**1. PREKIDI I SIGNALI**

**Upute za laboratorijske vježbe**

**SIGNALI**

Jezgra može poslati procesu signal. Taj signal može biti proizveden od same jezgre, proces ga može poslati sam sebi ili drugom procesu, ili ga može poslati korisnik.

Primjer signala koji potiče od jezgre je signal koji je poslan kada proces pokuša pristupiti memoriji koja nije u njegovom adresnom prostoru (SIGSEGV). Primjer signala koji proces šalje sam sebi je alarm (SIGALRM). Signal koji proces šalje drugom procesu je najčešće signal koji uništava procese, a šalje ga proces koji želi uništiti svoju "porodicu" procesa. Tipični korisnikov signal je prekidni signal. Standardno je postavljeno da se taj signal generira tipkom **DEL**, međutim, većina korisnika to mijenja u **Ctrl-C** (stty intr ^C).

Postoji tridesetak različitih signala (u različitim verzijama UNIX-a može ih biti manje ili više). Za većinu signala (izuzetak je SIGKILL i još neki) proces može kontrolirati što se dešava nakon što primi neki signal. Može prihvatiti ugrađenu akciju što (za većinu signala) rezultira uništavanjem procesa, može ga ignorirati ili može uhvatiti signal i izvesti određenu funkciju. Tip signala (cijeli broj) se prenosi u funkciju kao jedini argument i funkcija ne može otkriti izvor signala. Nakon povratka iz funkcije proces može nastaviti od mjesta prekida.

Značenje signala za procese je analogno značenje prekidnih signala na razini procesora. Signal može biti poslan u bilo kojem trenutku, ali ne mora biti primljen i ne mora nitko reagirati na signal. Ne sadrže nikakve informacije i može biti poslan samo određenom procesu ili procesima. Najčešće se ne koriste za normalnu komunikaciju, nego samo za posebne događaje.

Simbolička imena signala nalaze se u [signal.h](http://www.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html).

**Vrste signala**

Moderne implementacije UNIX-a definiraju oko tridesetak signala. Ovdje je pregled samo najznačajnijih, dok potpun popis daje [man -s 3HEAD signal](http://www.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html):

|  |  |
| --- | --- |
| SIGINT (2) | šalje se svakom procesu za koji je to kontrolni terminal u trenutku kada je pritisnuta tipka za prekid (standardno **DEL**, najčešće izmijenjeno u **Ctrl-C**). |
| SIGQUIT (3) | Slično kao SIGINT, ali se odnosi na tipku za kraj izvršavanja (standardno **Ctrl-\**, najčešće izmijenjeno u **Ctrl-X**). |
| SIGKILL (9) | Jedini siguran način uništavanja procesa je da mu se pošalje ovaj signal, jer ne može biti ignoriran niti uhvaćen. |
| SIGALARM (14) | šalje se kada istekne traženo vrijeme čekanja procesa. |
| SIGTERM (15) | Ovo je standardni signal za uništavanje procesa. Koristi se i kod isključivanja da ubije sve aktivne procese. Očekuje se da će proces koji ga primi uredno spremiti aktualno stanje prije završetka. |

***sigset* i njemu bliski sustavski pozivi**

void (\* [sigset](http://www.opengroup.org/onlinepubs/009695399/functions/sigpause.html)(int sig, void (\* func)())() ;

specificira što će se zbiti prilikom primitka određenog signala (maskira signal). *sig* je broj signala. Drugi argument je kazaljka na funkciju i može biti:

* SIG\_DFL. U ovom slučaju postavlja se automatska reakcija na signal. Za većinu signala to znači uništavanje procesa.
* SIG\_IGN. Ovime se postavlja da signal bude ignoriran. Signal SIGKILL se ne može ignorirati.
* SIG\_HOLD. Signal se prihvaća ali ne obrađuje i obradit će se onda kada se definicija ponašanja za taj signal promijeni. Može se čuvati samo po jedan signal svake vrste.
* Kazaljka na funkciju. Ovime se označava da po primitku signala treba pozvati funkciju. Ne može se upotrijebiti za SIGKILL. Funkciji će biti predan jedan cjelobrojni argument - vrsta signala.

Prototip je nešto teže razumijeti jer je jedan od argumenata kazaljka na funkciju, a i sama funkcija kao rezultat vraća kazaljku na funkciju - prethodnu definiciju ponašanja za zadani signal. To se može koristiti ako kasnije treba vratiti prijašnje stanje.

Nakon povratka iz funkcije za obradu signala, proces nastavlja s radom od mjesta na kojem je prekinut primitkom signala. Međutim, ako je signal prihvaćen za vrijeme čekanja na izvršenje poziva *read*, *write*, *open*, *ioctl*, *wait*, ili *pause* oni će biti prekinuti uz postavljanje greške EINTR u *errno* (ne odnosi se na pristup običnim datotekama).

Neposredno pred poziv funkcije za obradu signala *func*, ponašanje prilikom prijema dotičnog signala se mijenja u SIG\_HOLD. Na povratku iz te funkcije restaurira se *func* kao funkcija za obradu signala i otpušta eventualno pristigli signal. To se može obaviti i prije završetka same funkcije, pozivom *sigrelse*, što onda omogućava prijem novog signala prije nego što je prethodni obrađen.

int [sighold](http://www.opengroup.org/onlinepubs/009695399/functions/sigpause.html)(int sig) ;

int [sigrelse](http://www.opengroup.org/onlinepubs/009695399/functions/sigpause.html)(int sig) ;

Ovo su pozivi za manipulaciju sa signalima. *sig* je signal.

*sighold* i *sigrelse* se koriste za zaštitu kritičnih odsječaka u programu. *sighold* je analogan podizanju prioriteta i zadržavanju signala dok se prioritet ne spusti sa *sigrelse. sigrelse* obnavlja akciju procesa prethodno specificiranu u *sigset*, te otpušta ranije primljeni i zadržani signal.

**Ostali pozivi za manipulaciju signalima**

Iako su ranije opisani pozivi dovoljni za rješenje zadatka, slijedi opis još nekoliko važnijih funkcija i sustavskih poziva povezanih sa signalima:

int [pause](http://www.opengroup.org/onlinepubs/007908799/xsh/pause.html)(void) ;

zaustavlja proces do dolaska nekog signala. To ne može biti neki od signala koji se ignoriraju. Također, ako signal izazove prekid procesa, *pause* ne može ništa vratiti. Tek ako funkcija za hvatanje signala vraća nešto, *pause* se vraća s rezultatom -1 i *errno* postavljenim na EINTR.

unsigned [alarm](http://www.opengroup.org/onlinepubs/009695399/functions/alarm.html)(unsigned secs) ;

postavlja sat na vrijednost *secs* što predstavlja broj sekunda, a vraća vrijednost na koju je prethodno bio postavljen. Svaki proces ima sat spremljen u segmentu sustavskih podataka. Kada istekne postavljeno vrijeme šalje se signal SIGALRM.

unsigned [sleep](http://www.opengroup.org/onlinepubs/009695399/functions/sleep.html)(unsigned secs) ;

*alarm* i *pause* u kombinaciji formiraju standardnu funkciju *sleep*. Prvo se postavlja alarm na traženo vrijeme u sekundama, a zatim poziva *pause* da čeka prijem signala. Međutim, *sleep* može trajati i kraće od zadanog vremena jer *pause* ne čeka samo SIGALRM nego bilo koji signal. Također, *sleep* se brine o ranije postavljenim alarmima pa može završiti ranije ili će po svom završetku restaurirati od prije postojeći alarm. U svakom slučaju, *sleep* kao rezultat vraća "neprospavano" vrijeme (koliko je proces ranije probuđen zbog drugih signala). To vrijeme se onda može nadoknaditi dodatnim spavanjem.

int [usleep](http://www.opengroup.org/onlinepubs/007908799/xsh/usleep.html)(useconds\_t useconds);

Funkcija usleep je slična funkciji sleep, samo što može raditi s kraćim vremenskim intervalima od jedne sekunde. Odnosno, vrijeme 'spavanja' izražava se u mikosekundama.

int [kill](http://www.opengroup.org/onlinepubs/009695399/functions/kill.html)(int pid, int sig) ;

šalje signal procesu. *pid* je ID broj procesa koji prima signal, a *sig* je broj signala. Ako je *pid* jednak 0, signal se šalje svim procesima koji su istoj grupi kao i proces koji šalje signal. To se obično upotrebljava kod komande ljuske *kill*, čime se uništavaju svi procesi koji se izvode u pozadini, bez obzira na njihov ID.

U većini slučajeva *kill* se koristi za uništavanje procesa (od tuda dolazi ime) ili za testiranje ponašanja kod greške simuliranjem signala. Najčešće se koristi kao komanda ljuske, a ne kao poziv u programu.

void (\* [signal](http://www.opengroup.org/onlinepubs/009695399/functions/signal.html)(int sig, void (\* func)())() ;

specificira šta se dešava prilikom primitka određenog signala, slično kao i *sigset*. *signal* je stariji i razlikuje se po tome što ne poznaje SIG\_HOLD. Također, ponašanje na ulasku u funkciju za obradu signala čini ga manje upotrebljivim od *sigset*.

Neposredno pred poziv funkcije za obradu signala, ponašanje kod prijema dotičnog signala se mijenja u SIG\_DFL. To znači da će slijedeći signal istog tipa ubiti proces. Ovo ponašanje se može promijeniti unutar same funkcije za obradu signala, ali uvijek postoji interval (koji kod većeg opterećenja može biti i prilično dugačak) unutar kojeg prijem signala može ubiti proces.

Umjesto funckije sigset može se koristiti i naprednija funckija [sigaction](http://www.opengroup.org/onlinepubs/007908799/xsh/sigaction.html). Opis te funckije dan je i kratkim [primjerom](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/sigaction.c).

Osim direktno preko tipkovnice, proces može dobiti signal do drugog procesa (funkcija kill(pid,sig)). Također, iz komandne linije se naredbom [kill](http://www.opengroup.org/onlinepubs/007908799/xcu/kill.html) može poslati signal nekom od procesa (npr. kill -SIGINT 12345).

Kostur programa za obradu prekida dan je sljedećim kodom:

#include <stdio.h>

#include <signal.h>

void prekidna\_rutina(int sig)

{

/\* *obrada prekida* \*/

}

int main (void)

{

sigset (SIGTSTP, prekidna\_rutina);

printf("Poceo osnovni program PID=%d\n", getpid());

/\* *troši vrijeme da se ima šta prekinuti - 10 s* \*/

printf ("Zavrsio osnovni program\n");

return 0;

}

**Win32**

Ukoliko se želi raditi u Win32 okruženju stvari se malo kompliciraju, jer Win32 arhitektura drukčije obrađuje prekide. Naime, za obradu prekida stvara se nova dretva te program odjednom postane višedretveni!

[Više o usklađenosti te prenošenju UNIX programa na Win32.](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/misc/Unix2Win.htm)

**Vježba 1: Signali**

**Zadatak**

Napisati program koji će prihvaćati signale SIGQUIT (Ctrl+\, Ctrl+Ž na nekim tipkovnicama) i SIGTSTP (Ctrl+Z). Neka program prvih 10 sekundi svog izvođenja svake sekunde (ili i kraće, npr. svakih 300 ms) ispisuje svoje napredovanje (pogledati primjer ispisa ispod). Nakon 10. sekunde postavljaju se maske za navedene signale te program nastavlja s ispisom svog napredovanja. U proceduri za obradu signala ispisivati slovo Q ili Z (npr. svakih 100 ms) kojim se simulira obrada signala. Neka obrade prekida traju otprilike 6 sekundi.

**Napomene**

Pretpostavljeno ponašanje signala SIGQUIT jest zaustavljanje procesa, dok SIGTSTP samo suspendira izvođenje procesa i može ga se kasnije aktivirati (npr. naredbom *fg*). Ovo ponašanje može se i ispitati u prvih 10 sekundi programa, prije nego li su ta dva signala maskirana. Za prekidanje izvođenja programa može se koristiti signal SIGINT (Ctrl+C) koji se u ovom primjeru ne maskira. Ako se želi slova ispisivati jedna do drugih (kao što je prikazano u nastavku) onda je potrebno nakon svakog ispisa pozvati i funckiju fflush (npr. printf("Q"); fflush(stdout);) zbog načina na koji \*unix\* koristi konzolu za ispis.

Primjer ispisa nekoliko pokretanja programa (plavo nije ispis već dodani komentari, crveno su označeni znakovi koji su ispisani na zaslonu uslijed pritiska kombinacija tipki za izazivanje signala):

pinus:~/vj1> gcc vj1.c

pinus:~/vj1> ./a.out

Glavni program - pocetak

xxxxxxxxxxx**^C** (signal SIGINT, prekid programa)

pinus:~/vj1> ./a.out

Glavni program - pocetak

xxxxxxxxx**^Z** (signal SIGTSTP, program supendiran)

[1]+ Stopped ./a.out

pinus:~/vj1> ps

PID TTY TIME CMD

6615 pts/3 0:00 bash

7125 pts/3 0:00 a.out

pinus:~/vj1> fg (prvi program iz 'pozadine' se 'aktivira')

./a.out (ako je bio suspendiran, onda se i pokreće)

xxxxxxxxxx**^\Q**uit (signal SIGQUIT, prekid programa)

pinus:~/vj1> ps

PID TTY TIME CMD

6615 pts/3 0:00 bash

pinus:~/vj1> ./a.out

Glavni program - pocetak

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

SIGQUIT inicijaliziran (Ctrl+\)

SIGTSTP inicijaliziran (Ctrl+Z)

Glavni program - nastavak

XXXXXXXXXXX**^Z**SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSXXXXXXXXXXXXXXXX

XXXXXX**^\Q**QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQXXXXXXXXXXXXXXXXXXXXX

XXXXXX**^\Q**QQQQQQQQQQQQQQQQQQQQQ**^Z**QQQQQQQQQQQQQQQQQQQQQQQQQQQQQSSSSSSSSSSSSSSSSSSS

SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSXXXXXXXXXXXXXXXXXX**^C**

pinus:~/vj1>

**2. Višezadaćni rad**

**Upute za laboratorijske vježbe**

Višezadaćni rad moguće je ostvariti s pomoću više procesa ili s pomoću više dretvi.

**1. Ostvarenje višezadaćnog rada s pomoću više procesa**

Program je skup instrukcija i podataka koji se nalaze u datoteci na disku. U opisu datoteke ona je opisana kao izvršna i njen sadržaj je organiziran prema pravilima jezgre. Sve dok sadržaj datoteke odgovara pravilima i dok je označena kao izvršna, program može biti pokrenut. Kako bi se pokrenuo novi program prvo treba (pozivom jezgre) stvoriti novi proces koji je okolina u kojem se izvršava program.

Proces se sastoji od tri segmenta: segment instrukcija, segment korisničkih podataka i segment sustavskih podataka. Program inicijalizira segment instrukcija i korisničke podatke. Nakon inicijalizacije više nema čvste veze između procesa i programa koji on izvodi. Proces dobiva sredstva (više spremnika, datoteke, itd.) koji nisu prisutni u samom programu, mijenja podatke, itd. Iz jednog programa može se inicijalizirati više procesa koji se paralelno izvode.

**Sustavski poziv *fork***

Sustavskim pozivom *fork* zahtijeva se stvaranje novog procesa iz postojećeg. Kada proces koji se trenutno izvodi pokrene novi proces, pokrenuti proces postaje "dijete" procesa "roditelja" koji ga je pokrenuo. Dijete dobija kopije segmenta instrukcija i segmenta podataka roditelja. U stvari, budući se segment instrukcija najčešće ne mijenja, jezgra može uštediti vrijeme i memoriju tako da postavi taj segment kao zajednički za oba procesa (sve dok ga jedan od njih ne odluči inicijalizirati novim programom). Također, dijete nasljeđuje većinu sustavskih podataka roditelja.

int fork(void) ;

U ovaj sustavski poziv ulazi jedan proces, a iz njega izlaze dva odvojena procesa ("dijete" i "roditelj") koji dobivaju svaki svoju povratnu vrijednost. Proces dijete dobiva rezultat 0, a roditelj dobiva identifikacijski broj procesa djeteta. Ako dođe do greške, vraćena vrijednost je -1, a dijete nije ni stvoreno. *fork* nema nikakvih argumenata, pa programer ne može biti odgovoran za grešku već je ona rezultat nemogućnosti jezgre da kreira novi proces zbog nedostatka nekog od potrebnih sredstava.

Dijete nasljeđuje većinu atributa iz segmenta sustavskih podataka kao što su aktualni direktorij, prioritet ili identifikacijski broj korisnika. Manje je atributa koji se ne nasljeđuju:

* Identifikacijski brojevi procesa djeteta i roditelja su različiti, jer su to različiti procesi.
* Proces dijete dobiva kopije otvorenih opisnika datoteka (*file descriptor*) od roditelja. Dakle to nisu isti opisnici datoteka, tj. procesi ih ne dijele. Međutim, procesi dijele kazaljke položaja u datotekama (*file pointer*). Ako jedan proces namjesti kazaljku položaja na određeno mjesto u datoteci, drugi proces će također čitati odnosno pisati od tog mjesta. Za razliku od toga, ako dijete zatvori svoj opisnik datoteke, to nema veze s roditeljevim opisnikom datoteke.
* Vrijeme izvođenja procesa djeteta je postavljeno na nula.

Dijete se može inicijalizirati novim programom (poziv *exec*) ili izvoditi poseban dio već prisutnog programa, dok roditelj može čekati da dijete završi ili paralelno raditi nešto drugo. Osnovni oblik upotrebe sustavskog poziva *fork* izgleda ovako:

**if (fork() == 0) {**

***posao procesa djeteta***

**exit(0);**

**}**

***nastavak rada procesa roditelja (ili ništa);***

**wait(NULL);**

**Plavo** - izvodi proces roditelj, **zeleno** - izvode oba procesa (provjera povratne vrijednosti fork()-a), **crveno** - izvodi proces djeteta.

**Sustavski pozivi *exit,* *wait i getpid***

void exit(int status) ;

Poziv *exit* završava izvođenje procesa koji poziva tu funkciju. Prije završetka, uredno se zatvaraju sve otvorene datoteke. Ne vraća nikakvu vrijednost jer iza njega nema nastavka procesa. Za *status* se obično stavlja 0 ako proces normalno završava, a 1 inače. Roditelj procesa koji završava pozivom *exit* prima njegov *status* preko sustavskog poziva *wait*.

int wait(int \*statusp) ;

Ovaj sustavski poziv čeka da neki od procesa djece završi (ili bude zaustavljen za vrijeme praćenja), s tim da mu se ne može reći koji proces treba čekati. Funkcija vraća identifikacijski broj procesa djeteta koji je završio i sprema njegov status (16 bitova) u cijeli broj na koji pokazuje *statusp*, osim ako je taj argument NULL. U tom slučaju se status završenog procesa gubi. U slučaju greške (djece nema, ili je čekanje prekinuto primitkom signala) rezultat je ­1.

Postoje tri načina kako može završiti proces: pozivom *exit*, primitkom signala ili padom sustava (nestanak napajanja ili slično). Na koji je način proces završio možemo pročitati iz statusa na koji pokazuje *statusp* osim ako se radi o trećem slučaju (vidi man wait).

Ako proces roditelj završi prije svog procesa djeteta, djetetu se dodjeljuje novi roditelj - proces *init* s identifikacijskim brojem 1. *init* je važan prilikom pokretanja sustava, a u kasnijem radu većinom izvodi *wait* i tako "prikuplja izgubljenu djecu" kada završe.

Ako proces dijete završi, a roditelj ga ne čeka sa *wait*, on postaje proces-zombi (*zombie*). Otpuštaju se njegovi segmenti u radnom spremniku, ali se zadržavaju njegovi podaci u tablici procesa. Oni su potrebni sve dok roditelj ne izvede *wait* kada proces-zombi nestaje. Ako roditelj završi, a da nije pozvao *wait*, proces-zombi dobiva novog roditelja (*init*) koji će ga prikupiti sa *wait*.

pid\_t getpid() ;

Poziv *getpid* vraća identifikacijski broj procesa (PID).

**Pokretanje paralelnih procesa**

U ovoj vježbi trebat će pokrenuti više procesa tako da rade paralelno. To se može izvesti s dvije petlje. U prvoj se stvaraju procesi djeca pozivom *fork* a svako dijete poziva odgovarajuću funkciju. Iza poziva funkcije treba se nalaziti *exit* jer samo roditelj nastavlja izvršavanje petlje. Nakon izlaska iz prve petlje, roditelj poziva *wait* toliko puta koliko je procesa djece stvorio.

for (i = 0; i < N; i++)

switch (fork()) {

case 0:

*funkcija koja obavlja posao djeteta* **i**

exit(0);

case -1:

*ispis poruke o nemogućnosti stvaranja procesa;*

dafault:

*nastavak posla roditelja;*

}

while (i--) wait (NULL);

**ZAJEDNIČKI ADRESNI PROSTOR**

Nakon stvaranja novog procesa sa *fork*, procesi roditelj i dijete dijele segment s podacima koji se sastoji od stranica. Sve dok je stranica nepromjenjena oba procesa je mogu čitati. Ali čim jedan proces pokuša pisati na stranicu procesi dobiva odvojene kopije podataka. Tada niti globalne varijable nisu zajedničke za sve procese, pa ako jedan proces promjeni neku varijablu drugi to neće primijetiti. To je jedan od razloga za korištenje zajedničkog spremnika. Varijable koja trebaju biti zajedničke za sve procese moraju se nalaziti u zajedničkom spremniku kojeg prethodno treba zauzeti.

Zajednički spremnički prostor je najbrži način komunikacije među procesima. Isti spremnik je priključen adresnim prostorima dva ili više procesa. Čim je nešto upisano u zajednički spremnik, istog trenutka je dostupno svim procesima koji imaju priključen taj dio zajedničkog spremnika na svoj adresni prostor. Za sinkronizaciju čitanja i pisanja u zajednički spremnik mogu se upotrijebiti semafori, poruke ili posebni algoritmi.

Blok zajedničkog spremnika se kraće naziva segment. Može biti više zajedničkih segmenata koji su zajednički za različte kombinacije aktivnih procesa. Svaki proces može pristupiti k više segmenata. Segment je prvo stvoren izvan adresnog prostora bilo kojeg procesa, a svaki proces koji želi pristupiti segmentu izvršava sustavski poziv kojim ga veže na svoj adresni prostor. Broj segmenata je određen sklopovskim ograničenjima, a veličina segmenta može također biti ograničena.

**Sustavski pozivi za stvaranje i rad sa zajedničkim spremnikom**

typedef key\_t int;

int shmget(key\_t key, int size, int flags) ;

Ovaj sustavski poziv pretvara ključ (*key*) nekog segmenta zajedničkog spremnika u njegov identifikacijski broj ili stvara novi segment. Novi segment duljine barem *size* bajtova će biti stvoren ako se kao ključ upotrijebi IPC\_PRIVATE. U devet najnižih bitova *flags* se stavljaju dozvole pristupa (na primjer, oktalni broj 0600 znači da korisnik može čitati i pisati, a grupa i ostali ne mogu). *shmget* vraća identifikacijski broj segmenta koji je potreban u *shmat* ili -1 u slučaju greške.

Proces veže segment na svoj adresni prostor sa *shmat*:

char \*shmat(int segid, char \*addr, int flags) ;

Ako segment treba vezati na određenu adresu treba je staviti u *addr*, a ako je *addr* jednako NULL jezgra će sama odabrati adresu (moguće ako se kasnije ne koristi dinamičko zauzimanje spremnika s *malloc* ili slično). *flags* također najčešće može biti 0. *segid* je identifikacijski broj segmenta dobijen pozivom *shmget*. *shmat* vraća kazaljku na zajednički adresni prostor duljine tražene u shmget ili -1 ako dođe do greške. Dohvaćanje i spremanje podataka u segmente obavlja se na uobičajen način.

Segment se može otpustiti sustavskim pozivom *shmdt*:

int shmdt(char \*addr) ;

Zajednički spremnički prostor ostaje nedirnut i može mu se opet pristupiti tako da se ponovno veže na adresni prostor procesa, iako je moguće da pri tome dobije drugu adresu u njegovom adresnom prostoru. *addr* je adresa segmenta dobivena pozivom *shmat*.

Uništavanje segmenta zajedničke memorije izvodi se sustavskim pozivom *shmctl*:

int shmctl(int segid, int cmd, struct shmid\_ds \*sbuf) ;

Za uništavanje segmenta treba za *segid* staviti identifikacijski broj dobiven sa *shmget*, *cmd* treba biti IPC\_RMID, a *sbuf* može biti NULL. Greška je uništiti segment koji nije otpušten iz adresnog prostora svih procesa koji su ga koristili. *shmctl*, kao i *shmdt* vraća 0 ako je sve u redu, a -1 u slučaju greške. (Detaljnije o ovim pozivima u: man shmget, man shmop, man shmctl)

**Struktura programa sa paralelnim procesima i zajedničkim spremnikom**

*definiranje kazaljki na zajedničke varijable*

proces **k**

početak

*proces koji koristi zajedničke varijable*

...

kraj

...

glavni program

početak

zauzimanje zajedničke memorije

pokretanje paralelnih procesa

oslobađanje zauzete zajedničke memorije

kraj

**VAŽNO**: Varijablama u zajedničkom spremniku se nužno pristupa korištenjem kazaljki.

**Primjer programa sa paralelnim procesima i zajedničkim spremnikom**

Ovo je trivijalan primjer korištenja zajedničkog spremnika. Koristi se jedna cjelobrojna zajednička varijabla. Stvaraju se dva paralelna procesa, od kojih jedan upisuje vrijednost (različitu od 0) u tu varijablu, a drugi čeka da ona bude upisana.

#include <stdio.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

int Id; /\* *identifikacijski broj segmenta* \*/

int \*ZajednickaVarijabla;

void Pisac(int i)

{

\*ZajednickaVarijabla = i;

}

void Citac(void)

{

int i;

do {

i = \*ZajednickaVarijabla;

printf("Procitano %d\n", i);

sleep(1);

} while (i == 0);

printf("Procitano je: %d\n", i);

}

void brisi(int sig)

{

/\* *oslobađanje zajedničke memorije* \*/

(void) shmdt((char \*) ZajednickaVarijabla);

(void) shmctl(Id, IPC\_RMID, NULL);

exit(0);

}

int main(void)

{

/\* *zauzimanje zajedničke memorije* \*/

Id = shmget(IPC\_PRIVATE, sizeof(int)\*100, 0600);

/\* sizeof(int) je dovoljno memorije, ali u nekim slucajevima (pinus)

treba zatraziti vise memorije, pa je u primjeru broj pomnozen sa 100 \*/

if (Id == -1)

exit(1); /\* *greška - nema zajedničke memorije* \*/

ZajednickaVarijabla = (int \*) shmat(Id, NULL, 0);

\*ZajednickaVarijabla = 0;

sigset(SIGINT, brisi);//u slučaju prekida briši memoriju

/\* *pokretanje paralelnih procesa* \*/

if (fork() == 0) {

Citac();

exit(0);

}

if (fork() == 0) {

sleep(5);

Pisac(123);

exit(0);

}

(void) wait(NULL);

(void) wait(NULL);

brisi(0);

return 0;

}

Ako se segment zajedničkog spremnika ne uništi, zajednički adresni prostor ostaje trajno zauzet i nakon završetka svih procesa koji ga koriste, pa čak i nakon što korisnik koji ga je stvorio napusti računalo (logout). Budući je broj segmenata ograničen, to ubrzo može izazvati nemogućnost rada programa koji koriste zajednički spremnik. (Isto vrijedi i za ostala sredstva za međuprocesnu komunikaciju: skupove semafora i redove poruka.) Podaci o upotrijebljenim sredstvima za međuprocesnu komunikaciju mogu se dobiti naredbom: *ipcs*. Naredbom *ipcrm* mogu se uništavati pojedina sredstva (vidi: *man ipcrm, man ipcs*). Za lakše uništavanje zaostalih segmenata zajedničkog spremnika (kao i skupova semafora i redova poruka) može poslužiti jednostavan program *brisi*:

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/sem.h>

#include <sys/msg.h>

#include <values.h>

int main ( void )

{

int i;

for (i = 1; i < MAXLONG; i++) {

if (shmctl(i, IPC\_RMID, NULL) != -1)

printf("Obrisao zajednicku memoriju %d\n", i);

if (semctl(i, 0, IPC\_RMID, 0) != -1)

printf("Obrisao skup semafora %d\n", i);

if (msgctl(i, IPC\_RMID, NULL) != -1)

printf("Obrisao red poruka %d\n", i);

}

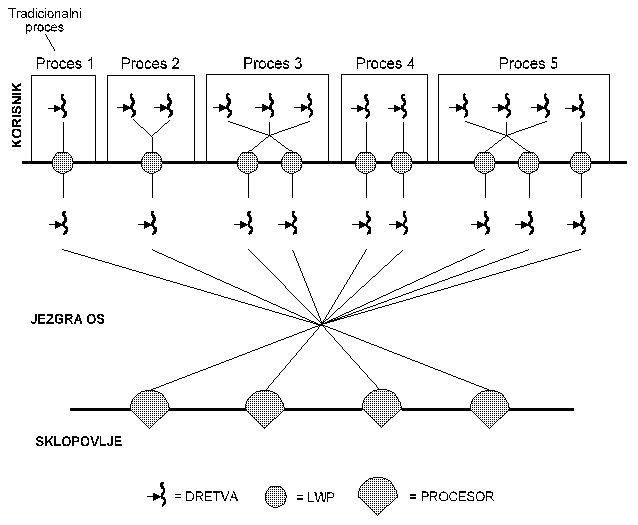
return 0;

}

**U svaki program uključiti mogućnost prijevremenog izlaska iz programa (ctrl+C) na način kao što je to ostvareno u prvoj vježbi. S tim da prekidna rutina briše zauzete sustavske resurse (semafore i zajedničku memoriju) prije no što program završi.**

**2. Višedretvenost**

Povijest višedretvenog programiranja počinje 60-tih, dok se njihova implementacija na UNIX sustavima pojavljuje sredinom 80-tih godina, a na ostalim sustavima nešto kasnije. Ideja višedretvenog programiranja jest u tome da se program sastoji od više jedinica koje se samostalno mogu izvoditi. Programer ne mora brinuti o redoslijedu njihova izvođenja, već to obavlja sam operacijski sustav. Štoviše, ukoliko je to višeprocesorski sustav, onda se neke jedinice-dretve mogu izvoditi istovremeno. Komunikacija među dretvama je jednostavna i brža u odnosu na komunikaciju među procesima, jer se obavlja preko zajedničkog adresnog prostora, te se može obaviti bez uplitanja operacijskog sustava.



***Slika: Arhitektura višedretvenog sustava Solaris 2.4***

Operacijski sustav za koji su predviđene ove laboratorijske vježbe jest *Sun*-ov *Solaris 2.4* (ili noviji)*.* Vidljivost dretvi jest samo unutar procesa, čiji su one dio i čija sredstva dijele (adresni prostor, otvorene datoteke, ...). Sljedeća stanja su jedinstvena svakoj dretvi: *identifikacijski broj dretve, registarsko stanje uključujući programsko brojilo i kazaljku stoga, stog, signalna maska, prioriti* i *privatni prostor same dretve.*

Upravljanje dretvi, odnosno, osiguravanje da se sve dretve izvode, obavlja s pomoću dretvene biblioteke *libthread*. Upravljanje se obavlja na korisničkoj razini, a ne na razini operacijskog sustava. Veza izmedu dretve i jezgre su tzv. *laki* (*lightweight*)procesi, za koje se u daljnjem tekstu koristi oznaka LWP. LWP jest osnovna dretva upravljanja na razini jezgre sustava. LWP se, sa strane programera, može predočiti jednostavno kao virtualni procesor.

Arhitektura višedretvenog sustava *Solaris 2.4* prikazana je na slici. Iz slike je vidljivo da standardni UNIX procesi imaju samo jedan LWP, odnosno, jednu dretvu. Također, svaka dretva ne mora imati vlastiti LWP, već ih više njih može koristiti iste LWP-ove, odnosno, razlikujemo dvije skupine dretvi: *vezane* (engl. *bound threads*) i *nevezane* (engl. *unbound threads*). Prve, *vezane*, su takve dretve kojima je pridjeljen vlastiti LWP koji izvodi samo njene instrukcije, dok druge, *nevezane*, nisu vezane za vlastiti LWP, već se mogu izvoditi na bilo kojem LWP-u koji se nalazi na raspolaganju procesu, a nije vezan za jednu dretvu. Vezane dretve zauzimaju nešto više sredstava sustava, ali su zbog izbjegavanja učestalih promjena dretvi na LWP-u, brže.

Kod drugih operacijskih sustava status dretvi je drugačiji, a na neki način bi se mogao shavatiti kao na gornjoj slici za procese 1 i 4, tj. svaku dretvu raspoređuje OS.

**Funkcije za rukovanje dretvama**

Iako *Solaris* ima vlastite funkcije za rukovanje dretvama u nastavku su objašnjene funkcije po POSIX standardu koje izgledaju vrlo slično. Pogledati man threads.

***Stvaranje dretvi***

Sve dretve, osim prve, inicijalne, koja nastaje stvaranjem procesa, nastaju pozivom pthread\_create:

int pthread*\_*create(pthread*\_*t \*thread, const pthread*\_*attr*\_*t \*attr,

void \*(\*start*\_*routine)(void \*), void \*arg);

thread je kazaljka na mjesto u memoriji gdje se sprema id novostvorene dretve i za razliku od *Solaris*-ove funkcije za stvaranje dretve ovaj parametar nesmije biti NULL. attr je adresa strukture koja sadrži podatke o atributima s kojima se želi stvoriti dretvu. Ako se za attr postavi NULL onda se uzimaju pretpostavljene vrijednosti (dovoljno dobre za lab. vježbe). start\_routine predstavlja pokazivač na početnu funkciju koju će novostvorena dretva imati kao početnu (npr. kao što glavna dretva ima funkciju main). arg je adresa parametra koji se prenosi dretvi (može biti NULL ako se ništa ne prenosi). Pošto se može prenijeti samo jedan parametar u slučaju potrebe prijenosa više parametara oni se pohranjuju u strukturu te se šalje pokazivač na tu struktru.

***Završetak rada dretve***

Normalan završetak dretve jest njen izlazak iz prve, inicijalne funkcije, ili pozivom funkcije pthread\_exit:

int pthread\_exit(void \*status);

status je kazaljka na stanje sa kojim dretva završava.

Dretva čeka na završetak druge dretve pozivom funkcije pthread\_join:

int pthread\_join( pthread\_t cekana\_dr, void \*\*stanje);

cekana\_dr je identifikacijski broj dretve na čiji se kraj čeka, ~~ukoliko nije nula, kada se čeka na završetak bilo koje dretve~~ (thr\_join). stanje je kazaljka na kazaljku izlaznog statusa dočekane dretve. Funkcija pthread\_join zaustavlja izvođenje pozivajuće dretve sve dok određena dretva ne završi sa radom. Nakon ispravnog završetka funkcija vraća nulu.

Normalni završetak višedretvenog programa zbiva se kada sve dretve završe s radom, odnosno, kada prva, početna dretva izađe iz prve funkcije (*main*)*.* Prijevremeni završetak zbiva se pozivom funkcije *exit* od strane bilo koje dretve, ili pak nekim vanjskim signalom (SIGKILL, SIGSEGV, SIGINT, SIGTERM, ...).

Primjer jednog višedretvenog programa koji koriste istu varijablu:

#include <stdio.h>

#include <pthread.h>

int ZajednickaVarijabla;

void \*Pisac(void \*x)

{

ZajednickaVarijabla = \*((int\*)x);

}

void \*Citac(void \*x)

{

int i;

do {

i = ZajednickaVarijabla;

printf("Procitano %d\n", i);

sleep(1);

} while (i == 0);

printf("Procitano je: %d\n", i);

}

int main(void)

{

int i;

pthread\_t thr\_id[2];

ZajednickaVarijabla = 0;

i=123;

/\* pokretanje dretvi \*/

if (pthread\_create(&thr\_id[0], NULL, Citac, NULL) != 0) {

printf("Greska pri stvaranju dretve!\n");

exit(1);

}

sleep(5);

if (pthread\_create(&thr\_id[1], NULL, Pisac, &i) != 0) {

printf("Greska pri stvaranju dretve!\n");

exit(1);

}

pthread\_join(thr\_id[0], NULL);

pthread\_join(thr\_id[1], NULL);

return 0;

}

Identifikacijski broj dretve moguće je dobiti pozivom funkcije pthread\_self:

pthread\_t pthread\_self(void);

*Napomene*

Prilikom prevođenja potrebno je postaviti zastavicu -D\_REENTRANT koja ukazuje na to da se koriste višedretvene inačice upotrijebljenih funkcija, ako takve postoje, te zastavicu -*lpthread* (npr. gcc -D\_REENTRANT -lpthread prvi.c -o prvi).

Stranice (manual) POSIX dretvi u kojima su detaljno opisane funkcije za rad s dretvama *pthread*: [pthread](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread.htm), [pthread\_create](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_create.htm), [pthread\_exit](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_exit.htm), [pthread\_detach](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_detach.htm), [pthread\_join](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_join.htm), [pthread\_mutex\_init](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_mutex_init.htm), [pthread\_mutex\_lock](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_mutex_lock.htm), [pthread\_mutex\_unlock](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_mutex_unlock.htm), [pthread\_mutex\_destroy](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_mutex_destroy.htm), [pthread\_cond\_init](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_cond_init.htm), [pthread\_cond\_wait](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_cond_wait.htm), [pthread\_cond\_signal](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_cond_signal.htm), [sem\_init](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/sem_init.htm), [sem\_wait](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/sem_wait.htm), [sem\_post](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/sem_post.htm), [sem\_destroy](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/sem_destroy.htm)...

**Win32**

Stvaranje procesa pod Win32 obavlja se funkcijom [CreateProcess](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/createprocess.asp)(). [Primjer.](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/misc/winprocesi.c)

Zajednička memorija ostvaruje se pomoću funkcija [CreateFileMapping](http://msdn2.microsoft.com/en-us/library/aa366537.aspx) i [MapViewOfFile](http://msdn2.microsoft.com/en-us/library/aa366761.aspx). [Primjer](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/misc/Unix2Win.htm#Windows_example:_shared_memory)

Stvaranje dretvi pod Win32 obavlja se funkcijom [CreateThread](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/createthread.asp)(). [Primjer.](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/misc/windretve.c)

**Dretve**

**Zadatak**

Napisati program koji stvara N novih dretvi (N zadati kao argument komandne linije). Svaka dretva u svakoj od M sekundi svog izvođenja (M zadan kao parametar komandne linije) generira slučajan prosti broj koji i ispisuje.

**Napomene**

Za generiranje slučajnog prostog broja može se koristiti kod [random\_prime.c](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/random_prime.c)

**Primjer ispisa prilikom pokretanja programa**

pinus:~/vj2> ./a.out 3 6

dretva tid=4, broj=22973

dretva tid=5, broj=10477

dretva tid=6, broj=25763

dretva tid=4, broj=7297

dretva tid=5, broj=15461

dretva tid=6, broj=28163

dretva tid=5, broj=28771

dretva tid=6, broj=15803

dretva tid=4, broj=11149

dretva tid=6, broj=19379

dretva tid=5, broj=19001

dretva tid=4, broj=26501

dretva tid=5, broj=2879

dretva tid=4, broj=29243

dretva tid=6, broj=21139

dretva tid=5, broj=157

dretva tid=4, broj=31357

dretva tid=6, broj=29129

Dretva tid=5 zavrsila!

Dretva tid=4 zavrsila!

Dretva tid=6 zavrsila!

pinus:~/vj2>

**Procesi**

**Zadatak**

Napisati program koji stvara N novih procesa (N zadati kao argument komandne linije). Svaki proces u svakoj od M sekundi svog izvođenja (M zadan kao parametar komandne linije) generira slučajan prosti broj koji i ispisuje.

**Napomene**

Za generiranje slučajnog prostog broja može se koristiti kod [random\_prime.c](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/random_prime.c)

**Primjer ispisa prilikom pokretanja programa**

pinus:~/vj3> ./a.out 3 6

dretva tid=4, broj=22973

dretva tid=5, broj=10477

dretva tid=6, broj=25763

dretva tid=4, broj=7297

dretva tid=5, broj=15461

dretva tid=6, broj=28163

dretva tid=5, broj=28771

dretva tid=6, broj=15803

dretva tid=4, broj=11149

dretva tid=6, broj=19379

dretva tid=5, broj=19001

dretva tid=4, broj=26501

dretva tid=5, broj=2879

dretva tid=4, broj=29243

dretva tid=6, broj=21139

dretva tid=5, broj=157

dretva tid=4, broj=31357

dretva tid=6, broj=29129

Dretva tid=5 zavrsila!

Dretva tid=4 zavrsila!

Dretva tid=6 zavrsila!

pinus:~/vj3>

**Dekkerov postupak međusobnog isključivanja**

**Zadatak**

Ostvariti sustav paralelnih procesa/dretvi. Struktura procesa/dretvi dana je sljedećim pseudokodom:

proces proc(**i**) /\* **i** [0..**n**-1] \*/

za **k** = 1 do 5 čini

**uđi u kritični odsječak**

za **m** = 1 do 5 čini

ispiši (**i**, **k**, **m**)

**izađi iz kritičnog odsječka**

kraj.

Međusobno isključivanje ostvariti za dva procesa/dretve međusobnim isključivanjem po Dekkerovom algoritmu.

**Dekkerov algoritam:**

zajedničke varijable: PRAVO, ZASTAVICA[0..1]

funkcija uđi\_u\_kritični\_odsječak(i,j) {

ZASTAVICA[i] = 1

dok je ZASTAVICA[j]<>0 čini {

ako je PRAVO=j onda {

ZASTAVICA[i] = 0

dok je PRAVO=j čini {

ništa

}

ZASTAVICA[i] = 1

}

}

} kraj.

funkcija izađi\_iz\_kritičnog\_odsječka(i,j) {

PRAVO = j

ZASTAVICA[i] = 0

} kraj.

**Upute:**

Ako se program rješava s procesima tada treba zajedničke varijable tako organizirati da se prostor za njih zauzme odjednom i podijeli među njima. Ovo je nužno zbog ograničenog broja segmenata i velikog broja korisnika.

Ovisno o opterećenju računala i broju procesa koji se pokreću, a da bi se vidjele razlike prilikom izvođenja programa može biti potrebno usporiti izvršavanje sa:

sleep(1);

nakon ispisi (**i**, **k**, **m**).

# 3. Sinkronizacijski mehanizmi

Sinkronizacijske mehanizme podijeliti ćemo na one koji služe za sinhronizaciju procesa te na one koji služe za sinkronizaciju dretvi.

## 1. Sinkronizacija procesa semaforima

Sinkronizacija procesa se uglavnom obavlja semaforima. Ostali mehanizmi (čija primarna namjena nije sinkronizacija, poput redova poruka, cjevovoda, i sl.) rijetko se koriste za sinkronizaciju procesa. Semafori se također mogu koristiti i kod dretvi (ima i posebnih semafora za dretve), ali se kod dretvi najčešće koriste mehanizmi međusobnog isključivanja korištenjem tzv. mutexa te uvjetnih varijabli.

Semafor je mehanizam koji sprečava da dva ili procesa pristupaju zajedničkom sredstvu istovremeno. Binarni semafor ima dva stanja: propusno i nepropusno. Opći semafor ima beskonačan broj stanja (ili barem vrlo velik). To je brojač koji se smanjuje za jedan kada se zahtijeva semafor, a povećava se za jedan kada se oslobađa. Ako je na nuli, a proces zahtijeva semafor, tada taj proces mora čekati dok drugi proces ne poveća vrijednost semafora. Semafor u UNIX-u (System V) ne može imati negativnu vrijednost iako su teoretski ostvarivi i semafori kod kojih bi bile dozvoljene i negativne vrijednosti. Upotreba semafora se obično razmatra kroz dvije jednostavne operacije: "dohvati" i "otpusti" (*acquire* i *release*) ili "čekaj" i "postavi" (*wait* - *signal*, odnosno, izvorno iz nizozemskog: P i V).

### Semafori u UNIX-u

Pozivi za korištenje semafora u UNIX-u su daleko od jednostavnosti operacija dohvati i otpusti. Dapače, to su izuzetno složeni mehanizmi (vidi: man semget, man semop, man semctl). Zbog toga ćemo se ograničiti samo na dio funkcionalnosti poziva za rad sa semaforima koji je dovoljan za većinu primjena.

### Dobavljanje semafora

Semafori se dobavljaju sustavskim pozivom *semget* slično kao i segment zajedničkog spremnika:

int semget(key\_t key, int nsems, int flags) ;

Ovaj sustavski poziv dohvaća skup semafora koji se mogu kontrolirati svi odjednom ili stvara novi skup semafora. Novi skup sa ukupno *nsem* semafora će biti stvoren ako se za ključ upotrijebi IPC\_PRIVATE. U devet najnižih bitova *flags* se stavljaju dozvole pristupa. *semget* vraća identifikacijski broj skupa semafora ili -1 u slučaju greške. Redni brojevi semafora u skupu počinju od nule. Zbog jednostavnosti bilo bi dobro imati uvijek samo po jedan semafor u skupu. Međutim, kao i kod zajedničkog spremnika, ukupan broj skupova semafora u sustavu je ograničen. Zbog velikog broja korisnika taj je broj lako premašiti, pa je poželjno sve potrebne semafore uvijek dobavljati odjednom, tj. u jednom skupu.

### Postavljanje početne vrijednosti semafora

Kada se stvara novi skup semafora, vrijednosti svih semafora u skupu se postavljaju na 0. Ova vrijednost se može promijeniti sustavskim pozivom *semctl*, ali to je samo jedna od brojnih operacija koje on obavlja:

union semun {

int val;

struct semid\_ds \*buf;

unsigned short \*array;

};

int semctl(int semid, int semnum, int cmd, union semun \*arg) ; //Solaris

int semctl(int semid, int semnum, int cmd, union semun arg) ; //Ostali, više/manje

*semid* je uvijek identifikacijski broj skupa semafora. Vrijednost semafora se može postaviti na dva načina:

1. Ako je *semnum* redni broj semafora, *cmd* SETVAL, a *arg* nenegativan cijeli broj (*arg.val*), postavlja se vrijednost određenog semafora.
2. Ako je *cmd* SETALL, a *arg* kazaljka na niz kratkih cijelih brojeva bez predznaka (*arg.array*), postavljaju se svi semafori u skupu na vrijednosti iz danog niza.

*semctl* pozvan na ovaj način vraća 0 ako je sve u redu ili -1 u slučaju greške (na primjer, vrijednost na koju treba postaviti semafor je prevelika ili negativna).

### Uništavanje semafora

Skup semafora treba uništiti nakon upotrebe jer u protivnom ostaje kao trajno zauzeto sredstvo u sustavu. Uništavanje semafora također se provodi sustavskim pozivom *semctl*. *semid* je identifikacijski broj skupa semafora, a *cmd* treba biti IPC\_RMID. Ostali argumenti nisu bitni.

### Operacije sa semaforima

### struct sembuf {

short sem\_num;

short sem\_op;

short sem\_flg;

};

int semop(int semid, struct sembuf (\*sops)[], int nsops) ;

*semop* se može upotrijebiti za nedjeljivo izvođenje niza operacija na skupu semafora. Međutim, najčešće je sasvim dovoljno izvesti jednu operaciju u jednom pozivu. U tom slučaju, *semid* je identifikacijski broj skupa semafora, *nsops* je 1, a *sops* je kazaljka na strukturu koja opisuje traženu operaciju. Unutar te strukture, *sem\_num* je redni broj semafora u skupu, *sem\_op* je tražena operacija koju dodatno opisuje i *sem\_flg*.

Za uobičajeno korištenje semafora dovoljne su dvije vrste operacija: povećavanje vrijednosti semafora (*sem\_op* je pozitivan) i smanjivanje vrijednosti semafora (*sem\_op* je negativan). U oba slučaja, *sem\_flg* treba biti 0. *semop* vraća rezultat 0 ako je sve u redu, ili -1 u slučaju greške.

Smanjivanje vrijednosti semafora odgovara operaciji "čekaj". Ukoliko je vrijednost semafora moguće umanjiti za zadani broj, a da ne postane negativna, vrijednost se umanjuje, a proces nastavlja rad. U suprotnom slučaju, proces čeka dok vrijednost semafora ne postane dovoljno velika da se umanjivanje može provesti.

Povećavanje vrijednosti odgovara operaciji "postavi". Vrijednost semafora se povećava za zadani broj, a proces nastavlja rad. Proces zaustavljen u redu semafora se oslobađa ako je ovim povećanjem vrijednost semafora postala dovoljno velika da je on može umanjiti, a da ipak ne postane negativna.

### Pomoćne funkcije za rad sa semaforima

Prethodna objašnjenja rada sa semaforima mogu se upotrijebiti za izradu pomoćnih funkcija za rad sa semaforima koje su mnogo jednostavnije za upotrebu od izravnih sustavskih poziva.

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

int SemId; /\* *identifikacijski broj skupa semafora* \*/

void SemGet(int n)

{ /\* *dobavi skup semafora sa ukupno* ***n*** *semafora* \*/

SemId = semget(IPC\_PRIVATE, n, 0600);

if (SemId == -1) {

printf("Nema semafora!\n");

exit(1);

}

}

int SemSetVal(int SemNum, int SemVal)

{ /\* *postavi vrijednost semafora* ***SemNum*** *na* ***SemVal*** \*/

return semctl(SemId, SemNum, SETVAL, &SemVal);

}

int SemOp(int SemNum, int SemOp)

{ /\* *obavi operaciju* ***SemOp*** *sa semaforom* ***SemNum*** \*/

struct sembuf SemBuf;

SemBuf.sem\_num = SemNum;

SemBuf.sem\_op = SemOp;

SemBuf.sem\_flg = 0;

return semop(SemId, & SemBuf, 1);

}

void SemRemove(void)

{ /\* *uništi skup semafora* \*/

(void) semctl(SemId, 0, IPC\_RMID, 0);

}

[ZAJEDNIČKI ADRESNI PROSTOR](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/visezadacnirad.html#ZAJEDNICKI_ADRESNI_PROSTOR)

## 2. Sinkronizacija dretvi

Istovremeni pristup podacima od strane više dretvi se ponekad ne smije dopustiti. To su situacije u kojima dretve koriste zajedničke varijable, a čija bi istovremena uporaba izazvala greške. Takvi se kritični odsječci programa zaštićuju međusobnim isključivanjem. Kod korištenja zajedničkih podataka treba također uzeti u obzir da promjena koju uzrokuje jedna dretva ne mora baš istog trenutka biti vidljiva svim ostalim dretvama. Odnosno, zbog toga što procesori koriste vlastite priručne spremnike, promjena podatka se privremeno može obaviti samo lokalno, a ažuriranje glavnog spremnika može se obaviti tek kasnije. Takvi se dijelovi programa također zaštićuju.

UNIX operacijski sustav (*SUN Solaris, a i Linux*) omogućuje sljedeće načine sinkronizacije među dretvama: *međusobno isključivanje, uvjetne varijable* i *semafori*. Kada je potrebno zaštititi neke dijelove programa od istovremenog korištenja više dretvi (kritični odsječci), tada se koriste funkcije međusobnog isključivanja, odnosno, dijelovi programa se zaključuju pomoću kontrolnih varijabli - ključeva. Inicijalizacija ključeva se obavlja funkcijom pthread\_mutex\_init, zaključavanje funkcijom pthread\_mutex\_lock, a otključavanje funkcijom pthread\_mutex\_unlock (*Solaris* ima i svoje vlastite funkcije za te operacije koje nemaju prefix "pthread\_")

int pthread\_mutex\_init(pthread\_mutex\_t \*ključ, const pthread\_mutexattr\_t \*attr);  
int pthread\_mutex\_lock(pthread\_mutex\_t \*ključ);  
int pthread\_mutex\_unlock(pthread\_mutex\_t \*ključ);

ključ pokazuje na varijablu zaključavanja, attr definira karakteristike stvorene varijable (NULL za pretpostavljene vrijednosti). Sve dretve, koje pokušaju zaključati već zaključanu varijablu ostaju blokirane na pozivu sve dok varijabla ostaje zaključana. Kada se varijabla otključa, onda samo jedna dretva ulazi slijedeća u kritični osječak, uz zaključavanje varijable. Zaključavanjem se smanjuje moguća istovremenost u izvođenju skupine dretvi, te je njihova upotreba prihvatljiva (obavezna) samo u kritičnim odsječcima.

***Uvjetne varijable*** se koriste kada želimo da neka dretva zaustavi svoje izvođenje te čeka da se određeni uvjet ispuni, tj. da ga ispuni neka druga dretva. Funkcijama za korištenje uvjetnih varijabli obavezno prethodi zaključavanje, budući su uvjetne varijable globalne, tj. zajedničke cijelom procesu.

Osnovne funkcije za rukovanje uvjetnim varijablama su pthread\_cond\_init, pthread\_cond\_wait, pthread\_cond\_signal i pthread\_cond\_broadcast.

int pthread\_cond\_init(pthread\_cond\_t \*uvjet, const pthread\_condattr\_t \*attr);  
int pthread\_cond\_wait(pthread\_cond\_t \*uvjet, pthread\_mutex\_t \*ključ);  
int pthread\_cond\_signal(pthread\_cond\_t \*uvjet);  
int pthread\_cond\_broadcast(pthread\_cond\_t \*uvjet);

uvjet je kazaljka na uvjetnu varijablu, ključ jest kazaljka na varijablu zaključavanja, attr odefinira karakteristike stvorene varijable (NULL za pretpostavljene vrijednosti). Pozivom pthread\_cond\_wait pozivajuća dretva se postavlja u stanje čekanja tj. premješta ju se u red uvjet i istovremeno se otljučava ključ. Čekanje te dretve završava neka druga dretva pozivima pthread\_cond\_signal ili pthread\_cond\_broadcast. Po primitku "signala" dretva prije nastavka rada najprije mora ponovno zaključati ključ. Poziv pthread\_cond\_signal ispunjuje uvjet za nastavaka samo jedne dretve iz reda čekanja na istu uvjetnu varijablu, dok poziv pthread\_cond\_broadcast omogućuje svim takvim dretvama nastavak izvođenja. Ako niti jedna dretva nije bila blokirana na uvjetnoj varijabli prilikom poziva pthread\_cond\_signali pthread\_cond\_broadcast, onda ovi pozivi nemaju nikakav učinak, tj. ako već u slijedećem trenutku neka dretva pozove pthread\_cond\_wait ona ostaje blokirana. Uvjetnim varijablama moguće je ostvariti sustav monitora - mehanizma kojim je moguće istovremeno provjeriti više uvjeta (zauzeti resurse) potrebnih za napredovanje dretve.

Poziv pthread\_cond\_wait(uvjet, ključ) može se shvatiti u dva koraka: u prvom se otključava ključ i dretva pomiče u red uvjet; u drugom, nakon primitka signala se mora ponovno zaključati ključ. Tj. mogli bi reći da funkcija izgleda:

pthread\_cond\_wait(uvjet, ključ) {  
   oslobodi(ključ); //funkcionalno skoro isto što i mutex\_unlock()  
   uvrsti\_u\_red\_uvjeta(uvjet);  
  
   zauzmi(ključ); //funkcionalno isto što i mutex\_lock()  
}

***Monitor*** se sastoji od skupa procedura i strukture podataka nad kojima procedure djeluju i koje nisu vidljive izvan monitora. Procedure se ne smiju izvoditi paralelno. Unutar monitora ispituje se uvjet koji utječe na odvijanje zadatka. Ukoliko je uvjet ispunjen, zadatak zauzima sredstva i obavlja potrebne akcije nad njima unutar monitora. Po završetku oslobađa sredstva te dozvoljava drugom zadatku ulazak u monitor te izlazi iz monitora. Ukoliko po ulasku u monitor uvjet nije ispunjen tada zadatak odlazi u red čekanja na taj uvjet, tj. prividno napušta monitor.

Ostvarenje monitora u višedretvenom programu prilično je jednostavna budući na raspolaganju stoje funkcije međusobnog isključavanja čime se jednostavno postiže da samo jedna dretva ulazi u monitor, te funkcije za rukovanje uvjetnim varijablama koje služe za ostvarenje reda uvjeta. Ostvarenje monitora koji zauzima sredstva p i q prikazan je u nastavku:

void uđi\_u\_monitor(cond\_t \*uvjet){

pthread\_mutex\_lock(&monitor);

while(p==0 || q==0)

pthread\_cond\_wait(uvjet,&monitor);

p=q=0;

pthread\_mutex\_unlock(&monitor);

}

void izađi\_iz\_monitora(cond\_t \*uvjet){

pthread\_mutex\_lock(&monitor);

p=q=1;

pthread\_cond\_broadcast(uvjet);

pthread\_mutex\_unlock(&monitor);

}

***Semafori*** se obično upotrebljavaju za pristup *ograničenim* sredstvima od strane većeg broja korisnika (dretvi), tj. služe kao brojači događaja. Osnovne funkcije za rad sa semaforima u višedretvenom programu su: sem\_init, sem\_postisem\_wait.

int sem\_init(sem\_t \*sem, int mprocesi, unsigned int koliko);  
int sem\_post(sem\_t \*sem);  
int sem\_wait(sem\_t \*sem);

sem je pokazivač na varijablu semafora, koliko je broj na koji se postavi semafor, mprocesi označava je li semafor predviđen za sinkronizaciju dretvi istog procesa (0) ili dretvi različitih procesa (nešto različito od nule). Funkcija sem\_post jedinično povećava semafor. Funkcija sem\_wait smanjuje vrijednost semafora za jedan, ako je vrijednost semafora veća od nule, u protivnom čeka dok se vrijednost ne poveća.

Stranice (manual) POSIX dretvi u kojima su detaljno opisane funkcije za rad s dretvama *pthread*: [pthread](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread.htm), [pthread\_create](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_create.htm), [pthread\_exit](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_exit.htm), [pthread\_detach](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_detach.htm), [pthread\_join](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_join.htm), [pthread\_mutex\_init](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_mutex_init.htm), [pthread\_mutex\_lock](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_mutex_lock.htm), [pthread\_mutex\_unlock](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_mutex_unlock.htm), [pthread\_mutex\_destroy](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_mutex_destroy.htm), [pthread\_cond\_init](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_cond_init.htm), [pthread\_cond\_wait](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_cond_wait.htm), [pthread\_cond\_signal](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/pthread_cond_signal.htm), [sem\_init](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/sem_init.htm), [sem\_wait](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/sem_wait.htm), [sem\_post](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/sem_post.htm), [sem\_destroy](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/pthread/sem_destroy.htm)...

**Semafori**

**Zadatak**

Ostvariti komunikaciju među procesima koji se paralelno izvode korištenjem međuspremnika ograničene duljine. Dva proizvođača trebaju na početku pročitati niz znakova s tipkovnice i nakon toga poslati znak po znak preko međuspremnika potrošaču. Potrošač prima znak po znak od proizvođača i nakon što je primio sve znakove od oba proizvođača, treba ih ispisati na zaslonu.

Struktura rješenja

Podaci zajednički za proizvođače i potrošače (smjestiti ih u [zajednički spremnik](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/visezadacnirad.html#ZAJEDNICKI_ADRESNI_PROSTOR)):

* cjelobrojne varijable ULAZ i IZLAZ
* međuspremnik M, polje od 5 mjesta
* semafori PIŠI, PUN, PRAZAN

Struktura procesa proizvođač:

proces proizvođač

pročitaj niz znakova sa tastature u polje s

i = 0

čini

čekaj\_PUN

čekaj\_PIŠI

M[ULAZ] = s[i]

ULAZ = (ULAZ+1) mod 5

postavi\_PIŠI

postavi\_PRAZAN

i = i+1

do kraja niza

kraj.

Struktura procesa potrošač:

proces potrošač

i = 0

čini

čekaj\_PRAZAN

s[i] = M[IZLAZ]

IZLAZ = (IZLAZ+1) mod 5

postavi\_PUN

i = i+1

do kraja oba niza

ispiši niz znakova iz polja s na zaslon

kraj.

**Napomene**

Pripaziti na ispravnu inicijalizaciju semafora i varijable ULAZ i IZLAZ. Čitanje znakova s tipkovnice te početak glavnih petlji proizvođača treba uskladiti korištenjem (dodatnih) semafora kako se ne bi desilo da jedan od procesa proizvođača završi kada drugi još prikuplja niz znakova s tipkovnice. Oznaku kraja niza također prenositi koristeći međuspremnik kako bi proces potrošač mogao znati kada nastupa kraj niza. Proces potrošač treba ispisati primljene znakove i završiti tek kada primi dvije oznake kraja niza. "Usporiti" rad procesa proizvođača s jednim pozivom sleep(1) unutar petlje.

**Primjer ispisa pokretanja programa**

# ./a.out

Unesi znakove za proizvodjaca 1: 12345678

Unesi znakove za proizvodjaca 2: abcdef

PROIZVOÐAČ1 -> 1

PROIZVOÐAČ2 -> a

POTROSAC <- 1

PROIZVOÐAČ1 -> 2

POTROSAC <- a

PROIZVOÐAČ2 -> b

POTROSAC <- 2

PROIZVOÐAČ1 -> 3

POTROSAC <- b

PROIZVOÐAČ2 -> c

POTROSAC <- 3

PRPROIZVOÐAČ1 1 -> 4

POTROSAC <- c

PROIZVOÐAČ1 -> 5

POTROSAC <- 4

PROIZVOÐAČ2 -> d

POTROSAC <- 5

PROIZVOÐAČ1 -> 6

POTROSAC <- d

PROIZVOÐAČ2 -> e

POTROSAC <- 6

PROIZVOÐAČ2 -> f

POTROSAC <- e

PROIZVOÐAČ1 -> 7

POTROSAC <- f

PROIZVOÐAČ2 ->

POTROSAC <- 7

PROIZVOÐAČ1 -> 8

POTROSAC <-

PROIZVOÐAČ1 ->

POTROSAC <- 8

POTROSAC <-

Primljeno je 1a2b3c45d6ef78

#

**Monitori**

**Zadatak**

S pomoću više dretvi riješiti problem pet filozofa koristeći koncept monitora. Pri svakoj promjeni program mora vizualno prikazati za sve filozofe što oni rade. Npr. kada filozof 4 ide jesti, tada treba ispis izgledati otprilike ovako: "Stanje filozofa: X o O X o" (X-jede, O-razmišlja, o-čeka na vilice).

*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 5.1. Pseudokod dretve filozof

Potrebno je pravilno sinkronizirati rad filozofa. Uporabom binarnog semafora za pojedine štapiće može doći do potpunog zastoja (kada svi istovremeno uzmu lijevi štapić). Problem se može riješiti uvođenjem dodatnog općeg semafora koji će ograničavati broj filozofa za stolom. Međutim, u općem slučaju kada zadaci dijele više sredstava i koriste ih više istovremeno semafori mogu uzrokovati potpuni zastoj. Upravo zbog takvih problema dijeljenja više sredstava među više zadataka uvodi se novi mehanizam za sinkronizaciju: monitori.   
Monitor se sastoji od skupa procedura i strukture podataka nad kojima procedure djeluju i koje nisu vidljive izvan monitora. Procedure su tako konstruirane da djeluju samo jedna po jedna . Unutar monitora ispituje se uvjet koji utječe na odvijanje zadatka. Ukoliko je uvjet ispunjen, zadatak zauzima sredstva i obavlja potrebne akcije. Po završetku oslobađa sredstva te dozvoljava drugom zadatku ulazak u monitor te izlazi iz monitora. Ukoliko po ulasku u monitor uvjet nije ispunjen tada zadatak odlazi u red čekanja na taj uvjet, tj. prividno napušta monitor.   
Za ostvarenje monitora u višedretvenom programu stoje na raspolaganju funkcije međusobnog isključavanja te funkcije za rukovanje uvjetnim varijablama koje služe za ostvarenje reda uvjeta.

**Struktura rješenja**

Pseudokod na slici 5.1 pretvoriti u dio programa koji radi svaka dretva tako da misliti i jesti postaju funkcije. Funkcija misliti() treba simulirati trošenje vremena (npr. sleep(3)). Funkcija jesti(n) simulira postupak hranjenja filozofa s uključivo potrebnim kodom sinkronizacije koji je skiciran u nastavku.

jesti(n){

/\* n - redni broj filozofa \*/

uđi\_u\_kritični\_odsječak;

filozof[n] = 'o';

dok (vilica[n] == 0 || vilica[(n + 1) % 5] == 0)

čekaj\_u\_redu\_uvjeta(red[n]);

vilica[n] = vilica[(n + 1) % 5] = 0;

filozof[n] = 'X';

ispiši\_stanje(n);

izađi\_iz\_kritičnog\_odsječka;

njam\_njam; /\* sleep(2); \*/

uđi\_u\_kritični\_odsječak;

filozof[n] = 'O';

vilica[n] = vilica[(n + 1) % 5] = 1;

oslobodi\_dretvu\_iz\_reda(red[(n - 1) % 5]);

oslobodi\_dretvu\_iz\_reda(red[(n + 1) % 5]);

ispiši\_stanje(n);

izađi\_iz\_kritičnog\_odsječka;

}

**Primjer ispisa pokretanja programa**

# ./a.out

X 0 0 0 0 (1)

X o X 0 0 (3)

O o X o o (1)

O o X o X (5)

O o O o X (3)

O X O o X (2)

O X O o O (5)

O O O o O (2)

O O O X O (4)

X O O X O (1)

X O o O O (4)

X O X O O (3)

O O X O O (1)

O O X O X (5)

O o O O X (3)

O X O O X (2)

O X O o O (5)

O X O X O (4)

o O O X O (2)

X O O X O (1)

X O o O O (4)

X O X O O (3)

^C

#

**4. 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\_CREAT 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 promjeniti 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*

**ZADATAK**

Ostvariti komunikaciju među procesima tipa proizvođača i potrošača korištenjem redova poruka. Osnovna struktura procesa dana je pseudokodom:

proces proizvođač (niz znakova)   
  **i** = 0;  
    čini  
        pošalji poruku sa **i**-tim znakom;  
     **i** = **i**+1;  
    do kraja niza;  
    pošalji poruku sa oznakom kraja;  
kraj.

proces potrošač  
    niz = prazan;  
    dok ima poruka čini  
        primi poruku sa jednim znakom  
        dodaj znak na kraj niza  
        ako je znak = oznaka kraja onda  
            ispiši niz  
            niz = prazan  
kraj.

**Proizvođač i potrošač trebaju biti odvojeni izvršni programi.** (Na primjer, mogu se zvati sender i receiver.) Proizvođač prima niz znakova kao argument kod poziva. Svi procesi dohvaćaju ključ reda poruka iz varijable okoline MSG\_KEY. Ukoliko red ne postoji, proces ga kreira. Potrošač uništava red poruka prije završetka.

Primjer pokretanja procesa (zasebno):

./sender 123456789 & ./sender abcdefghi & ./sender ABCDEFGHI & ./receiver

**Napisati program koji će povezati rad proizvođača i potrošača.** Programom treba odrediti koliko proizvođača se želi stvoriti i unijeti nizove znakova za svakoga od njih. Zatim treba (obavezno) kreirati novi red poruka i njegov ključ zapisati u varijablu okoline MSG\_KEY. Tek tada se stvaraju svi potrebni procesi uz prenošenje parametara i čeka da svi oni završe. Potrošaču se prepušta uništavanje reda poruka.

Primjer pokretanja s glavnim programom ([primjer ispisa](http://www.zemris.fer.hr/predmeti/os/pripreme/ispis_za_poruke.txt)):

./main 123456789 abcdefghi ABCDEFGHI

**UPUTE**

1. Nije jednostavno odrediti kada proces potrošač treba prestati očekivati nove poruke. Na primjer, svaki puta kada primi oznaku kraja, slijedeću poruku može čekati sa postavljenim IPC\_NOWAIT. Ako ne dobije odmah poruku, može odspavati malo (sleep(1)) i pokušati ponovo. Ako poruku ne dobije u nekoliko uzastopnih pokušaja može zaključiti kako nema više poruka koje bi mogle stići i završiti s radom.
2. Jedinstvenost ključa reda poruka među različitim korisnicima može se osigurati tako da se za MSG\_KEY stavi identifikacijski broj korisnika (*uid*). On se može dobiti pregledom /etc/passwd sa:

grep ime /etc/passwd

Traženi broj nalazi se u trećem polju. *uid* se također može dobiti iz programa pozivom getuid() (vidi man getuid).

1. Poruke koje se prenose u programu su vrlo kratke. Da bi se vidjeli efekti paralelnog rada više proizvođača sa jednim redom poruka potrebno je smanjiti duljinu reda poruka korištenjem *msgctl* sa naredbom IPC\_SET. Potrebne vrijednosti za popunjavanje strukture su: *uid* se dobiva sa getuid(), *gid* sa getgid(), *mode* se može postaviti na 0600, a *msg\_qbytes* može biti 5. Skraćivanje reda poruka mogu slobodno obaviti svi procesi potrošači prije nego što pošalju prvu poruku.
2. Ključ reda poruka koji je kreiran (*msgget*) sa *key* IPC\_PRIVATE može se dobiti korištenjem naredbe IPC\_STAT u *msgctl*. Traženi ključ je polje *msg\_perm.key* u vraćenoj strukturi.

Primjer rada sa porukama: [kirk.c](http://www.zemris.fer.hr/predmeti/os/pripreme/spock.c) [spock.c](http://www.zemris.fer.hr/predmeti/os/pripreme/kirk.c)

[Dodatne preporuke za rješavanje zadatka](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/vj4_poruke_upute.txt)

**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 naslijeđuje 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[] = "Kroz cijev!";/\* *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 r*ead* ni *write* se ne zablokiraju nego jave 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](http://www.zemris.fer.hr/predmeti/os/pripreme/imen_cj.c)

[Dodatne upute](http://www.zemris.fer.hr/predmeti/os/pripreme/upute/vj5_cijevi_upute.txt)

**ZADATAK**

Napisati program za provjeru znanja aritmetičkih operacija. Program može čitati pripremljene zadatke iz neke datoteke ili ih generirati nekim postupkom. Zadatak ispisuje na standardni izlaz, dohvaća odgovor sa standardnog ulaza i uspoređuje ga sa ispravnim odgovorom. Broj postavljenih zadataka treba biti ograničen. Ispravan odgovor treba tražiti koristeći program *bc* (aritmetički jezični procesor)*.*

Program treba pokrenuti *bc* i pomoću dva cjevovoda se povezati s njegovim standardnim ulazom i izlazom. Zatim svaki izraz koji treba izračunati šalje se na ulaz *bc* koji će dati neki rezultat na izlaz (cjevovodi povezani na standardni ulaz i izlaz programa *bc*). Taj rezultat treba analizirati i usporediti sa brojem koji je unijet sa tipkovnice ako je numerički podatak. Inače treba pretpostaviti da je u izrazu bilo grešaka i ispisati odgovarajuću poruku. Ako je unijeta točna vrijednost ispisati ISPRAVNO, inače NEISPRAVNO, točan odgovor je [točan odgovor].

Primjer ulaza: (zadatak se najprije ispiše na ekran, korisnik unese rješenje, nakon provjere s bc-om ispiše se status, itd.)  
**1+1=2                     
ISPRAVNO  
1\*2=4  
NEISPRAVNO, točan odgovor je 2  
2+\*3=5  
NEISPRAVAN IZRAZ**

 Zadatak ostvariti običnim ili imenovanim cjevovodom, prema uputi na početnoj stranici (ovisno o mbr-u).

**UPUTE *bc***

*bc* je jezik koji prepoznaje aritmetičke izraze i ima sintaksu sličnu jeziku C (vidi man bc, isprobati pokrenuti *bc* iz komandne linije). Ovdje je bitno samo da za aritmetički izraz napisan u jednom redu standardnog ulaza daje rezultat na standardnom izlazu. *bc*, kao i *dc* završava s radom kada dođe do kraja datoteke na standardnom ulazu (Ctrl+D, ako se pokreće iz ljuske).

## 5. Straničenje na zahtjev

Kada je međuspremnik pun, a stranica koja se zahtjeva nije u međuspremniku/spremniku, potrebno je odrediti koju ćemo stranicu izbaciti, odnosno nadomjestiti s novom stranicom. (*U procesu straničenja potrebno je primjeniti jednu od metoda pomoću koje će se prije dohvaćanja potrebne stranice iz sekundarnog spremnika u primarni odlučiti koja će se stranica izbaciti iz primarnog spremnika.)* Za to postoji mnoštvo metoda (teorijskih i praktičnih) od kojih su najpoznatije slijedeće strategije izbacivanja stranica: FIFO, LRU, LFU i OPT.

FIFO (*First In First Out*) strategija - iz radnog se spremnika (primarnog spremnika) izbacuje stranica koja je najdulje u radnom spremniku.

LRU (*Least Recently Used*) strategija - izbacuje se stranica koja se najdulje u prošlosti nije upotrebljavala.

LFU (*Least Frequently Used*) strategija - izbacuje se ona stranica od onih u radnom spremniku koja se najrjeđe koristila.

OPT (*Optimal*) strategija - izbacuje se ona stranica koja će se najkasnije u budućnosti koristiti.

Inačica LRU algoritma koja se koristi u više inačica operacijskog sustava UNIX naziva se *satnim algoritmom* (*Clock algorithm*). Algoritam djeluje tako da se sve stranice u radnom spremniku slažu u kružnu listu po redu prispijeća. Prilikom traženja stranice za izbacivanje posebnom se kazaljkom pretražuje lista, počevši od zadnje stranice učitane u primarni spremnik. Ukoliko stranica ima oznaku (zastavicu) A=1 (tj. stranica se koristila u nekom prethodnom "bliskom" trenutku), ta se oznaka postavlja na 0 te se kazaljka pomiče na slijedeću stranicu. Ako stranica ima oznaku A=0 tada se dotična stranica izbacuje iz radnog spremnika i na isto mjesto se učitava nova stranica (kazaljka se ne pomiče). Svaki puta kada se neka stranica koristi njena se oznaka A postavlja u jedinicu.

**ZADATAK**

**Studenti čiji matični broj završava s 0 ili 1 trebaju ostvariti simulaciju FIFO strategije izbacivanja stranica.  
Studenti čiji matični broj završava s 2 ili 3 trebaju ostvariti simulaciju LRUstrategije izbacivanja stranica.  
Studenti čiji matični broj završava s 4 ili 5 trebaju ostvariti simulaciju LFU strategije izbacivanja stranica.  
Studenti čiji matični broj završava s 6 ili 7 trebaju ostvariti simulaciju OPT strategije izbacivanja stranica.  
Studenti čiji matični broj završava s 8 ili 9 trebaju ostvariti simulaciju satnog mehanizma.**

**UPUTA**

Za svaku od strategija potrebna je odgovarajuća prateća podatkovna struktura. Pretpostaviti da će na raspolaganju biti samo 4-10 okvira za stranice u radnom spremniku (broj okvira je parametar i zadaje se u komandnoj liniji) te da će zahtjeva biti između 10 i 100 (također parametar komandne linije). Program treba na početku generirati zahtjeve za stranicama (redni brojevi stranica se kreću od 1 do 8) te koristeći zadanu strategiju ispisivati povijest radnog spremnika. Pretpostaviti da je u početku radni spremnik prazan.

**Primjer ispisa programa za FIFO metodu:**

$ ./str\_fifo 4 10

Zahtjevi: 5,2,3,8,2,3,4,1,2,4

#N  1   2   3   4

------------------

5  [5]  -   -   -

2   5  [2]  -   -

3   5   2  [3]  -

8   5   2   3  [8]

2   5  (2)  3   8   #pogodak

3   5   2  (3)  8   #pogodak

4  [4]  2   3   8

1   4  [1]  3   8

2   4   1  [2]  8

4  (4)  1   2   8

Kod FIFO strategije za svaki od okvira potrebno je pamtiti: stranicu koja se u njemu nalazi te nekakvu oznaku vremena.

Npr. za ispisani primjer stanja varijabli po okvirima mogla bi biti:

#N  1   2   3   4     (n,t) (n,t) (n,t) (n,t)

---------------------------------------------

5  [5]  -   -   -     (5,1) (0,0) (0,0) (0,0)

2   5  [2]  -   -     (5,1) (2,2) (0,0) (0,0)

3   5   2  [3]  -     (5,1) (2,2) (3,3) (0,0)

8   5   2   3  [8]    (5,1) (2,2) (3,3) (8,4)

2   5  (2)  3   8     (5,1) (2,2) (3,3) (8,4)

3   5   2  (3)  8     (5,1) (2,2) (3,3) (8,4)

4  [4]  2   3   8     (4,5) (2,2) (3,3) (8,4)

1   4  [1]  3   8     (4,5) (1,6) (3,3) (8,4)

2   4   1  [2]  8     (4,5) (1,6) (2,7) (8,4)

4  (4)  1   2   8     (4,5) (1,6) (2,7) (8,4)

Stranica sa najmanjom vremenskom oznakom se izbacuje.

Kod LRU strategije može se gornja metoda modificirati tako da se i za pogodke upisuje nova vremenska oznaka u "pogođeni" okvir te primijeni isti princip (stranica sa najmanjom vremenskom oznakom se izbacuje).

Kod LFU strategije se umjesto vremenske oznake za svaki okvir koristi poseban brojač. Prvi se puta (kada se stranica učita u okvir) brojač postavlja na 1, a svaki se slijedeći put (pogodak) vrijednost tog brojača povećava za jedan. Izbacuje se ona stranica sa najmanjom vrijednošću brojača. Ako više okvira ima istu vrijednost odabir je proizvoljan (odabrati okvir s najmanjim brojem).

Kod optimalne strategije lista zahtjeva pretražuje se prema naprijed te se može za svaki okvir izračunati kada će se (za koliko iteracija) pojaviti zahtjev za stranicom (u okviru) u budućnosti. Odabire se okvir s najvećim brojem. Ukoliko se stranica ne pojavljuje u budućnosti vrijednost brojača postaviti na maksimalnu (definirati neku vrijednost, npr. 100). Ako više okvira ima istu vrijednost odabir je proizvoljan (odabrati okvir s najmanjim brojem).

Kod strategije satnog mehanizma predvidjeti polje za zastavice (za svaki okvir) te jednu varijablu za kazaljku. U početku sve zastavice imaju vrijednost nula te kazaljka pokazuje na prvi okvir.

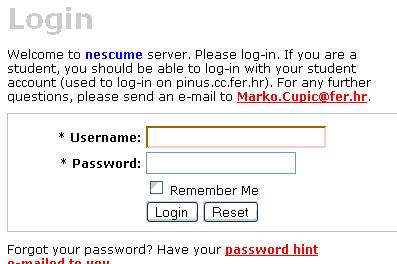
(Ova vježba može se izraditi u bilo kojem okruženju, jer samo simulira neku funkcionalnost a ne koristi neke specifične pozive jezgre, kako je to u ostalim vježbama.)

**Upute za upload laboratorijskih vježbi**

Molimo pročitati upute do kraja prije samostalnih pokušaja uploada.

Adresa poslužitelja za upload jest:

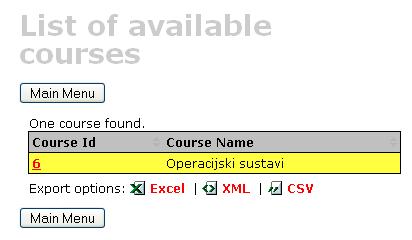
<http://studtest.zemris.fer.hr:8080/nescume/>



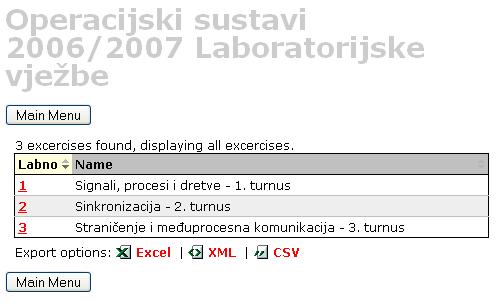
Username/Password je onaj s računala na pinus.cc.fer.hr. Ukoliko odbija login, provjeriti ispravnost "logiranjem" na pinus korištenjem nekog ssh klijenta (npr. putty). Ukoliko se ni sa putty-jem ne možete spojiti javite se djelatnicima CIPa (C zgrada). U slučaju da se sa putty-jem možete spojiti, ali ne i ovdje obratite se na: http://www.zemris.fer.hr/predmeti/os/leonardo_mail.gif.

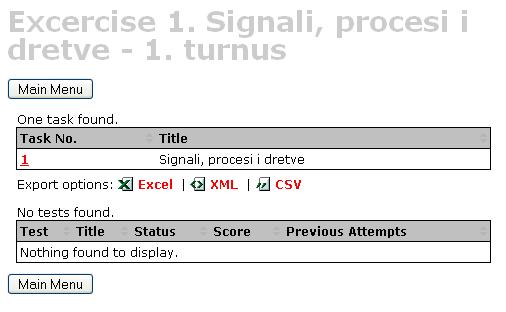
Logiranjem dolazite do prvog izbornika. Slijedećih neko stranica prikazano je na slikama:

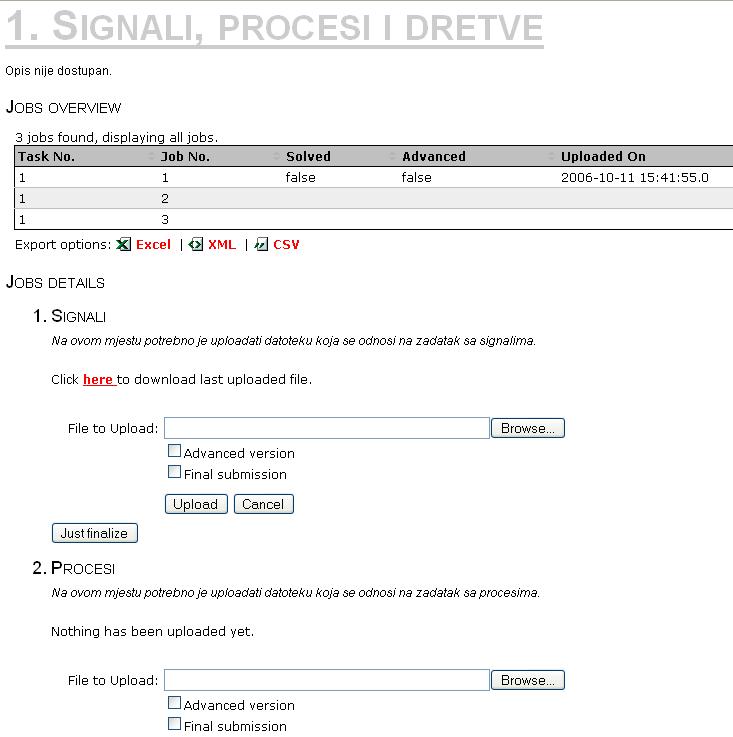








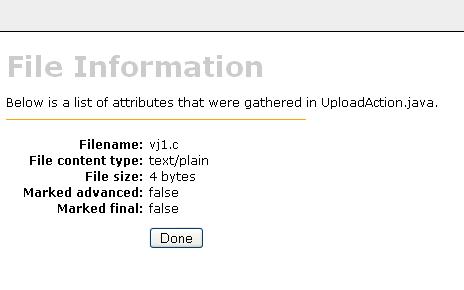


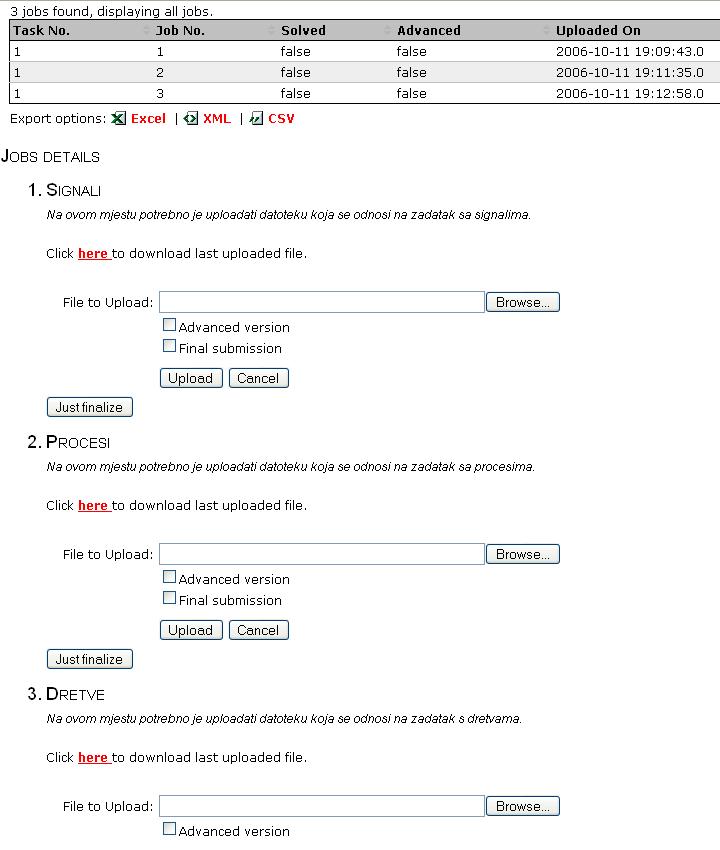


Na slici iznad prikazana je dio stranice koji omogućava upload pojedinih datoteka prvog turnusa. Za sva tri dijela labosa predviđena je jedna datoteka koju treba pohraniti na server. Gumbom "Browse" odabire se datoteka za upload te potom gumbom "Upload" datoteka se pohranjuje na poslužitenja.

**Opcije "Advanced version" i "Final submission" NE OZNAČAVATI!**

Po uploadu pojavljuju se informacije o pohranjenoj datoteci.





Nakon uploada datoteka(e) se može i pogledati (Click here to download last uploaded file). PREPORUČA SE DA SE NAKON UPLOADA POGLEDA SADRŽAJ DATOTEKE, je li cijela pohranjena na server, da slučajno nije kriva datoteka odabrana i slično. Ako nešto ne valja sa datotekom na poslužitelju, datoteka se ponovo može "uploadati".

Nakon što je PROVJERENO da su na poslužitelju prave datoteke, i SAMO NAKON TOGA može se tj. mora se to i potvrditi pritiskom na gumb "Just finalize". Nakon toga ta se vježba smatra pohranjenom i ne može se više mjenjati.

Na slici ispod, datoteka za drugi dio (procesi) je "finalizirana" i ne može se više mjenjati.

