Osnovni koncepti operacijskih sustava

Upravljanje dretvama: sinkronizacija

POSIX sučelje

Koncepti ↔ implementacija

- Razlike implementiranih operacija od koncepata?
- Proširene mogućnosti implementacija?
- Restrikcije u korištenju implementiranih operacija?

Pregled prezentacije

- POSIX dretve
  - uvodna razmatranja (zašto dretve)
  - stvaranje, zaustavljanje, upravljanje dretvama
  - privatni podaci dretve

- Sinkronizacija
  - varijable međusobnog isključivanja, uvjetne varijable
  - zaključavanja na čitanje i pisanje, barijera, radno zaklj.
  - POSIX semafori
  - “UNIX” semafori

POSIX

- Prije POSIX-a:
  - puno “vlasničkih” standarda
    - (gotovo) svaka UNIX distribucija je imala vlastito sučelje
    - programi nisu bili prenosivi na razini izvornog koda
- **POSIX - *P*ortable *O*perating *S*ystem *I*nterface *[for Unix]***
- Skup povezanih normi definiranih od strane IEEE radi definicije sučelja prema programima (API)
- sučelja koja definira POSIX nalaze se u gotovo svim sustavima, samo je format drukčiji, ali ista funkcionalnost
  - POSIX omogućuje prenosivost izvornih kodova
- Nastao iz projekta koji je krenuo oko 1985. godine
- IEEE Std 1003, ISO/IEC 9945
- http://en.wikipedia.org/wiki/POSIX (kratki pregled)
- http://www.unix.org/2008edition (pregled sučelja)
- http://www.opengroup.org

POSIX dretve

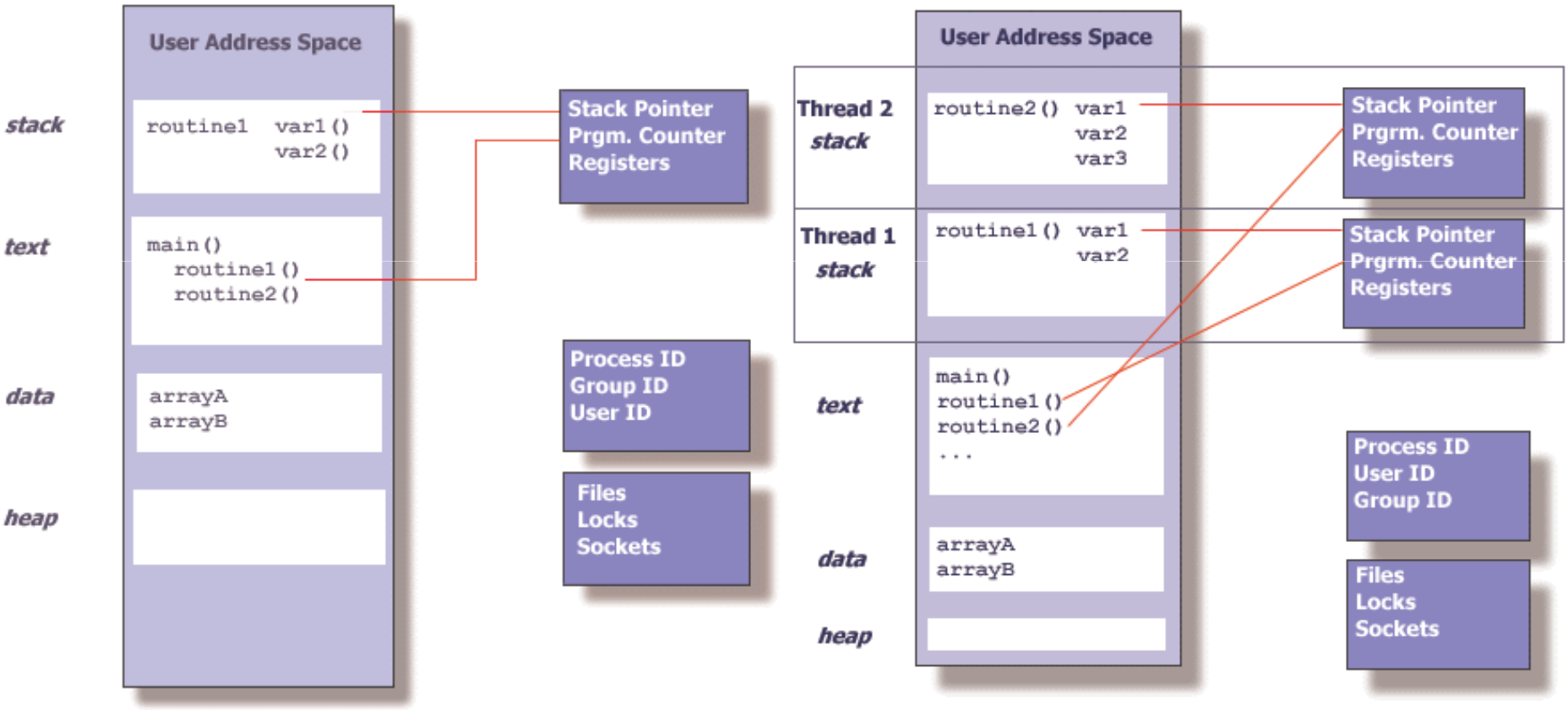
- POSIX **definira sučelje, ne implementaciju**
  - e.g. http://www.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html

- **Implementacije** ne moraju sadržavati potpunu implementaciju svih navedenih sučelja ili opisane funkcionalnosti

- Zašto toliko pažnje prema dretvama, a ne procesima?

Dretve naspram procesa: zauzeće sredstava

- Stvaranje nove dretve u postojećem procesu zauzima znatno manje sredstava sustava od novog procesa



Proces i početna dretva Proces sa više dretvi

Dretve naspram procesa : vremena stvaranja

- Stvaranje 50,000 novih procesa/dretvi (u sekundama, izvor: https://computing.llnl.gov/tutorials/pthreads/)

Platforma	fork()			pthread_ create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Dretve naspram procesa

- Dretve su nezahtjevne
  - traže manje sredstava sustava
  - značajno kraće vrijeme stvaranja
  - brža zamjena konteksta (zamjena dretvi unutar istog procesa naspram zamjeni dretvi različitih procesa)
  - jednostavnija i brža komunikacija među dretvama
  - jednostavnije dijeljenje podataka = cijeli adresni prostor
    - nekontrolirani pristup može dovesti do grešaka!

- Dretve su manje sigurne
  - jedna dretva može zaustaviti (ugasiti) cijeli proces
  - ako dretve rade odvojene poslove, može ih se odvojiti u zasebne procese
    - primjer: Google Chrome stvara novi proces za svaki novi “tab” – svaka se Web stranica obrađuje u zasebnom procesu; kada se stranica zatvara svi se njeni podaci automatizmom brišu; sigurnost;

Stvaranje dretvi

- Pokretanjem programa stvara se proces a sa njime i početna dretva
- Iz perspektive C programera:
  - početna dretva počinje s funkcijom "**`main`**"

```
int main (…)
{
  …
}
```

- Ako su potrebne i druge dretve, treba zatražiti njihovo stvaranje iz programa (iz početne dretve)
- POSIX sučelje za stvaranje dretve: **`pthread_create`**

pthread_create - parametri

```
int pthread_create (
      pthread_t *opisnik,
      pthread_attr_t *atributi,
      void *(*pocetna_funkcija)(void*),
      void *argument);
```

- **opisnik** – adresa za pohranu opisnika stvorene dretve
- **atributi** – razni atributi koji se mogu zadati za novu dretvu (podaci o načinu raspoređivanja, stog, …)
- **pocetna_funkcija** – početna funkcija dretve (kao što je “main” za početnu dretvu)
- **argument** – argument koji se šalje u početnu funkciju (adresa)

Primjer stvaranja dretve – (pthread_create.c)

```
#include <pthread.h>
#include <stdio.h>
void *nova_dretva (void *p) {
   int *n = p;
   int num = *n;
   printf("U dretvi %d\n", num);
   return p; //ili pthread_exit(p);
}
int main () {
   pthread_t t1, t2;
   int n1 = 1, n2 = 5, *status1, *status2;
   pthread_create(&t1, NULL, nova_dretva, (void *) &n1);
   pthread_create(&t2, NULL, nova_dretva, (void *) &n2);
   pthread_join(t1, (void *) &status1);//cekaj kraj dretve t1
   pthread_join(t2, (void *) &status2);
   printf("Dohvaceni statusi: %d %d\n", *status1, *status2);
   return 0;
}
```

Prevođenje:

```
#gcc e1.c -lpthread
#./a.out
U dretvi 1
U dretvi 5
Dohvaceni statusi: 1 5
#
```

Parametar za početnu funkciju

Povratna vrijednost dretve (status)

Povratna vrijednost procesu roditelju (ljusci)

Slanje parametara novoj dretvi

- Ne slati adresu varijable petlje!

```
for (i = 0; i < N; i++)
    pthread_create(&t, NULL, nova_dretva, (void *) &i);
```

- Varijabla petlje se mijenja i željena vrijednost ne mora biti predana (obično tada sve dretve pročitaju zadnju vrijednost: `N`)!
- Ako treba poslati broj, može se i izravno (adresa je broj!):

```
for (i = 0; i < N; i++)
    pthread_create(&t, NULL, nova_dretva, (void *) i);
```

U dretvi se broj dohvaća:

```
void * nova_dretva (void *p) {
    int num = (int) p;
    ...
```

- Ako je potrebno više parametara, koristiti strukture, npr.

```
struct param { int a, b, c; double d; … } p[N];
for (i = 0; i < N; i++) {
    //inicijalizirati p[i] podacima za dretvu 'i'
    pthread_create(&t[i],NULL,nova_dretva,(void *)&p[i]);
}
```

Upravljanje dretvama

- Stvorene dretve završavaju svoje izvođenje:
  - izlaskom iz početne funkcije
  - pozivom **`pthread_exit`**
    - poziv je pogodan ako dretva trenutno nije u početnoj funkciji
- Početna dretva (ili neka druga) može čekati na završetak druge dretve sa:
  - **`pthread_join`**
- Dretva može “nasilno” prekinuti drugu dretvu sa:
  - slanjem signala dretvi
    - **`pthread_kill (thread, signal)`**
    - a u obradi signala poziva se **`pthread_exit`**
  - traženje prekida dretve
    - **`pthread_cancel (thread)`**
- Ako početna dretva završi, završi cijeli proces (a s njime i sve ostale dretve) !!!

Oslobađanje sredstava koje zauzima dretva

- Za svaku stvorenu dretvu rezerviraju se potrebna sredstva u jezgri
  - opisnik dretve
  - stog i privatni podaci unutar adresnog prostora procesa
- Kada dretva završi, zauzeta se sredstva ne oslobađaju automatski
- Sredstva se oslobađaju kada:
  - je pozvan **`pthread_join`** (druga dretva je pozvala),

  ili
  - dretva je označena kao "odvojiva" ("*detachable*")
    - korištenjem **`attr`** parametra pri stvaranju dretve, ili naknadno sa **`pthread_detach`**
    - kada "odvojiva" dretva završi, njena se sredstva automatski oslobađaju i mogu se iskoristiti za nove dretve

Primjer: dretve u Web poslužitelju

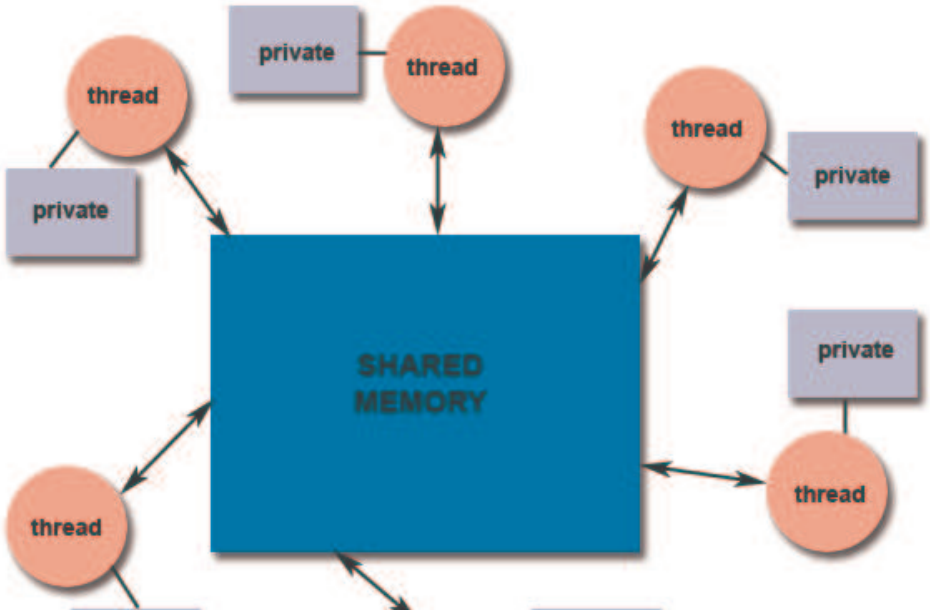
```
int main() {
    //inicijalizacija u početnoj dretvi koja će čekati na spajanja klijenata
    pthread_attr_t attr;
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    //navedenom zastavicom, zauzeta sredstva dretve će se po završetku osloboditi
    ...
    //osnovna petlja – čekanje na zahtjeve te njihovo prosljeđivanje u dretve na obradu
    while (nije_kraj) {
        klijent = malloc(sizeof(struct Klijent));
        klijent->socket = accept(srv, &klijent->addr, &klijent->len);
        pthread_create(&tid,&attr, dretva_obrade, (void *)klijent);
    }
    ...
}

//dretva za obradu zahtjeva
void *dretva_obrade (void *p) {
    struct Klijent *klijent = p;
    //obradi klijentov zahtjev
    ...
}
```

Privatni podaci dretve

- Privatni podaci dretve – vezani uz pojedinu dretvu

- Kada trebamo privatne podatke?
  - princip je isti kao i s globalnim varijablama
    - ovdje želimo da te “globalne varijable” vidi samo dretva
  - može se smanjiti broj parametara za pozive funkcija
  - može se koristiti kada se parametri ne mogu izravno poslati: npr. pri obradi signala



Primjer – privatni podaci dretve (pthread_specific.c)

```
#include <pthread.h>
pthread_key_t kStatus, kPodaci;//globalne varijable, dijeljene među dretvama
...
//početna dretva – inicijalizacija 'ključeva', potrebno za privatne podatke dretve
pthread_key_create(&kStatus, oslobodi);
pthread_key_create(&kPodaci, oslobodi);
//inicijalno, NULL vrijednost je povezana uz ključeve
//za sve dretve
...
pthread_create...
...
//radna dretva - inicijalizacija
stat = malloc(sizeof(struct Status));     //stat - lokalna varijabla
podaci = malloc(sizeof(struct Podaci)); //podaci - lokalna varijabla

//povezi 'stat' s kljucem 'kStatus' u trenutnoj dretvi
pthread_setspecific(kStatus, stat);
pthread_setspecific(kPodaci, podaci);
...
//radna dretva – u nekoj funkciji
s = pthread_getspecific(kStatus);//dohvati podatke povezane uz ključ kStatus
p = pthread_getspecific(kPodaci);
... //koristi 's' i 'p'
```

```
void *oslobodi (void *d) {
    if (d != NULL)
        free(d);
    return NULL;
}
```

Sinkronizacija

- Dostupni mehanizmi:
  - varijable međusobnog isključivanja:
    - **`pthread_mutex_lock/unlock/init`**
  - uvjetne varijable
    - **`pthread_cond_wait/signal/broadcast/init`**
  - Zaključavanja na čitanje ili pisanje
  - Barijera
  - Radno zaključavanje (engl. *spin lock*)
  - Semafori (RT biblioteka)
  - “UNIX” semafori
- Mehanizmi su prikazani kroz primjere

Primjer monitora: Problem starog mosta

- Stari most preko rijeke predstavlja ograničenja na promet:
  - automobili mogu preko njega voziti samo u jednom smjeru istovremeno: most je uzak
  - istovremeno na njemu ne smije biti više od tri automobila
    - konstrukcije je loša (ili je most kratak, pa više i ne stane)
- Simulirati automobile dretvama
  - koristiti koncept monitora
  - simulacija mora poštivati gornja ograničenja
- U idućoj implementaciji stanje mosta definirano je sa:
  - brojem automobila na mostu: **`auti_na_mostu`**
  - smjerom kretanja automobila na mostu: **`smjer_na_mostu`**

Problem starog mosta - rješenje (stari_most.c)

```
void *novi_auto (void *p) {
    struct Automobil *Auto = p;
    pthread_mutex_lock(&m);
    while(auti_na_mostu>2 || (smjer_na_mostu!=-1 && smjer_na_mostu!=Auto->smjer))
        pthread_cond_wait(red_a[Auto->smjer], &m);
    auti_na_mostu++;
    smjer_na_mostu = Auto->smjer;
    printf("Auto %2d na mostu, ide na %s (na mostu %d, idu na %s)\n",
        Auto->id, tsmjer[Auto->smjer], auti_na_mostu, tsmjer[smjer_na_mostu+1]);
    pthread_mutex_unlock(&m);
    usleep(5000000); //prelazak preko mosta
    pthread_mutex_lock(&m);
    auti_na_mostu--;
    if (auti_na_mostu > 0) {
        pthread_cond_signal(red_a[Auto->smjer]);
    }
    else {
        smjer_na_mostu = -1;
        pthread_cond_broadcast(red_a[1-Auto->smjer]);
    }
    printf("Auto %2d sisao s mosta, ide na %s (na mostu %d, idu na %s)\n",
        Auto->id, tsmjer[Auto->smjer], auti_na_mostu, tsmjer[smjer_na_mostu+1]);
    pthread_mutex_unlock(&m);
    free(Auto);
    return NULL;
}
```

```
int main () {
    pthread_t thr_id; pthread_attr_t attr;
    int id_auta = 0; struct Automobil *Auto;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,
        PTHREAD_CREATE_DETACHED);
    while (1) {
        Auto = malloc(sizeof(struct Automobil));
        Auto->id = ++id_auta;
        Auto->smjer = rand() & 1;
        printf("Novi auto %d iz smjera %s\n",
        id_auta, tsmjer[1 + 1 - Auto->smjer]);
        pthread_create(&thr_id, &attr,novi_auto,
                (void *) Auto);
        usleep(2000000);
    }
    return 0;
}
```

Ovo rješenje nije pravedno!
Za bolju pravednost bilo bi potrebno brojanje prelazaka.

Zaključavanje na čitanje ili pisanje (citaci_pisaci.c)

```
void *citac (int p) {
  while (1) {
     pthread_rwlock_rdlock(&cpkljuc);
     br_citaca++;
     usleep(2000000);
     br_citaca--;
     pthread_rwlock_unlock(&cpkljuc);
     usleep(1000000 * p);
  }
}
void *pisac (int p) {
  while (1) {
     pthread_rwlock_wrlock(&cpkljuc);
     br_pisaca++;
     usleep(1000000);
     br_pisaca--;
     pthread_rwlock_unlock(&cpkljuc);
     usleep(1000000 * p * 5);
  }
}
```

Osnove mehanizma zaključavanja na čitanje ili pisanje:

- Kada čitač zaključa, samo ostali čitači mogu prolaziti kroz zaključavanje
- Kada pisač zaključa, nitko drugi ne može proći zaključavanje dok pisač ne otključa
- Kada pisač čeka, čitačima se ne dozvoljava prolaz, iako možda je objekt zaključan na čitanje
  - ovo ponašanje je teorijski opisano u opisu funkcija, negdje nije podržano

Barijera (barijera.c)

```
pthread_barrier_t barijera;
void *dretva (int p) {
   while (1) {
        usleep(1000000 * p);
        na_barijeri++;
        pthread_barrier_wait(&barijera);
        na_barijeri--;
        if(!na_barijeri)
                printf("---Prosli barijeru---\n");
        usleep(1000000 * p);
   }
}
int main () {
   ...
   pthread_barrier_init(&barijera, NULL, 5);
   pthread_create(&thr_id, &attr, dretva, (void *) 1);
   ...
   pthread_create(&thr_id, &attr, dretva, (void *) 5);
   usleep(50000000); //vrijeme simulacije
   return 0;
}
```

- barijera blokira dretve dok sve do nje ne dođu
- kada zadnja dretva dođe do barijere, sve dretve se propuštaju, a barijera se resetira (postavlja na početnu vrijednost)
- pri inicijalizaciji barijere postavlja se broj dretvi

Radno zaključavanje (spinlock.c)

Pogledati opterećenje sustava! (visoko)

```
pthread_spinlock_t kljuc;
void *dretva (int p) {
   while (1) {
        printf("Dretva %d spremna\n", p);
        pthread_spin_lock(&kljuc);
        printf("Dretva %d unutar K.O.\n", p);
        usleep(1000000 * p);
        printf("Dretva %d izlazi iz K.O.\n", p);
        pthread_spin_unlock(&kljuc);
        usleep(1000000);
   }
}
int main () {
   pthread_t thr_id;
   pthread_spin_init(&kljuc, PTHREAD_PROCESS_PRIVATE);
   pthread_create(&thr_id, NULL, dretva, (void *) 1);
   ...
   pthread_create(&thr_id, NULL, dretva, (void *) 6);
   usleep(50000000); //vrijeme simulacije
   return 0;
}
```

Semafor (semafor.c)

```
#include <semaphore.h> //POSIX RT
sem_t sem;
void *dretva (int p) {
   while (1) {
        printf("Dretva %d spremna\n", p);
        sem_wait(&sem);
        printf("Dretva %d unutar K.O.\n", p);
        usleep(1000000 * p);
        printf("Dretva %d izlazi iz K.O.\n", p);
        sem_post(&sem);
        usleep(1000000);
   }
}
int main () {
   pthread_t thr_id;
   sem_init(&sem, 0, 1);
   pthread_create(&thr_id, NULL, dretva, (void *) 1);
   ...
   pthread_create(&thr_id, NULL, dretva, (void *) 6);
   usleep(50000000); //vrijeme simulacije
   return 0;
}
```

Pogledati opterećenje sustava! (malo)

Početna vrijednost semafora

“UNIX” semafori (semafor3.c)

```
#include <sys/sem.h>
int sem;
void *thread (int p) {
    struct sembuf op;
    op.sem_num = 0;//indeks prvog semafora
    op.sem_flg = 0;
    while (1) {
        op.sem_op = -1; //smanjiti vrijednost za jedan => ČekajSem
        semop(sem, &op, 1);
        printf("Dretva %d unutar K.O.\n", p);
        usleep(1000000 * p);
        printf("Dretva %d izlazi iz K.O.\n", p);
        op.sem_op = 1; //povećati vrijednost za jedan => PostaviSem
        semop(sem, &op, 1);
        usleep(1000000);
    }
}
int main () {
    sem = semget(IPC_PRIVATE, 1, 0600 | IPC_CREAT);
    semctl(sem, 0, SETVAL, 1);
    ...
```

`semop (sem_id, sem_op, n)`

- `sem_id` – id skupa semafora
- `sem_op` – niz operacija (provode se kao **atomarne**)
- `n` – broj operacija

Dohvati skup semafora, sa samo jednim semaforom

Početna vrijednost semafora 0

Proširene funkcionalnosti

- “Tempirano” čekanje
  - `pthread_mutex_timedlock(mutex, vrijeme)`
  - `pthread_cond_timedwait`
  - `pthread_rwlock_timedrdlock/timedwrlock`
  - `sem_timedwait`

  - dretve neće biti blokirane dulje od zadanog vremena
    - ako vrijeme isteče i dretva nije prošla kroz funkciju, dretva se odblokira
    - korištenjem povratne vrijednosti, javlja se istek vremena, tj. da objekt nije zauzet

- Npr. ako se pri pokretanju programa, korisniku daje mogućnost da unutar 10 sekundi pritiskom na tipku može zadati drukčije parametre pokretanja, tada se u funkciji koja blokira dretvu (na ulaz tipkovnice) može zadati 10 sekundi kao najdulje vrijeme blokiranja

Proširene funkcionalnosti

- Neblokirajuće “pokušaj” funkcije:
  - **`pthread_mutex_trylock`**
  - **`pthread_rwlock_tryrdlock/trywrlock`**
  - **`pthread_spin_trylock`**
  - **`sem_trywait, semop`** sa **`IPC_NOWAIT`** zastavicom
  - ako se objekt ne može odmah zaključati (jer ga je zaključala druga dretva), ne blokirati dretvu već samo povratnom vrijednošću pokazati neuspjelo zauzeće

- Npr. pri prosljeđivanja zahtjeva slobodnom poslužitelju može se koristiti kôd:

```
for (i = 0; i < br_posluzitelja; i++)
  if (sem_trywait(&p[i]) == 0)
      break; //pošalji zahtjev “i”-tom poslužitelju
```

Problemi višedretvenog programiranja

- Višedretveno sigurne funkcije (engl. *thread-safe, MT-safe (MultiThreading), reentrant*) (dalje *MT sigurne* funkcije)
  - funkcije se mogu istovremeno pozivati iz raznih dretvi i dalje raditi ispravno (kao u slučaju korištenja samo jedne dretve, slijednog pozivanja)
- Neke biblioteke i pozivi (NE jezgrine funkcije!) nisu MT sigurne (npr. **`gethostbyname`**, **`rand`**)!
  - funkcije koriste (interno) neke globalne varijable (međuspremnike, kazaljke, identifikatore, …)
  - prije korištenja detaljno proučiti opis funkcije (*man* stranice)
    - npr. http://www.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_09
  - ako je potrebno, zaštititi te nesigurne funkcije (npr. monitorom)
- Izgrađivati MT sigurne funkcije – izbjegavati globalne varijable ili ih koristiti kontrolirano (u kritičnom odsječku)