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 da 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, tj. pokrenuti drugi program primjerice naredbom *exec*). 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 se ne može definirati na koji proces treba čekati (dočekuje se prvi proces dijete koji završi). 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 u stranicu, procesi dobivaju 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 koje 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 dobiven 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, mada 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), 0600);

   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`).

Ukoliko se koristi zajedničko računalo za više korisnika i 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*:

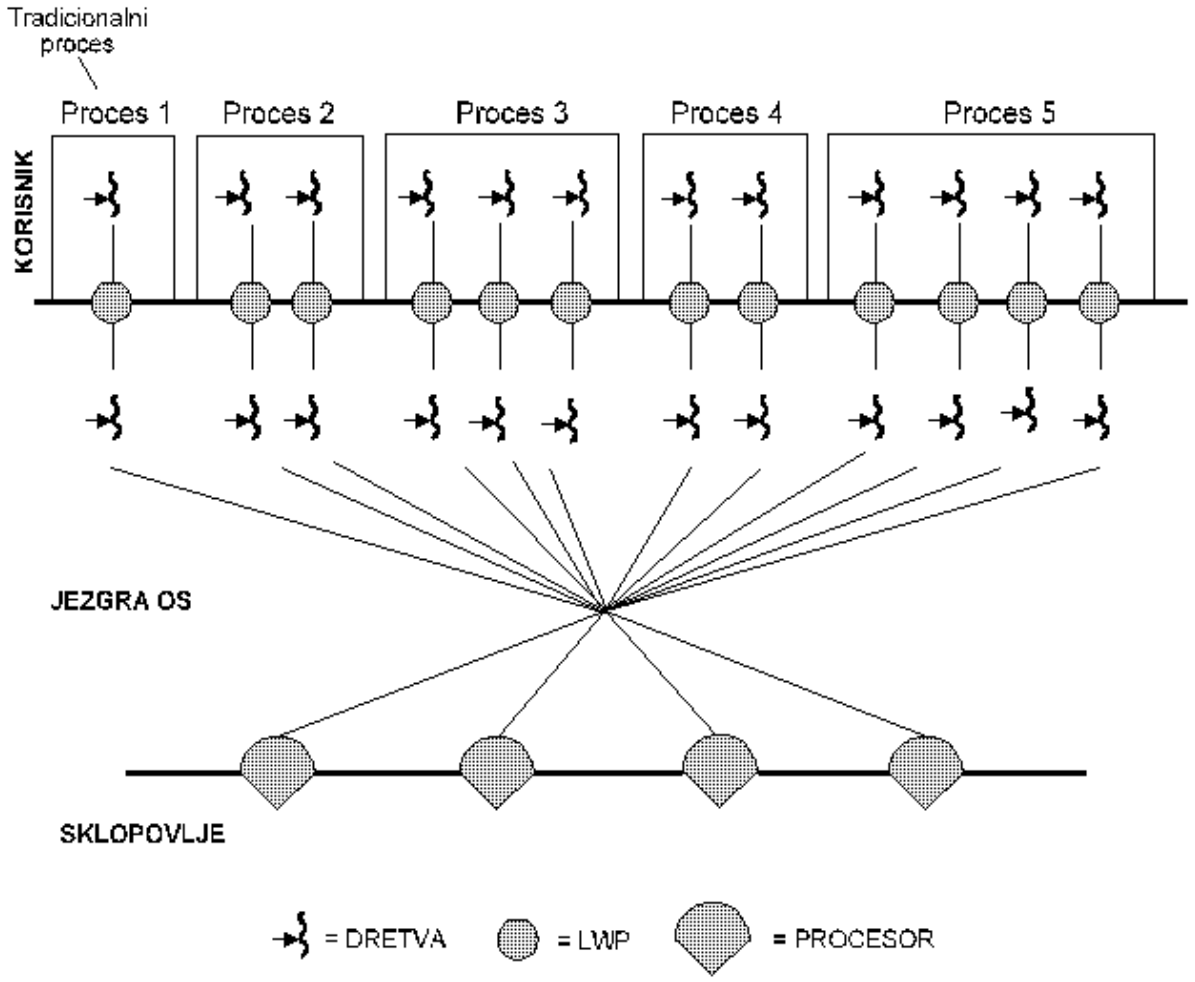
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/msg.h>
#include <values.h>

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 zajednički spremnik) prije no što program završi.

---

2. Višedretvenost

Povijest višedretvenog programiranja počinje 60-ih, dok se njihova implementacija na UNIX sustavima pojavljuje sredinom 80-ih 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

Operacijski sustav za koji su predviđene ove laboratorijske vježbe jest UNIX sustav koji podržava POSIX dretve. Gornja slika prikazuje primjere procesa s jednom, dvije, tri, dvije i četiri dretve. Uobičajeno je da operacijski sustav raspoređuje dretve na raspoložive procesore te se u gornjoj slici svaka dretva vidi i u operacijskom sustavu, tj. svakoj dretvi pripada virtualni procesor, na slici označen s LWP (Light Weight Process).
Neki sustavi dozvoljavaju i djelomično upravljanje dretvama u procesima, pa tako broj dretvi u procesu može biti i veći nego što operacijski sustav vidi (pogledati poziv thr_create u Solarisu i pojmove "bound" i "unbound" dretve; "fiber"-i na Win* i sl.).

Funkcije za rukovanje dretvama

U nastavku su objašnjene funkcije po POSIX standardu (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. `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). Budući da se može prenijeti samo jedan parametar, a 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 s 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 (*thr_join*). `stanje` je kazaljka na kazaljku izlaznog statusa dočekane dretve. Funkcija `pthread_join` zaustavlja izvođenje pozivajuće dretve sve dok određena dretva ne završi s 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 koristi 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 *`-pthread`* koja ukazuje na to da se koristi višedretveni program (npr. `gcc -pthread prvi.c -o prvi`).

(U inačici UNIX operacijskog sustava *Solaris* 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`, `pthread_create`, `pthread_exit`, `pthread_detach`, `pthread_join`, `pthread_mutex_init`, `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_mutex_destroy`, `pthread_cond_init`, `pthread_cond_wait`, `pthread_cond_signal`, `sem_init`, `sem_wait`, `sem_post`, `sem_destroy`...

Win32

Stvaranje procesa pod Win32 obavlja se funkcijom CreateProcess(). Primjer.

Zajednička memorija ostvaruje se pomoću funkcija CreateFileMapping i MapViewOfFile. Primjer

Stvaranje dretvi pod Win32 obavlja se funkcijom CreateThread(). Primjer.