imali na svom sistemu veliku bazu sa više nezavisnih procesa.
- Da bi UNIX prililižila database serverima, UNIX System V sadrži u sebi file i record locking mehanizam.
- **file locking** mehanizam zabranjuje pristup (r/w) datoteci ostalim procesima, osim tom jednom koji je lock-ovao datoteku

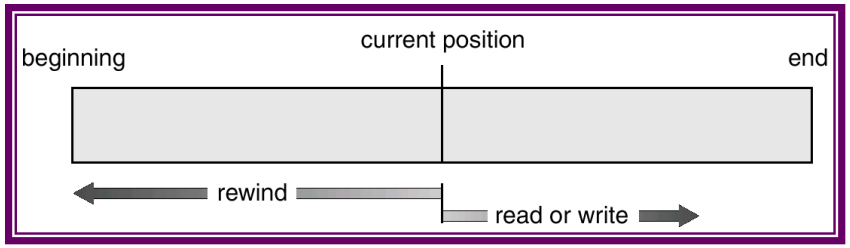
- **record locking** mehanizam zabranjuje pristup (r/w) recordu ostalim procesima, osim tom jednom koji je lock-ovao record.
- **Record** je **logička celina** u **datoteci**, može biti bilo koji deo datoteke, ima svoj offset i veličinu.

Podešavanje pozicije u datoteci - lseek

- Obično korišćenje **read i write SC** obezbeđuje sekvencijalni pristup datoteci, dok **lseek** SC omogućava random pristup datoteci.

- Sintaksa za lseek SC je:

- **position = lseek(fd, offset, reference)**



- pri čemu su parametri:
- **fd file descriptor** dobijen od open SC
- **offset** je je relativni bajt offset
- **reference** ukazuje u odnosu na na šta je offset:
  - u odnosu na **početak datoteke** (0)
  - u odnosu na **tekuću poziciju** (1)
  - u odnosu na **kraj datoteke** (2)
- **position** je offset u kome će sledeće čitanje ili upis da se dogode

lseek - example

- #include <fcntl.h>
- **/*process A*/**
- main(argc, argv)
- int argc
- char *argv[]
- {
- int fd, skval;
- char c;
- if (argc != 2) exit();
- **fd = open(argv[1], O_RDONLY);**
- if (fd == -1) exit();
- while ((skval = **read**(fd, &c, 1)) == 1)
- {
- printf("char %c\n",c)
- skval = **lseek**(fd, 1023L, 1);
- printf("new seek val %d \n",skval)
- }
-
- Program čita svaki 1024 bajt datoteke, tako što prvo pročita jedan bajt, onda obavi **lseek** za 1023. **SC lseek** praktično ne radi ništa sa diskom, on jednostavno podesi offset u FT, koji će se korisititi u sledećoj r/w instanci.

close SC

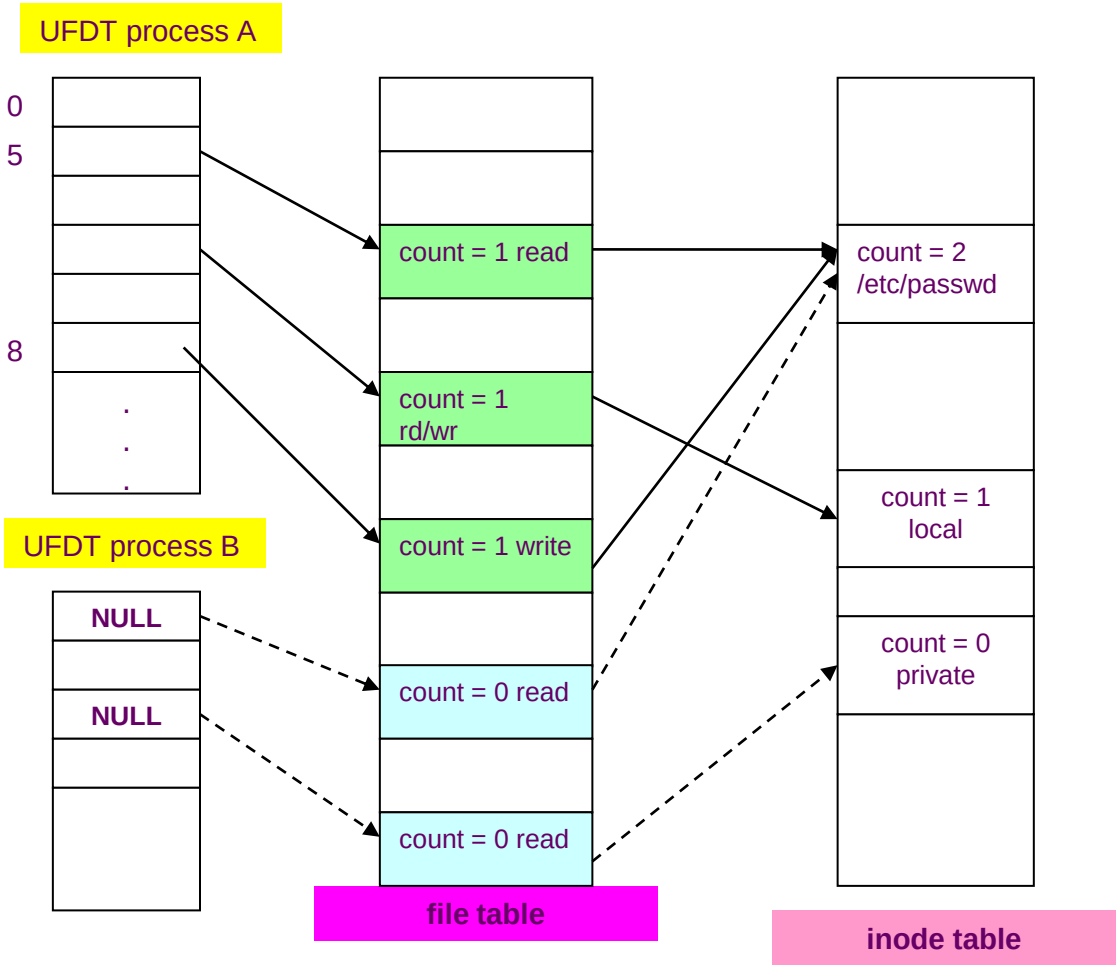
- Proces zatvara otvorenu datoteku kada više ne želi da joj pristupa, preko close SC.
- Sintaksa za close SC je krajnje jednostavna:
- **close(fd);**

- Kernel obavlja **close operaciju** tako što obavi manipulacije u **UFDT**, **FT** i **in-core** inode.
  - Ako je **reference count** u **FT** veći od 1, zbog dup ili fork SC, kernel će dekremenirati broj u FT.
  - Ako taj broj posle dekrementa padne na 0, kernel **oslobađa ulaz** u **FT** i otpušta in-core inode (**iput**), originalno kreiran nakon open SC.
  - Ako ostali procesi takođe koriste taj **in-core inode**, broj RC se dekrementira.
  - Nakon toga se se briše ulaz u **UFDT**.

- Kada proces obavi **exit(SC)** sve njegove otvorene datoteke se moraju zatvoriti.

process B closes all own files

- Za **in-core** inode za private datoteku, **RC** pada na nulu čime se otpušta taj inode, **/etc/passwd** datoteku FC pada na 2, oslobađaju se 2 ulaza u FT koja su pripadala procesu B, a njegovi **UFTD** ulazi se anuliraju.



Primeri sistemskih pozivi na Linux OS

Linux U/I sistemski pozivi nizeg nivoa

- **viši SC**
  - **fopen**
  - **fprintf, fputc, fputs i fwrite** koje **upisuju** podatke u datoteku (stream)
  - **fscanf, fgetc, fgets i fread** koje **ucitavaju** podatke.

- Datoteka (stream) je predstavljena kao **FILE* pokazivac**. Kada otvorite datoteku sa fopen vi dobijate FILE* pokazivac. Kada zavrsite mozete je zatvoriti sa fclose.
- **low-level SC**

- Sa Linux-ovim operacijama nizeg nivoa, koristimo postupak nazvan **deskriptor datoteke** umesto FILE* pokazivaca.

- Deskriptor datoteke je celobrojna vrednost koja upucuje na odredjenu datoteku u pojedinacnom procedu.

- Moze biti otvoren za citanje, za pisanje ili i za citanje i za pisanje.
- Deskriptor datoteke ne mora da upucuje na otvorenu datoteku; moze predstavljati vezu sa drugom sistemskom komponentom koja moze da prima i salje podatke.

SC open na Linux OS - flagovi

- **fd=open (pathname, flags, modes)**
- **flags**
  - **O_RDONLY**, datoteka je otvorena samo za citanje
  - **O_WRONLY** stvara da bude samo za upis.
  - **O_RDWR** deskriptor datoteke se navodi da bude upotrebljena i za upis i za citanje.
- Mozete navesti dodatne funkcije koriscenjem bita operacija ili sa ovom vrednoscu sa jednim ili vise indikatora. Ovo su najcesce koriscene vrednosti:
- **O_TRUNC** da anulirate datoteku, ako prethodno postoji. Podatci upisani u deskriptor datoteke ce zameniti prethodni sadrzaj datoteke.
- **O_APPEND** da dodate u postojecu datoteku. Podatci upisani u deskriptor datoteke ce biti dodati na kraj datoteke.
- **O_CREAT** da kreirate novu datoteku. Ako ime datoteke koje navedete ne postoji, nova datoteka ce biti kreirana. Ako datoteka vec postoji, onda ce se samo otvoriti.
- **O_EXCL sa O_CREAT** da nasilno kreirate novu datoteku. Ako datoteka vec postoji, poziv open ce biti bezuspesan.

SC open na Linux OS (modovi)

- Ako pozovete open sa O_CREAT, dodajte treci dodatni argument zadajuci dozvole za novu datoteku.
- **Umasks**
- Kada kreirate novu datoteku sa open, neki bitovi prava pristupa koje ste predhodno namestili mogu biti iskljuceni. To je zato sto je vasa umaska podesena na ne nulte vrednost. Procesov umask odredjuje bitove koji su maskirani van svih novokreiranih dozvola. Aktuelna koriscena prava prtistupa su nivoi bita i pristipna prava koja odredimo za otvaranje i nivoi bita dopunjeni sa umask.

- Da bi promenili vas umask iz komandnog interpretera, koristite umask komandu, i odredite numericku vrednost maske, u oktalnom sistemu. Da bi promenili umask za tekuci proces, koristite funkciju umask, prosledjujuci mu zeljenu vrednost kako bi je koristili za sledecu open funkciju.

- Na primer, pozivanjem ovog reda u programu
- **umask (S_IRWXO | S_IWGRP);**

- ili pozivajuci ovu komandu
- **$ umask 027**
- odredjujete dozvolu za pisanje za clanove grupe i citanje, pisanje i pristup drugima ce uvek biti maskiran van novih dozvola.

(create-file.c)
Kreiranje nove datoteke

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
/* Putanja gde ce se kreirati nova datoteka. */
char* path = argv[1];
/* Dozvole za novu datoteku. */
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH;
/* Kreira datoteku. */
int fd = open (path, O_WRONLY | O_EXCL | O_CREAT, mode);
if (fd == -1) {
/* Dolazi do greske. Ispisuje se poruka o gresci. */
perror (“open”);
return 1;
}
return 0;
}
```

(create-file.c)
Kreiranje nove datoteke

- program u akciji:

- **$ ./create-file testfile**
- **$ ls -l testfile**
- -rw-rw-r-- 1 samuel users 0 Feb 1 22:47 testfile

- **$ ./create-file testfile**
- open: File exists

- Primetite da je velicina nove datoteke 0 zato sto program nije upisao nikakve podatke u nju.

read SC: hexdump.c

- Listing **hexdump** prikazuje prostu demonstraciju funkcije read.
- Program ispisuje heksadecimalne ostatke sadrzaja datoteke, precizirane na comandnoj liniji.
- Svaki red prikazuje
  - pomeraj u datoteci
  - i
  - sledecih 16 bajtova.

read SC: hexdump.c

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])

{
unsigned char buffer[16];

size_t offset = 0;
size_t bytes_read;

int i;
```

read SC: hexdump.c

```
/*Otvara datoteku za citanje*/
int fd = open (argv[1], O_RDONLY);

do { /*Citanje bafera. */
bytes_read = read (fd, buffer, sizeof (buffer));

/*Pisanje pomeraja u datoteku, pracen je sa samim bajtovima*/
printf (“0x%06x : “, offset);
for (i = 0; i < bytes_read; ++i)
printf (“%02x “, buffer[i]);
printf (“\n”);
/*Pamti nasu poziciju u datoteci */
offset += bytes_read;
} while (bytes_read == sizeof (buffer));
/*Sve gotovo.*/
return 0;
}
```

read SC: hexdump.c

- hexdump u akciji.
- Prikazuje ispis izvrsne datoteke hexdump:

- **$ ./hexdump hexdump**
- 0x000000 : 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
- 0x000010 : 02 00 03 00 01 00 00 00 c0 83 04 08 34 00 00 00
- 0x000020 : e8 23 00 00 00 00 00 00 34 00 20 00 06 00 28 00
- 0x000030 : 1d 00 1a 00 06 00 00 00 34 00 00 00 34 80 04 08
- ...

- Vas izlaz moze igledati drugacije,
- u zavisnosti od prevodioca koji koristite
- da prevedete hexdump
- i prevedenih(kompajliranih) indikatora (flags) za prevodjenje koje ste selektovali.

write SC: timestamp.c

- Program u timestamp.c dodaje tekuce vreme u datu datoteku.
- Ako datoteka ne postoji, kreirace se.

- Ovaj program takodje koristi
  - time
  - localtime
  - asctime
- funkcije za dobijanje i formatiranje tekuceg vremena.

write SC: timestamp.c

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

/* Vraca string koji predstavlja tekuci datum i vreme. */
char* get_timestamp ()
{
time_t now = time (NULL);
return asctime (localtime (&now));
}
```

write SC: timestamp.c

- int main (int argc, char* argv[]) {
- /* Datoteka u koju treba da se doda vremenska oznaka. */
- char* filename = argv[1];
- /* Uzima tekucu vremensku oznaku */
- char* timestamp = **get_timestamp ();**

- /* Otvara datoteku za upis. Ako postoji, pristupa joj. */
- int fd = **open**(filename, O_WRONLY | O_CREAT | O_APPEND, 0666);

- /* Racuna duzinu timestamp stringa. */
- size_t length = strlen (timestamp);

- /* Upisuje timestamp u datoteku. */
- **write**(fd, timestamp, length);

- /* Sve gotovo. */
- **close** (fd);
- return 0; }

write SC: timestamp.c

- Evo kako timestamp program radi:

- **$ ./timestamp tsfile**
- $ cat tsfile
- Thu Feb 1 23:25:20 2001

- **$ ./timestamp tsfile**
- $ cat tsfile
- Thu Feb 1 23:25:20 2001
- Thu Feb 1 23:25:47 2001

- Zapamtite da prvi put kad pozovemo timestamp,
- on kreira tsfile,
- dok kad drugi put pozovemo taj program
- on mu samo dodaje vreme.

lseek

- **off_t position = lseek (file_descriptor, offset, SEEK_xxx);**
- **lseek** funkcija vam omogucava da pozicionirate deskriptor datoteke u datoteci. Prosledite ga deskriptoru datoteke i dva dodatna argumenta, **offset** i **SEEK_SET,** koji odredjujuci novu poziciju.
- Ako je **treci argument**
  - **SEEK_SET**, lseek tumaci drugi argument kao poziciju, u bajtovima, od pocetka datoteke.
  - **SEEK_CUR**, lseek tumaci drugi argument kao pomeraj , koji moze biti pozitivan ili negativan, od trenutne pozicije.
  - **SEEK_END**, lseek tumaci drugi argument kao pomeraj od kraja datoteke. Pozitivna vrednost pokazuje poziciju izvan kraja datoteke.
- Funkcija **lseek** vraca novu poziciju, kao pomeraj od pocetka datoteke.
- tip pomeraja je **off_t.**
- Ako se pojavi greska, **lseek** vraca -1.
- **lseek** ne mozete koristit sa nekim vrstama deskriptora datoteka, kao sto je uticnica(socket) deskriptor datoteke.

lseek iza kraja datoteke

- Linux vam omogucava da lseek funkcijom pozicionirate deskriptor datoteke izvan datoteke.
- Normalno, ako je deskriptor datoteke pozicioniran na kraju datoteke i vi nesto upisujete u njega, Linux ce automatski da produzi datoteku kako bi napravio mesta za nove podatke. Ako pozicionirate deskriptor datoteke izvan datoteke i onda nesto upisujete u njega:
- Linux prvo prosiruje datoteku kako bi prilagodio rupu koju ste napravili sa lseek operacijom i onda upisuje na kraj datoteke.
- Ova praznina, kako bilo, **ne zauzima mesto na disku**; naprotiv, Linux samo pravi poruku koliko je to veliko.
- Ako kasnije pokusate da procitate do iz datoteke, vasem programu ce izgledati kao da je ta praznina popunjena sa 0 bajtovima.

lseek primer -lseek-huge.c

- Koriscenjem ovog ponasanja **lseek**,
  - moguce je kreirati veoma velike datoteke
  - koje ne zauzimaju skoro nikakav prostor na disku.
- Program **lseek-huge** radi to.
- On uzima sa komandne linije
  - ime datoteke i
  - velicinu trazene datoteke u MB.
- Program otvara novu datoteku,
  - pozicinira se do kraja datoteke koristeci lseek,
  - i
  - onda upise jedinstveni 0 bajt pre nego sto ga zatvori.

lseek primer -lseek-huge.c–početak koda

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main (int argc, char* argv[])
{

int zero = 0;

const int megabyte = 1024 * 1024;

char* filename = argv[1];

size_t length = (size_t) atoi (argv[2]) * megabyte;
```

lseek primer -lseek-huge.c-nastavak koda

- /*Otvaranje nove datoteke */
- int **fd** = **open** (filename, O_WRONLY | O_CREAT | O_EXCL, 0666);
- /* Prelazimo na 1 bajt manje od mesta gde hocemo da bude kraj datoteke. */
- **lseek (fd, length - 1, SEEK_SET);**
- /* Pisemo jedinstven 0 bajt*/
- **write (fd, &zero, 1);**
- /*Sve gotovo.*/
- **close (fd);**
- return 0;
- }

lseek primer -lseek-huge.c

- Koriscenjem lseek-huge, napravicemo datoteku od 1 GB(1024MB).
- Proverite slobodno mesto na disku pre i posle operacije.
- **$ df -h .**
- Filesystem Size Used Avail Use% Mounted on
- /dev/hda5 2.9G 2.1G 655M 76% /

- **$ ./lseek-huge bigfile 1024**
- % ls -l bigfile
- -rw-r----- 1 samuel samuel 1073741824 Feb 5 16:29 bigfile

- **$ df -h .**
- Filesystem Size Used Avail Use% Mounted on
- /dev/hda5 2.9G 2.1G 655M 76% /

lseek primer -lseek-huge.c

- Neprimetna kolicina memorije na disku je potrosena, uprkos nenormalnoj velicini bigfile.
- Opet, iako otvorimo bigfile i citamo iz njega, on ce izgledati kao da je popunjen sa nulama u vrednosti od 1GB.
- Za primer, mozemo da testiramo njegov sadrzaj sa hexdump programom iz predhodnog primera.
- **$ ./hexdump bigfile | head -10**
- 0x000000 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
- 0x000010 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
- 0x000020 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
- 0x000030 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
- 0x000040 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
- 0x000050 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
- ...
- -end of Linux examples

File Creation

- Dok **open** SC otvara postojeće datoteke, **creat SC** kreira novu datoteku.
- Sintaksa za creat SC je:

- **fd = creat(pathname, modes)**

- gde svi parametri imaju isto značenje kao i kod **open** SC.

- **Ako** takva **datoteka ne postoji**, kernel kreira novu datoteku 0-te veličine, pod zadatim imenom i sa zadatim pravima.
- **Ako datoteka postoji, kernel odseca datoteku na veličinu 0, oslobađjući sve data blokove.**

creat SC

- **algorithm creat**
- inputs: {filename, file permissions (for creation type of **open**)}
- output: file descriptor

- {
- get inode for file name (algorithm **namei**);
- if (**file already exists**)
- {
- if (not permitted access)
- {
- release inode (algorithm **iput**);
- return(error)
- }
- }

creat SC

- **else /*file does not exist yet*/**
- {
- assign free inode from FS (algorithm **ialloc**);
- create new directory entry in parent directory: include new file name and newly assigned inode number
- }
-
- allocate **file table entry** for **inode**, initialize count, offset; **#FT**
- allocate **user file descriptor entry**, set pointer to file table entry; **#UFTD**
- if (**file did exist at time of create**) free all file blocks (algorithm **free**)
-
- unlock(**inode**);
- **return(user file descriptor**);
- }

creat SC – discussion

- **SC create** može da se izvede sa **open SC** koristeći 2 flag-a:
  - **O_CREAT (create)**
  - **O_TRUNC(truncate)**
- Kernel razbija pathname koristeći algoritam **namei**, dok ne naiđe na zadnju komponentu.
  - Ukoliko u zadnjoj grani nađe zadnje ime ono se mora odseći na 0.
  - U protivnom dešava se kreacija novog objekta u direktorijumu, novi FCB.
- Ako ima mesta u direktorijumskom bloku, dobro u protivnom, direktorijum se mora proširiti sa novim blokom što izaziva brojne akcije (alokaciju, inode promenu..).
- Potom se pronalazi **slobodan inode** za novi datoteku (**ialloc**) koji se inicijalizuje i upisuje na disk preko **bwrite** algoritma.
- Ukoliko je datoteka postojala, kernel zahteva od procesa da ima write pravo na nju da bi mogao da je odseče na 0, svi data blokovi se oslobađaju (**free** algoritam), ali se ne menja vlasništvo i grupa datoteke.
- Na kraju se inicijaliziju po jedan ulaz u **UDFT** i **FT**, kao kod **open** SC.

Kreiranje specijalnih datoteka

- Kreiranje specijanih datoteka, kao što su device files, named pipes i direktorijumi se odvija preko **mknod** SC, koji ima sličnosti sa create tako što dodeljuje novi inode za datoteku.

- Sintaksa za mknod SC je:
- **mknod (pathname, type and permissions, dev)**

- pri čemu je
  - **pathname** ime specijalne datoteke,
  - **type and permissions** daju tip specijalne datoteke (**node, pipe, directory**) i prava pristupa koja joj se dodeljuju
  - **dev** specificira major i minor broj za specijalne blok i karakter datoteke

mknod - algorithm

- **algorithm mknod**
  - **inputs:** {filename, file type, permissions, major and minor device number}
  - **output:** none
- {
- if (new node not name pipe and user **not super user**) return (error);
-
- get inode for file name (algorithm **namei**);
- if (**new node already exist**)
- {
- release parent inode (algorithm **iput**);
- return(**error**)
- }
- else /* **file does not exist yet***/
- {
  - assign **free inode** from FS (algorithm **ialloc**);
  - create new directory entry in parent directory: include new node name and newly assigned inode number;
  - release parent inode (algorithm **iput**);
- }
-
- if (new node is **block** or **character special** file) write major and minor numbers into inode structure;
- release new node inode (algorithm **iput**);
- }

mknod - discussion

- Kernel u direktorijumskoj strukturi traži ime specijalne datoteke koju treba da kreira i
- ako **ime ne postoji**, dodeljuje se novi inode,
- u direktorijumu se upisuje novi FCB.
- u inode se upisuje tip datoteke (node, pipe, directory) i prava pristupa.

- Ukoliko je **nod** u pitanju, upisuju se **major** i **minor** number u inode.

- Za **direktorijum** se moraju alocirati data blok i popuniti sa prva dva ulaza
- **.** **(itself)**
- **..** **(parent)**

- Na kraju se radi **iput** za novu specjilanu datoteku

Change directory, change root

- Kada se sistem podiže,
- **prvi proces** koji se kreira,
  - uzima root direktorijum za svoj tekući direktorijum,
  - tako što obavi **iget** za */*,
  - pa pa ga čuva u u-area kao svoj tekući direktorijum,
  - i obavlja inode unlock (iput).

- Kada proces kreira svoje dete sa fork SC,
- novi proces nasleđuje tekući direktorijum od procesa roditelja,
- a kernel povećava **RC** za inode tekućeg direktorijuma.

chdir - syntax

- Algoritam **chdir** menja tekući direktorijum procesa. Sintkasa za **chdir SC** je:
-
- **chdir(pathname)**
- gde je pathname ime direktorijuma koji će postati novi tekući direktorijum procesa.

- Kernel razbija **pathname po granama** preko **namei** algoritma
- dok ne dođe do zadnje komponente,
- kada proverava prava pristupa na tu granu.
- Ako ima prava pristupa,
  - uzima se inode te grane (iget) i inkremetira RC za novi direktorijum,
  - oslobađa se (iput), izbacuju se stari inode a RC--,
  - a upisuje nov u u-areu procesa.
- Pošto proces **promeni tekući direktorijum**, **njegov inode** je **polazna tačka** za sva pretraživanja (namei) za sve reference koje ne počinju sa apsolutnom putanjom /....

- Inode za tekući direktorijum se otpušta i RC--, samo ako nastupi novi **chdir** ili proces završi aktivnosti (exit)

algorithm change directory (chdir)

- **algorithm change directory (chdir)**
  - input: new directory name
  - output: none
- {
- get inode for new directory name (algorithm **namei**);
- if(inode not that of directory or process not permitted access to file)
- {
  - release inode (algorithm **iput**);
  - return(error);
- }
- unlock inode;
- release "old" current directory inode (algorithm **iput**);
- place **new inode** into current directory slot in **u-area**
- }

chroot

- Procesi podrazumevaju da **/** označava root direktorijum **celog UNIX stabla**
  - to je globalna sistemska varijabla koja sadrži inode root direktorijuma,
  - koji se dobija preko **iget SC prilikom podizanja UNIX-a**.
- Procesi mogu promeniti
  - **svoju oznaku** za **FS root direktorijum** preko **chroot SC**
  - to je **korisno** kada procesi **simulairaju sopstvenu FS hijerarhiju**.
- Sintaksa za **chroot SC** je
- **chroot(pathname)**
  - **pathname** ime direktorijuma koji će se tretirati kao **root direktorijum procesa**.
- Algoritmam za **chroot** je praktično isti kao i za **chdir**.
- Preko **namei** se dobija inode novog root direktorijima koji se upisuje u **u-areu** procesa.
- **Novi inode** je **logički root** za sve **reference koje počinju sa /.**

Change owner, change mode

- Promena vlasništva ili prava pristupa su operacije koje se obavljaju isključivo nad inodom, ništa se sa datotekom ne radi.

- Sintakse su:
- **chown(pathname, owner, group)**
- **chmod(pathname, mode)**

- Da bi promenio vlasništvo, kernel konvertuje pathname u inode preko **namei** algoritma.
- Proces mora biti sa root privilegijom ili da bude vlasnik datoteke, u protivnom ne može.
- Nakon dobijanja inode, kernel modifikuje UID i GID polje, briše set user i set group flagove, a potom otpušta inode sa iput.

- Potpuno ista procedura se radi i za chmod, gde se modifikuju prava pristupa.

stat, fstat

- SC stat i fstat omogućavaju procesu da
- postavi **upit za status datoteke**, a
- **upit će vratiti informaciju** o
  - tipu datoteke,
  - vlasništvu,
  - ACL,
  - broju linkova,
  - broju inoda i
  - vremenima pristupa.

- **Sintakse su:**
- **stat(pathname, statbuffer)**
- **fstat(fd, statbuffer)**
- gde je
  - **pathname** ime datoteke, fd je file descriptor koji je vratio prethodni open SC,
  - **statbuffer** je korisnikov bafer za prijem podataka o statusu datoteke gde će stat i fsfstat da napune podatke.