Operativni sistemi

Uvod

Prevođenje programa:

```
gcc -Wall -Wextra FAJL.c -o NAZIV
```

Otklanjanje **upozorenja za argumente komandne linije** kada ih ne koristimo:

```
#define UNUSED(arg) ((void)arg)
UNUSED(argv);
```

Makro za greške:

```
#define check_error(expr, msg)\
  do {\
    if(!(expr)) {\
      perror(msg);\
      fprintf(stderr, "File: %s\nFunction: %s\nLine: %d\n", __FILE__,
__func__, __LINE__);\
      exit(EXIT_FAILURE);\
    }\
  } while(0)
```

Kodovi grešaka upisuju se u promenljivu `errno` koja se nalazi unutar zaglavlja `errno.h`. Pre poziva funkcije koja može dovesti do greške treba je inicijalizovati na 0. Kodovi:

- **ENOENT** - ne postoji traženi resurs (npr. fajl)
- **EACCESS** - nemamo prava pristupa

Niske

- `strcpy` - kopira jednu nisku u drugu
- `strncpy` - kopira prvih n karaktera jedne niske u drugu

- `strcat` - nadovezuje dve niske
- `strtol` - učitava broj u nisku, moguće zadavanje osnove
- `fgets` - čita string iz fajla

Prava pristupa

Postoje tri tipa korisnika:

1. **vlasnik** (**user**) - vlasnik fajla
2. **grupe** (**groups**) - svaki korisnik može da pripada različitim grupama
3. **ostali** (**others**) - svi koji ne pripadaju nekoj od grupa kojoj mi pripadamo su za nas ostali

Svakom tipu korisnika dodeljuju se tri prava:

1. **r - read** - čitanje
2. **w - write** - menjanje
3. **x - execute** - izvršavanje

Prava pristupa nad fajlom su u obliku "rwxrwxrwx" gde se prva 3 odnose na korisnika, druga 3 na grupe, a poslednja 3 na ostale. Mogu se zapisati i binarno i oktalno.

Informacije o fajlovima i korisnicima

Sistemski poziv `getpwnam` vraća `passwd` strukturu koja sadrži informacije o korisnicima kada se prosleđuje ime korisnika. Isto radi i `getpwuid` samo što se prosleđuje ID korisnika umesto imena. Više informacija na `man 3 getpwnam`. Struktura `passwd` je statički alocirana i NIKADA SE NE OSLOBAĐA. Funkcije `setpwent`, `getpwent` i `endpwent` omogućavaju čitanje jednog unosa passwd fajla:

- `setpwent` - otvara passwd fajl i postavlja offset na 0
- `getpwent` - čita 1 liniju passwd fajla
- `endpwent` - zatvara passwd fajl

Kada dobijemo upozorenje da imamo implicitne deklaracije funkcija, to verovatno znači da nismo uključili neki makro. U man stranama tražimo datu funkciju i gledamo koji makro fali. Ako piše `#define XOPEN_SOURCE >= 500`, onda na vrh programa dodajemo `#define XOPEN_SOURCE 700`.

Informacije o grupi dobijamo putem funkcija `getgrnam` i `getgrgid` koje vraćaju strukturu `group`. Ponovo imamo funkcije `setgrent`, `getgrent` i `endgrent` za prikaz svih grupa.

Za informacije o fajlovima koristi se sistemski poziv `stat`. Informacije na `man 2 stat`. Makroi za proveru tipa fajla i prava pristupa nalaze se u `man 7 inode`. Koriste se i `fstat` koji umesto fajla prima njegov fajl deskriptor, kao i `lstat` koji ne prati linkove do fajlova već vraća informacije o samim linkovima.

Funkcija `fileno` vraća fajl deskriptor prosleđenog fajl strima (FILE). Funkcija `fdopen` otvara fajl strim prosleđenog fajl deskriptora. Zatvaranjem fajl strima automatski se zatvara i njegov fajl deskriptor.

Sistemski pozivi

Sistemski pozivi se u `man` stranama nalaze na drugoj strani.

- `mkdir` - kreiranje direktorijuma
- `rmdir` - brisanje direktorijuma
- `unlink` - brisanje fajlova
- `open` - otvaranje fajlova, na kraju se zatvara dobijeni fajl deskriptor sa `close(fd)`
- `read` i `write` - čitanje i pisanje, rade sa bajtovima pa je potrebno dodati terminirajuću nulu pri čitanju
- `lseek` - pomera offset u fajlu
- `chmod` - menja prava pristupa fajlu
- `umask` - postavlja novu vrednost umask i vraća staru, korisno samo pri kreiranju novih fajlova

Vreme

Vreme modifikacije i poslednjeg pristupa fajlu nalaze se u `stat` strukturi. Za promenu tih vremena koristi se poziv `utime` pri čemu se prosleđuje popunjena `utimebuf` struktura.

Čitanje trenutnog vremena:

```
time_t now = time(NULL)
```

Ostale funkcije:

- `ctime` - prima sekunde od početka epoha i ispisuje formatirano vreme
- `localtime` - vraća `tm` strukturu koja sadrži informacije o vremenu, ova struktura se NE OSLOBAĐA
- `mktime` - vraća sekunde od prosleđene `tm` strukture. Možemo je pozvati nakon modifikacije te strukture da bi se ažurirala ostala njena polja.
- `gettimeofday` - u strukturu tipa `timevall` upisuje precizne informacije o vremenu (sekunde i mikrosekunde)
- `strftime` - omogućava formatiran prikaz vremena koristeći `tm` strukturu

Obilazak direktorijuma

Prvi način:

```
void sizeOfDir(char* putanja, unsigned* psize) {
    struct stat fInfo;
    check_error(lstat(putanja, &fInfo) != -1, "lstat");

    /* OBRADA FAJLA */

    if (!S_ISDIR(fInfo.st_mode))
        return;

    DIR* dir = opendir(putanja);
    check_error(dir != NULL, "opendir");
    check_error(chdir(putanja) != -1 , "chdir");

    struct dirent* dirEntry = NULL;
    errno = 0;
    while ((dirEntry = readdir(dir)) != NULL) {
        if (!strcmp(dirEntry->d_name, ".") || !strcmp(dirEntry->d_name,
"..")) {
            check_error(stat(dirEntry->d_name, &fInfo) != -1, "stat");
            /* POSEBNA OBRADA ZA "." i ".." */
            errno = 0;
            continue;
        }
        sizeOfDir(dirEntry->d_name, psize);
        errno = 0;
    }

    check_error(errno != EBADF, "Greska");
    check_error(chdir("..") != -1, "chdir");
```

```
    check_error(closedir(dir) != -1, "closedir");
}
```

Drugi način - pozivamo funkciju `nftw` koja kao argument prima funkciju

```
int IME(const char* fpath, const struct stat* sb, int typeflag, struct FTW*
ftwbuf)
```

koja vršu obradu fajla. Ova funkcija treba da vrati 0 u slučaju uspešne obrade.

Procesi

ID trenutnog procesa i roditeljskog procesa dobija se sa `getpid()` i `getppid()`. Novi proces se kreira sa `fork()` koji vraća ID kreiranog dete procesa. Na osnovu toga odvajamo roditeljsku granu i granu deteta:

```
pid_t child = fork()
if(child) {...} /* RODITELJ */
else {...} /* DETE */
```

Roditeljski proces treba da sačeka dete proces da ne bi došlo do nastajanja zombi pocesa:

```
int status;
wait(&status); // waitpid(child, &status, 0);
if(WIFEXITED(status)) { /* DETE ZAVRŠENO */
    int code = WEXITSTATUS(status); /* STATUS ZAVRŠETKA */
}
else {...} /* DETE NIJE ZAVRŠILO */
```

Dete i roditelj mogu komunicirati preko pajpa koji se kreira pre forkovanja:

```
int pipeFds[2];
pipe(pipeFds);
```

Pajp služi za jednosmernu komunikaciju. Na početku koda i roditelj i dete zatvaraju po jedan kraj koji ne koriste, a drugi kraj koriste kao bilo koji fajl deskriptor. Kraj za čitanje ima vrednost 0, a kraj za pisanje vrednost 1. Na kraju upotrebe i roditelj i dete zatvaraju i drugi kraj pajpa. Ako je potrebna dvosmerna komunikacija kreiraju se dva pajpa.

Sistemski poziv `exec` izvršava drugi program.Prvi argument je putanja do fajla koji treba izvršiti, drugi argument je po konvenciji sam naziv fajla, a zatim slede argumenti tog

programa ili kao lista argumenata same funkcije ili kao jedan argument u obliku niza. U oba slučaja poslednji argument programa je uvek NULL.

- `execl` - argumenti programa se zadaju kao lista argumenata funkcije
- `execlp` - isto kao prethodni poziv, ali se dodatno pretražuju i direktorijumi PATH promenljive
- `execv` - argumenti programa se zadaju kao niz
- `execvp` - isto kao prethodni poziv, ali se dodatno pretražuju i direktorijumi PATH promenljive

Signali

Na vrhu je potrebno uključiti `#define _DEFAULT_SOURCE` kako bi obrada signala bila konzistentna. Obradu signala vršimo tako što povežemo funkciju za obradu sa signalom. Možemo sve signale povezati sa istom funkcijom, a u njoj onda raditi različite stvari u skladu sa signalom koji je stigao. Za povezivanje se koristi funkcija `signal` koja vraća `SIG_ERR` ako je došlo do greške. Funkcija za obradu ima sledeći potpis:

```
void NAZIV(int signum) {...}
```

Nekad je bolje kompleksnu obradu signala ostaviti za drugu funkciju ili main, a u ovoj funkciji za obradu samo naznačiti koji je signal došao. Signali se preko terminala šalju na sledeći način:

```
kill -SIGNAL_NUM PROCESS_ID
```

Signale možemo čekati u petlji na sledeći način:

```
do {
    pause();
} while(!shouldTerminate)
```

FIFO fajlovi

FIFO fajlovi služe za komunikaciju između procesa koji nisu u odnosu roditelj-dete. Funkcionišu isto kao pajpovi. FIFO fajl se kreira pozivom `mkfifo`. Nakon kreiranja FIFO fajl se otvara pozivom `open` koji blokira izvršavanje sve dok drugi proces ne otvori taj FIFO u

suprotnom modu (čitanje-pisanje). Zatvaranjem jedne strane FIFO fajla automatsi se zatvara i druga strana. Kraj za čitanje ima vrednost 0, a kraj za pisanje vrednost 1.

Deljena memorija

Pri prevođenju programa koji rade sa deljenom memorijom potrebno je dodati opciju `-ltr`. Funkcija kojom proces kreira blok deljene memorije date veličine na datoj putanji:

```
void* createMemoryBlock(const char* filePath, unsigned size) {
    /* Kreiranje, prava i mod zavise od načina upotrebe */
    int memFd = shm_open(filePath, O_RDWR | O_CREAT, 0600);
    check_error(memFd != -1, "shm_open failed");
    /* Podešavanje veličine */
    check_error(ftruncate(memFd, size) != -1, "ftruncate failed");
    /* Mapiranje objekta deljene memorije u RAM */
    void* addr;
    check_error((addr = mmap(0, size, PROT_READ | PROT_WRITE, \
            MAP_SHARED, memFd, 0)) != MAP_FAILED, "mmap failed");
    close(memFd);
    return addr;
}
```

Nakon završetka upotrebe potrebno je odmapirati blok deljene memorije:

```
check_error(munmap(niz, size) != -1, "unmap failed");
```

Samo jedan proces kreira blok deljene memorije, a ostali ga otvaraju i koriste:

```
void *getMemoryBlock(const char* filePath, unsigned* size) {
    /* Otvaranje */
    int memFd = shm_open(filePath, O_RDONLY, 0);
    check_error(memFd != -1, "shm_open failed");
    /* Uvek koristiti fstat za dobijanje veličine bloka deljene memorije */
    struct stat fInfo;
    check_error(fstat(memFd, &fInfo) != -1, "fstat failed");
    *size = fInfo.st_size;
    void* addr;
    check_error((addr = mmap(0, *size, PROT_READ, MAP_SHARED, memFd, 0)) \
    != MAP_FAILED, "mmap failed");
    close(memFd);
    return addr;
}
```

Nakon završetka upotrebe ponovo se vrši unmap. Ukoliko je potrebno obrisati blok deljene memorije to se radi sa:

```
check_error(shm_unlink(argv[1]) != -1, "shm_unlink failed");
```

Ukoliko više procesa istovremeno koristi isti blok deljene memorije, sinhronizacija se vrši pomoću semafora. U jednu strukturu ubacujemo podatke koji se koriste, kao i dve promenljive tipa `sem_t` koje sinhronizuju čitanje i pisanje. Samo jedan proces inicijalizuje semafore funkcijom `sem_init` i to tako što se svaki semafor inicijalizuje na broj dostupnih resursa, a ako nema dostupnih resursa inicijalizuje se na 0 (treći argument). Kao drugi argument prosleđuje se 1 što označava da se radi o globalnom semaforu za sve procese (ako se prosledi 0 onda se radi o lokalnom semaforu koji se odnosi na niti jednog procesa). Na semaforu čekamo pre nego radimo sa deljenom memorijom funkcijom `sem_wait`, a obaveštavamo druge procese da je deljena memorija bezbedna za neku radnju funkcijom `sem_post`.

Preusmeravanje tokova i baferisanje

Tokove preusmeravamo funkcijom `dup2`. Pozivom

```
dup2(pipeFds[PIPE_WR], STDOUT_FILENO);
```

standardni izlaz preusmeravamo na kraj za pisanje pajpa, pa će npr. funkcija `printf` pisati u pajp umesto na standardni izlaz. Ako želimo da se vratimo na prethodno stanje možemo napraviti kopiju fajl deskriptora za standardni izlaz pomoću funkcije `dup` i onda ponovo iskoristiti funkciju `dup2` sa obrnutim redosledom argumenata.

Pod baferisanjem se podrazumeva definisanje onoga što vrši okidanje fizičke IO operacije.

- Nebaferisani IO znači da je okidač za IO operacije sama operacija čitanja ili pisanja. (default za stderr)
- Linijski baferisani IO znači da je okidač za IO operacije karakter za novi red. (default za terminale)
- Potpuno baferisani IO znači da je okidač za IO operacije napunjenost bafera. Fizička IO operacija se vrši samo onda kada se stigne do kraja bafera. (default za regularne fajlove)

Mod baferisanja može da se kontroliše pomoću funkcije `setvbuf`:

```
setvbuf(stdout, NULL, _IONBF, 0); // nebaferisani IO
setvbuf(stdout, NULL, _IOLBF, 0); // linijski baferisani IO
```

```
setvbuf(stdout, NULL, _IOFBF, 0); // potpuno baferisani IO
```

Baferi mogu i ručno da se prazne funkcijom `fflush` koja prima bafer kao argument. Ako se prosledi NULL prazne se svi baferi.

Zaključavanje fajlova

Da bismo zaključali fajl potrebno je prvo popuniti `flock` strukturu sa odgovarajućim informacijama, a zatim za zaključavanje koristimo funkciju `fcntl`. Moguće ja pokušati zaključavanje na 3 načina:

- `F_SETLK` - ako ne može da zaključa fajl, odmah puca i vraća se nazad
- `F_SETLKW` - ako ne može da zaključa fajl, sačekaće dok se resurs ne oslobodi
- `F_GETLK` - koristi se za ispitivanje da li može da se postavi željeni katanac (ako može da se postavi katanac za pisanje, onda može i katanac za čitanje)

Tip katanca može biti `F_RDLCK` i `F_WRLCK` za čitanje i pisanje, kao i `F_UNLCK` za otključavanje. Funkcija `ftell` vraća poziciju u fajlu gde će biti izvršena sledeća IO operacija.

Niti

Sve funkcije za rad sa nitima vraćaju 0 u slučaju uspeha, a vrednost veću od 0 u slučaju neuspeha. Makro za obradu grešaka funkcija koje rade sa nitima:

```
#define pthreadCheck(pthreadErr, userMsg)\
    do {\
        int _pthreadErr = pthreadErr;\
        if (_pthreadErr > 0) {\
            errno = _pthreadErr;\
            check_error(false, userMsg);\
         }\
    } while (0)
```

Programi koji rade sa nitima se prevode uz dodatne opcije `-ltr` i `-pthread`. Funkcija koja implementira nit je oblika:

```
void* osThreadFunction(void* arg)
```

Argument funkcije, kao i povratnu vrednost ako postoji, potrebno je smestiti u neku strukturu jer rad sa osnovnim tipovima može napraviti problem. U samoj funkciji ili pri radu sa vrednošću koju funkcija vrati potrebno je kastovati `void*` u tu odgovarajuću strukturu. Nit se kreira pozivom funkcije `pthread_create` koja prima adresu gde smešta ID kreirane niti, drugi argument koji je uvek NULL, funkciju koja implementira rad niti i adresu strukture koja predstavlja argument. Svaka nit ima jedinstven ID koji se dobija sa `pthread_self()`. Na završetak rada niti čeka se komandom `pthread_join`, gde drugi argument predstavlja tip povratne vrednosti koji se čeka. Ako jedna nit pozove `exit` automatski se završavaju i sve ostale niti. Ako je potrebno završiti samo jednu nit onda se koristi `pthread_exit` ili `return` naredba u funkciji koja implementira rad te niti. Ukoliko ne želimo da čekamo na neku nit koristimo `pthread_detach`. Nit koja je detach-ovana ne sme da budu join-ovana, a takođe nije dozvoljeno join-ovati istu nit više puta.

Ukoliko više niti koristi isti resurs, onda taj resurs treba spakovati u strukturu zajedno sa muteksom tipa `pthread_mutex_t`. Pre kreiranja niti potrebno je inicijalizovati muteks pomoću `pthread_mutex_init`. Nakon završetka rada muteks se uništava sa `pthread_mutex_destroy`. Zaključavanje resursa se vrši pomoću funkcija `pthread_mutex_lock` i `pthread_mutex_unlock`.

Drugi način je upotreba atomičkih promenljivih iz zaglavlja `stdatomic.h`. Na primer, ako računamo neku sumu potrebno je deklarisati globalnu promenljivu tipa `atomic_int`. Pre upotrebe se inicijalizuje pomoću funkcije `atomic_init`. Sabiranje se u kritičnoj sekciji vrši sa `atomic_fetch_add`. Na isti način se radi sa drugim tipovima i operacijama.