GRADIVO ZA PRVI ZADATAK I 2 PRIMJERA:

KOMUNIKACIJA PORUKAMA

REDOVI PORUKA

Poruka je mala količina podataka (na primjer, do nekoliko stotina bajtova) koja može biti poslana u red poruka. Poruci se može pridijeliti tip po kojem se može prepoznati. Svaki proces s odgovarajućom dozvolom može primiti poruku iz reda.

Red poruka može poslužiti kao semafor: stavljanje poruke u red je ekvivalentno otvaranju semafora, a uzimanje poruke iz reda ekvivalentno je zatvaranju semafora. Poziv za uzimanje poruke se normalno zablokira ako je red prazan što odgovara stanju kada je semafor na nuli.

Sustavski pozivi za rad s redovima poruka

Rad sa redovima poruka je ovdje opisan nešto detaljnije nego što su bili opisani pozivi za rad sa zajedničkom memorijom i skupovima semafora. Ipak, više detalja se može naći sa: `man msgget`, `man msgop` i `man msgctl`. Podaci potrebni za rad sa redovima poruka definirane su u datotekama <sys/types.h>, <sys/ipc.h> i <sys/msg.h> koje treba uključiti na početku programa. Opis bitnih struktura podataka može se naći sa `man intro`.

```
int msgget(key_t key, int flags);
```

Sustavski poziv *msgget* stvara red poruka, ili vraća identifikator reda poruka ako red već postoji. Poziv je analogan sustavskom pozivu *open*. Kao parametar prima ključ *key* i vraća identifikator reda, odnosno -1 ako dođe do greške.

Identifikator reda je vrlo sličan opisniku datoteke, osim što ga može koristiti bilo koji proces koji poznaje taj broj. Ako je postavljen bit `IPC_CREATE` u *flags*, red se kreira ako već ne postoji, a devet najnižih bitova su dozvole za korištenje reda. Dozvola za pisanje dopušta da poruka bude poslana, a dozvola za čitanje dopušta primanje poruke. Ako `IPC_CREATE` nije postavljen onda red mora postojati i u tom slučaju ova funkcija samo pronalazi identifikator reda. (Ako se za *key* stavi `IPC_PRIVATE` onda se kreira novi red bez obzira na `IPC_CREAT`.)

Dozvole pristupa u *flags* su definirane na slijedeći način:

```
00400Receive message by user
00200Send message by user
00040Receive message by group
00020Send message by group
00004Receive message by others
00002Send message by others

struct msgbuf {

     long mtype;
     char mtext[1];

};

int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg);
```

Sustavski poziv *msgsnd* šalje poruku u red čiji je ID *msqid* dobiven primjerice od *msgget*. *msgp* pokazuje na strukturu u kojoj na prvom mjestu mora biti dugačak cijeli broj veći od nule - vrsta poruke. Ostatak te strukture ovisi o podacima koji se šalju. Interno se ostatak poruke prihvaća kao niz znakova (bajtova) duljine *msgsz*. Tip poruke omogućava primaocu da odabere iz reda poruke koje želi izvaditi, odnosno može čekati određeni tip poruke. *msgflg* je obično 0, što uzrokuje da se *msgsnd* zablokira dok je red pun. Druga mogućnost je `IPC_NOWAIT` što uzrokuje da poziv *msgsnd* vrati grešku ako je red pun. *msgsnd* vraća 0 ako uspije ili -1 ako dođe do greške.

```
int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);
```

Sustavski poziv *msgrcv* poziva primalac poruke. *msgsz* mora biti veličina najveće poruke koja može stati u prostor na koji pokazuje *msgp*. Obzirom da primljena poruka može biti manja od toga, ovaj poziv vraća veličinu poruke ili -1 ako dođe do greške. Ako primalac želi određenu vrstu poruke onda je stavi u *msgtyp*. Inače se stavi 0 čime se uzima najstarija poruka iz reda (bez obzira na vrstu poruke). Ako je red prazan ili u njemu nema poruka tražene vrste onda će se *msgrcv* zablokirati, osim ako je *msgflg* (*msgflg* je obično 0) `IPC_NOWAIT` u kojem slučaju će se odmah vratiti -1 (greška).

```
struct ipc_perm {

      ushort cuid;/* creator user id*/
      ushort cgid;/* creator group id*/
      ushort uid;/* user id*/
      ushort gid;/* group id*/
      ushort mode;/* r/w permission*/
      ushort seq;/* slot usage sequence #*/
      key_t key;/* key*/

} ;

struct msg {

      struct msg *msg_next;/* ptr to next message on queue */
      long msg_type;/* message type */
      short msg_ts;/* message text size */
      short msg_spot;/* message text map address */

};

struct msqid_ds {

      struct ipc_perm msg_perm;/* message operation permissions*/
      struct msg *msg_first; /* ptr to the first message on the queue*/
      struct msg *msg_last; /* ptr to the last message on the queue*/
      ushort msg_cbytes; /* current number of bytes on the queue*/
      ushort msg_qnum; /* nr of messages currently on the queue*/
      ushort msg_qbytes; /* max nr of bytes allowed on the queue*/
      ushort msg_lspid; /* last process that performed msgsnd*/
      ushort msg_lrpid; /* last process that performed msgrcv */
      time_t msg_stime; /* time of the last msgsnd operation*/
      time_t msg_rtime; /* time of the last msgrcv operation*/
      time_t msg_ctime; /* time of the last msgctl operation*/

} ;
```

Red poruka se nakon uporabe treba obrisati. Npr. pozivom `msgctl(msqid, IPC_RMID, NULL)`. Sustavski poziv

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

obavlja jednu od tri operacije u ovisnosti o *cmd*:

`IPC_STAT` popunjava strukturu *buf* vrijednostima za red poruka *msqid*.

`IPC_SET` mijenja *msg_perm.uid*, *msg_perm.gid*, *msg_perm.mode* i *msg_qbytes* za red poruka *msqid* sa vrijednostima iz *buf*.

`IPC_RMID` uništava red poruka *msqid* i bilo koji zablokirani poziv vraća grešku u tom slučaju.

OKOLINA (*environment*)

Okolina je niz znakovnih nizova oblika "*ime=vrijednost*" koji se predaje svakom programu prilikom pokretanja. *ime* je varijabla okoline. Uobičajeno je za imena tih varijabli upotrebljavati velika slova iako to nije obavezno.

Varijable okoline se najčešće postavljaju korištenjem korisničke ljuske (vidi: `man sh`, `man csh`). U ljusci *sh* se sa:

```
ime=vrijednost
```

postavlja varijabla za samo ljusku. Tek ako se izvede:

```
export ime
```

ista varijabla se uključuje i u okolinu koju ljuska predaje programima koje poziva. `export` daje popis svih varijabli koje se predaju kao okolina programima, dok `set` daje popis svih varijabli koje poznaje sama ljuska. `unset ime` poništava definiciju varijable *ime*.

U ljusci *csh* se varijabla okoline programa definira sa:

```
setenv ime vrijednost
```

Varijable same ljuske se postavljaju sa:

```
set ime=vrijednost
```

Neke od najčešće korištenih varijabli: *logname*, *home*, *path*, *user* i *term* automatski se uključuju i u okolinu nakon ove naredbe, pa za njih nije potrebno upotrebljavati *setenv*. Prilikom uključenja u okolinu, imena ovih varijabli se pišu velikim slovima. `setenv` izlistava okolinu koja se predaje programima, dok `set` daje popis svih varijabli koje poznaje sama ljuska.

Kako se poziva *main*

Prototip prema kojem se poziva funkcija *main* svakog programa u UNIX-u je:

```
int main(int argc, char *argv[], char *envp[]);
```

argc je broj argumenata navedenih kod poziva programa, a *argv* je niz od *argc* kazaljki na te argumente kao nizove znakova. Prvi od tih nizova je ime samog pozvanog programa. *envp* je niz kazaljki na nizove znakova oblika "*ime=vrijednost*" koji čine okolinu. Posljednja kazaljka je `NULL`. Okolini se može pristupiti i na praktičniji način nego korištenjem *envp*. Zbog toga se *main* može definirati i kao:

```
int main(int argc, char *argv[]) { ... }
```

Također, program koji ne koristi nikakve ulazne parametre može definirati *main* kao:

```
int main(void) { ... }
```

main treba vratiti cjelobrojnu vrijednost jer poziv možemo pojednostavljeno zamisliti kao:

```
exit(main(argc, argv, envp));
```

Ako program završava pozivom *exit* na nekom mjestu, onda ne dolazi do povratka iz *main*. Međutim, ako *main* normalno završava, onda je potrebna povratna vrijednost koja će postati argument poziva *exit*.

Pristup varijablama okoline iz programa

envp nije pogodan za pristup varijablama okoline jer je poznat samo unutar main, a ne i u ostalim funkcijama programa. Zato postoji globalna varijabla:

```
extern char *environ[];
```

koja je, također, niz kazaljki na nizove znakova koji čine okolinu. Toj varijabli se može pristupati izravno ili korištenjem funkcija *getenv* i *putenv*.

```
char *getenv(char *name);
```

name pokazuje na niz znakova s *imenom* varijable okoline kojoj treba pristupiti. Rezultat je kazaljka na *vrijednost* te varijable u nizu oblika "*ime=vrijednost*" ili `NULL` ako varijabla nije nađena. Npr. ako u okolini postoji "*nadimak=pero*", tada će getenv("*nadimak*") vratiti pokazivač na "*pero*".

```
int putenv(char *string);
```

string pokazuje na niz znakova oblika "*ime=vrijednost*". *putenv* ga uključuje u okolinu umjesto postojećeg niza koji počinje istim *imenom* ili ga dodaje u okolinu. Rezultat je različit od 0 samo ako *putenv* nije dobio potrebnu memoriju za proširenje okoline.

putenv mijenja okolinu na koju pokazuje *environ* i kojoj se pristupa pomoću *getenv*. Međutim, pri tome se ne mijenja *envp* koji je predan funkciji *main*. Niz znakova na koji pokazuje kazaljka *string* postaje dio okoline.

POKRETANJE PROGRAMA (sustavski pozivi *exec...*)

Sustavski pozivi *exec* (u svim oblicima) inicijaliziraju proces novim programom. Jedino pomoću njih se izvršavaju programi u UNIX-u. Postoji šest poziva koji se uglavnom razlikuju po načinu prijenosa parametara (vidi `man exec`):

```
int execl(char *path, char *arg0, char *arg1,....., char *argn, char *null)

int execv(char *path, char *argv[])

int execle(char *path, char *arg0,...., char *argn, char *null, char *envp[])

int execve(char *path, char *argv[], char *envp[])

int execlp(char *file, char *arg0,...., char *argn, char *null)

int execvp(char *file, char *argv[])
```

Pozivom neke verzije poziva *exec* izvršava se navedeni program od početka, tj. pozivom funkcije *main*. Ako je poziv uspio, iz njega nema povratka. U slučaju greške rezultat je -1.

Argument *path* mora sadržavati put do datoteke sa izvršnom verzijom programa ili tekstom koji se može interpretirati (počinje sa `#!`) nekim drugim programom, najčešće ljuskom. Kod *execlp* i *execvp*, dovoljno je da argument *file* bude samo ime takve datoteke, a ona se traži u direktorijima koji su navedeni kao vrijednost varijable okoline "`PATH`".

execl, *execle* i *execlp* imaju varijabilan broj argumenata. Prvi argument *arg0* uvijek mora biti ime izvršne verzije prorama, a `NULL` je oznaka kraja argumenata. Od tih argumenata se kreira *argv* koji se predaje funkciji *main* novog programa.

Kod *execv*, *execve* i *execvp* predaje se izravno *argv*. Po dogovoru, i on mora imati barem jednu kazaljku koja pokazuje na niz znakova s imenom programa. Ostale pokazuju na argumente programa. Posljednja kazaljka mora biti `NULL` kako bi se znalo gdje je kraj i moglo izračunati *argc*.

envp u *execle* i *execve* je niz kazaljki na nizove znakova koji čine okolinu. Posljednja kazaljka mora biti `NULL`. Kod ostalih poziva, novi program dobiva postojeću okolinu (*environ*).

Otvoreni opisnici datoteka ostaju otvoreni kroz poziv *execl*. Ako to nije potrebno, treba ih zatvoriti prije nekog od ovih poziva. Kao i kod sustavskog poziva *fork*, većina sustavskih atributa ostaje nepromjenjena.

Primjer upotrebe *exec* i *fork*

Obično *exec* služi za inicijalizaciju procesa djeteta kreiranog sustavskim pozivom *fork*. Slijedeći primjer pokazuje kako se *fork* i *exec* obično pozivaju:

```
switch (fork()) {

      case -1:
            printf("Ne mogu kreirati novi proces\n");
            break;
      case 0:
            execl("./ime","ime",NULL);
            exit(1);
      default:
            wait(NULL);
}
```

Ako *fork* ne uspije, rezultat je -1. Novi proces nije kreiran i dovoljno je ispisati odgovarajuću poruku ili pokušati ponovo. Ako je rezultat 0, nalazimo se u procesu djetetu i inicijaliziramo ga s programom `ime` bez dodatnih argumenata. Normalno nema povratka iz *execl*, ali ako on ne uspije, dijete ipak treba završiti sa *exit*. U slučaju nekog drugog rezultata poziva *fork*, radi se o nastavku procesa roditelja koji treba pričekati da dijete završi.

UPUTE za rad s naredbama ljuske operacijskog sustava za oslobađanje zauzetih računalnih resursa (zajedničke memorije, semafora i redova poruka) ukoliko dođe do nepredvidivog (!?) prekida izvođenja programa koji ih zauzima:

Naredba *ipcs*

Ova naredba daje informacije o sredstvima koja sudjeluju u komunikaciji među procesima. Bez opcija ispisuje informacije o postojećim redovima poruka, zajedničkoj memoriji i skupovima semafora.

Poziva se sa: `ipcs [ opcije ]`.

Opcije:

```
-q ispisuje informacije o aktivnim redovima poruka
-m ispisuje informacije o aktivnim segmentima zajedničke memorije
-s ispisuje informacije o aktivnim semaforima
```

Ako niti jedna od ovih opcija nije specificirana, tada se ispis može kontrolirati slijedećim opcijama:

```
-b ispisuje najveću dozvoljenu veličinu informacije (na primjer, najveći dozvoljeni broj bajtova u redu poruka)
-c ispisuje ime korisnika i njegove grupe
-o ispisuje broj poruka u redu i ukupan broj bajtova u redu poruka, odnosno broj procesa priključenih zajedničkoj memoriji
-p ispisuje identifikacijski broj procesa ( koji je zadnji poslao poruku, priključio zajedničku memoriju i slično)
-t ispisuje informacije o vremenu koje ima nekakve veze sa semaforima, redovima poruka ili zajedničkom memorijom
-a upotrijebiti sve opcije
```

Stanje se može promijeniti dok se izvršava ova naredba. pa je slika koju daje samo približna.

Naredba *ipcrm*

Ova naredba uklanja red poruka, skup semafora ili oslobađa zajedničku memoriju. U stvari uklanjaju se identifikacijski brojevi. Poziva se sa: `ipcrm [ opcije ]`.

Opcije:

```
-q msqid uklanja identifikator reda poruka msqid iz sistema
-m shmid uklanja identifikator zajedničke memorije shmid iz sistema
-s semid uklanja identifikator semafora semid
-Q msgkey uklanja identifikator reda poruka koji je kreiran s ključem msgkey
-M shmkey uklanja identifikator zajedničke memorije zauzete s ključem shmkey
-S semkey uklanja identifikator semafora kreiranog s ključem semkey
```

Primjer rada s redovima poruka: kirk.c spock.c

Pripazite! U navedenim primjerima ključ koji se koristi prilikom dobavljanja reda poruka je postavljen na 12345. Ukoliko više studenata odjednom pokreće primjer, doći do greške. Naime, red poruka je već stvoren red s pravima pristupa 0600 i nitko drugi nema pravo slati ili čitati u taj red poruka! Stoga, promijenite ključ u primjerice identifikator korisnika - UID, kojeg možete dobaviti funkcijom getuid().

PRIMJERI:

```c
/*
** kirk.c -- writes to a message queue
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;
    char text[]="Kirk: We are attacked. Spock, send reinforcement.";

    key = 12345;

    if ((msqid = msgget(key, 0600 | IPC_CREAT)) == -1) {
        perror("msgget");
        exit(1);
    }

    memcpy(buf.mtext, text, strlen(text)+1);
    buf.mtype = 1;

    if (msgsnd(msqid, (struct msgbuf *)&buf, strlen(text)+1, 0) == -1)
            perror("msgsnd");

    printf("Kirk: Reported attacks to Spock, he will send help!\n");

    return 0;
}
```

```c
/*
** spock.c -- reads from a message queue
*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <signal.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int msqid;

void retreat(int failure)
{
    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl");
        exit(1);
    }
    exit(0);
}

int main(void)
{
    struct my_msgbuf buf;
    key_t key;

    key = 12345;

    if ((msqid = msgget(key, 0600 | IPC_CREAT)) == -1) { /* connect to the queue */
        perror("msgget");
        exit(1);
    }

    sigset(SIGINT, retreat);

    for(;;) { /* Spock never quits to his captain! */
        if (msgrcv(msqid, (struct msgbuf *)&buf, sizeof(buf)-sizeof(long), 0, 0) == -1) {
            perror("msgrcv");
            exit(1);
        }
        printf("Spock: received: \"%s\". \nSending reinforcement!\n", buf.mtext);
    }

    return 0;
}
```

ZADATAK 1:

Vrtuljak

Zadatak

Modelirati vrtuljak (ringišpil) s dva tipa procesa: procesima *posjetitelj* (koje predstavljaju posjetitelje koji žele na vožnju) te jednim procesom *vrtuljak*. Procesima *posjetitelj* se ne smije dozvoliti ukrcati na vrtuljak kada više nema praznih mjesta (kojih je ukupno 4) te prije nego li svi prethodni posjetitelji sidu. Vrtuljak se može pokrenuti tek kada je pun. Na početku glavni proces, koji ujedno predstavlja proces vrtuljak, stvara 8 procesa *posjetitelja*. Procesi međusobno komuniciraju uz pomoć **reda poruka**.

Ispravno sinkronizirati 8 procesa *posjetitelja* i jednog procesa *vrtuljak* koristeći

- **raspodijeljeni centralizirani protokol** gdje je proces *vrtuljak* čvor koji je odgovoran za međusobno isključivanje (rješavaju studenti čija je **zadnja** znamenka JMBAG **parna**) ili
- protokol s putujućom značkom (rješavaju studenti čija je **zadnja** znamenka JMBAG **neparna**).

Sve što u zadatku nije zadano, riješiti na proizvoljan način.

```
Dretva posjetitelj(K) {
   ponavljaj 3 puta{
      spavaj X milisekundi; // X je slučajan broj između 100 i 2000
      pošalji vrtuljku poruku "Želim se voziti";
      po primitku poruke "Sjedi" sjedni, ispiši "Sjeo posjetitelj K" i čekaj;
      po primitku poruke "Ustani" ustani, sidi i ispiši "Sišao posjetitelj K";
   }
   ispiši i pošalji poruku vrtuljku "Posjetitelj K završio."
}

Dretva vrtuljak() {
   dok ima posjetitelja {
      čekaj posjetitelje da zažele vožnju {
          čekaj 4 poruke "Želim se voziti";
          odgovori na svaku poruku porukom "Sjedi";
      }
      pokreni vrtuljak i ispiši "Pokrenuo vrtuljak";
      spavaj X milisekundi; // X je slučajan broj između 1000 i 3000
      zaustavi vrtuljak i ispiši "Vrtuljak zaustavljen";
      pošalji posjetiteljima na vrtuljku poruku "Ustani"
   }
}
```

GRADIVO I PRIMJERI ZA 2 ZADATAK:

CJEVOVODI I IMENOVANI CJEVOVODI

CJEVOVODI

Općenito se procesi povezuju cjevovodom na slijedeći način:

1. Napraviti cjevovod.
2. Napraviti proces dijete koji će čitati.
3. U djetetu zatvoriti kraj cjevovoda na kojeg se piše i obaviti druge pripreme ako je potrebno.
4. Izvršiti program za dijete koje čita.
5. U roditelju zatvoriti kraj cjevovoda s kojeg se čita i obaviti druge pripreme ako je potrebno.
6. Ako drugo dijete treba pisati u cjevovod, stvoriti proces i izvršiti program.
7. Ako roditelj treba pisati, neka piše.

Čitanje i pisanje u cjevovod je slično radu s običnim datotekama, ali postoje i bitne razlike. Pisanje i čitanje se odvija po principu FIFO-redova: čitač čita redom kojim je pisač zapisivao. Ako čitač isprazni cjevovod on čeka podatke, a ako pisač popuni cjevovod on mora čekati dok ga čitač malo ne isprazni i oslobodi mjesto u cjevovodu. Svaki pročitani znak više ne postoji u cjevovodu i može se vratiti samo tako da ga pisač ponovo upiše.

Osnovni nedostatak komuniciranja preko cjevovoda je da procesi moraju biti povezani, na primjer roditelj i dijete. Cjevovod se ne može kreirati nakon što su procesi već stvoreni, zato što proces koji stvara cjevovod ne može prenjeti opisnik datoteke drugom procesu. Opisnici datoteka se prenose samo kod kreiranja procesa djeteta. Zato se prvo stvori cjevovod, a zatim kreira dijete koje će naslijediti opisnik datoteka cjevovoda. Procesi koji komuniciraju preko cjevovoda mogu biti roditelj i dijete, ili dvoje djece, ili "djed" i "unuk". Važno je samo da su u "srodstvu" i da je cjevovod prenesen kod "rođenja".

Cjevovodi koriste međuspremnike (*buffer cache*) veličine jednog bloka (obično 512 bajtova) kao i obične datoteke, što se može iskoristiti za povećanje efikasnosti rada. Svakim pozivom *write* moguće je upisati jedan blok podataka. Ako pisač ne piše kompletne blokove, a čitač pokušava čitati cijeli blok, čitač će dobivati nekompletne blokove. Ali ako je pisač brži od čitača onda će čitač ipak čitati kompletne blokove.

Korisnici mogu upotrebljavati cjevovode pozivima naredbi iz komandne linije, na primjer:

```
ls | wc
```

Tok podataka ima samo jedan smjer, od *ls* prema wc.

Sustavski pozivi i funkcije za rad sa cjevovodima

Cjevovod je predstavljen s dva opisnika datoteka a kreira se sustavskim pozivom *pipe*:

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

Ako je uspješno izvršen, *pipe* vraća 0, a u slučaju greške -1. *fd[1]* je deskriptor ulazne strane cjevovoda. Pisanjem u njega stavljaju se podaci u cjevovod, a čitanje iz *fd[0]* (deskriptor izlazne strane cjevovoda) vadi podatke van. Dobiveni deskriptori mogu se koristiti u pozivima za rad s datotekama: *close*, *dup*, *fcntl*, *fstat*, *read*, *write*. Izuzetak je *lseek* jer se cjevovodu može pristupati samo sekvencijalno (FIFO).

```
int close(int fd);
```

zatvara opisnik datoteke *fd*. Rezultat je 0, ili -1 u slučaju greške.

U radu s cjevovodima često je potrebno duplicirati postojeće opisnike datoteka (da bi se cjevovod povezao na standardni ulaz ili izlaz, da bi se opisnici za standardni ulaz i izlaz mogli spremiti i vratiti nakon zatvaranja cjevovoda, itd.). Za dupliciranje opisnika datoteka služi sustavski poziv *dup*:

```
int dup(int fd);
```

dup kopira postojeći opisnik datoteke *fd* i vraća novi opisnik datoteke ili -1 u slučaju greške. Poziv ne uspjeva, na primjer ako *fd* nije otvoren ili je već otvoren maksimalan broj opisnika datoteka (obično 20). Novi opisnik datoteke ima drugačiji broj od originalnog! Pravilo je da se kod otvaranja bilo kojeg novog opisnika datoteke uzima najmanji slobodni broj (to vrijedi i kod *open* i *pipe*). Najmanji brojevi 0, 1 i 2 su opisnici standardnog ulaza, standardnog izlaza i standardnog izlaza za greške.

Koristeći to pravilo, kraj cjevovoda iz kojeg se može čitati se povezuje kao standardni ulaz na slijedeći način: zatvori se opisnik datoteke 0 i duplicira se opisnik kraja za čitanje cjevovoda. *dup* će vratiti opisnik datoteke 0. Ako se zatim pokrene proces koji čita standardni ulaz, on će čitati iz cjevovoda. (Prethodno treba zatvoriti polazni opisnik kraja za čitanje cjevovoda ako ovaj više nije potreban.) Slično se postiže da opisnik datoteke 1 (standardni izlaz) bude kraj za pisanje u cjevovod. Na taj način ljuska povezuje procese koje korisnik poziva sa:

```
proc1 | proc2
```

Ako kasnije treba restaurirati opisnike datoteka za standardni ulaz i izlaz, onda ih je potrebno prvo duplicirati i zapamtiti tako dobivene opisnike. Tada se opisnici 0 i 1 mogu zatvoriti i zamijeniti, a kada ih je potrebno vratiti, primjenjuje se isti postupak za zamjenu opisnika. Na primjer, za restauraciju standardnog ulaza prvo se zatvara opisnik 0, a zatim se duplicira prije dobivena kopija standardnog ulaza. (Nakon toga se kopija može zatvoriti.)

Ponekad izbor najnižeg slobodnog broja za opisnik nove datoteke nije poželjan. Tada se umjesto *dup* može koristiti *fcntl* na slijedeći način:

```
int fcntl(fd, cmd, arg);
```

fd je deskriptor kojega se duplicira. *cmd* treba biti F_DUPFD, a odabrat će se novi deskriptor veći ili jednak *arg*.

BSD UNIX (ali ne i System V) ima još i poziv:

```
int dup2(int oldd, int newd);
```

Deskriptor *oldd* se duplicira u *newd*. Ako je *newd* već bio zauzet, on se prethodno zatvara. Rezultat je *newd* ili -1 u slučaju greške.

Čitanje i pisanje se provodi pozivima read i write kao i kod običnih datoteka:

```
int read(int fd, char *buf, unsigned nbyte);
```

pokušava pročitati *nbyte* znakova iz datoteke s opisnikom *fd* na adresu spremnika *buf*. Rezultat je broj stvarno pročitanih znakova (koji može biti manji od *nbyte*). U slučaju greške, rezultat je -1.

Ako je cjevovod prazan, *read* će čekati. Međutim, neće nužno čekati dok ne bude prisutno *nbytes* znakova nego će vratiti onoliko znakova koliko je prisutno u cjevovodu (ako ih ima manje od *nbytes*). *read* će vratiti znak kraja datoteke (rezultat 0) samo kada se zatvori opisnik kraja za pisanje u cjevovod.

```
int write(int fd, char *buf, unsigned nbyte);
```

pokušava zapisati *nbyte* znakova iz spremnika *buf* u datoteku *fd*. Rezultat je broj stvarno zapisanih znakova ili -1 u slučaju greške.

Ako se cjevovod napuni, *write* čeka dok se ne oslobodi prostor. *write* se neće djelomično obaviti već će čekati dok ne bude dovoljno mjesta za svih *nbyte* znakova. Kapacitet cjevovoda je tipično 5120 znakova (10 blokova). Ako se zatvori opisnik kraja za čitanje iz cjevovoda, *write* će završiti s greškom.

Pozivi *read* i *write* se rijetko upotrebljavaju izravno u programima jer C ima bogatu biblioteku praktičnijih funkcija za pristup datotekama (*fread*, *fwrite*, *printf*, *scanf*, itd.). Međutim, te funkcije ne rade sa opisnicima datoteka već sa kazaljkama na strukturu koja opisuje datoteku. Ta kazaljka se za obične datoteke dobiva funkcijom *fopen* (vidi man fopen), ali se može dobiti i iz opisnika datoteke funkcijom *fdopen*:

```
FILE *fdopen(int fd, char *type);
```

vraća kazaljku na strukturu koja opisuje datoteku za opisnik datoteke *fd*. U slučaju greške rezultat je NULL. *type* je niz znakova koji opisuje način pristupa datoteci i mora odgovarati načinu na koji je otvoren opisnik datoteke *fd*:

"r" čitanje

"w" pisanje

Ovo nije potrebno raditi ako se cjevovod povezuje na standardni ulaz ili izlaz jer uvijek postoje kazaljke *stdin*, *stdout* i *stderr* za standardni ulaz, standardni izlaz i standardni izlaz za poruke o greškama.

Primjer korištenja cjevovoda

Ovo je jednostavan primjer korištenja cjevovoda za prenošenje poruke iz jednog procesa u drugi. Proces prvo stvara cjevovod, a zatim jedno dijete. Dijete nasljeđuje sve podatke i opisnike, pa tako i cjevovod. Taj proces zatvara opisnik za pisanje u cjevovod i čita jednu poruku iz cjevovoda koju zatim ispisuje. S druge strane, roditelj, nakon kreiranja djeteta, zatvara opisnik za čitanje u cjevovod. Zatim šalje poruku cjevovodom i čeka da dijete završi.

Zbog jednostavnosti, uglavnom se ne provjerava je li došlo do grešaka kod pojedinih poziva. Upotrijebljeni opisnici cjevovoda se ne zatvaraju jer to ionako radi *exit*.

```
#include <stdio.h>
#include <string.h>
#define MAXREAD 20/* najveća duljina poruke*/
int main(void)
{
      int pfd[2];
      char buf[MAXREAD] = "";
      char message[] = "Ovo sijevi";/* poruka*/
      if (pipe(pfd) == -1)/* stvaranje cjevovoda*/
            exit(1);
      switch (fork()) {
            case -1: /* dijete nije kreirano*/
                  exit(1);
            case 0:/* dijete čita */
                  close(pfd[1]);/* zato zatvara kraj za pisanje*/
                  (void) read(pfd[0], buf, MAXREAD);
                  puts(buf);
                  exit(0);
            default:/* roditelj piše */
                  close(pfd[0]);/* zato zatvara kraj za čitanje*/
                  (void) write(pfd[1], message, strlen(message) + 1);
                  wait(NULL);/* roditelj čeka da dijete završi*/
            }
            exit(0);/* zatvara sve deskriptore */
      }
```

IMENOVANI CJEVOVODI

Imenovani cjevovodi (FIFO redovi) su kombinacija datoteka i cjevovoda. Oni imaju svoje ime i mjesto u sustavu datoteka, ali su označeni kao posebna vrsta datoteka (oznaka p - *pipe*). Pristupa im se korištenjem imena, kao i datotekama, pa procesi ne moraju biti u srodstvu da bi komunicirali preko njih. Imenovani cjevovod je potrebno prvo kreirati sustavskim pozivom mknod, a zatim ga treba otvoriti dva puta: jednom za čitanje, a jednom za pisanje. Ovisno o načinu pristupa navedenom kod otvaranja datoteke, proces otvara kraj za čitanje ili kraj za pisanje imenovanog cjevovoda. Nakon što su otvoreni s njima se radi kao s cjevovodima. Kapacitet imenovanih cjevovoda ovisi o implementaciji.

Kad se imenovani cjevovod otvara za čitanje, *open* čeka dok ga neki drugi proces ne otvori i za pisanje. Vrijedi i obrnuto. To dozvoljava procesima da se sinkroniziraju prije nego počne prenošenje bilo kakvih podataka.

S druge strane, kada je u *open* za čitanje postavljena zastavica O_NDELAY, on neće čekati odgovarajući *open* za pisanje, a kada je postavljena u *open* za pisanje, on će vratiti grešku ako niti jedan čitač nema otvoren isti imenovani cjevovod. Svrha ovoga je da procesi ne pišu u cjevovode koje u tom trenutku nitko ne čita zato što UNIX ne sprema podatke u njih trajno. Ako u imenovanom cjevovodu ostanu podaci nakon zatvaranja svih opisnika datoteka koji su s njim povezani, bit će izgubljeni bez dojave greške.

Ako zastavica O_NDELAY nije postavljena, *read* se zablokira kada nema podataka u imenovanom cjevovodu, a *write* se zablokira ako je prepunjen. Ako je zastavica postavljena, ni *read* ni *write* se ne zablokiraju nego javе grešku.

Sustavski pozivi za rad sa imenovanim cjevovodima

```
int mknod(char *path, int mode, int dev);
```

stvara novu datoteku čiji put (uključujući i ime) je *path*. Ovaj poziv je rezerviran za superkorisnika, osim u slučaju stvaranja imenovanog cjevovoda kada *mode* mora biti kombinacija zastavice `S_IFIFO` (oktalno 0010000, definirano u `<sys/stat.h>`) i dozvola pristupa u donjih 9 bitova, a *dev* nije bitno). Rezultat je 0, osim u slučaju greške kada je -1.

Korisnik može stvoriti imenovani cjevovod naredbom:

```
/etc/mknod ime p
```

```
int open(char *path, int flags [, int mode] );
```

path pokazuje na put (s imenom) datoteke, *flags* je neka kombinacija zastavica definiranih u `<fcntl.h>`:

- `O_RDONLY` otvori za čitanje;
- `O_WRONLY` otvori za pisanje;
- `O_RDWR` otvori za čitanje i pisanje;
- `O_NDELAY` utječe na postupak otvaranja i kasniji rad sa *read* i *write* - ako se operacija ne može obaviti bez čekanja, nema čekanja nego se vraća greška;
- `O_APPEND` dodavanje na kraj datoteke;
- `O_SYNC` write će čekati sve dok podaci ne budu stvarno zapisani na disk;
- `O_CREAT` ako datoteke nema, kreira se nova sa pravima pristupa danim u *mode*;
- `O_TRUNC` ako datoteka postoji, njen sadržaj se briše;
- `O_EXCL` ne dozvoljava korištenje postojeće datoteke ako je postavljeno `O_CREAT` .

Ako je datoteka (ili imenovani cjevovod) uspješno otvorena, *open* vraća opisnik datoteke (uvijek najmanji koji može). U slučaju greške, rezultat je -1.

Neke od zastavica postavljenih kod otvaranja, mogu se u kasnijem radu mijenjati korištenjem sustavskog poziva *fcntl* (vidi `man fcntl`):

```
int fcntl(int fd, int cmd, int arg);
```

fd je opisnik datoteke, *cmd* treba biti `F_SETFL`, a *arg* kombinacija zastavica koja se postavlja. Smatra se da je poziv uspio ako je rezultat različit od -1, ali se samo neke zastavice mogu mijenjati (na primjer, `O_NDELAY`).

Ako je *cmd* jednak `F_GETFL`, ovim pozivom se kao rezultat dobiva stanje svih zastavica za otvorenu datoteku s opisnikom *fd*.

Dalji rad sa imenovanim cjevovodima je isti kao i sa običnim cjevovodima. Koriste se pozivi *read* i *write* ili funkcije iz standardne biblioteke koje unutar sebe također koriste ove pozive za pristup podacima.

Primjer programa s imenovanim cjevovodima

PRIMJER:

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

#define MAXREAD 20 /* najveca duljina poruke*/

int main(void)
{
        int pfd;
        char buf[MAXREAD] = "";
        char message[] = "Kroz cijev!";

        unlink("./cjev");

        if (mknod("./cjev", S_IFIFO | 00600, 0)==-1)
                exit(1);

        switch (fork()) {
        case -1: /* dijete nije kreirano*/
                exit(1);

        case 0:/* dijete cita */
                pfd = open("./cjev", O_RDONLY);
                (void) read(pfd, buf, MAXREAD);
                puts(buf);
                exit(0);

        default:/* roditelj pise */
                pfd = open("./cjev", O_WRONLY);
                (void) write(pfd, message, strlen(message) + 1);
                wait(NULL);/* roditelj ceka da dijete zavrsi*/

        }
        return 0;
}
```

ZADATAK 2:

Problem pet filozofa

Problem pet filozofa. Filozofi obavljaju samo dvije različite aktivnosti: misle ili jedu. To rade na poseban način. Na jednom okruglom stolu nalazi se pet tanjura te pet štapića (između svaka dva tanjura po jedan). Filozof prilazi stolu, uzima lijevi štapić, pa desni te jede. Zatim vraća štapiće na stol i odlazi misliti.
Ako rad filozofa predstavimo jednim zadatkom onda se on može opisati na sljedeći način:

```
filozof i
    ponavljati
        misliti;
        jesti;
    do zauvijek;
```

Slika 1. Pseudokod zadatka kojeg obavlja proces filozof

Zadatak

Potrebno je pravilno sinkronizirati rad pet **procesa** filozofa koristeći:

- Lamportov raspodijeljeni protokol (rješavaju studenti čija je **predzadnja** znamenka JMBAG **parna**) ili
- protokol Ricarta i Agrawala (rješavaju studenti čija je **predzadnja** znamenka JMBAG **neparna**).

Svi procesi ispisuju poruku koju šalju i poruku koju primaju.

Sve što u zadatku nije zadano, riješiti na proizvoljan način.