# 1. Kratke upute za rad u UNIX/Linux ljusci

**Natuknice**

- računalo za rad: bilo koji UNIX/Linux sustav, pravi i virtualni, laptop, ...
  - preporuka: ako već nemate neki UNIX/Linux sustav, koristiti VMware/VirtualBox ili slično, ili
  - ako imate windows 10 možete koristiti i Windows Subsystem for Linux
    - upute kako to ostvariti na webu
    - kako doći do datoteka koje su tamo?
    - pokrenuti explorer iz komandne linije: `explorer.exe . &`
- spajanje: terminal, `putty`, `ssh`, ...
- osnovne naredbe: `ls`, `pwd`, `cd`, `mkdir`, `cp`, `mv`, `rm` (!)
- ljuska: `bash` (tab, strelice)
- tekstualni editor: `pico` (`nano`, `vi`, `vim`)
- prevođenje (kompajliranje): `gcc` (`g++`, `cc`, `CC`)
- pokretanje programa: `./a.out`
- rad s procesima: `ps`, `kill -9 PID`
- rad s resursima: `ipcs`, `ipcrm`
- prijenos datoteka: copy/paste, `scp`, `winscp`, `dos2unix`, `unix2dos`

**Primjer 1.1. Primjer programa, prevođenje, pokretanje**

Napisati program, pohraniti ga pod imenom `vj1.c`, prevesti (kompajlirati) i pokrenuti ga.

```
$ pico vj1.c
```

Upisati program (paziti na stil pisanja: razmaci, uvlačenja, zagrade, prozračnost!)

```
1 #include <stdio.h>
2 int main()
3 {
4         printf("Hello World!\n");
5         return 0;
6 }
```

Izlaz iz editora: `Ctrl + x`, `yes`, `yes`

```
$ gcc vj1.c             // program se pokreće sa "./a.out"
$ gcc vj1.c -o prog     // program se pokreće sa "./prog"
```

**Najčešće greške na labosima**

- `ime.c` != `ime.C`
  - prevoditelj (gcc) prepoznaje prvo (`.c`) kao C, a drugo (`.C`) kao C++ program
  - `.cpp .cxx .cc .C .CC` $\Longrightarrow$ C++

- Greške: `Segmentation fault`, `Core dumped`, `Bus error`
  - kazaljka pokazuje izvan procesnog adresnog prostora
  - najčešći razlozi: neinicijalizirana kazaljka; neinicijalizirana varijabla koja se koristi kao indeks polja
- funkcije pozivati sa zagradama ()
  - pri prepisivanju pseudokoda u kod, ako se poziva funkcija (i bez argumenata) obavezno staviti zagrade
  - npr. u psedudokodu piše: `zabrani_prekidanje`,
    a u implementaciji postoji funkcija `void zabrani_prekodanje(){...}`,
    onda ju u kodu pozvati sa `zabrani_prekodanje();`
  - ako se izostave zagrade, prevoditelj neće javiti grešku, ali se funkcija neće pozvati!

**Stil pisanja koda – preporuke**

- kod najčešće nije namijenjen samo osobi koja ga stvara
- u praksi će taj kod dijeliti s drugima, koji će surađivati/nastaviti s radom
- također, često se ne kreće od nule već od postojećeg koda
- na FER-u taj kod još gledaju asistenti
- stoga kod MORA biti pisan prema uobičajenim pravilima za taj programski jezik
- postoji nekoliko sitnih varijacija pravila za C, odabrati bilo koju i konzistentno ju se držati
- preporuka je koristiti pravila koja se koriste za kod Linux jezgre
- osnovni zahtjev: čitljivost – struktura koda mora biti vidljiva
  - uvlačenje koda: “tab” ili razmaci (preporuka: tabulator)
  - KONZISTENTNO koristiti tab ILI razmake
  - NE miješati tab i razmake
  - podesiti alat koji se koristi (editor) na jedno ili drugo i konstantno ga koristiti (ili sve koji se koriste podesiti na isto)
- poželjno:
  - komentirati kod (preporuka, ne uvjet)
  - lomljenje predugih redova radi čitljivosti (npr. 80./100. stupac)
  - “prozračnost” koda – umetanje praznih linija i razmaka
  - razmaci: `a␣=␣b␣+␣c␣*␣d;` ne `a=b+c*d;`
  - zagrade `()`, `[]`, `{}`: npr. `if␣(a[4][2]␣==␣5){`
    (zagrada `{` u novi red samo za funkcije ili iznimno zbog preglednosti)

# 2. Signali (u UNIX-u)

## 2.1. Općenito o signalima u UNIX-u

- signale primaju procesi, šalje ih jezgra, iz svojih razloga ili kao posrednik
- signale upućuje (izvor signala):
  1. jezgra OSa (npr. prilikom greške u pristupu memoriji – `SIGSEGV`)
  2. proces sam sebi (npr. nakon isteka vremena `SIGALRM`)
  3. drugi proces (preko sučelja OS-a, npr. funkcijom `kill`)
  4. korisnik (`Ctrl+C`, preko sučelja OS-a, naredbom `kill` iz ljuske)
- analogno prekidnim signalima na razini procesora (sklopovlja), signali su mehanizam na razini OS-a (primaju ih procesi)
- kao i prekidi, signali služe za obradu asinkronih pojava – proces radi nešto drugo kad mu se signal uputi
- signali se mogu ignorirati (ne svi) ili su povod akciji
- proces može reagirati na signal na sljedeće načine:
  1. prihvatiti i obraditi signal pretpostavljenom funkcijom (npr. prekid izvođenja na SIGINT Ctrl+C)
  2. prihvatiti i obraditi signal zadanom funkcijom (zadana u programu, npr. sa `sigaction`)
  3. zadržati signal (za eventualnu kasniju obradu) (NE RADI u Pythonu)
  4. ignorirati signal
- izuzetak od gornjeg pravila je signal `SIGKILL` (broj 9) koji uništava proces
- signal ne sadrži nikakve dodatne informacije (osim u nekim proširenjima)
- pri prihvatu signala ponašanje procesa za taj signal se mijenja u "zadržati signal" dok se on ne obradi; kad obrada završi isti se signal ponovno može prihvatiti na isti način (slično kao kod prihvata prekida: prvo se zabranjuje daljnje prekidanje...)
- OS pamti samo po jedan signal pojedinog tipa za svaki proces – samo jedan signal istog tipa može biti na čekanju. Npr. ako se trenutno obrađuje `SIGINT` i za vrijeme njegove obrade dođe još 5 signala `SIGINT`, samo će se jedan zapamtiti i kasnije obraditi.
- paziti kako koristiti `sleep` kojega budi bilo koji signal, a vraća broj neprospavanih sekundi
  - `sleep(x)` ⇒ `for (i = 0; i < x; i++) sleep(1);`
- UNIX ima oko 30-tak signala (simbolička imena nalaze se u "signal.h"), neki od njih:
  - `SIGINT` – `Ctrl + C`; `SIGQUIT` – `Ctrl + \ (ž)`; `SIGTSTP` – `Ctrl + Z`
  - `SIGALARM`, `SIGTERM`, `SIGKILL`
  - `SIGUSR1`, `SIGUSR2`
- (info) MS Windowsi ne ponašaju se isto za signale (koriste nešto drugo)

## 2.2. Postavljanje ponašanja za signal (maskiranje signala)

### Sučelje `sigaction`

- postoji više sučelja, u nastavku je prikazan samo `sigaction`
- za prihvat signala prvo treba drefinirati funkciju za obradu signala oblika:
  - `void obrada(int sig){ obradi signal; }`
- potom pozivom `sigaction` povezati signal s tom funkcijom
  - `sigaction(SIGXXXX, &act, NULL)`

**Primjer rada sa signalima**

```
#include <stdio.h>
#include <signal.h>

void obrada ( int sig )
{
    obradi signal;
}

int main ()
{
    struct sigaction act;
    act.sa_handler = obrada; //SIG_DFL, SIG_IGN
    act.sa_flags = 0;
    sigemptyset ( &act.sa_mask );

    sigaction(SIGINT, &act, NULL); //maskiraj SIGINT - Ctrl+C

    radi nešto korisno; //to se prekida signalom

    return 0;
}
```

### Privremeno blokiranje/dozvoljavanje prekidanja sa signalima (u funkciji obrade)

- dok je program u obradi signala novi isti takav signal neće prekinuti obradu
- ipak, ako se želi da se u dijelu koda obrade ponovno prihvaća isti signal (kao u algoritmu za programski prihvat prekida prema prioritetu) onda se to može omogućiti tako da se u kod funkcije obrade doda:

```
void obrada(int sig) { //funkcija registrirana za SIGINT
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    ...
    sigprocmask(SIG_UNBLOCK, &set, NULL); //dozvoli prekidanje sa SIGINT
        nekritični dio koda; //može se prekidati novim signalom SIGINT
    sigprocmask(SIG_BLOCK, &set, NULL); //ponovno blokiraj signal
    ...
}
```

- u `set` se mogu dodati i drugi signali ako je i njih potrebno blokirati/odblokirati
- slično je moguće postići i za drugi signal dok smo u obradi prvog, kombinacijom `SIG_BLOCK` i `SIG_UNBLOCK` i obratno.

# 3. Procesi i dretve u UNIX-u

## 3.1. Općenito o procesima i dretvama

Osnovni pojmovi: program, proces, dretva

- poglavlje u skripti: 4.1. Osnovni pojmovi – program, proces, dretva

Pri pokretanju *programa* operacijski sustav (OS) stvara *proces*:

- rezervira mjesto u memoriji za proces (adresni prostor za procesa)
- stvara opisnik procesa, opisnike datoteka koje proces koristi, ... (u memoriji OS-a)
- stvara početnu dretvu procesa (i njen opisnik)
- dretva može krenuti s radom

*Adresni prostor procesa* sastoji se od:

1. segmenta s instrukcijama (program)
2. segmenta s podacima (program)
3. gomile
4. stoga

Adresni prostor procesa dohvatljiv je svim dretvama procesa!

Jedinstveno za svaku dretvu:

- identifikacijski broj dretve
- kontekst (stanje registara procesora)
- kazaljka stoga i sam stog
- prioritet
- *razne zastavice (npr. signalna maska)*
- *privatni prostor svake dretve*

U adresnom prostoru OS-a:

- segment sustavskih podataka procesa:
  - opisnik procesa, opisnici dretvi
  - opisnici naprava, otvorenih datoteka, . . .
  - međuspremnici

Stvaranje novog procesa je zahtjevnija operacija od stvaranja nove dretve i traži više sredstava od sustava (npr. više memorije). Stoga se uglavnom preporuča korištenje stvaranja više dretvi unutar istog procesa. Iznimke su uglavnom radi sigurnosti, kad se želi sakriti podatke među dretvama (npr. u web pregledniku se ne želi dopustiti da jedna stranica ("tab") može doći do podataka s druge).

## 3.2. Pokretanje više dretvi u UNIX okruženju

**Stvaranje dretvi prema POSIX sučelju**

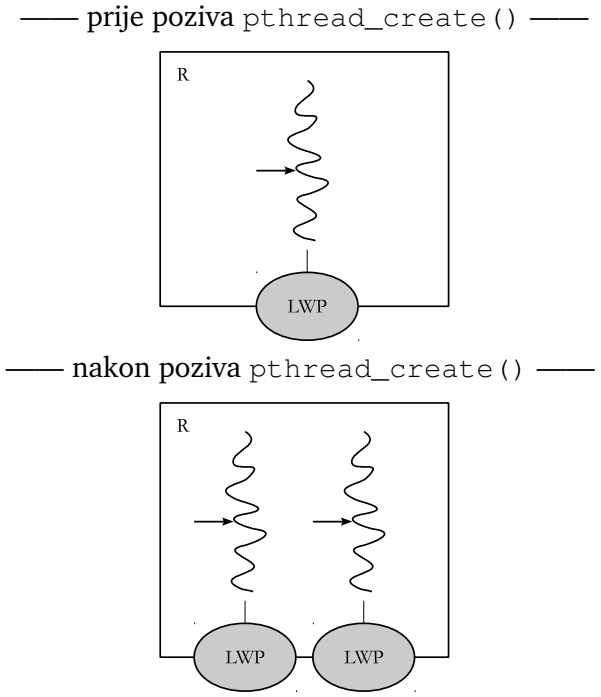
POSIX (engl. *The Portable Operating System Interface*) nastoji omogućiti prenosivost programa na razini izvorna koda - da se program napisan za jedan operacijski sustav (i okruženje) može jednostavno prevesti i pokretati na nekom drugom.

POSIX sučelje za stvaranje nove dretve je funkcija `pthread_create` s parametrima:

1. `pthread_t *opisnik` – mjesto za spremanje opisnika dretve
2. `pthread_attr_t *svojstva` – parametri za novu dretvu (može NULL)
3. `void *(*početna_funckija)(void *)` – početna funkcija za novu dretvu
4. `void *pparam` – parametar koji se šalje u početnu funkciju

Npr. `pthread_create (&opisnik, NULL, dretvena_funkcija, &podaci)`

**Grafički prikaz:**




**Slika 3.1. Prije i poslije poziva `pthread_create`**

Primjer programa koji stvara više dretvi:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <unistd.h>
5
6  void *dretva ( void *rbr )
7  {
8          int i, *d = rbr;
9          for ( i = 1; i <= 5; i++ ) {
10                 printf ( "Dretva %d, i=%d\n", *d, i );
11                 sleep (1);
12         }
13         return NULL;
14 }
15 int main ()
16 {
17         int i, j, BR[3];
18         pthread_t t[3];
19
20         for ( i = 0; i < 3; i++) {
21                 BR[i] = i;
22                 if ( pthread_create ( &t[i], NULL, dretva, &BR[i] ) ) {
23                         printf ( "Ne mogu stvoriti novu dretvu!\n" );
24                         exit (1);
25                 }
26         }
27         for ( j = 0; j < 3; j++)
28                 pthread_join ( t[j], NULL ); //čekaj kraj dretve t[j]
29
30         return 0;
31 }
32 // gcc -lpthread ... ili na Linuxu gcc -pthread ...
```

**Prijenos parametara u dretvu**

- kada treba svim dretvama dati istu vrijednost:
  - preko globalne varijable
  - parametar = kazaljka na istu strukturu
- kada treba svakoj dretvi dati njenu zasebnu vrijednost:
  - za svaku dretvu napraviti zasebnu strukturu podataka i kazaljka na nju poslati dretvi
  - sve drukčije je KRIVO !
    - npr. umjesto `&RBR[i]` staviti `&i` je KRIVO
      - glavna dretva i stvorene dretve rade paralelno, a glavna mijenja varijablu `i`

**Završetak rada dretve**

- izlaskom iz početne funkcije (navedene u `pthread_create`)
- pozivom `pthread_exit`
- završetkom rada početne dretve (dretve koja kreće s funkcijom `main`) završava proces – i sve ostale dretve tada budu prekinute! (vrijedi za C/C++, ali ne za Python)

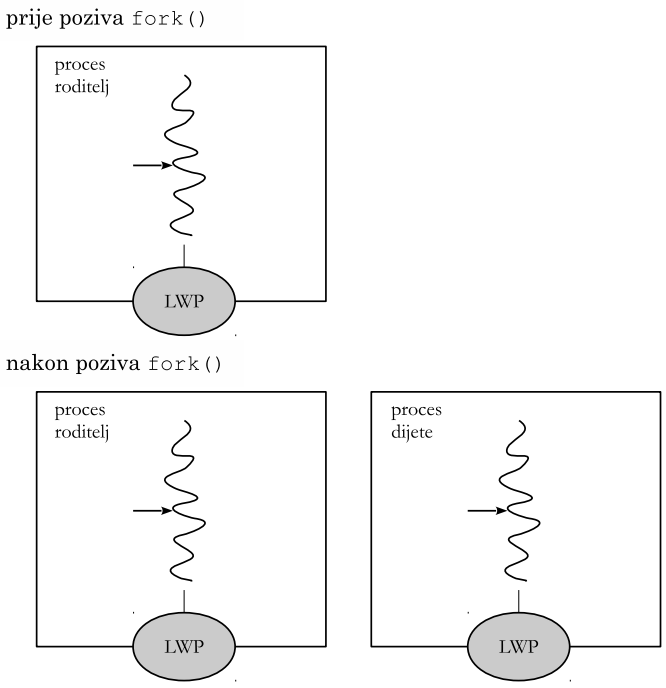
## 3.3. Pokretanje više procesa u UNIX okruženju s `fork()` (info)

- proces *roditelj* stvara proces *dijete*
- dijete dobiva *kopiju* trenutnog adresnog prostora roditelja
  - podaci NISU zajednički (logički gledano, o implementaciji u 8. poglavlju)
  - u C-u: globalne varijable imaju istu vrijednost u oba procesa samo dok se ne napravi promjena u jednom od procesa – ta je promjena vidljiva samo u tom procesu, nije u onom drugome (tamo je i dalje ona stara vrijednost)
- prilikom međuprocesne komunikacije upliće se OS

**Sučelje `fork()`**

- prototip: `int fork (void);`
- u funkciju ulazi jedan proces, izlaze dva
- dijete kao povratnu vrijednost dobiva 0, a proces roditelj dobiva PID djeteta
- PID – jedinstveni identifikacijski broj procesa (svaki proces ima svoj PID)
- PID roditelja != PID djeteta
- u slučaju greške povratna vrijednost je -1 i novi proces nije stvoren
- greška: sustav nema dovoljno resursa za stvaranje nova procesa
- proces dijete je kopija procesa roditelja
- sredstva sustava su jedino što procesi dijele:
  - npr. otvorene datoteke, dijeljeni spremnik, semafori, redovi poruka

**Grafički prikaz fork-a:**




**Slika 3.2. Prije i poslije poziva `fork`**

Uobičajeni kraći način korištenja `fork`-a:

```
if ( fork() == 0 ) //oba procesa uspoređuju povratnu vrijednost s nulom!
{
    posao_procesa_djeteta;
    exit (0); // završava proces, izlazni status = 0 (uspješno završen)
}
nastavak rada procesa roditelja;
wait ( NULL ); //zaustavlja roditelja do proces dijete ne završi
```

Ili pravilnije, s ispitivanjem povratne vrijednosti:

```
pid_djeteta = fork ();
switch ( pid_djeteta )
{
case -1:
    printf("Ne mogu stvoriti novi proces!\n");
    exit(1);
case 0:
    Posao djeteta;
    exit(0);
default:
    Posao roditelja za novostvoreni proces;
}
nastavak rada roditelja;
wait ( NULL );
```

**Završetak rada procesa**

- proces završava ("svojevoljno", bez grešaka ili signala):
  - izlaskom iz početne funkcije (`main`)
  - pozivom funkcije `exit(izlazni_status)` (npr. `exit(0)` ako je program napravio sve ili npr. `exit(1)` ako je bila kakva greška)
- proces roditelj može čekati na završetak procesa djeteta s funkcijom `wait(&status)`
  - ukoliko samo želimo čekati kraj nekog procesa djeteta (prvog koji završi) a status nas ne zanima može se koristiti poziv `wait(NULL)`
  - ako želimo status, onda argument treba biti kazaljka na cijeli broj preko kojeg ćemo dohvatiti status (npr. `int status; wait(&status);`)
  - ako želimo čekati na točno određni proces, treba koristiti funkciju `waitpid(pid, &status, 0)` (`man waitpid` za upute)
- svaki proces u operacijskom sustavu ima svog roditelja
  - iznimka su neki početni procesi
  - npr. pogledati stupce PID i PPID (id procesa i id procesa roditelja) u ispisu naredbe `ps -Al`
- proces roditelj može završiti i prije procesa djeteta
  - u tom slučaju dijete dobije novog roditelja (jednog od onih početnih)
  - kaže se da proces dijete tada postaje "zombie"
  - najčešće takvo ponašanje nije poželjno, osim kada se pokreću neki procesi "u pozadini" (npr. servisi), kada se želi te procese odijeliti od njihova roditelja (npr. ljuske) i datoteka koje je od njega naslijedio (stdin, stdout, stderr, ...)

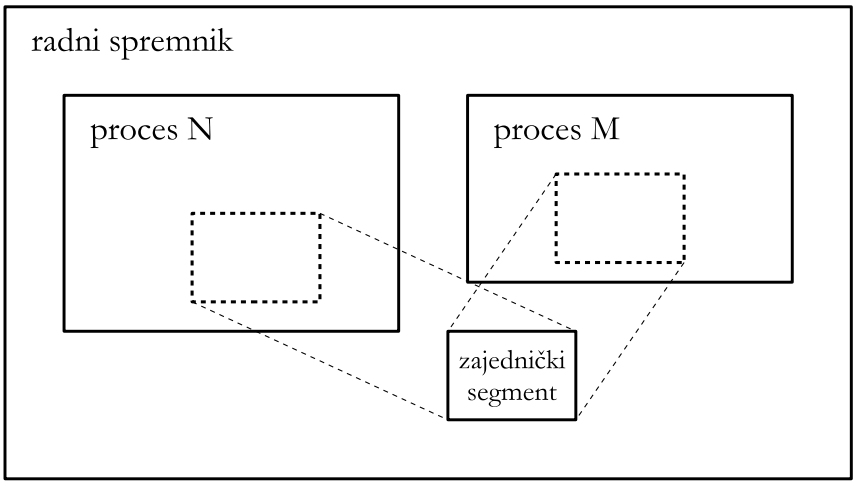
## 3.4. Korištenje zajedničkog spremnika (info)

Što je to i kako koristiti "zajednički spremnik"?

- kod dretvi istog procesa
  - "zajednički spremnik" je zapravo cijeli adresni prostor procesa
  - najjednostavnije je koristiti globalne varijable
- kod dretvi različitih procesa
  - zajednički spremnik treba stvoriti preko sučelja OS-a:
    1. stvoriti segment zajedničkog spremnika i povezati ga s procesom
    2. globalne varijable (kazaljke) usmjeriti na taj segment
    3. stvoriti nove procese s fork
    4. globalne varijable = kazaljke ne mijenjati – mijenjati/koristiti sadržaj na koji one pokazuju, a koji je u zajedničkom spremniku!

U nastavku pod "zajednički spremnik" smatramo zajednički spremnik za komunikaciju dretvi različitih procesa.

**Zajednički spremnik za procese**




**Slika 3.3. Zajednički spremnik (idejno ostvarenje)**

Primjeri s fork() i zajedničkom memorijom

# 4. Sinkronizacijski mehanizmi

## 4.1. Semafori u UNIX-u

Postoje dva sučelja:

- `semop` – "tradicionalni" semafori
- `sem_*` – "noviji" semafori – njih koristiti u vježbama

### 4.1.1. Sučelja za `semop`

Od operacijskog sustava se dohvaća skup semafora sa funkcijom semget:

```
    id_skupa = semget(ključ, koliko_semafora, zastavice)
```

Primjer:

```
    SemId = semget(IPC_PRIVATE, n, 0600);
```

Postavljanje početne vrijednosti i dodatna kontrolna podešavanja i brisanje (semctl)

```
    semctl(id_skupa, koji_semafor_u_skupu, naredba, argumenti)
npr.
    semctl(SemId, SemNum, SETVAL, SemVal); //postavi vrijednost
    semctl(SemId, 0, IPC_RMID, 0); //briši skup semafora
```

Povećavanje/smanjivanje sa semop:

```
    struct sembuf SemBuf;
    SemBuf.sem_num = SemNum;
    SemBuf.sem_op = SemOp; //SemOp je 1 za Postavi, -1 za Čekaj
    SemBuf.sem_flg = 0;
    semop(SemId, &SemBuf, 1);
```

Primjeri korištenja na webu (u uvodu sinkronizacije) i u "konceptima"

### 4.1.2. Sučelja za `sem_*`

Sučeljem `sem_*` ostvaruje se opći semafor (OSEM s predavanja). Ulazak i izlazak iz kritična odsječka može se ostvariti ovim semaforom uz početnu vrijednost 1.

`int sem_init(sem_t *sem, int mprocesi, unsigned int koliko );`
`int sem_post(sem_t *sem);` – PostaviOSEM
`int sem_wait(sem_t *sem);` – ČekajOSEM
`int sem_destroy(sem_t *sem);` – obriši semafore

**Koncept korištenja**

```
sem_t s; //globalna varijabla

    //inicijalizacija, npr. u funkciji main() prije stvaranja dretvi
    sem_init(&s, 0, 5); //početna vrijednost = 5

        //u novim dretvama
        sem_wait(&s); //ČekajOSEM
        ...
        sem_post(&s); //PostaviOSEM

    //u funkciji main, nakon završetka ostalih dretvi
    sem_destroy(&s);
```

**Korištenje za dretve različitih procesa**

```
sem_t *sem; //globalna varijabla = za kazaljku na objekt u zajedničkoj memoriji

void proces(int id)
{   ...
    sem_wait(sem); i/ili sem_post(sem);
    ...
}
int main()
{
    ...
    ID = shmget(IPC_PRIVATE, sizeof(sem_t), 0600);
    sem = shmat(ID, NULL, 0);
    shmctl(ID, IPC_RMID, NULL); //moze odmah ovdje, nakon shmat, ili na kraju nakon
         shmdt

    sem_init(sem, 1, 5); //početna vrijednost = 5, 1=>za procese

    ... fork() ...
    ...
    ... wait(NULL) ...

    sem_destroy(sem);
    shmdt(sem);

    return 0;
}
```

Primjeri na webu (osnovni koncepti OSa: semafor)

## 4.2. Monitori

### Sučelja

```
//globalne varijable
pthread_mutex_t ključ;
pthread_cond_t uvjet;

pthread_mutex_init(&ključ, NULL);
pthread_cond_init(&uvjet, NULL);

pthread_mutex_lock(&ključ);

pthread_cond_wait(&uvjet, &ključ);
pthread_cond_signal(&uvjet);
pthread_cond_broadcast(&uvjet);

pthread_mutex_unlock(&ključ);
```

### Koncept korištenja

```
#include <pthread.h>
//globalne varijable
pthread_mutex_t m;
pthread_cond_t red;

    //inicijalizacija, npr. u funkciji main() prije stvaranja dretvi
    pthread_mutex_init(&m, NULL);
    pthread_cond_init(&red, NULL);

        //u novim dretvama(jedan ili oba iduća bloka)
        //prvi blok
        pthread_mutex_lock(&m);          // Uđi_u_monitor(m)
        while (uvjet_nije_zadovoljen)
            pthread_cond_wait(&red, &m);     // Čekaj_u_redu_uvjeta(red, m)
        ... (napravi nešto u monitoru)
        pthread_mutex_unlock(&m);            // Izađi_iz_monitora(m)

        //drugi blok
        pthread_mutex_lock(&m);          // Uđi_u_monitor(m)
        ... (napravi nešto u monitoru)
        pthread_cond_signal(&red);           // Oslobodi_iz_reda_uvjeta(red)
        pthread_mutex_unlock(&m);            // Izađi_iz_monitora(m)

    //u funkciji main, nakon završetka ostalih dretvi
    pthread_mutex_destroy(&m);
    pthread_cond_destroy(&red);
```

Monitor se ostvaruje s pomoću *varijabli međusobnog isključivanja* i *uvjetnih varijabli*

1. varijable međusobnog isključivanja (*mutex*)
   - kritični odsječak se zaključa s pomoću kontrolnih varijabli (ključeva)
   - "zaključavanje" => ulaz u monitor => `pthread_mutex_lock(&m)`
   - "otključavanje" => izlaz iz monitora => `pthread_mutex_unlock(&m)`
2. uvjetne varijable = red uvjeta
   - red uvjeta se koristi isključivo unutar monitora!
   - korištenje reda uvjeta pri "zauzimanju sredstava":
     a) dretva najprije ulazi u monitor
     b) u monitoru provjerava može li zauzeti tražena sredstva, tj. može li nastaviti s radom – provjera se izvodi korištenjem varijabli koje definiraju stanje sustava u programu, van jezgre – npr. jesu li lijevi i desni štapić slobodni (kod problema pet filozofa)
     c) ako ne može nastaviti, dretva poziva `pthread_cond_wait(&red, &m)`
        – funkcija `pthread_cond_wait(&red, &m)` uključuje tri operacije
          i) izbacuje dretvu iz monitora `m` (i propušta iduću unutra)
          ii) blokira dretvu i stavlja ju u red uvjeta `red`
          iii) nakon što je dretva odblokirana iz reda uvjeta, dretvu ponovno stavlja u monitor `m` ili u red za ulaz u monitor (ako je u njemu već neka druga dretva)
        – prve dvije operacije (i, ii) su "atomarne", obavljaju se u jednoj jezgrinoj funkciji
        – treća operacija (iii) obavlja se nakon odblokiranja te dretve
        – blokiranu dretvu u redu uvjeta može odblokirati samo druga dretva
   - korištenje reda uvjeta pri "oslobađanju sredstava":
     a) dretva najprije ulazi u monitor
     b) u monitoru mijenja zajedničke varijable (opcionalno)
     c) dretva oslobađa samo prvu ili sve dretvu iz reda uvjeta pozivima: `pthread_cond_signal(&red)` ili `pthread_cond_broadcast(&red)`
     d) dretva izlazi iz monitora (i omogućava drugima ulaz)
     e) ukoliko dretva pozove jednu od navedenih funkcija, ali u tom trenutku nije bilo blokiranih dretvi, taj poziv neće ništa napraviti – to nije semafor koji bi u takvom slučaju povećao vrijednost semafora – uz red uvjeta se ne veže vrijednost!
        – poziv `pthread_cond_wait(&red, &m)` uvijek blokira pozivajuću dretvu
        – poziv `pthread_cond_signal(&red)` odblokira dretvu ako ima takva dretva u zadanom redu uvjeta, u protivnom ne radi ništa

Varijable međusobnog isključivanja i redovi uvjeta trebaju biti globalne varijable.

Primjer na webu — stari most (OS koncepti)

**Isječak kôda 4.1. Primjer s ping-pong dretvama**

```
#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <stdlib.h>
#include <unistd.h>

pthread_mutex_t m;     //monitor
pthread_cond_t red[2]; //redovi uvjeta

char *ispis[] = {"ping", "pong"};
int na_redu = 0;
int radi = 0;
void *dretva(void *p);

int main() {
        pthread_t t;
        void *x[] = {NULL, (void*)1};

        pthread_mutex_init(&m, NULL);
        pthread_cond_init(&red[0], NULL);
        pthread_cond_init(&red[1], NULL);

        while(1) {
                pthread_create(&t, NULL, dretva, x[rand()%2]);
                sleep(1);
        }
        return 0;
}

void *dretva(void *p) {
        int tip = 0;
        if (p)
                tip = 1;

        pthread_mutex_lock(&m);
        printf("Kreće dretva %s\n", ispis[tip]);
        while(na_redu != tip || radi)
                pthread_cond_wait(&red[tip], &m);
        radi = 1;
        pthread_mutex_unlock(&m);

        printf("%s\n", ispis[tip]);

        pthread_mutex_lock(&m);
        radi = 0;
        na_redu = 1 - tip;
        pthread_cond_signal(&red[1-tip]);
        pthread_mutex_unlock(&m);

        return NULL;
}
```