Operativni sistemi

beleške za pismeni deo ispita

github.com/kotur95/skripte

22.09.2017.

Sadržaj

Predgovor

Poštovani čitaoci ovo je radna verzija skripte, pa postoji velika mogućnost da nije najnovija verzija. Skripta je nastala iz potrebe za materijalima iz predmeta `Operativni Sistemi` (za praktični deo ispita, tj. sa vežbi) koji se održava na drugoj i trećoj godini studija na Matematičkom fakultetu, smer informatika. Rad na skripti je započeo Nebojša Koturović student matematičkog fakulteta koja je prvenstveno bila namenjena za sopstvene potrebe a onda je to podelio sa svojim kolegama i omogućio im da i oni učestvuju u izradi ove skripte.

Izvorni (LaTeX) kod skripte je dostupan na adresi `http://github.com/kotur95/skripte`, a skripta se može pronaaći i u PDF formatu na adresi `http://alas.matf.bg.ac.rs/~mi15139/#files/skripte`.

Svako ko želi da unapredi ovu ili neku drugu drugi skriptu koja se nalazi u repozitorijumu `skripte` na `github-u` veoma je dobrodošao. Ideja je da se poboljša kvalitet dostupnih materijala iz kojih studenti mogu da uče, dakle svako ko želi da podeli svoje znanje dobro nam došao.

NAPOMENA: Skripta je konstantno u fazi izrade, i može sadržati greške različitog tipa. Ukoliko primetite neku grešku poželjno je da sami izmenite izvorni kod ili možete poslati `e-mail` sa greškom na: `mi15139@alas.matf.bg.ac.rs`, preporuke su takođe dobrodošle.

1 Obrada grešaka

1.1 Funkcije za obradu grešaka

Funkcija `ast()` služi za obradu grešaka nastalih u sistemskim pozivima, tačnije ukoliko data greška modifikuje `errno` promenljivu `<errno.h>`.

Funkcija `err_check()` obrađuje standardne grešaka koje mogu nastati tokom izvrsavanja programa.

Funkcija `pth_check()` služi za obradu gresaka kod funkcija koje rade sa posix threadovima.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <errno.h>

/* Error handling */
void ast(int statement, const char * usr_msg);
void err_chk(int statement, const char * usr_msg);
void pth_check(int pth_err, const char * usr_msg);

void ast(int statement, const char * usr_msg)
{
	if (!statement) {
		perror(usr_msg);
		exit(EXIT_FAILURE);
	}
}

void err_check(int statement, const char * usr_msg)
{
	if (!statement) {
		fprintf(stderr, "%s\n", usr_msg);
		exit(EXIT_FAILURE);
	}
}

void pth_check(int pth_err, const char * usr_msg)
{
	if (pth_err > 0) {
		errno = pth_err;
		ast(false, usr_msg);
	}
}
```

Primer 1: Funkcije za obradu grešaka

Primer poziva funkcije za obradu greske prilikom open sistemskog poziva:

```
ast((fd = open(fpath, O_RDONLY)), "Greska prilikom otvaranja fajla za citanje");
```

Primer poziva funkcije za proveru broja argumenata bio bi:

```
err_check(argc == 2, "Greska - program prima 2 argumenta");
```

Primer poziva funkcije za obradu greske prilikom poziva funkcije `pthread_create()` bio bi:

```
pth_check(pthread_create(thread_id, NULL, thr_func, args), "Error creating thread");
```

Funkcije date u Primeru 1 biće korišćene u narednim primerima.

1.2 Primeri obrade grešaka.

U ovoj sekciji biće predstavljeni primeri upotrebe funkcija za obradu gresaka

1.2.1 Pogrešan broj argumenata

```
int main(int argc, char * argv[])
{
      err_check(argc == 2, "Invalid number of arguments");
      printf("Argument je: %s ", argv[1]);

      return 0;
}
```

Primer 2: Pogresan broj argumenata

1.2.2 Otvaranje regularnog fajla

U Primeru 3 možemo videti primer gde se ograđujemo od raznih grešaka prilikom otvaranja fajla.

```
// Provera postojanja file-a
ast(-1 != access(argv[1], F_OK), "Unable to access file");
int fileFd;
// Otvaranje preko fd-a putem open f-je
ast( -1 != (fileFd = open(argv[1], O_RDWR | O_NONBLOCK)), \
            "Error opening file for reading");

struct stat sbuf;
// Provera da li je fajl regularan
ast(-1 != fstat(fileFd, &sbuf), "Error stat");
err_check((sbuf.st_mode & S_IFMT) != S_IFREG, "Not regular file");
```

Primer 3: Otvaranje regularnog fajla

`access(file, F_OK)` — proverava da li fajl uopšte postoji. Slično možemo pokušati da vidimo da li nam je fajl dostupan za čitanje/pisanje promenom drugog argumenta u `R_OK` ili `W_OK`.

Izraz na osnovu kojeg je kreirana provera da li je fajl regularan `(sbuf.st_mode & S_IFMT) == S_IFREG)` se može pronaći u man strani za `stat()` funkciju: `man 2 stat`.

1.2.3 Kreiranje niti

```
/* Provera za niti */

pthread_t thread;
int status = pthread_create(&thread, NULL, threadfunc2, NULL);
pth_check(status, "Error creating thread");
```

Primer 4: Obrada greske za funkcije koje rade sa threadovima

2 Praktični primeri

2.1 Obrada signala

Signal je obaveštenje procesu da je neki događaj nastupio. Tip signala se određuje u zavisnosti od vrednosti signala, u tom smislu signal je određen celobrojnom promenljivom pomoću koje se identidikuje. Funkcije i signalne promenljie (identifikatori) se nalaze u zaglavlju `<signal.h>`. Signali se nekada opisuju kao *hardware interupt-s* jer su analogni njima, tj. prekidaju normalan tok izvršavanja programa. U većini slučajeva nije moguće predvideti kada će neki signal nastupiti.

Postoje takozvane uobičajene reakcije na signal koji je poslat procesu, i njih možemo videti u man strani za signal sistemski poziv (`man 2 signal`).

Signal sistemski poziv prima 2 argumenta, prvi argument je celobrojna vrednost signala predstavljena simboličkim imenom koje možemo naći u `man 2 signal`, drugi argument je `signal handler` funkcija koja obrađuje taj, a ponekad i više signala,`signal handler` prima jedan argument a to je redni broj `signal-a` koji je nastupio. Ukoliko ne želimo da menjamo uobičajeno ponačanje kao drugi argument `signal()` sistemskog poziva prosleđujemo vrednost `SIG_DFL`, a ukoliko želimo da zanemarimo signal `SIG_IGN`.

```
void (*signal(int sig, void (*func)(int)))(int);
```

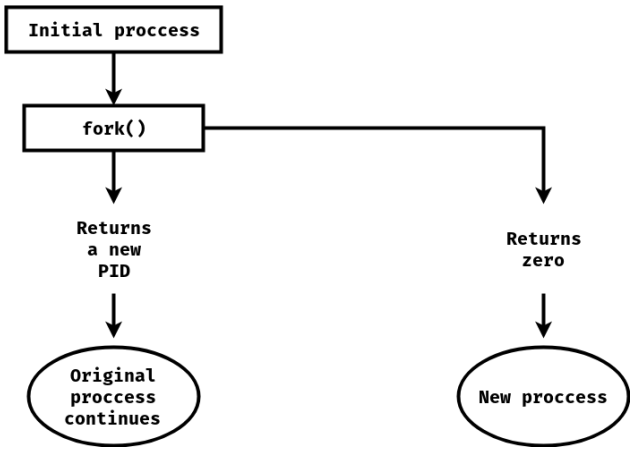
Povratna vrednost: Ukoliko je signal obrađen, povratna vrednost je ista kao povratna vrednost `signal handler-a`, za poslednji signal tog tipa. Ukoliko je došlo do greške povratna vrednost je `SIG_ERR` i `errno` promenljiva je postavljena na odgovarajuću vrednost.

```c
void sig_handler(int signal)
{
	/* U signal handler-im treba koristiti samo funkcije koje
	 * su reentrant safe, tj. one koje garantuju korektan
	 * natavak izvrsavanja nakon prekida. printf() nije
	 * reentrant safe, ali je data radi ilustracije. */
	switch (signal)
	{
		case SIGKILL : printf("Kill Ocurred");break;
		case SIGINT  : printf("Interupt Ocurred");break;
		case SIGSTOP : printf("Stop Ocurred");break;

		/* strsignal() vraca stringovni opis signala koji je
		 *  nastupio, a kao argument prima redni broj signala. */
		default: printf("Signal occured: %s\n", strsignal(signal));
	}
}

int main()
{
	int retval = signal(SIGINT, sig_handler)
	ast(retval != SIG_ERR, "Error setting signal");

	/* Cekanje signala da nastupi */
	for(;;)
		sleep(1);

	return 0;
}
```

Primer 5: Obrada signala

2.2 Kreiranje deteta

Funkcija `fork()` kreira novi proces duplirajući proces iz kojeg je pozvana. Novi proces je dete procesa iz kojeg je `fork()` funkcija pozvana. Oba procesa imaju zaseban memorijski prostor sa istim sadržajem, i međusobno ne dele memoriju.



Slika 1: Fork sistemski poziv

Dete je identičan duplikat roditelja, ali neke stvari ne nasleđuje:

1) `PID-ovi` su im različiti.
2) `PID` roditelja ostaje nepromenjen.
3) Dete ne nasleđuje `lock-ove` memorije roditelja.
4) Dete ne nasleđuje izmene semafora od roditelja.

   itd ...

Prototip funkcije je: `pid_t fork(void);`

Povratna vrednost: `PID` deteta roditelju i 0 detetu. U slučaju greške funkcija vraća `-1` roditelju i dete ne biva kreirano. Vrednost `errno` promenljive se postavlja na odgovarajucu vrednost.

```c
pid_t pid = fork();
ast(-1 != pid, "Unable to create child");

if (pid  > 0) {
      // Parent branch
      printf("Hello from parent, pid: %d\n", pid);
} else {
      // Child branch
      printf("Hello from child, pid: %d\n", pid);
      exit(EXIT_SUCCESS);
}

/* Child call's exit() before reaching this part of code,  *
 * Parent conitnue's executing after leaving parent branch */
```

Primer 6: Fork poziv

Standardni ulaz:	**Standardni izlaz:**
	`Hello from parent, pid: 7896` `Hello from child, pid: 0`

2.3 Exec familija funkcija

U unix-olikim operativnim sistemima zamena tekućeg procesa nekim drugim procesom (programom) se vrši pomoću exec() familije sistemskih poziva. Ovaj metod se najčešće koristi u kombinaciji sa fork() sistemskim pozivom, u kome se dete procesa zamenuje nekim drugim procesom (programom).

Sistemski pozivi koji pripadaju exec() familiji:

```
int execl(const char *path, const char *arg, ...
/* (char  *) NULL */);
int execlp(const char *file, const char *arg, ...
/* (char  *) NULL */);
int execle(const char *path, const char *arg, ...
/*, (char *) NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
char *const envp[]);
```

Prvi argument kod ovih funkcija predstavlja ime izvršnog fajla koji će biti pokrenut kao zamena trenutne slike procesa. Funkcije execlp(), execvp(), i execvpe() šta više ne zahtevaju tačnu putanju do programa, tj. ukoliko je prvi argument bez vodeće / date funkcije tragaju za izvršnim fajlom u direktorijumu koji je uključen u PATH enviroment promenljivoj shell programa.

Postoje sledeće varijacije exec() sistemskog poziva u odnosu na sufiks, od kojih zavisi drugi argument.

'p' - Funkcija uzima filename i koristi PATH da nađe izvršni fajl.
'l' - Funkcija uzima listu argumenata umesto pokazivača.
'v' - Argument je vektor argv[]
'e' - Funkcija uzima envp[] niz umesto trenutnog enviroment-a

Povratna vrednost: Nema povratka u slučaju uspeha, -1 u slucaju greške - errno promenljiva se postavlja na odgovarajuću vrednost.

```c
/* Primer poziva execvp gde je drugi argument
 * vektor, tj. niz argumenata koji se prosledjuje
 * izvrsnom fajlu, a funkcija takodje traga za
 * izvrsnim fajlom u PATH promenljivoj */
char * args[] = {"ls", "-l",  NULL};
int retval = execvp("ls", args);

/* Umesto prvog argumenta funkcije ls mogli smo
 * koristiti apsolutnu putanju do ls programa,tj.
 * poziv je mogao  izgledati na sledeci nacin: */
retval = execvp("/bin/ls", args);

/* Naravno potrebno je i obraditi gresku
 * u slucaju neuspesnog poziva */
ast(-1 != retval, "Error overriding process image");
```

Primer 7: Exec sistemski poziv

Rezultat rad programa iz 7 je ispisivanje informacija o tekućem folderu na standardni izlaz, kao prilikom komande shell-a: $ ls -l.

2.4 Pravljenje PIPE-a

Funkcija `pipe()` kreira pipe, kao argument prima niz tipa `int` dužine 2, i dodeljuje mu vrednosti fajl deskriptora koje je otvorila. Fajlovi na koje referišu ti fajl deskriptori su (`pipefd[0]`) tj. strana `pipe-a` za čitanje i (`pipefd[1]`) strana `pipe-a` za pisanje.

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

Povratna vrednost: `0` u slučaju uspešnog kreiranja `pipe-a`, `-1` u slucaju greške.

`PIPE` je struktura koja omogućava jednosmernu komunikaciju, dat je primer `pipe-a` u shell programu:

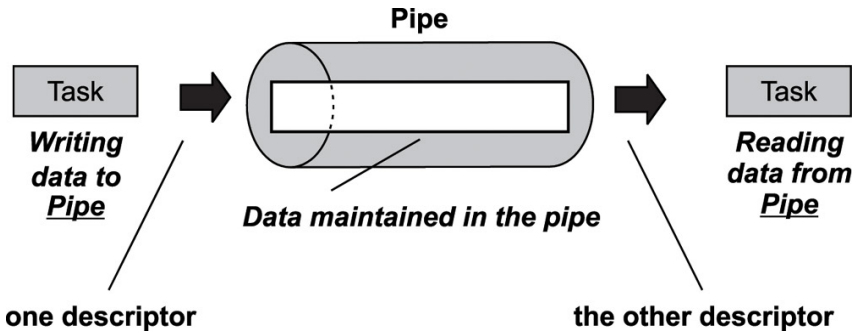
```
$ ls | wc -l
```

Ova komanda ispisuje broj fajlova u tekućem direktorijumu, tako što izlaz programa `ls` postaje ulaz programa `wc` koji potom obrađuje taj ulaz i štampa na standardni izlaz broj linija (fajlova). U sledećem primeru pokučaćemo da prikažedmo jednu od implementacija ovog `PIPE-a`.

```c
/* READ_SIDE ima vrednost (0), a WRITE_SIDE (1) */
int pipeFd[2];
ast(-1 != pipe(pipeFd), "Error creating PIPE");

pid_t pid = fork();
ast(-1 != pid, "Creating child error");

if (pid  > 0) {
      // Parent process
      close(pipeFd[WRITE_SIDE]); // Zatvaramo pipe za pisanje

      /* Cekamo na dete proces da zavrsi izvrsavanje */
      ast(-1 != wait(NULL), "Wait error");

      /* dup2() u datom pozivu duplira fd read pipe-a i dodeljuje mu
       vrednost (STDIN_FILENO). */
      ast(-1 != dup2(pipeFd[READ_SIDE], STDIN_FILENO), "Error assigning FD"
          );
      close(pipeFd[READ_SIDE]); // zatvaramo fd jer je prosledjen na stdin

      char * args[] = {"wc", "-l",  NULL};
      ast(-1 != execvp("wc", args), "Error executing wc");
} else {
      // Child process
      close(pipeFd[READ_SIDE]); // Zatvaramo pipe za citanje

      /* dup2() u datom pozivu duplira write fd pipe-a i dodeljuje mu  *
       * vrednost 2 tj. (STDOUT_FILENO). */
      ast(-1 != dup2(pipeFd[WRITE_SIDE], STDOUT_FILENO), \
                  "Error assigning FD");
      close(pipeFd[1]); // zatvaramo fd jer je prosledjen na stdout

      char * args[] = {"ls", NULL};
      ast(-1 != execvp("ls", args), "Error executing ls");
}

/* Ako se program izvrsi ocekivano, ovaj deo koda nikada nece biti dosegnut
     */
```

Primer 8: Pipe (ls | wc -l) izmedju roditelja i deteta

Dakle rezultat rada programa iz Primera 8 je ispisuje broja fajlova u direktorijumu iz kog je pokrenut.



Slika 2: Komunikacija putem PIPE-a

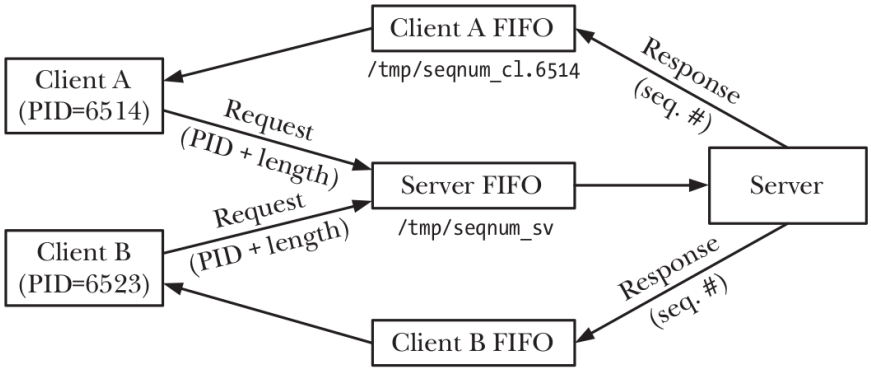
2.5 Jednosmerna komunikacija između procesa (FIFO)

`FIFO` je varijacija `PIPE` koncepta, `FIFO` se definiše kao `byte stream` što znači da se ne može podacima pristupiti u proizvoljnom redosledu, već onako kako oni putuju kroz `FIFO`. `PIPE` nam je omogućio komunikaciju između roditeljskog procesa i deteta, dok nam `FIFO` omogućuje komunikaciju između 2 "nepovezana" procesa. Otvaranje `FIFO-a` je slično kao kod regularnog fajla, i `PIPE-a`.

Fifo se može kreirati na dva načina:

1) shell komandom: `mkfifo [-m mode] pathname`
2) sistemskim pozivom: `mkfifo()`

Nakon kreiranja `FIFO` se otvara `open()` sistemskim pozivom, ali sa flag-ovima: `O_RDONLY` ili `O_WRONLY`, jer se koristi za komunikaciju u jednom smeru. Može ga otvoriti svaki proces, ako ima odgovarajuće privilegije.



Slika 3: Fifo client-server model

Na Slici 3 prikazan je client-server model komunikacije putem `fifo-a`.

Potrebno je uvesti i konvenciju po kojoj će se određivati dužina poruke koju `client` tj. `server` proces prima. Postoji više mogućih načina da se ovo realizuje, navešćemo neke od njih:

1) `delimiter character` — Poruke se razdvajaju nekim karakterom, npr. `'\n'`.
2) `fixed-length` — Poruke su unapred određene fiksne dužine.
3) `fixed-size header` — Svaka poruka ima header fiksne dužine u kome je navedena dužina poruke.

U primerima 9 i 10 prikazana je moguća implementacija `client-server` modela, u kome se poruke sa `standardnog ulaza` servera prosleđuju na `standardni izlaz` klijenta.

```
char * fifo_path = argv[1];
ast(-1 != mkfifo(fifo_path, 0755), "Error creating fifo");

int fifoFd;
ast(-1 != (fifoFd = open(fifo_path, O_WRONLY)), "Error opening fifo");

char buf[BUF_MAX];
int bytesRead;

while (1) {
      bytesRead = read(STDIN_FILENO, buf, BUF_MAX);
      ast(bytesRead != -1, "Reading error");
      buf[bytesRead-1] = '\0'; // Umesto '\n' ubacujemo '\0'
      ast(-1 != write(fifoFd, buf, bytesRead), "Writing error");
}
```

Primer 9: Fifo server

```
char * fifo_path = argv[1];
int fifoFd;
ast(-1 != (fifoFd = open(fifo_path, O_RDONLY)), "Error opening fifo");

char buf[BUF_MAX];
int bytesRead;

while (1) {
      ast(-1 != (bytesRead = read(fifoFd, buf, BUF_MAX)), "Reading error");
      ast(-1 != write(STDOUT_FILENO, buf, strlen(buf)), "Writing error");
      ast(-1 != write(STDIN_FILENO, "\n", 1), "Writing error");

      if (strcasecmp(buf, "kraj") == 0)
            break;
}
```

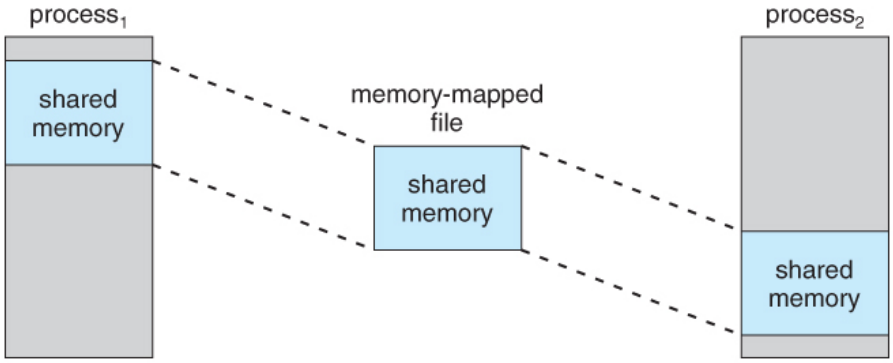
Primer 10: Fifo klijent

Standardni ulaz fifo servera	**Standardni izlaz fifo klijenta**
Write in C	Write in C
Hello World	Hello World
kraj	

Dati primer `FIFO client-server` komunikacije koristi `delimiter character` metod za razdvajanje poruka, taj karakter je `'\0'` tj. terminalna nula, što omogućava udobnu komunikaciju pomoću string-ova.

2.6 Deljena memorija

Procesi često imaju potrebu da pristupaju zajedničkoj memoriji, tj. da čitaju iz nje, pišu u nju. Da bi bili u mogućnosti da je koriste potreban im je način za kreiranje i način za pristupanje toj deljenoj memoriji. U `Linux` okruženju postoje funkcije koje se bave ovom problematikom.



Slika 4: Deljenje memorije

Deljena memorija predstavlja fajl u `linux fajl sistemu` koji sadrži podatke (memoriju). Tu memoriju možemo potom možemo mapirati u adresni prostor programa.

```
int shm_open(const char * name, int oflag, mode_t mode);
int shm_unlink(const char * name, int oflag, mode_t mode);
```

Povratna vrednost: `shm_open()` u slučaju uspeha vraća fajl deskriptor, `-1` u slučaju greške. `shm_unlink()` u slučaju uspeha vraća 0, u suprotnom `-1`.

Za kreiranje/pristup deljenoj memoriji koristimo funkciju `shm_open()` i to:

1) Za kreiranje: `shm_open(naziv, O_RDWR | O_CREAT, 0755);`
2) Za otvaranje: `shm_open(naziv, O_RDWR, 0);`
3) Za brisanje: `shm_unlink(naziv);`

Kada otvorimo deljenu memoriju, potrebno ju je još i mapirati u adresni prostor.

```
void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
int munmap(void *addr, size_t length);
```

Prvi argument nam nije potreban i njega ćemo zanemariti, tj. prosledićemo `NULL`, drugi argument predstavlja veličinu memorije, treći su prava pristupa (`PROT_READ, PROT_WRITE`). Argument `flags` je tip memorije koji alociramo, najčešće ćemo koristiti `MAP_SHARED`. Zatim ide fajl deskriptor deljene memorije, i na kraju ofset od početka fajla. `munmap()` "unmapira" memoriju. Od argumenata prima adresu i dužinu memorije.

Povratna vrednost: `mmap()` u slučaju uspeha vraća adresu mapirane memorije, u slučaju greške vraća `MAP_FAILED`. `munmap()` vraća `0`, a u slučaju greške `-1`.

NAPOMENA: Kada kompiliramo program koji koristi funkcije koje rade sa deljenom memorijom, potrebno je navesti i argument -lrt.

```
gcc shm.c -o shm -std=c99 -Wall -Wextra -lrt
```

2.6.1 Primeri deljenja memorije

Dati su primeri kreiranja deljenja memorije (Primer 11) i čitanja iz deljene memorije (Primer 12 na strani 17). Prvo se pokreće program iz primera 11 a nakon njegovog uspečnog izvršavanja pokreće se i program 12. Program koji kreira deljenu memoriju učitava niz sa `standardnog ulaza` u deljenu memoriju, a program koji otvara deljenu memoriju ispisuje taj niz na `standardni izlaz`. `_shared_array` je struktura koja nam olakšava razmenu podataka.

```
typedef struct
{
      double array[MAX_LEN];
      int length;
}_shared_array;

void * create_mem_block(const char * shmName, unsigned size)
{
      int memFd = shm_open(shmName, O_RDWR | O_CREAT, 0755);
      ast(-1 != memFd, "Error creating shared memory");

      /* Obavezno postavite fajl na odg. velicinu */
      ast(-1 != ftruncate(memFd, size), "Error truncating file");

      void * addr;
      addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED , \
                        memFd, 0);
      ast(addr != MAP_FAILED, "Error mapping shm");
      close(memFd);

      return addr;
}

int main(int argc, char * argv[])
{
      /* Program prima putanju do deljenje pemorije */
      err_check(argc == 2, "Invalid number of arguments");
      char * name = argv[1];

      unsigned mem_size = sizeof(_shared_array);
      _shared_array * shm_array = create_mem_block(name, mem_size);

      scanf("%d", &shm_array->length);
      for (int i=0; i < shm_array->length; i++)
            scanf("%lf", &shm_array->array[i]);

      munmap(shm_array, mem_size);

      return 0;
}
```

Primer 11: Kreiranje deljene memorije

Program iz primera 11 čita niz sa `standardnog ulaza` i upisuje ga u `deljenu memoriju`.

Program iz primera 12 čita niz iz deljene memorije i ispisuje ga na `standardni izlaz`.

```c
typedef struct {
      double array[MAX_LEN];
      int length;
}_shared_array;

void * get_mem_block(const char *fpath, unsigned *size)
{
      int memFd = shm_open(fpath, O_RDWR, 0);
      ast(memFd != -1, "Error opening shared memory");

      struct stat finfo;
      ast(-1 != fstat(memFd, &finfo), "Unable to get file stats");
      *size = finfo.st_size;

      addr = mmap(0, *size, PROT_READ | PROT_WRITE, MAP_SHARED, memFd, 0);
      ast(addr != MAP_FAILED, "Error maping memory");

      close(memFd);
      return addr;
}

int main(int argc, char * argv[])
{
      /* Program prima putanju do deljenje pemorije */
      err_check(argc == 2, "Invalid number of arguments");

      char * name = argv[1];
      unsigned data_size;
      _shared_array * shm_array = get_mem_block(name, &data_size);
      for (int i=0; i < shm_array->length; i++)
            printf("%.2lf ", shm_array->array[i]);

      printf("\n");

      munmap(shm_array, data_size);
      shm_unlink(name);

      return 0;
}
```

Primer 12: Otvaranje deljene memorije

Ulaz programa iz primera 11	**Izlaz programa iz primera 12**
5 1.22 2.51 3.14 2.72 1.81	1.22 2.51 3.14 2.72 1.81

Primetimo da ovaj program ima jedan nedostatak, istovremeno pokretanje oba programa može dovesti do problema. Na ovaj primer ćemo se vratiti kada budemo izučavali neke od metoda sinhronizacije.

2.7 Semafori

Semafori obezbeđuju sinhronizaciju između procesa (često radi pristupanja deljenoj memoriji). Semafor je struktura koja ima svoju vrednost (celobrijnu promenljivu tipa `sem_t`) i skup operacija koje se mogu izvršiti nad njom, te operacije su:

1) Uvećanje trenutne vrednost semafora.
2) Umanjenje trenutne vrednosti semafora.
3) Čekanje da vrednost semafora postane 0
4) Postavljanje vrednosti semafora na njenu apsolutnu vrednost.

Kada kreiramo semafor, pre nego što počnemo da ga koristimo potrebno je inicijalizovati ga na određenu vrednost. Ta vrednost predstavlja broj `procesa/threadova` koji mogu pristupiti resursu koji semafor "štiti". Kada je vrednost semafora `0` onda je resurs dostupan. Promenljiva tipa `sem_t` tj. semafor treba biti vidljiv svim procesima/threadovima koji ga koriste (npr. globalna promenljiva ili statički alocirana promenljiva koju možemo proslediti kao argument).

Funkcije pomoću kojih su ove operacije realizovane su:

```
int sem_init(sem_t *sem, int pshared, int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

`sem_init()` — Inicijalizuje semafor čiji se pokazivač prosleđuje kao prvi argument, drugi argument određuje da li će semafor biti deljen između procesa (1) ili izmedju threadova jednog procesa (0). Treći argument je vrednost na koju se inicijalizuje.
`sem_wait()` — Umanjuje vrednost semafora za `1` i pristupa resursu ili čeka ukoliko je resurs zauzet (ukoliko je vrednost semafora `0`).
`sem_post()` — Uvećava vrednost semafora za `1` i signalizira mogućnost neke operacije koja sledi i sl.

Povratne vrednosti: `0` u slučaju uspešnog izvršavanja, u slučaju greške `-1` i `errno` promenljiva je postavljena na odgovarajuću vrednost.

Vrednost semafora najčešće stavljamo zajedno za podacima, što omogućava udobnije korišćenje semafora.

```c
#define BUF_LEN (1024)

typedef struct {
      char buf[BUF_LEN];
      sem_t safe_to_read;
      sem_t safe_to_write;
}_podatak;
```

Primer 13: Struktura koja sadrži semafore

NAPOMENA: Kada kompiliramo program koji koristi funkcije koje rade sa semaforima, potrebno je navesti i argument -lpthread.

```
gcc sem.c -o sem -std=c99 -Wall -Wextra -lpthread
```

Primeri 14 i 15 predstavljaju modifikacije primera 11 i 12 koji nisu koristili nikakvu sinhronizaciju, dok novi primeri koriste sinhronizaciju putem semafora. Ulazne i izlazne vrednosti ostaju nepromenjene, ali je ipak uklonjena mogućnost bilo kakvog nedeterminističkog ponašanja.

```
typedef struct {
      double array[MAX_LEN];
      int length;
      sem_t safe_to_read;
      sem_t safe_to_write;
}_shared_array;

int main(int argc, char * argv[])
{
      /* Program prima putanju do deljenje pemorije */
      err_check(argc == 2, "Invalid number of arguments");
      char * name = argv[1];

      unsigned mem_size = sizeof(_shared_array);
      _shared_array * shm_array = create_mem_block(name, mem_size);

      /* Incijalizacija semafora se vrsi samo u jednom programu */
      ast(-1 != sem_init(&shm_array->safe_to_read, 1, 0), "Sem init err");
      ast(-1 != sem_init(&shm_array->safe_to_write, 1, 1), "Sem init err");

      ast(-1 != sem_wait(&shm_array->safe_to_write), "Sem wait err");
      scanf("%d", &shm_array->length);
      for (int i=0; i < shm_array->length; i++)
            scanf("%lf", &shm_array->array[i]);

      sem_post(&shm_array->safe_to_read);
      munmap(shm_array, mem_size);

      return 0;
}
```

Primer 14: Modifikacija primera 11

```
int main(int argc, char * argv[])
{
      /* Program prima putanju do deljenje pemorije */
      err_check(argc == 2, "Invalid number of arguments");

      char * name = argv[1];
      unsigned data_size;
      _shared_array * shm_array = get_mem_block(name, &data_size);
      ast(-1 != sem_wait(&shm_array->safe_to_read), "Sem wait err");
      for (int i=0; i < shm_array->length; i++)
            printf("%.2lf ", shm_array->array[i]);

      printf("\n");

      munmap(shm_array, data_size);
      shm_unlink(name);

      return 0;
}
```

Primer 15: Modifikacija primera 12

2.8 Zaključavanje fajlova

Dok jedan proces pristupa fajlu, neće doći do problema prilikom čitanja tj. pisanja u taj fajl. Ali kada više procesa pristupa istom fajlu, dolazi do tzv. trke za resursima. Da bismo rešili taj problem treba nam opet neka vrsta sinhronizacije. U linux okruženju imamo određene sistemske pozive koji se bave ovom problematikom.

Kod zaključavanja fajla (postavljanja katanca) imamo dve operacije:

1) `acquire` — Postavlja katanac na fajl/njegov deo
2) `release` — Skida katanac sa fajla/dela.

Ove se realizuju prosledjivanjem argumenata odgovarajućih argumenata `fcntl()` sistemskom pozivu.

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

Povratna vrednost: Zavisi od tipa poziva, tj. argumenata koje smo prosledili. U slučaju greške je `-1` i `errno` promenljiva se postavlja na odgovarajuću vrednost.

`fcntl()` sistemski poziv izvršava neku od operacija nad fajlom koji je prosleđen putem fajl deskriptora kao što su dupliranje fajl deskriptora, promena vrednosti fajl deskriptora i slično. Između ostalog uz pomoć ovog sistemskog poziva mogu se staviti i tzv. katanci koji onemogućuju drugim procesima da manipulišu podacima u datom fajlu.

Katanci `fcntl()` sistemskog poziva su katanci za sinhronizaciju, oni neće onemogućiti nekome da piše/čita iz fajla sem ukoliko želi sinhronizaciju i sam proveri da li postoje katanci. Dakle ova vrsta zaključavanja neće onemogućiti nekog ko ima jaku nameru da pristupi našem fajlu, već ga može obavestiti: “Hej, ovaj segment fajla je zaključan.”. Na tom principu rade katanci sistemskh poziva `fcntl()` i `flock()`.

Zaključavanje se vrši tako što se kao drgi argument prosledi `F_SETLK` i kao treći struktura tipa `struct flock` čiji član strukture (promenljiva) `l_type` određuje tip katanca npr(`F_WRLCK ili F_RDLCK`).

Za otključavanje drugi argument je takođe F_SETLK, a tip katanca `l_type` treba da bude `F_UNLCK`.

```
struct flock
{
      short l_type;    /* Tip katanca: F_RDLCK, F_WRLCK, F_UNLCK */
      short l_whence;  /* Odakle je start: SEEK_SET, SEEK_CUR, SEEK_END */
      off_t l_start;   /* Pocetni ofset od vrednosti whence */
      off_t l_len;     /* Broj bajtova koji zelimo da zakljucamo */
      pid_t l_pid;     /* PID procesa koji nam blokira zakljucavanje
                        * fajla Postavlja se automatski ako fcntl
                        * prosledimo  F_GETLK ili F_OFD_GETLK) */
};
```

Primer 16: Struktura flock

Struktura `flock` data u primeru 16 korišćena je u primeru 17 na strani 21 kod postavljanja katanca u fajlu (zaključavanja).

U Primeru 17 na `standardni ulaz` se prosleđuju ime fajla, od kog bajta ga treba zaključati i vreme trajanja katanca. Katanac biva zadržan na traženom delu fajla onoliko sekundi koliko smo uneli sa standardnog ulaza (sačuvali u promenljivu `lock_time`).

```c
int from, len, lock_time, fd;
char fpath[MAX_LEN];
printf("Enter: fpath, from, len, lock_time");
scanf("%s, %d, %d, %d", fpath, &from, &len, &lock_time);
ast(-1 != (fd = open(fpath, O_RDWR)), "Opening file error");

/* Pretpostavicemo da je fajl veci od from + len bajtova */
struct flock lock;
lock.l_type = F_WRLCK; // write lock
lock.l_whence = SEEK_SET;
lock.l_start = from;
lock.l_len = len;

/* pokusavamo da zakljucamo fajl na zeljeni nacin:
* F_SETLK: ako ne moze da zakljuca fajl, odmah puca i vraca se nazad
* F_SETLKW: ako ne moze da zakljuca, sacekace dok se resurs ne oslobodi
* F_GETLK: sluzi za ispitivanje da li moze da se postavi zeljeni katanac
* u ovom slucaju nije lose biti pohlepan i uvek ispitivati maksimalne
* mogucnosti, tj. treba postaviti tip katanca na F_WRLCK. Ako fajl moze
* da se zakljuca u  w modu, sigurno moze i u r modu. Vodite se parolom
* "ko moze vise moze i manje". Kada se vrsi zakljucavanje uvek treba
* zakljucavati minimum potrebnih bajtova da bi se omogucio sto
* veci paralelizam. */

ast(-1 != fcntl(fd, F_SETLK, &lock), "Locking file failed");

printf("Lock acquired, holding %d seconds.\n", lock_time);
sleep(lock_time); // Uspavljujemo proces (simulramo drzanje katanca)

lock.l_type = F_UNLCK; // Potrebno je otkljucati fajl
ast(-1 != fcntl(fd, F_SETLK, &lock), "Unlocking file failed");
printf("Lock released ...\n");

/* zatvaranjem deskriptora sa automatski oslobadaju svi katanci */
close(fd);
```

Primer 17: Zaključavanje fajla

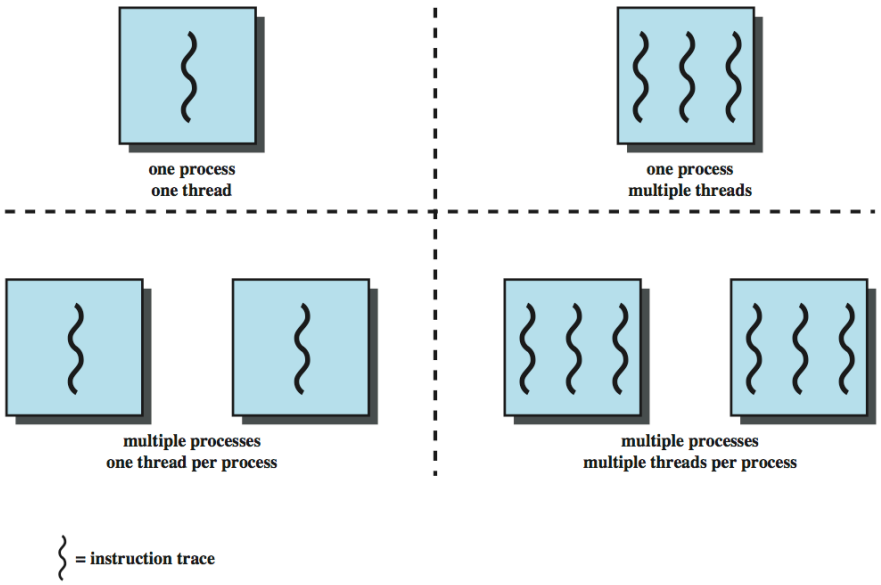
Standardni ulaz:	**Standardni izlaz:**

`file.txt 10 30 20`	`Enter: fpath from len lock_time`
	`Lock acquired, holding 20 seconds.`
	`Lock released ...`

Dodatne informacije o zaključavanju fajlova potražite u: `man 2 fcntl`.

2.9 Rad sa nitima

Niti nam obezbeđuju mogućnost da se u okviru jednog procesa izvršava više radnji istovremeno (svaka nit posebnu radnju). Niti dele deskriptore i memoriju procesa u okviru koga su pokrenute. Sinhronizacija prilikom korišćenja niti može biti značajno jednostavnija nego između procesa, a značajno je i to što na višeprocesorskim sistemima dobijamo poboljšane performanse.



Slika 5: Izvršavanje više niti istovremeno

```c
/* Kreira nit i zapocinje izvrsavanje od prosledjene
 * funkcije (arg start routine) */
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
              void *(*start_routine) (void *), void *arg);

/* Ceka da nit zavrsi za izvrsavanjem i koristi
 * se za uzimanje povratne vrednosti niti */
int pthread_join(pthread_t thread, void **retval);

/* Inicijalizuje vrednost mutex-a primer 1 je dinamicki
 * metod, a primer 2 staticki. */
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
           const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

/* Unistava mutex objekat (moze biti ponovo
 * inicijalizovan) pomocu mutex_init(). */
int pthread_mutex_destroy(pthread_mutex_t *mutex);

/* Funkcije za otkljucavanje/zakljucavanje mutex-a */
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Primer 18: Funkcije za rad sa threadovima

Greške kod funkcija datih u Primeru 18 treba obrađivati funkcijom `pth_check()` koju možete videti u Primeru 1 na strani 7.

U Primeru 19 možemo videti jedostavno kreiranje i izvršavanje niti. Funkciju korišćenu za obradu greške (`pth_check()` možete naći u Primeru 1 na strani 1.

```c
void * threadfunc2() {
      printf("Stampam iz prve niti.\n");
      pthread_exit(0);
}

void * threadfunc3() {
      printf("Stanmpam iz druge niti.\n");
      pthread_exit(0);
}

int main()
{
      /* Jedna nit programa je tekuca nit */
      /* Kreiranje druge niti programa */
      pthread_t thr2;
      int retval = pthread_create(&thr2, NULL, threadfunc2, NULL);

      /* Obrada greske */
      pth_check(retval, "Error creating thread");

      /* Kreiranje trece niti programa */
      pthread_t thr3;
      retval = pthread_create(&thr3, NULL, threadfunc3, NULL);
      pth_check(retval, "Error creating thread");

      /* Cekanamo da niti zavrse */
      retval = pthread_join(thr2, NULL);
      pth_check(retval, "Error waiting thread");

      retval = pthread_join(thr3, NULL);
      pth_check(retval, "Error waiting thread");

      return 0;
}
```

Primer 19: Kreiranje niti

Standarni ulaz:	**Standardni izlaz:**
	Stampam iz prve niti. Stampam iz druge niti

NAPOMENA: Kada kompiliramo program koji koristi funkcije koje rade sa posix threadovima, potrebno je navesti i argument -lpthread.

```
gcc pthreads.c -o pthreads -std=c99 -Wall -Wextra -lpthread
```

U Primeru 20 možemo videti na koji način se mogu kreirati niti, kako im se prosleđuju argumeni kao i uzimanje povratne vrednosti iz funkcije niti. Promenljiva `static double retval` živi od početka do kraja programa a vidljivost ima na nivou funkcije. Potrebno je pozivanje funkcije `thr3()` da bi uzela vrednost `2.72`, inače bi bila `3.14`.

```c
void * threadfunc2(void * arg)
{
      printf("Druga nit, string je: %s \n", (char *) arg);
      pthread_exit(0);
}

void * threadfunc3(void * arg)
{
      printf("Treca nit, string je: %s \n", (char *) arg);
      static double retval = 3.14;
      retval = 2.72;
      pthread_exit(&retval);
}
                  /* static modifikator nam omogucava        */
int main()        /* kreiranje promenljivih koje postoje ceo */
{                 /* zivotni vek programa (sl. kao glob.)    */
      static char str1[MAX_LEN];
      static char str2[MAX_LEN];

      printf("Unesite 2 stringa (argumente niti): \n");
      scanf("%s %s", str1, str2);

      /* 1 nit vec postoji ('glavna'). Kreiranje 2. niti */
      pthread_t thr2;
      int status = pthread_create(&thr2, NULL, threadfunc2, str1);
      pth_check(status, "Error creating thread");

      /* Kreiranje 3. niti */
      pthread_t thr3;
      status = pthread_create(&thr3, NULL, threadfunc3, str2);
      pth_check(status, "Error creating thread");

      void * retval_thr3; // Pokazivac za povratnu vrednost iz niti 3
      /* Cekamo niti da zavrse */
      pth_check(pthread_join(thr2, NULL), "pth join error");
      pth_check(pthread_join(thr3, &retval_thr3), "pth join error");

      printf("Povratna vrednost thr3: %.2lf\n", *((double *) retval_thr3));

      return 0;
}
```

Primer 20: Izvršavanje programa sa vise niti

Standarni ulaz:	**Standardni izlaz**

`Zdravo svete`	`Unesite 2 stringa (argumente niti):`
	`Treca nit, string je: svete`
	`Druga nit, string je: Zdravo`
	`Povratna vrednost thr3: 2.72`

Primetimo da pri izvršavanju datog programa ne možemo reći koja nit će se pre izvršiti (nit2 ili nit3). Ukoliko nam je redosled izvršavanja bitan, morali bismo da pozovemo funkciju `pthread_join()` nakon kreiranja druge niti a pre kreiranja treće tako bismo ovezbedili izvršavaju u redosledu nit2, nit3.

2.10 Mutual exclusion (mutex)

Proces ima više niti, te niti se mogu izvršavati paralelno. Ukoliko niti imaju porebu da menjaju zajedničke resurse npr. globalnu promenljivu itd. može doći do problema u sinhronizaciju. Za sinhronizaciju niti koristimo `mutex-e` (Mutual exclusion).

Mutex-ima ograničavamo tzv. kritične delove koda, npr. deo koda u kome se izmenjuje neki deljeni resurs. Kada nit zaključa `mutex`, bilo koja druga nit koja pokša da ga zaključa biće blokirana, tako isto i sledeća itd. Sve dok se `mutex` ne otključa, tada se jedna nit odblokira, pa za nju isto važi.

```c
/* Inicijalizacija  */
/* Dinamicka */
 int pthread_mutex_init(pthread_mutex_t *restrict mutex,
           const pthread_mutexattr_t *restrict attr);

/* Staticka */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

/* Funkcije za zaklj./otklj. */
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

/* Ukoliko nam se mutex nalazi u strukturi, mozemo
 * ga staticki inicijalizovati na sledeci nacin */

typedef struct
{
      pthread_mutex_t mutex;
      double cena;
      char ime_proizvoda[MAX_LEN];
}_shared_data;

/* Staticka inicijalizacija* mutex-a *
 * koji se nalazi u strukturi */
 _shared_data jabuka =
{
      .cena = 97,
      .ime_proizvoda = "jabuka",
      .mutex = PTHREAD_MUTEX_INITIALIZER

 };
```

Primer 21: Funkcije za rad sa mutex-ima

NAPOMENA: Kada kompiliramo program koji koristi funkcije koje rade sa mutex-ima, potrebno je navesti i argument -lpthread.

```
gcc mutex.c -o mutex -std=c99 -Wall -Wextra -lpthread
```

2.11 Bonus primeri

U narednoj sekciji biće dati primeri za koje nije napisan detaljan tekstualni opis već samo konkretni primeri programa.

```c
typedef struct {
	char buffer[BUF_LEN];
	pthread_mutex_t mutex_lock;
	pthread_cond_t cond_var;
	int signal_occured;
	/* promenljiva signal_occured mozda nije potrebna, ali
	 * sluzi da zabelezi signal ukoliko se desio pre  nego
	 * sto smo ga cekali sa: pthread_cond_wait();, kada
	 * ne bi postojala mozda bi cekali zauvek :) */
}_sh_data;

_sh_data shd = {
	.mutex_lock = PTHREAD_MUTEX_INITIALIZER,
	.cond_var = PTHREAD_COND_INITIALIZER,
	.signal_occured = 0
};

void * procitaj_sa_stdin()
{
	pthread_mutex_lock(&shd.mutex_lock);
	printf("Citanje iz 1 threada: ");
	fgets(shd.buffer, BUF_LEN, stdin);

	/* saljemo signal da thr2 - moze da pise na stdout */
	shd.signal_occured = 1;
	pthread_cond_signal(&shd.cond_var);
	pthread_mutex_unlock(&shd.mutex_lock);

	pthread_exit(NULL);
}

void * ispisi_na_stdout()
{
	pthread_mutex_lock(&shd.mutex_lock);

	/* Ako je signal vec poslat nastavi, ako nije onda ga cekaj */
	if (!shd.signal_occured)
		pthread_cond_wait(&shd.cond_var, &shd.mutex_lock);

	shd.signal_occured = 0;

	fprintf(stdout, "Ispis iz 2 threada: %s", shd.buffer);

	pthread_mutex_unlock(&shd.mutex_lock);

	pthread_exit(NULL);
}

int main()
{
	/* Kreiramo 2 thread-a, jedan ce da cita sa stdin a drugi
	* da ispisuje na stdout, ali drugi mora da ceka prvi da
	* procita sa stdin pa tek onda da ispise na stdout */

	pthread_t thr1, thr2;
	pthread_create(&thr1, NULL, procitaj_sa_stdin , NULL);
	pthread_create(&thr2, NULL, ispisi_na_stdout , NULL);
	pthread_join(thr1, NULL);
	pthread_join(thr2, NULL);

	return 0;
}
```

Primer 22: Conditional variables

```c
#include <stdio.h>
#include <stdatomic.h>
/* Standard >= C11 - potrebno */

/* Atomicna promenljiva */
atomic_int gaInt;

int main()
{

      /* Koriscenjem atomicnih funkcija za
       * incijalizaciju i dodelu promenljive */

      atomic_init(&gaInt, 0);
      atomic_fetch_add(&gaInt, 256);

      int vr = atomic_load(&gaInt);

      printf("%d\n", vr);

      return 0;
}
```

Primer 23: Korišćenje atomične promenljive

```c
int numFds=1;
struct pollfd fds[numFds];

for (int i=0; i<numFds; i++) {
      fds[i].fd = STDIN_FILENO;
      fds[i].events = 0;
      fds[i].events |= POLLIN;
}

char buf[1024];
int pret;
int br;

while(1) {

      pret = poll(fds, numFds, 5000);

      if (pret == 0)
            break;
      else {
            br = read(fds[0].fd, buf, 1024);
            buf[br] = '\0';
            fputs(buf, stdout);
      }
}
```

Primer 24: Poll - Čitanje iz pipe-ova i prosleđivanje na stdout

```c
/* Citanje kroz pipe-ove i ispisivanje na stdout */

int epfd = epoll_create(1);

struct epoll_event event;
union epoll_data data;
data.fd = STDIN_FILENO;

event.events = 0;
event.events |= EPOLLIN;
event.data = data;
int epret;

epoll_ctl(epfd, EPOLL_CTL_ADD, STDIN_FILENO, &event);
int bytesRead;
char buf[1024];

while (1)
{
	epret = epoll_wait(epfd, &event, 1, 5000);

	if (epret == 0)
		break;
	else {
		for (int i=0; ; i++) {

			if (event.events & EPOLLIN) {
				bytesRead = read(data.fd, buf, 1024);
				buf[bytesRead] = '\0';

				fputs(buf, stdout);
				break;
			}
		}

	}
}
```

Primer 25: Primer epoll-a - čitanje sa pipe-ova

3 Korisne stvari

3.1 Čitanje liniju po liniju

Funkcija `getline()` čita liniju sa iz toka `stream-a`, to može biti standardni ulaz npr.

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

Funkcija `getdelim()` radi isto što i `getline()` ali delimiter karakter se prosleđuje (zamenjuje '\n').

```
ssize_t getdelim(char **lineptr, size_t *n, int delim, FILE *stream);
```

Povratna vrednost: Broj pročitanih bajtova uključujući i `delimiter` karakter, ali be `'\0'` karaktera u slučaju uspeha. -1 u slučaju neuspešnog čitanja i `errno` se postavlja na odgovarajuću vrednost.

```c
char * line = NULL;
size_t line_size = 0;

while (-1 != getline(&line, &line_size, stdin)) {
      /* U telu petlje radimo sa linijom sta
       * nam je potrebno */
}
```

Primer 26: Čistanje liniju po liniju sa stdout

3.2 Saveti za ispit

- Možete pretražiti sve man strane komandom:

```
man -k . | grep KLJUCNA_REC
```

- Možete pretražiti man strane sistemskih poziva komandom:

```
man -2 syscalls | grep KLJUCNA_REC
```

- Ukoliko ste uključili sva potrebna zaglavlja a kaze vam da funkcija nije definisana, postoji velika mogucnost da ste zaboravil neku definiciju tipa:

```
#define _GNU_SOURCE (500)
```

- Kompilirajte kod tako da vam pokazuje sve moguće greške i upozorenja:

```
gcc primer.c -o primer -std=c99 -Wall -Wextra
```

- Proverite povratnu vrednost u slučaju uspeha/greške komandom `shell-a`:

```
echo $?
```

- Ukoliko vam se javlja greška u fazi linkovanja, verovatno ste zaboravili da kompajleru navedete biblioteku koja se dinamičk povezuje:

```
gcc primer.c -o primer -lpthread -lrt -lm -lNESTO
```

- Ponekad se mogu iskoristiti primeri koji se nalaze u man stranama.
- Budite pažljivi prilikom obrade grešaka