Osnovni i napredni sistemski pozivi za rad sa datotekama

Sadržaj

Osnovni sistemski pozivi za rad sa datotekama

Verovatno prva U/I funkcija sa kojom se studenti sreću kada započnu učenje programskog jezika C jeste ***printf***. Ova funkcija formatira tekstualni niz (ovo je neki string) i potom ga ispisuje preko standardnog izlaza. Uopštena verzija ove funkcije, ***fprintf***, može da ispiše tekst i u neku datoteku (*stream*), dakle – nije ograničena na štampanje isključivo preko standardnog izlaza.

Datoteka se predstavlja kao **FILE* pokazivač**. Pa tako, kada se datoteka otvori pomoću poziva ***fopen***, nazad se dobija pokazivač FILE*. Kada se završi sa radom, datoteka se može zatvoriti preko ***fclose***.

Pored ***fprint***, mogu se koristiti i druge funkcije, kao što su ***fputc***, ***fputs*** i ***fwrite*** koje upisuju podatke u datoteku, a za čitanje podataka se mogu koristiti funkcije ***fscanf***, ***fgetc***, ***fgets*** i ***fread***.

Sa Linuksovim operacijama nižeg nivoa se koristi **deskriptor datoteke** umesto FILE* pokazivača. Deskriptor datoteke je celobrojna vrednost koja pokazuje na određenu datoteku. Deskriptor može biti otvoren za čitanje, za pisanje, ili i za čitanje i za pisanje. Deskriptor datoteke ne mora da upućuje na otvorenu datoteku; može predstavljati i vezu sa nekom drugom sistemskom komponentom koja ima mogućnost da prima i šalje podatke. Npr, veza između uređaja je predstavljena pomoću deskriptora datoteke kao otvorena utičnica ili kao jedan kraj cevovoda (*pipe*).

Pomenuto je ranije da su U/I funkcije standardne C biblioteke implementirane povrh ovih U/I funkcija niskog nivoa. Ponekad je zgodno:

- da se koriste standardne funkcije sa deskriptorima datoteke,
- ili da se koriste U/I funkcije niskog nivoa na *FILE *stream* strukturi standardne biblioteke.

GNU/Linuks omogućuje obe opcije.

Ukoliko je datoteka otvorena koristeći *fopen*, može se dobiti deskriptor datoteke koristeći ***fileno*** funkciju. Funkcija ***fileno*** uzima argument FILE* i vraća deskriptor datoteke. Na primer, ako se otvori datoteka standardnom *fopen* funkcijom (funkcija visokog nivoa), upis u nju korišćenjem funkcije ***writev*** (funkcija niskog nivoa) može se postići uz pomoć sledećeg koda:

```
FILE* stream = fopen (filename, "w");
int file_descriptor = fileno (stream);
writev (file_descriptor, vector, vector_length);
```

Treba zapamtiti da FILE* *stream* i *file_descriptor* odgovaraju istoj otvorenoj datoteci. Ako se pozove sledeća linija, više se ne može upisivati u tu datoteku tj. u *file_descriptor*:

```
    fclose (stream);
```

Slično, ako se pozove sledeća liniju, više se nećete moći upisivati u *stream* FILE*:

```
close (file_descriptor);
```

Da bi se išlo obrnuto, tj. da se od deskriptora datoteke dođe do *stream* strukture (datoteke) FILE*, koristi se ***fdopen*** funkcija. Ona konstruiše *stream* strukturu FILE* koja odgovara deskriptoru datoteke. Funkcija *fdopen* uzima kao argumente deskriptor datoteke i *string* argument koji specificira način (*mod*) na koji se kreira *stream* struktura. Sintaksa ovog *mod* argumenta ista je kao i drugog argumenta funkcije *fopen*, on mora biti kompatibilan sa deskriptorom datoteke. Na primer, treba naznačiti *r* za čitanje deskriptora datoteke ili *w* za pisanje u deskriptor datoteke. Kao i kod funkcije *fileno*, datoteka (*stream*) i deskriptor datoteke ukazuju na istu otvorenu datoteku, pa ako se zatvori jedan, ni onaj drugi se neće moći koristiti.

Otvaranje datoteke

Da bi se otvorila datoteka i naveo deskriptor datoteke za pristup toj datoteci, može se koristiti funkcija ***open***. Ona uzima za argumente putanju datoteke koju treba otvoriti (putanja se zadaje u vidu slovnog niza) i indikator koji opisuje opcije sa kojima ta datoteka da se otvori. Poziv *open* se može koristiti da se napravi nova datoteka; u tom slučaju treba dodati argumente koji opisuju koja će pristupna prava imati nova datoteka.

Ako je u pozivu *open* drugi argument **O_RDONLY**, datoteka je otvorena samo za čitanje; ako se pokuša upisivanje u nju – doći će do greške. Slično, **O_WRONLY** čini da deskriptor bude samo za upis. Sa **O_RDWR**, deskriptor datoteke se navodi da bi datoteka bila upotrebljena i za upis i za čitanje.

Treba znati da ne mogu sve datoteke biti otvorene na sva tri načina. Na primer, dozvole nad datotekom mogu da zabrane određenom procesu da je otvori za čitanje ili za upis; a datoteka na uređaju koji je predviđen samo za čitanje, kao što je to CD-ROM, ne može biti otvorena za upis.

Možete navesti dodatne opcije, a ovo su najčešće korišćene vrednosti:

- Navedite **O_TRUNC** da odbacite sadržaj otvorene datoteke u slučaju da ona već postoji. Podaci upisani u deskriptor datoteke će zameniti prethodni sadržaj datoteke.

- Navedite **O_APPEND** da dodate sadržaj u postojeću datoteku. Podaci upisani u deskriptor datoteke biće dodati na kraj datoteke.

- Navedite **O_CREAT** da napravite novu datoteku. Ako ime datoteke koje navedete ne postoji, nova datoteka će biti napravljena (pod pretpostavkom da postoji direktorijum koji sadrži datoteku i da proces ima prava da pravi datoteku u tom direktorijumu). Ako datoteka već postoji, onda će ona samo biti otvorena.

- Navedite **O_EXCL** sa **O_CREAT** ukoliko isključivo želite da napravite novu datoteku. Ako datoteka već postoji, poziv *open* će biti bezuspešan.

Ako pozovete *open* sa O_CREAT, dodajte i treći dodatni argument zadajući dozvole za novu datoteku.

Na primer, sledeći program, ***create-file.c***, pravi novu datoteku sa imenom zadatim u komandnoj liniji. Koristi se **O_EXCL** indikator sa *open*, pa ako datoteka već postoji – dolazi do greške. Za novu datoteku su vlasniku i njegovoj grupi data prava čitanja i pisanja, a ostalima samo pravo čitanja (ako je *umask* podešen na ne-nulte vrednosti, aktuelne dozvole mogu biti restriktivnije).

```c
#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* putanja = argv[1];

    // Dozvole za novu datoteku
    mode_t opcije = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH;

    // Kreira se datoteka
    int fd = open(putanja, O_WRONLY | O_EXCL | O_CREAT, opcije);

    if(fd == -1){
        // Dolazi do greske i ispisuje se poruka
        perror("open");
        return 1;
    }

    return 0;
}
```

Prevedite program i pokrenite ga navodeći ime nove datoteke kao argument:

```
$ gcc -o create-file create-file.c
$ ./create-file novadat
$ ls -l novadat
-rw-rw-r--  1  ftn  users  0 Feb 1     22:47   novadat
```

Pokušajte ponovo da napravite istu datoteku:

```
$ ./create-file novadat
open: File exists
```

Primetite da je veličina nove datoteke 0 (nula) zato što program nije upisao nikakve podatke u nju.

Umask

Kada kreirate novu datoteku koristeći *open*, neki bitovi prava pristupa koje ste predhodno podesili mogu biti isključeni. To je zato što je *umask* podešen na ne-nultu vrednost. Proces preko ***umask*** određuje bitove koji su maskirani van svih novo-napravljenih dozvola. Aktuelno korišćena prava

pristupa su kombinacija bita pristupnih prava (koja odredimo za otvaranje) i bita dopunjenih pomoću *umask*.

Da biste promenili *umask* iz komandnog interpretera, koristite *umask* komandu i odredite numeričku vrednost maske (u oktalnom sistemu). Da biste promenili *umask* za tekući proces, koristite funkciju *umask* prosleđujući joj željenu vrednost da biste je koristili za sledeću *open* funkciju.

Na primer, pozivanjem ovog reda:

```
    umask (S_IRWXO | S_IWGRP);
```

u okviru programa, ili pozivajući ovu komandu:

```
    $ umask 027
```

u terminalu, ukidate dozvolu za upisanje za članove grupe, a ukidate čitanje, pisanje i pristup drugima.

Čitanje podataka

Odgovarajuća komanda za čitanje podataka je *read*. Kao i kod komande *write*, i ona zahteva deskriptor datoteke, pokazivač ka baferu, i brojač.

Brojač broji koliko je bajtova pročitano iz deskriptora datoteke u bafer. Komada *read* vraća -1 ako je došlo do greške ili onoliko bajtova koliko je zapravo pročitano. Taj broj može biti manji od broja bajtova koji su zahtevani za čitanje, npr. ako nema više bajtova u datoteci.

Čitanje DOS/Windows tekstualnih datoteka

Vašem programu će verovatno ponekad trebati čitanje neke datoteke na DOS ili Windows sistemu. Zato je bitno naglasiti jednu važnu razliku u strukturi datoteka ove dve platforme.

U GNU/Linuks tekstualnim datotekama, svaki red je odvojen od sledećeg karakterom za novi red. Karakter za novi red je konstanta „**\n**”, koja ima ASCII kod 10. U Windows sistemu, redovi su odvojeni kombinacijom 2 karaktera: **CR** karakterom (tj. konstanta „**\r**”, ciji je ASCII kod 13), koji je propraćen karakterom za novi red.

Neki GNU/Linuks tekstualni editori prikazuju **^M** na kraju svakog reda kod čitanja Windows tekstualne datoteke - to je njihova oznaka za *CR* karakter. *Emacs* prikazuje Windows tekstualne datoteke pravilno, ali ih označava pokazivanjem u statusnoj (*mode*) liniji na dnu bafera. Neki Windows editori, npr. *Notepad*, prikazuju sav tekst u GNU/Linuks tekstualnoj datoteci u jednom redu jer očekuju *CR* karakter na kraju svakog reda. Drugi programi, koji su i za GNU/Linuks i Windows i rade sa tekstom, mogu da prijave dosta grešaka kada im se ubaci pogrešan format teksta.

Ako program obavlja čitanje tekstualne datoteke napravljene nekim Windows programom, verovatno će se želeti da se sekvenca „**/r/n**” zameni jednim novim redom. Slično, ako program piše

tekstualne datoteke koje treba da pročita Windows program, treba zameniti karakter za novi red kombinacijom „**/r/n**”. Ovo se mora uraditi bilo da se koristi niži nivo U/I funkcija, bilo da se koristi standardne U/I funkcije jezika C.

Sledeći program, ***hexdump.c***, je prosta demonstracija funkcije *read*. Program ispisuje heksadecimalne ostatke sadržaja datoteke određene u komandnoj liniji. Svaki red prikazuje pomeraj u datoteci i sledećih 16 bajtova.

```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 bafer[16];
    size_t ofset = 0;
    size_t procitani_bajtovi;

    // Otvara se datoteka za citanje
    int fd = open(argv[1], O_RDONLY);

    /*
    Cita se iz datoteke, karakter po karakter. Postupak se nastavlja sve
    dok citanje ne postane "kratko" (tj. kada broj procitanih bajtova
    postane manji od trazenog), ovo ukazuje da se stiglo do kraja datoteke.
    */
    do {
        // Citanje bafera
        procitani_bajtovi = read(fd, bafer, sizeof(bafer));
        // Upisivanje pomeraja u datoteku, pracen samim bajtovima
        printf("0x%06x : ", ofset);
        for (int i = 0; i < procitani_bajtovi; ++i)
            printf("%02x ", bafer[i]);
        printf("\n");
        // Pamti nasu poziciju u datoteci
        ofset += procitani_bajtovi;
    } while (procitani_bajtovi == sizeof(bafer));

    // Zavrseno
    return 0;
}
```

U terminalu prevesti i pokrenuti program. Program prikazuje ispis ostataka izvršne datoteke *hexdump*, tj. same sebe.

```
$ gcc -o hexdump hexdump.c
$ ./hexdump hexdump
```

```
0x000000 : 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
0x000010 : 03 00 3e 00 01 00 00 00 e0 10 00 00 00 00 00 00
0x000020 : 40 00 00 00 00 00 00 00 30 3a 00 00 00 00 00 00
```

Izlaz može izgledati drugačije, u zavisnosti od prevodioca koji koristite da prevedete *hexdump* i prevedenih (kompajliranih) indikatora (*flags*) za prevođenje koje ste selektovali.

Upisivanje podataka

Podsetite se sistemskog poziva ***write***. Upisivanje podataka u deskriptor datoteke se obavlja korišćenjem poziva *write*. Za poziv *write* potrebno je obezbediti sledeće ulazne argumente:

- deskriptor datoteke;
- pokazivač na bafer podataka;
- broj bajtova za upis.

Upišite podatke u deskriptor datoteke koristeći poziv *write*. Obezbedite deskriptoru datoteke pokazivač na bafer podataka i broj bajtova za upis. Deskriptor datoteke mora biti otvoren za upis. Podaci upisani u datoteku ne moraju biti string, tj. *write* kopira proizvoljne bajtove iz bafera u deskriptor datoteke.

Program **timestamp.c** dodaje tekuće vreme u zadatu datoteku ca komandne linije. Ako ta datoteka ne postoji, biće napravljena; ako već postoji, biće nadovezano na njen kraj. Ovaj program takođe koristi *time*, *localtime* i *asctime* funkcije za dobijanje i formatiranje tekućeg vremena.

```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));
}

int main(int argc, char *argv[]) {
    // Datoteka u koju treba da se doda vremenska oznaka
    char* imedatoteke = argv[1];

    // Uzima tekucu vremensku oznaku
```

```
    char* timestamp = get_timestamp();

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

    // Racuna duzinu timestamp stringa
    size_t duzina = strlen(timestamp);

    // Upisuje timestamp u datoteku
    write(fd, timestamp, duzina);

    // Zavrseno
    close(fd);

    return 0;
}
```

Program *timestamp.c* dodaje tekuće vreme u datu datoteku sa komandne linije. Ako datoteka ne postoji, biće napravljena nova. Prevedite i izvršite ovaj program, a zatim prikažite sadržaj datoteke koju ste naveli kao argument.

```
$ gcc -o timestamp timestamp.c
$ ./timestamp vreme.txt
$ cat vreme.txt
Sun Dec 20 21:25:11 2020
```

Pokušajte ponovo da pokrenete program navodeći istu datoteku kao argument.

```
$ ./timestamp vreme.txt
$ cat vreme.txt
Sun Dec 20 21:25:11 2020
Sun Dec 20 21:26:18 2020
```

Zapamtite da prvi put kad pozovemo *timestamp*, on pravi datoteku *blackadder*, dok kod drugog poziva samo dodajemo u već napravljenu datoteku.

Kretanje po datoteci

Deskriptor datoteke pamti svoju poziciju u datoteci. Dok čitate iz njega ili pišete u njemu, njegova pozicija napreduje paralelno sa brojem bajtova koje čitate ili pišete. Ponekad ćete se kretati po datoteci bez čitanja ili upisivanja podataka. Na primer, želite da pišete po sredini datoteke bez menjanja početka datoteke, ili želite da se vratite na početak datoteke i ponovo da pročitate nešto a da ne morate ponovo da otvarate datoteku i krećete iz početka.

Funkcija ***lseek*** vam omogućava da odredite poziciju deskriptora datoteke u datoteci. Prosledite ga deskriptoru datoteke uz dva dodatna argumenta, time određujući novu poziciju.

- Ako je treći argument **SEEK_SET**, *lseek* tumači drugi argument kao poziciju, datu u bajtovima, od početka datoteke.
- Ako je treći argument **SEEK_CUR**, *lseek* tumači drugi argument kao pomeraj, koji može biti pozitivan ili negativan, u odnosu na trenutnu poziciju.
- Ako je treći argument **SEEK_END**, *lseek* tumači drugi argument kao pomeraj od kraja datoteke. Pozitivna vrednost pokazuje poziciju iza kraja datoteke.

Funkcija *lseek* vraća novu poziciju, kao pomeraj u odnosu na početak datoteke.

Vrsta pomeraja je *off_t*. Ako se javi greška, *lseek* vraća -1; *lseek* ne možete koristiti sa nekim vrstama deskriptora datoteka, kao što je to „utičnica" (*socket*) datoteke.

Ako želite da nađete poziciju deskriptora datoteke u datoteci, a da ga ne menjate, odredite pomeraj 0 u odnosu na trenutnu poziciju, npr:

```
    off_t position = lseek (fd, 0, SEEK_CUR);
```

Linuks omogućava da se *lseek* funkcijom pozicionirate deskriptor datoteke izvan datoteke. Normalno, ako je deskriptor datoteke na kraju datoteke i vi nešto upisujete u njega, Linuks će automatski da proširi datoteku kako bi napravio mesta za nove podatke. Ako pozicionirate deskriptor datoteke izvan datoteke i onda nešto upisujete u njega, Linuks prvo proširuje datoteku da bi „popunio" rupu koju ste napravili *lseek* operacijom, a onda vrši upisivanje na kraju datoteke. Ova praznina ne zauzima mesto na disku, Linuks samo pravi poruku koliko je to veliko. Ako kasnije pokušate da to pročitate iz datoteke, programu će izgledati kao da je ta praznina popunjena bajtovima nula.

Korišćenjem ovakvog ponašanja *lseek*, moguće je kreirati veoma velike datoteke koje ne zauzimaju skoro nikakav prostor na disku. Program **lseek-huge.c** upravo radi to. On uzima iz komandne linije ime datoteke i veličinu tražene datoteke u MB. Program otvara novu datoteku, pozicionira se iza kraja datoteke koristeći *lseek*, a onda upiše jedinstveni nula-bajt pre nego što zatvori datoteku.

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

int main(int argc, char *argv[]) {
    int nula = 0;
    const int megabajt = 1024 * 1024;

    char* imedatoteke = argv[1];
    size_t duzina = (size_t) atoi(argv[2]) * megabajt;

    // Otvaranje nove datoteke
    int fd = open(imedatoteke, O_WRONLY | O_CREAT | O_APPEND, 0666);

    // Prelazimo na bajt manje od mesta gde hocemo da bude kraj
    lseek(fd, duzina - 1, SEEK_SET);
```

```
    // Upisujemo jedinstven nula-bajt
    write(fd, &nula, 1);

    // Zavrseno
    close(fd);

    return 0;
}
```

Prevedite program i iskoristite ga da napravite datoteku od 1GB (1024MB). Proverite koliko ima slobodnog prosotra na disku pre i posle operacije.

```
$ gcc -o lseek-huge lseek-huge.c
$ 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     marko      users      1073741824      Feb 5 16:29      bigfile
$ df -h .
Filesystem      Size       Used       Avail      Use%       Mounted on
/dev/hda5       2.9G       2.1G       655M       76%        /
```

Neprimetna količina memorije na disku je potrošena, uprkos ogromnoj veličini datoteke *bigfile*. Opet, ako otvorimo *bigfile* i čitamo iz njega, on će izgledati kao da je popunjen nulama u vrednosti od 1GB. Na primer, možemo da testiramo njegov sadržaj našim već napisanim programom *hexdump*.

```
$ ./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
```

Ako ovo sami pokrenete, verovatno ćete hteti da ga prekinete operaciju pomoću prečice Ctrl +C, da ne biste morali čekati da on ispiše 1.073.741.824 „0” bajtova.

Upamtite da su ove „magične” praznine u datotekama posebne karakteristike *extended* sistema datoteka *(ext2, ext3, ext4)* koji se obično koriste za GNU/Linuks diskove. Ako pokušate da koristite *lseek-huge* za stvaranje datoteke na nekim drugim tipovima sistema datoteka, kao što je *fat* ili *vfat* sistem datoteka, videćete da datoteka koju dobijete kao rezultat zaista popunjava određenu količinu prostora na disku.

Treba samo još napomenuti da Linuks ne dozvoljava da se pozicionirate pomoću *lseek* na poziciju koja se nalazi pre početka datoteke.

Napredni sistemski pozivi za rad sa datotekama

U prethodnoj vežbi smo obradili primere za sistemske pozive ***open***, ***read***, ***write***, ***lseek***. U ovom predavanju ćemo dopuniti znanje primerima za sledeće sistemske pozive:

- vektorsko čitanje i pisanje (***writev***, ***readv***)
- brzi prenos podataka (***sendfile***)
- čitanje sadržaja simboličke veze (***readlink***)
- zaključavanje datoteka (***fcntl***)
- prikazivanje informacija iz *i-node* strukture datoteke (***stat***)
- proveru prava pristupa (***access***)
- rad sa direktorijumima
- brisanje i pomeranje, odnosno promena imena.

Vektorsko čitanje i pisanje

Poziv *write* uzima kao argumente:

- deskriptor datoteke u koju se upisuje,
- pokazivač na početak bafera podataka,
- dužinu tog bafera.

Poziv ***write*** upisuje kontinualnu oblast memorije u deskriptor datoteke. Program ponekad treba da zapiše nekoliko stavki podataka, a svaka se nalazi u različitim delovima memorije. Ako bi koristio jedan *write* poziv, program mora ili da kopira sve stavke podatka u kontinualni deo memorije, a to dovodi do dodatnih memorijskih ciklusa CPU, ili bi program morao da napravi više *write* poziva, što takođe nije efikasno.

Za neke aplikacije, višestruki *write* pozivi su neefikasni ili nepoželjni. Na primer, kada pišemo na mrežni soket, dve *write* funkcije mogu da pruzrokuju da dva paketa budu poslata kroz mrežu, a ti isti podaci, mogu biti poslati u jednom paketu, preko jednog poziva *write*.

Poziv ***writev*** omogućava Vam da više ne-kontinualnih regiona memorije upišete u deskriptor datoteke pomoću jedne operacije. Ovo se naziva **vektorsko pisanje** (***vector write***). Ono što treba dodatno da se uradi prilikom korišćenja poziva *writev*, treba podesiti strukturu podataka specificiranjem početka i dužine svakog dela memorije. Ova struktura podataka se naziva niz (*array*) elemenata strukture ***iovec***. Svaki element specificira jedan deo memorije za upis, pri čemu polja ***iov_base*** i ***iov_len*** određuju početnu adresu dela memorije i njenu dužinu. Ako unapred

znate kolika vam oblast treba, možete prosto da deklarišete niz promenljivih strukture ***iovec***; ako broj regiona može da varira, onda morate da dodelite dinamički niz.

Ulazni parametri funkcije ***writev*** su:

- deskriptor datoteke u koju se upisuje,
- niz elemenata strukture *iovec,*
- broj elemenata u nizu.

Povratna vrednost je ukupan broj upisanih bajtova.

Program **write-arg.c** upisuje argumente sa komandne linije u datoteku koristeći jedan *writev* poziv. Prvi argument je ime datoteke, a drugi i svi ostali argumenti se upisuju u datoteku sa tim imenom, i to svaki u posebnom redu. Program alocira niz sa elementima structure *iovec* koji je ima duplo više elemenata od broja argumenata koji se upisuju – program za svaki argument upisuje tekst svojih argumenata, kao i karakter za novi red. Pošto ne znamo unapred broj argumenata, niz se alocira korišćenjem funkcije *malloc*.

```c
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int fd;
    struct iovec *vec;
    struct iovec *vec_sledeci;

    /*
    Trebace nam "bafer" koji sadrzi karaktere za novi red.
    Koristite uobicajene char promenljive za to.
    */
    char novalinija = '\n';
    // Prvi argument u komandnoj liniji je ime datoteke
    char *imedatoteke = argv[1];

    /*
    Preskacemo prva dva elementa sa liste argumenata.
    Element 0 je ime programa, a element 1 je ime datoteke.
   */
    argc -= 2;
    argv += 2;

    /*
    Dodeljivanje niza elemenata iovec. Za to nam trebaju
    po dva za svaki element liste argumenata, jedan za sam
```

```
    tekst, a drugi za novi red.
    */
    vec = (struct iovec*) malloc(2 * argc * sizeof(struct iovec));

    // Petlja preko liste argumenata, pravljenje iovec upisa
    vec_sledeci = vec;
    for(int i = 0; i < argc; ++i){
        // Prvi element je tekst samog argumenta
        vec_sledeci->iov_base = argv[i];
        vec_sledeci->iov_len = strlen(argv[i]);
        ++vec_sledeci;
        /*
        Drugi red je samo karakter za novi red. U redu je ako
        visestruki elementi iz niza struktura iovec pokazuju na
        isti deo memorije.
        */
       vec_sledeci->iov_base = &novalinija;
       vec_sledeci->iov_len = 1;
       ++vec_sledeci;
    }

    // Upisivanje argumenta u datoteku
    fd = open(imedatoteke, O_WRONLY | O_CREAT);
    writev(fd, vec, 2 * argc);
    close(fd);
    free(vec);

    return 0;
}
```

Prevedite program i pokrenite ga navodeći *izlaznadat* kao argument.

```
$ gcc -o write-arg write-arg.c
$ ./write-arg izlaznadat stefan uros nemanja dusan
```

Deklarišemo izlaznu datoteku pod imenom *izlaznadat* i tri tekstualna niza koji su ulazni argumenti za *writew* funkciju. Rezultat izvršavanja programa je:

```
$ cat izlaznadat
stefan
uros
nemanja
dusan
```

Brzi prenosi podataka

Sistemski poziv ***sendfile*** omogućava efikasan mehanizam kopiranja podataka iz jednog deskriptora datoteke u drugi. Deskriptori mogu biti otvoreni za disk datoteke, sokete ili druge uređaje.

Da bi se kopiralo iz jednog deskriptora datoteke u drugi, program dodeljuje bafer fiksne veličine, kopira neke podatke iz jednog deskriptora u bafer, a zatim iz istog bafera upisuje u drugi deskriptor, a to ponavlja sve dok svi podaci ne budu kopirani. Ovo zahteva dodatnu memoriju za bafer i pravi suvišnu kopiju podataka u baferu, pa je neefikasno po pitanju i vremena i prostora.

Korišćenjem funkcije *sendfile*, posrednički bafer se eliminiše.

Poziv ***sendfile*** ima sledeće ulazne parametre:

- deskriptor datoteke u koju treba da se upisuje,
- deskriptor datoteke iz koje se čita,
- pokazivač za promenljivu pomeraja (*offset*), koja ukazuje odakle treba da se počne iz ulazne datoteke,
- broj bajtova koje treba prebaciti.

Promenljiva (*offset*) sadrži pomeraj u ulaznoj datoteci od koga čitanje treba da počne (0 označava početak datoteke) i ova promenljiva se ažurira na poziciju u datoteci nakon transfera. Vrednost koju *sendfile* vraća je broj prebačenih bitova. Uključite **<sys/sendfile.h>** u vaš program ukoliko se koristi funkcija *sendfile*.

Sledeći program, **copy.c**, je jednostavna ali veoma efikasna primena kopiranja datoteka.

Kada se ovaj program pozove sa dva imena za datoteke na komandnoj liniji, program kopira sadržaj prve datoteke u drugu. Program koristi *fstat* da odredi veličinu izvorne tj. prve datoteke u bajtovima.

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

int main(int argc, char *argv[]) {
    int read_fd, write_fd;
    struct stat stat_baf;
    off_t ofset = 0;

    // Otvaranje ulazne datoteke
    read_fd = open(argv[1], O_RDONLY);

    // Dobijanje velicine ulazne datoteke
```

```
    fstat(read_fd, &stat_baf);

    /*
    Otvaranje izlazne datoteke za upis, i to sa istim dozvolama
    kao kod ulzane datoteke.
    */
    write_fd = open(argv[2], O_WRONLY | O_CREAT, stat_baf.st_mode);

    // Brzo kopiranje
    sendfile(write_fd, read_fd, &ofset, stat_baf.st_size);

    // Zatvaramo datoteku
    close(read_fd);
    close(write_fd);

    return 0;
}
```

Prevedite i pokrenite program.

```
$ gcc -o copy copy.c
$ ./copy /etc/termcap /tmp/termcap.1
```

Deklarišemo ulaznu datoteku /etc/termcap i izlaznu datoteku /tmp/termcap.1. Prva datoteka predstavlja nešto veću datoteku na operativnom sistemu Linuks, a /tmp direktorijum je izabran zato što korisnik tamo ima pravo da upisuje svoje datoteke (a pogodnost je i što će oni biti automatski obrisani nakon nekog vremena).

Rezultat izvršavanja programa možete proveriti na sledeći način:

```
$ ls -l /tmp/termcap.1
```

Videćete novokreiranu datoteku *termcap.1* koju je kreirao sistemski poziv *sendfile*. Uporedite vremena. Prvo ćemo obrisati datoteku:

```
$ rm /tmp/termcap.l
```

Merimo vreme za klasičnu *cp* komandu:

```
$ time cp /etc/termcap /tmp/termcap.1
```

Potom ćemo ponovo obrisati datoteku:

```
$ rm /tmp/termcap.1
```

Merimo vreme za klasični *copy* program koji koristi sistemski poziv *sendfile*:

```
$ time ./copy /etc/termcap /tmp/termcap.1
```

Uporedite ta dva vremena.

Sistemski poziv *sendfile* ima veliku primenu jer se njime kopiranje čini efikasnijim. Primer je njegova upotreba na Web serveru ili nekom drugom mrežnom *daemon*-u, koji prenosi sadržaj

datoteke, preko mreže do klijentskog programa. Zahtev se obično prima sa soketa klijentskog računara. Serverski program otvara lokalnu datoteku da bi dobio podatke i upisuje sadržaj datoteke na mrežni soket. Korišćenje funkcije *sendfile* može znatno da ubrza ovu operaciju. Treba preduzeti i druge korake da bi se mrežni transfer učinio što efikasnijim, kao što je pravilno postavljanje soket parametara.

Čitanje sadržaja simboličkih veza (linkova)

Sistemski poziv ***readlink*** obezbeđuje ime objekta na koji pokazuje simbolička veza. On uzima tri argumenta:

- putanju do simboličke veze,
- bafer koji prima putanju do datoteke na koju pokazuje veza, tj. original datoteke (*target*),
- dužinu tog bafera.

Poziv ***readlink*** u ovom slučaju ne stavlja *null* karakter („**\0**"), koji označava kraj stringa u bafer koji sadrži ime originalne datoteke (*target*). Ali umesto toga, *readlink* vraća broj karaktera koji upisuje u bafer, tako da je upis oznake kraja bafera krajnje jednostavan (samo dodate nulu).

Ako prvi argument za *readlink* pokazuje na datoteku koja nije simbolička veza, *readlink* postavlja vrednost *errno* na **EINVAL** i vraća vrednost -1.

Sledeći program, **print-symlink.c**, prikazuje put do datoteke na koju pokazuje simbolička veza (simbolička veza se unosi preko komandne linije).

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    char ciljna_putanja[256];
    char* sim_putanja = argv[1];

    // Pokusaj da se procita ono na sta ukazuje sim. veza
    int len = readlink(sim_putanja, ciljna_putanja, sizeof(ciljna_putanja));

    if(len == -1){
        // Neuspesan poziv
        if(errno = EINVAL)
            // Ovo nije simbolicka veza
            fprintf(stderr, "%s nije simbolicka veza\n", sim_putanja);
        else {
            // Neki drugi problem, ispisujemo genericku poruku
            perror("readlink");
```

```
            return 1;
        }
    } else {
        // Niz ciljna_putanja se mora zavrsiti NULL karakterom
        ciljna_putanja[len] = '\0';
        // Ispisujemo putanju do ciljne datoteke
        printf("%s\n", ciljna_putanja);
    }
    return 0;
}
```

Prevedite program:

```
$ gcc -o print-symlink print-symlink.c
```

Evo primera kako možete napraviti simboličku vezu i iskoristiti program *print-symlink* da je pročitate:

```
$ ln -s /usr/bin/wc my_link
$ ./print-symlink my_link
/usr/bin/wc
```

Zakjučavanje

Sistemski poziv ***fcntl*** je način da se pristupi nekim naprednim operacijama na deskriptorima datoteka. Prvi argument za *fcntl* je deskriptor otvorene datoteke, a drugi je vrednost koja označava koja operacija treba da bude izvršena. Za neke operacije, *fcntl* uzima dodatni argument. Ovde ćemo opisati jednu od najkorisnijih *fcntl* operacija, zaključavanje datoteka (***file locking***). Pogledajte *fcntl man* stranu, za informacije o ostalim operacijama.

Fcntl sistemski poziv omogućava programu da postavi zaključavanje za čitanje (***read-lock***) ili zaključavanje za upis (***write-lock***) na datoteku, što je u nekoj meri slično *mutex* semaforima. Zaključavanje za čitanje (*read-lock*) se postavlja na deskriptor datoteke sa koga se može čitati, a zaključavanje za upis (*write-lock*) se postavlja na deskriptor datoteke u koji se može pisati. Više od jednog procesa može držati zaključavanje za čitanje (*read-lock*) na istoj datoteci u isto vreme, ali samo jedan proces može držati zaključavanje za upis (*write-lock*). Ista datoteka ne može biti zatvorena (zaključana) i za čitanje i za pisanje.

Važna napomena: Upamtite da zaključavanje zapravo ne sprečava ostale procese da otvore datoteku, čitaju iz nje ili upisanju u nju, osim ako i oni takođe ne traže zaključavanje preko *fcntl*.

Da bi se postavilo zaključavanje (*lock*) na datoteku, prvo napravite i anulirajte strukturu ***flock***. Postavite **l_type** polje strukture na **F_RDLCK** za zaključavanje za čitanje (*read-lock*) ili **F_WRLCK** za zaključavanje za upis (*write-lock*). Zatim pozovite *fcntl* sa tri ulazna argumenta:

- deskriptor te datoteke,
- F_SETLCKW je operacijski kod,
- pokazivač na *struct flock* promenljivu.

Ako neki drugi proces drži zaključavanje koje sprečava primenu novog zaključavanja, *fcntl* se blokira dok se to zaključavanje ne ukloni.

Program **lock-file.c** (dat je ispod) otvara datoteku za pisanje čije ime se daje u komandnoj liniji, a onda se zaključavanje za upis (*write-lock*) postavlja na tu datoteku. Program čeka da korisnik pritisne taster <Enter>, a onda otključava i zatvara datoteku.

```c
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    char* datoteka = argv[1];
    int fd;
    struct flock lock;

    printf("Otvaranje datoteke %s\n", datoteka);
    // Otvaranje deskriptora datoteke
    fd = open(datoteka, O_WRONLY);

    printf("Zakljucavanje\n");
    // Inicijalizacija flock strukture
    memset(&lock, 0, sizeof(lock));

    lock.l_type = F_WRLCK;
    // Postavljanje zabrane pisanja (write lock) na datoteku
    fcntl(fd, F_SETLKW, &lock);

    printf("Zakljucano! Pritisni <Enter> za otkljucavanje...\n");

    // Cekamo na <Enter>, ali u sustini ne proveravamo, moze bilo koje dugme
    getchar();
    printf("Otkljucavanje!\n");

    // Otkljucavanje
    lock.l_type = F_UNLCK;
    fcntl(fd, F_SETLKW, &lock);
    close(fd);

    return 0;
}
```

Prevedite i pokrenite program navodeći ime datoteke /tmp/test-file kao argument:

```
$ gcc -o lock-file lock-file.c
$ touch /tmp/test-file
$ ./lock-file /tmp/test-file
```

```
Otvaranje datoteke /tmp/test-file
Zakljucavanje
Zakljucano! Pritisni Enter za otkljucavanje...
```

Nakon ovoga, u drugom prozoru pokušajte da pokrenete *lock-file*, ponovo na istoj datoteci.

```
$ ./lock-file /tmp/test-file
Otvaranje datoteke /tmp/test-file
Zakljucavanje
```

Primetite da je druga instanca blokirana dok pokušava da zaključa datoteku. Vratite se na prvi prozor i pritisnite Enter. Dobićete poruku:

```
Otključavanje
```

Program koji je otvoren u drugom prozoru odmah dobija šansu da obavi zaključavanje za upis (*write-lock*) i dobija se ispis kao kod prvog programa.

Ako se želi da se *fcntl* ne blokira u slučaju kada poziv ne može da dobije zaključavanje koje je traženo, treba koristiti **F_SETLK** umesto **F_SETLKW**. Tada, ako se zaključavanje ne može postići, *fcntl* odmah vraća -1.

Linuks obezbeđuje još jednu primenu zaključavanja datoteka pozivom *flock*. Primena *fcntl* ima jednu veliku prednost: radi sa NFS datotekama (ukoliko je NFS server relativno nov i ispravno konfigurisan). Ako imate pristup preko dve mašine koje aktiviraju isti sistem datoteka preko NFS, možete ponoviti prethodni primer koristeći dve različite mašine. Treba pokrenuti *lock-file* na jednoj mašini, specificirajući datoteku na NFS i pokrenuti ga ponovo na drugoj mašini, specificirajući istu datoteku. NFS budi drugi program kada se zaključavanje na upis oslobodi, od strane prvog programa.

Informacije iz *i-node* strukture datoteke

Korišćenjem poziva *open* i *read*, možete videti sadržaj datoteke. Ali šta je sa ostalim informacijama? Na primer, pozivanje ***ls -l*** prikazuje informacije kao što su veličina datoteke, poslednje promene u datoteci, dozvole i vlasnika datoteke.

Funkcija ***stat*** nam daje te informacije o datoteci. Funkcija ***stat*** ima dva ulazna argumenta:

- **putanja** (*path*) do datoteke koji se traži
- **pokazivač** na promenljivu tipa ***struct stat***

Ako *stat* odradi sve uspešno, onda vraća vrednost 0 i popunjava polja informacijama o datoteci, u suprotnom, *stat* vraća vrednost -1.

Ovo su neka od najkorišćenijih polja u strukturi *stat*:

- ***st_mode*** sadrži prava pristupa te datoteke,

- kao dodatak na prava pristupa, *st_mode* koduje tip datoteke u svojim višim bitovima,
- ***st_uid*** i ***st_gid*** sadrže ID vlasnika datoteke i grupe kojoj datoteka pripada,
- ***st_size*** sadrži veličinu datoteke u bajtovima,
- ***st_atime*** sadrži vreme kada je poslednji put bilo pristupano toj datoteci (pisanje ili čitanje),
- ***st_mtime*** sadrži vreme kada je ta datoteka poslednji put bila promenjena.

Sledeći makroi proveravaju vrednost polja *st_mode* da bi odredili koju vrstu tj. tip datoteke je analizirao sistemski poziv *stat*. Makro ocenjuje da je to istina (*true*, tj. 1), ako je datoteka određenog tipa:

- **S_ISBLK** (mode) – blok uređaj,
- **S_ISCHR** (mode) – karakter uređaj,
- **S_ISDIR** (mode) – direktorijum,
- **S_ISFIFO** (mode) – fifo,
- **S_ISLNK** (mode) – simbolički link,
- **S_ISREG** (mode) – regulama datoteka,
- **S_ISS0CK** (mode) – soket.

Polje ***st_dev*** sadrži glavni (*major*) broj i sporedni (*minor*) broj hardverskog uređaja, tačnije – drajvera, kojima blok ili karakter datoteka odgovara. Glavni (*major*) broj uređaja se pomera ulevo za osam bitova, a sporedni (*minor*) broj uređaja obuhvata osam najmanje značajnih bitova. Polje ***st_ino*** sadrži *inode* broj datoteke, pri čemu *inode* locira datoteku u sistemu datoteka.

Ako pozovete *stat* za simboličku vezu, *stat* prati link i možete dobiti informacije o datoteci na koju taj link pokazuje, ali ne i o samoj simboličkoj vezi! Zbog takvog rada, S_ISLINK nikada neće biti istinit (*true*) kao rezultat *stat*. Koristite ***lstat*** funkciju ako ne zelite da pratite simboličku vezu, zato što ova funkcija dobija informacije o samoj vezi, a ne o njenom originalu. Ako pozovete *lstat* za datoteku koja nije simbolička veza, onda ona ima isti rezultat kao *stat*. Pozivanjem funkcije *stat* na prekinuti *link* (*link* koji pokazuje na nepostojeću ili nepristupačnu originalnu datoteku), kao rezultat se pojavljuje greška, dok se pozivanjem funkcije *lstat* greška ne prijavljuje.

Ako već imate otvorenu datoteku za čitanje ili pisanje, pozovite ***fstat*** umesto *stat*. Poziv *fstat* uzima deskriptor datoteke kao njegov prvi argument umesto putanje.

Program **read-file.c** prikazuje funkciju koja alocira bafer koji je dovoljno veliki da prihvati sadržaj datoteke i onda ga učitava u bafer. Funkcija koristi *fstat* da odredi veličinu bafera koja će biti dovoljna za alociranje, a takođe se proverava da li je to regularna datoteka.

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

/*
```

```c
/*
Funkcija ucitava sadrzaj datoteke (IMEDAT) u bafer. Velicna bafera
se cuva u *duzina. Funkcija vraca bafer koji korisnik treba da izprazni.
Ako makro ne odgovori da li je datoteka regularna - vraca se NULL.
*/

char *citaj_datoteku(const char *imedat) {
    int fd;
    struct stat file_info;
    char *bafer;
    size_t *duzina = malloc(sizeof(size_t));

    // Otvaramo datoteku
    fd = open(imedat, O_RDONLY);
    // Uzimamo informacije o njoj
    fstat(fd, &file_info);
    *duzina = file_info.st_size;

    // Obratite paznju da li je u pitanju regularna datoteka
    if (!S_ISREG(file_info.st_mode)) {
        // Nije regularna, odustajemo
        close(fd);
        return NULL;
    }

    // Dodeljujemo dovoljno veliki bafer za sadrzaj datoteke
    bafer = (char *)malloc(*duzina);

    // Ucitavamo datoteku u bafer
    read(fd, bafer, *duzina);

    // Zavrsavamo rad
    close(fd);
    return bafer;
}

int main(int argc, char *argv[]) {
    char *imedat = argv[1];
    char *bafer = citaj_datoteku(imedat);
    printf("Bafer je: \n%s\n", bafer);

    return 0;
}
```

Prevedite program *read-file.c*

```
$ gcc -o read-file read-file.c
```

Pokrenite program na test datoteci /etc/hosts:

```
$ ./read-file /etc/hosts
Bafer je:
127.0.0.1       localhost
172.20.3.100    linux.site linux
. . .
```

Izlistajte sadržaj datoteke /etc/hosts koristeći naredbu *cat* i uporedite sadržaj:

```
$ ./read-file /etc/hosts
127.0.0.1       localhost
172.20.3.100    linux.site linux
. . .
```

Provera prava pristupa datoteci

Sistemski poziv ***access*** određuje da li pozivni proces ima prava pristupa datoteci. Može da proveri bilo koju kombinaciju *read*, *write* i *execute* prava, a takođe proverava i postojanje datoteke.

Poziv *access* ima dva ulazna argumenta:

- Prvi je putanja do datoteke koji treba da se proveri.
- Drugi je bit (*bitwise*) ili simbolička vrednost R_0K, W_0K, i X_0K, koja odgovara *read*, *write*, i *execute* dozvoli.

Povratna vrednost je 0 ako proces ima sve specificirane dozvole. Ako datoteka postoji, ali pozivni proces nema specificirane dozvole, *access* vraća -1 i postavlja *errno* na vrednost EACCES (ili na vrednost EROFS, ako se dozvola za upis proverava za datoteku u *read-only* sistemu datoteka).

Ako je drugi argument F_0K, *access* prosto proverava postojanje datoteke. Ako ona postoji, vrednost koja se vraća je 0; ako ne postoji, ta vrednost je -1 i *errno* se postavlja na vrednost ENOENT. Zapamtite da *errno* može biti postavljen na EACCES ako se bilo kom direktorijumu u putanji datoteke ne može pristupiti.

Program **check-access.c** koristi sistemski poziv *access* da proveri postojanje datoteke i proveri *read* i *write* dozvole. Potrebno je u komandnoj liniji dati ime datoteke, koja se proverava.

```c
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char  argv[])
{
    char* path = argv[1];
    int rval;

    // Provera postojanja datoteka
    rval = access (path, F_OK);
    if (rval == 0) printf ("%s postoji\n", path);
    else
    {
        if (errno == ENOENT) printf ("%s ne postoji\n", path);
        else if (errno == EACCES)
            printf ("ne mogu da pristupim %s \n", path);
        return 0;
    }
    // Provera read dozvola
    rval = access (path, R_OK);
    if (rval == 0) printf ("%s moze da se cita\n", path);
    else printf ("%s ne moze da se cita\n" path);

    // Proverava write dozvolu
    rval = access (path, W_OK);
    if (rval == 0) printf ("U %s moze da se pise\n", path);
    else if (errno == EACCES)
        printf ("U %s ne moze da se pise (nema w dozvole)\n", path);
    else if (errno == EROFS)
        printf ("U %s ne moze da se pise (read-only FS)\n", path);

    return 0;
}
```

Prevedite program *check-access.c*

```
$ gcc -o check-access check-access.c
```

Pokrenite program na test datoteci /etc/hosts:

```
$ ./check-access /etc/hosts
/etc/hosts postoji
/etc/hosts moze da se cita
U /etc/hosts ne moze da se pise (nema w dozvole)
```

Na primer, da bi se proverile pristupne dozvole za datoteku README na CD-ROM medijumu, primenite sledeće:

```
$ ./check-access /mnt/cdrom/README
/mnt/cdrom/README postoji
/mnt/cdrom/README moze da se cita
U /mnt/cdrom/README ne moze da se pise (read-only FS)
```

Rad sa direktorijumima

Par operacija nad direktorijumima koje mogu da budu korisne su:

- **getcwd** – daje trenutni radni direktorijum. Ima dva argumenta, bafer i dužinu tog bafera. *getcwd* kopira putanju trenutnog radnog direktorijuma u bafer.

- **chdir** – menja radni direktorijum u kojem se trenutno nalazite, na putanju koju stavite kao ulazni argument.

- **mkdir** – pravi novi direktorijum. Njegov prvi argument je putanja gde će se nalaziti taj novi direktorijum. Drugi argument su prava pristupa za taj novi direktorijum. Interpretacija prava pristupa je ista kao kod trećeg argumenta za funkciju *open* i modifikuju se na isti način preko *umask* procesa.

- **rmdir** – briše direktorijum. Njegov argument je putanja do direktorijuma.

Čitanje sadržaja direktorijuma

GNU/Linuks obezbeđuje funkcije za čitanje sadržaja direktorijuma. Iako one nisu direktno vezane za U/I funkcije nižeg nivoa, ovde su objašnjene jer su često korisne u raznim aplikativnim programima.

Da biste pročitali sadržaj direktorijuma, pratite sledeće korake:

- Pozovite ***opendir*** funkciju, zadajući putanju direktorijuma koji želimo da ispitamo. Funkcija *opendir* vraća **DIR*** identifikator, koji ćete da koristite za pristup sadržaju direktorijuma. Ako se pojavi neka greška, funkcija vraća NULL.

- Pozovite ***readdir*** sa ponavljanjem, prosleđući DIR* identifikator koji ste dobili sa *opendir*. Svaki put kada pozovete *readdir*, on vraća pokazivač na strukturu ***dirent*** koji odgovara sledećem direktorijumskom ulazu. Kada stignete do kraja sadržaja tog direktorijuma, *readdir* vraća NULL vrednost. Struktura *dirent* koju dobijate od funkcije *readdir* ima polje ***d_name*** koje sadrži ime direktorijumskog ulaza.

- Pozovite ***closedir***, prosleđujući DIR* identifikator, da bi se završila operacija listanja.

Uključite **<sys/types.h>** i **<dirent.h>** ako koristite ove funkcije u svom programu.

Ukoliko želite da sadržaj direktorijuma bude poređan u određenom redosledu, to morate sami da uradite.

Program **listdir.c** ispisuje sadržaj direktorijuma. Direktorijum može biti zadat u komandnoj liniji, ali ako nije zadat, program koristi trenutni radni direktorijum. Program prikazuje tip i putanju svake datoteke, tj svake stavke u direktorijumu koristeći sistemski poziv ***lstat*** funkciji ***get_file_type***.

```
#include <assert.h>
#include <dirent.h>
```

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

// Funkcija koja vraca tip datoteke
const char* daj_tip_datoteke(const char* putanja){
    struct stat st;
    lstat(putanja, &st);

    if (S_ISLNK(st.st_mode))
        return "simbolicka veza";
    else if (S_ISDIR(st.st_mode))
        return "direktorijum";
    else if (S_ISCHR(st.st_mode))
        return "karakter uredjaj";
    else if (S_ISBLK(st.st_mode))
        return "blok uredjaj";
    else if (S_ISFIFO(st.st_mode))
        return "fifo";
    else if (S_ISSOCK(st.st_mode))
        return "soket";
    else if (S_ISREG(st.st_mode))
        return "regularna datoteka";
    else
        // Neocekivan rezultat, datoteka mora da bude jedno od gorenavedenog
        assert(0);
}

int main(int argc, char *argv[]) {
    char* dir_putanja;
    DIR* dir;
    struct dirent* stavka;
    char putanja_stavke[PATH_MAX + 1];
    size_t duz_putanje;

    if(argc >= 2)
        // Ako je direktorijum zadat kao argument, koristimo ga
        dir_putanja = argv[1];
    else
        // U suprotnom, koristimo trenutni direktorijum
        dir_putanja = ".";

    // Kopiramo putanju direktorijuma u putanja_stavke
    strncpy(putanja_stavke, dir_putanja, sizeof(putanja_stavke));
    duz_putanje = strlen(dir_putanja);
```

```
    // Ako se putanja direktorijuma ne zavrsava kosom crtom, ona se dodaje
    if(putanja_stavke[duz_putanje - 1] != '/'){
        putanja_stavke[duz_putanje] = '/';
        putanja_stavke[duz_putanje + 1] = '\0';
        ++duz_putanje;
    }

    // Pocetak operacije listanja direktorijuma
    dir = opendir(dir_putanja);
    while ((stavka = readdir(dir)) != NULL) {
        const char* tip;
        strncpy(putanja_stavke + duz_putanje, stavka->d_name,
                    sizeof(putanja_stavke) - duz_putanje);
        // Odredjivanje tipa datoteke (stavke u direktorijumu)
        tip = daj_tip_datoteke(putanja_stavke);
        // Ispisivanje tipa i punog imena sa putanjom
        printf("%-18s: %s\n", tip, putanja_stavke);
    }

    // Kraj programa
    closedir(dir);
    return 0;
}
```

Prevedite i izvršite program *listdir.c*, deklarišite ulaznu datoteku i demonstrirajte sistemske pozive za direktorijume.

Prevedite program *listdir.c* i pokrenite ga navodeći direktorijum /dev kao argument.

```
$ gcc -o listdir listdir.c
$ ./listdir /dev
direktorijum       : /dev/.
direktorijum       : /dev/..
direktorijum       : /dev/pts
simbolicka veza    : /dev/stderr
simbolicka veza    : /dev/stdout
simbolicka veza    : /dev/stdin
blok uredjaj       : /dev/sdb
karakter uredjaj   : /dev/sg1
soket              : /dev/log
regularna datoteka : /dev/MAKEDEV
fifo               : /dev/initctl
...
```

Da biste ovo proverili, možete koristiti komandu *ls* u istom direktorijumu. Navedite *-u* indikator da *ls* ne bi sortirao, i stavite *-a* indikator da biste uzrokovali da trenutni direktorijum (**.**) i roditeljski direktorijum (**..**) budu uračunati. Prvi karakter svakog reda koji prikazuje komada *ls* je tip ulaza.

Brisanje, pomeranje i promena imena

Sistemski poziv ***unlink*** briše datoteku. Njegov argument je putanja do datoteke. Ova funkcija takođe može biti korišćena za brisanje drugih objekata sistema datoteka, kao što su imenovane *pipe* datoteke (*named pipes*) i uređaja (/dev). U stvari, *unlink* ne mora da obriše sadržaj datoteke. Kao što njegovo ime govori, on raskida vezu izmedju datoteke i direktorijuma u kome se nalazi. Datoteka se više ne ispisuje u tom direktorijumu kada ga listamo, ali ako bilo koji proces ima otvoren deskriptor datoteke za tu datoteku, sadržaj datoteke se ne briše sa diska. Samo kada više nema procesa koji ima otvoren deskriptor datoteke – onda se sadržaj datoteke briše. Tako da, ako neki process otvori datoteku da bi je čitao ili pisao u nju, a onda drugi proces uradi *unlink* nad tom datotekom i napravi novu datoteku sa istim imenom, prvi proces će videti stari sadržaj datoteke, a ne novi, osim ako ne zatvor, pa ga ponovo otvori.

Sistemski poziv ***rename*** preimenuje ili pomera datoteku. Njegova dva argumenta su njegova stara putanja i nova putanja do datoteke. Ako su putanje ka različitim direktorijumima, *rename* pomera datoteku samo ako su obe u različitim sistemima datoteka. Možete takođe koristiti *rename* da pomerate direktorijume ili druge objekte sistema datoteka.