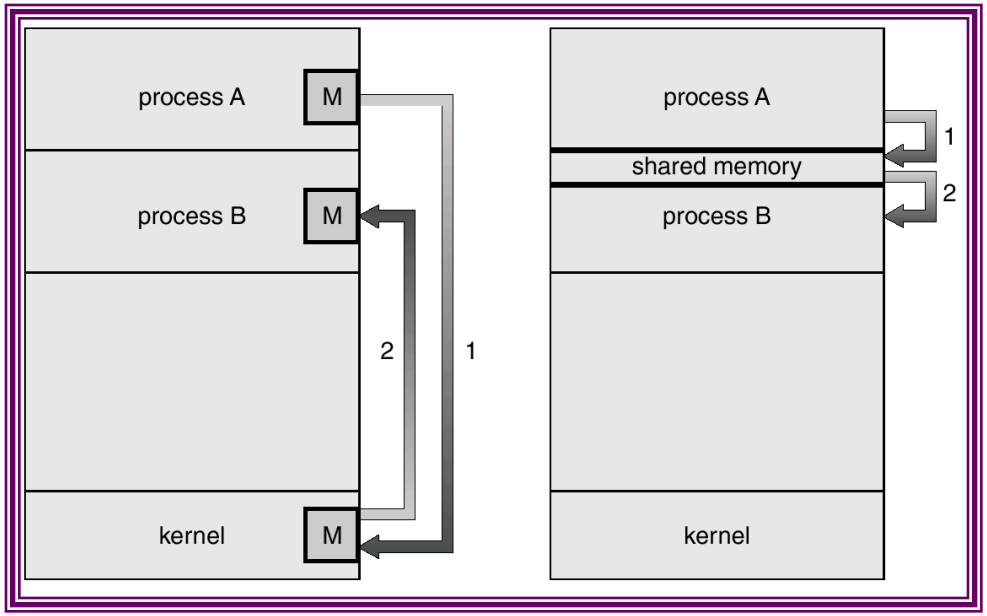
Interprocess communication

- **Interprocess komunikacioni mehanizam** dozvoljava **procesima** da:
  - **razmenjuje podatke**
  - **sinhronišu izvršavanje**
- Već smo razmatrali razne forme IPC kao što su:
  - **pipes**
  - **named pipes**
  - **signali**
- **Pipes (unamed)** imaju nedostatak što za njih znaju samo procesi koji su **naslednici** procesa koji je obavio **pipe SC**, ostali procesi ne mogu pristupati tom unamed pipe-u, **nema komunikacije**.
- Mada **named-pipes** dozvojavaju procesima koji nisu povezani sa pipe-om da komuniciraju, **oni se ne mogu generalno koristiti preko mreže**, kao što ne **dozvoljavaju višestruke IPC**. Nemoguće je naterati named pipe da obezbedi privatni kanal za par komunikacionih procesa.
- Procesi mogu komunicirati slanjem signala preko kill SC, ali cela poruka je samo "**signal number**".

Interprocess communication

- Ovo poglavlje opisuje druge forme IPC.
- Počinje sa ispitivanjem **proces tracing**, gde jedan proces prati (trace) i kontroliše izvršavanje drugog procesa,



- System V IPC paket:
- **poruke**
- **shared memory**
- **semafori**

- Objašnjavaju se tradicionalni metodi za IPC po mreži, i
- na kraju se daje user-level pregled za **BSD socket**.
- Ovde se ne diskutuju mrežni protokoli, adresiranje, name servisi.

System V IPC

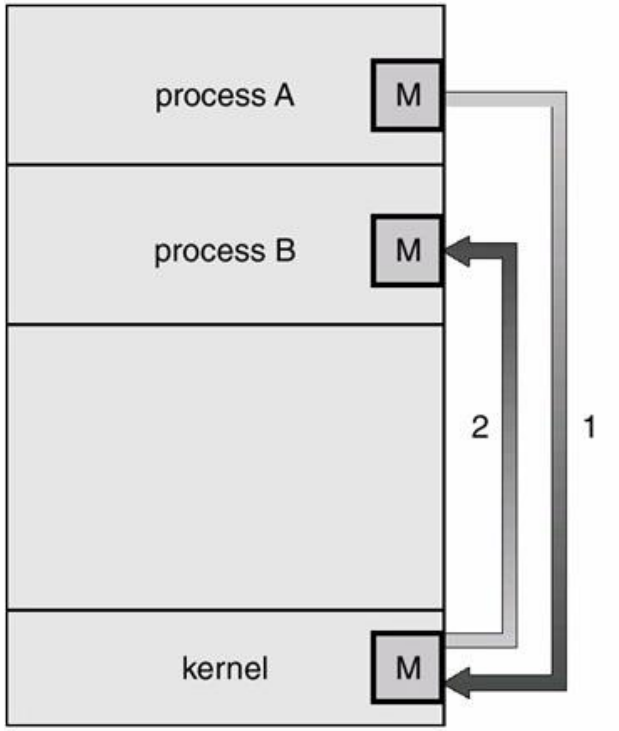
- UNIX System V IPC paket sastoji se od **3 mehanizma**:
  - **poruke** dozvoljavaju procesima da šalju formatirane nizove procesima
  - **shared memory** dozvoljava procesima da dele delove svojih adresnih prostora
  - **semafori** dozvoljavaju procesima da sinhronišu svoja izvršavanja
- **Svaki mehanizam** sadrži **tabelu** čiji **ulazi** opisuju **sve instance mehanizma**
- Svaku **ulaz** sadrži **numerički key**, koji je ime izabrano od strane usera
- **Svaki mehanizam** sadrži **get SC** za kreiranje **novog ulaza** ili da **dobijanje postojećeg ulaza** i parametre za SC koji **uključuju ključ i flagove**.
  - **Kernel pretražuje odgovarajuću tabelu** za ulaz koji sadrži zadati **key**.
  - **Procesi mogu pozvati get SC** sa ključem **IPC-PRIVATE** da obezbede **potpuno novi neiskorišćeni ulaz.**
  - **Procesi mogu postaviti IPC_CREAT** bit u flag polju da kreiraju novi ulaz ako **takav sa zadatim ključem ne postoji**, a mogu da forsiraju objavu greške postavljenjem **IPC_EXCL i IPC_CREAT** flagovima, **ako takav ulaz već postoji**.
  - **SC get vraća kernel-chosen deskriptor** za **korišćenje u drugim SC** i **get je analogan FS SC(open i creat)**

System V IPC

- **Za svaki IPC mehanizam**, kernel koristi sledeću formulu da **nađe index** u tabelu data struktura iz deskrtiptora
- **index = descriptor % [number of entries in the table]** **[moduo]**
- Na primer, ako tabela za message struktura **sadrži 100 ulaza**, deskriptori **za ulaz 1** su 1, 101, 201.
  - Kada proces uklanja ulaz, kernel inkrementira deskriptor dodeljen za broj ulaza u tabeli: inkrementirana vrednost postaje novi deskriptor za ulaz kada se ponovo pozove get SC.
  - Procesi koji pokušavaju da priđu ulazu sa svojim starim desktriptorom dobijaju otkaz (fail). Na primer, ako je deskriptor dodeljen sa (ulazom 1) =201, pa se ukloni, kernel mu dodeljuje novi deskriptor, 301.
- Svaki IPC ulaz ima **zaštitnu strukturu**, koja uključuje user ID i grupni ID procesa koji je kreirao ulaz, a UID i GID se setuju preko control SC i **rwx** za vlasnika, grupu i ostale.
- Svaku ulaz sadrži **druge statusne informacije**, kao što je **PID procesa** koji je zadnji ažurirao ulaz (send a message, receive a message, ..) i **vreme kad se update dogodio**.
- Svaki mehanizam ima **"control" SC**, da postavi **upit na ulaz**, da postavi statusne informacije ili da ukloni ulaz iz sistema. Kada proces postavi upit za status ulaza, kernel proverava da li proces ima r pravo i kopira podatke iz ulaza u user prostor. Slično, da bi se postavili parametri ulaza, kernel verifikuje da li je UID procesa = UID creator-a ili je UID=root (w pravo nije dovoljno, može samo vlasnik). Kernel kopira user data u ulaz, postavljajući UID, GID, prava, i druga polja zavisno od mehanizma. Kernel ne menja vlasnika ulaza, a samo vlasnik ili root mogu ukloniti ulaz.

Messages

- Postoje 4 SC za poruke:
- **msgget: vraća** (ili **kreira**) MD (**message descriptor**) koji označava **MQ** (**message queue**) za korišćenje u **drugim SC**
- **msgctl:** ima opcije za **setovanje i čitanje parametra** MD, a **ima i opciju da ukloni MD**
- **msgsnd: šalje** poruku
- **msgrcv: prima poruku**



msgget SC

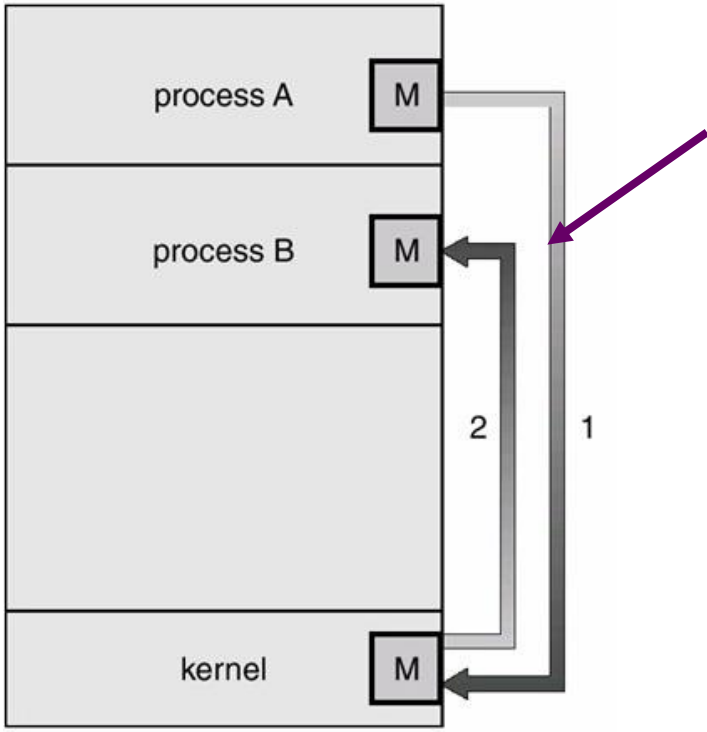
- **Sintaksa** za **msgget SC** je:
- **msgqid = msgget(key, flag);**
- gde je
  - **msgqid** deskriptor koji vraća SC,
  - **key i flag** imaju semantiku opisanu kao za get SC.

- **Kernel čuva poruke na linkovanoj listi** (queue) za deskriptor i koristi msgqid kao index u polje **MQH (mesage queue headers).**

- U dodatku IPC permissions polja, **queue struktura** sadrži sledeća polja:
  - **pointer na prvu i zadnju poruku** u linkovanoj listi
  - **broj poruka u ukupna broj data bajtova na linkovanoj listi**
  - **maksimalni broj data bajtova koji mogu biti na linkovanoj listi**
  - **PID ove poslednjih procesa** koji su **slali ili primali poruke**
  - **TS (time stamps)** poslednjih **msgsnd**, **msgrcv** i **msgctl** operacija

- **Kada user pozove msgget da kreira novi deskriptor**, kernel pretražuje **message queues da se vidi ima li neki ulaz taj key**. **Ako nema ulaza** za **zadati ključ**, kernel alocira **novu queue strukturu**, inicijalizuje je, i vraća **msgqid useru**. U protivnom, **provere se prava pristupa i vraća se**

msgsnd



- Proces koristi msgsnd SC da pošalje poruku:
- **msgsnd(msgqid, msg, count, flag);**
- gde je
  - **msgqid** deskriptor koji vraća msgget SC,
  - **msg** je pointer na **strukturu** koja se sastoji od
    - user-chosen integer tip i polja od karaktera,
  - **count** daje veličinu data polja, a
  - **flag** specificira akciju koju kernel preuzima ako se prekorači interni bafer space.

algorithm msgsend

- **algorithm msgsend /* send a message*/**
  - input:
    - (1) **message queue descriptor**
    - (2) **address of message structure**
    - (3) **size** of **message**
    - (4) **flags**
  - output: number of byte sent
- { **check legality of descriptors, permission;**
- **while(not enough space to store messages)**
- { **if(flags specify not to wait) return;**
- **sleep**(until event enough space available);}
- **get message header**;
- **read message text** from **user space to kernel**;
- **adjust data structure**:
  - enqueue message **header**;
  - **message header points to data**, count, time stamps, PID;
- **wakeup all processes waiting to read message from queue**;
- }

msgsnd

- Kernel proverava
  - **da li sending proces ima w** pravo na **MD**,
  - **da li dužina poruke ne pravazilazi sistemski limit**,
  - **da li MQ ne sadrži suviše bajtova**, i
  - da li je **tip poruke +integer**.

- Ako **svi testovi prođu**,
  - kernel alocira prostor za poruku iz mape poruka i
  - kopira data iz user space-a.

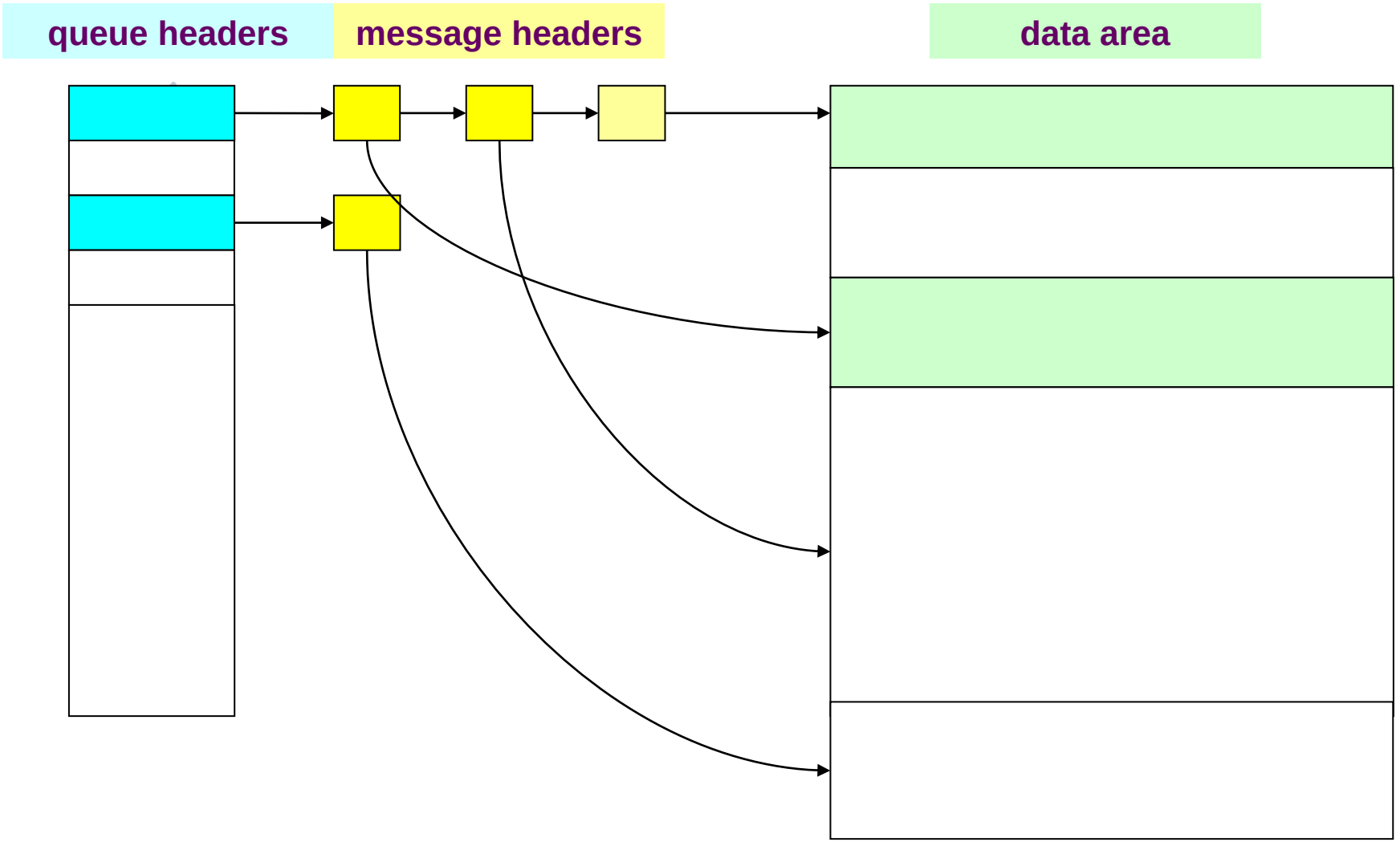
- Kernel alocira MH i stavlja ga na kraj **linkovane liste MQH**
  - **upisuje tip poruke i veličinu u MH**
  - **postavlja MH da ukazuje na message data** i
  - **ažurira različita statistička polja** (broj poruka i bajtova u MQ, TS i uid sendera) u queue header
- Kernel zatim **budi procese koji spavaju**, čekajući da poruke stignu u MQ

- Ako broj bajtova **prekorači queue limit, proces spava sve dok se druge poruke** ne **izbace-read iz queue**.

- Ako proces specificira flag **IPC_NOWAIT**, on se vraća bez error uslova.

MQH, MH, messages

- Slika prikazuje **poruke na MQ**, prikazuje **queue headers**, linkovanu listu **MH**, pointere iz MH u **data area PT**



msgsnd example

- Posmatrajmo sledeći program:
- proces poziva **msgget** da dobije descriptor za **MSGKEY.**
- Zatim se **setuje poruka dužine 256 bajtova**,
- **mada** se **koristi samo prvi integer**,
- **kopira** se **PID u text poruke**,
- dodeljuje se **tip poruke =1**
- **poziva** se **msgsnd SC** da **pošalje poruku**.

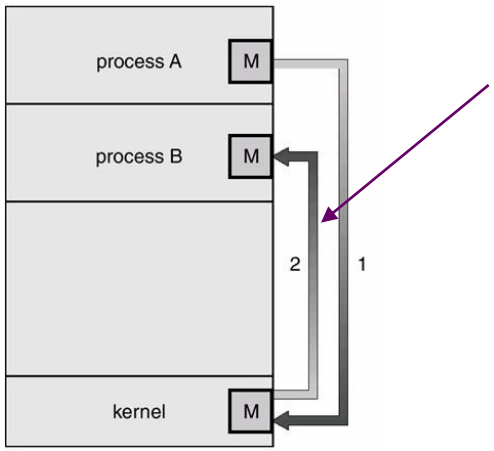
```
struct msgform { long mtype;  char  mtext[256]; }msg;
```

type	body of message

msgsnd example

- client process
- #include <sys/types.h> #include <sys/ipc.h>
- #include <sys/msg.h> #define MSGKEY 75
- **struct msgform { long mtype; char mtext[256]; }msg;**
- **main()**
- {
- **struct msgform msg; int msgid, pid, *pint;**
- **msigid = msgget(MSGKEY, 0777);**
- pid = getpid();
- **pint = (int *) msg.mtext;**
- ***pint=pid; /*copy pid into message text*/**
- **msg.mtype=1;**
- **msgsnd(msigid, &msg, sizeoff(int),0);**
- **msgrcv(msigid, &msg, 256, pid,0) /*pid is used as msg type*/**
- **printf("client: receive from pid %d", *pint);**
- }

msgrcv



- Proces prima poruku preko **msgrcv SC**
- **count = msgrcv(id, msg, maxcount, type, flag)**
  - gde je:
  - **id message deskriptor**,
  - **msg** je **pointer na user strukturu** koja će **sadržavati primljenu poruku**,
  - **maxcount je veličinu data polja** u msg
  - **type** (sledeca strana) je opisan kasnije
  - **flag** specificira **akciju** koju kernel **preuzima ako poruka nije u MQ**.
  - **count**, povratna vrednost, je **broj bajtova koje je dobio user.**

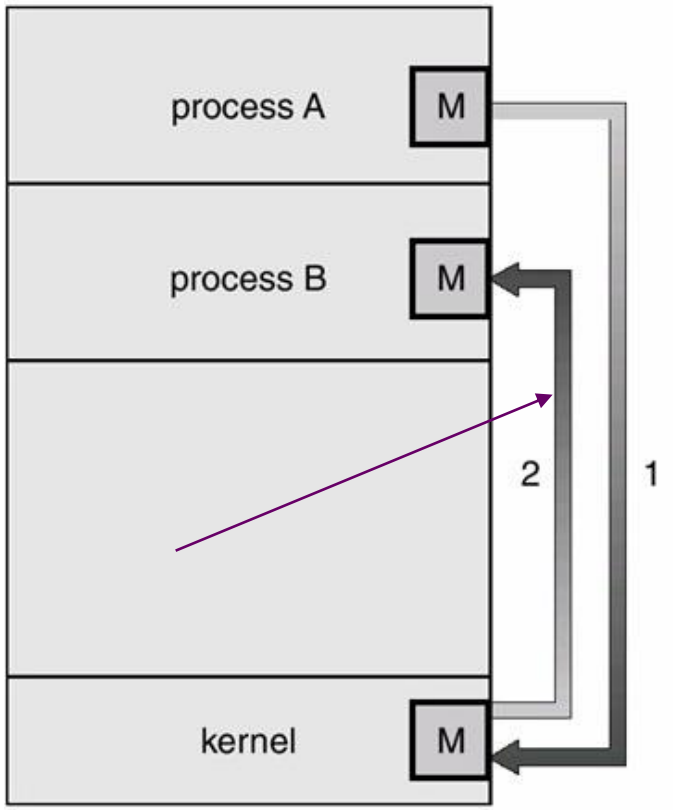
msgrcv

- **Kernel proverava dali user ima potrebna prava** na **MQ**.
- Ako je **type=0**, **kernel nalazi prvu poruku** na **linkovanoj listi**.
- Ako je **veličina poruke manja** ili **jednaka veličini maxcount**,tada **kernel**
- **kopira poruku** u **user space** = msg,
- **podešava MQ**:
  - **dekremetira broj poruka u queue,**
  - **setuje vreme prijema** i
  - **PID procesa koji je primio poruku**,
  - **podešava linkovanu listu**
  - oslobađa prostor koji je zauzimala poruka.
- Iza toga kernel **budi sve procese koji su čekali da se oslobodi mesto u MQ**.

- **Ako je poruka veća** od **maxcount**, **kernel vraća grešku u SC**, i **ostavlja poruku na queue.**
- Ako **proces ignoriše size-ograničenja**,(bit MSG_NOERROR je setovan u flag), kernel odseca poruku, vraća zahtevani broj bajtova i **uklanja celu poruku** iz MQ.

algorithm msgrcv

- **algorithm msgrcv /* receive a message*/**
- input:
  - (1) message descriptor
  - (2) address of data array for incoming message
  - (3) size of data array
  - (4) requested message type
  - (5) flags
- output: number of byte in returned message
- { **check permissions;**
- **loop**: **check legality of message descriptor**;
- **/*find message to return to user*/**
- **if(**requested **message type == 0**)
  - **consired first message** on **queue**;
- else if(requested **message type > 0**)
  - **consired first message** on queue **with given type**;
- else **if(requested message type < 0**)
  - **consired first** of **lowest message on queue**;
  - such that its **type is** <= **absolute value of requested type**;



algorithm msgrcv

- **if(there is a message**)
- {
- **adjust message size** or **return error** if **user size is too small**;
- **copy message type, text from kernel space** to **user space**;
- **unlink message** from **queue**;
- return;
- }

- /***no message***/
- if(**flag** specify **not to sleep**) **return with error;**
- **sleep(event message arrives on queue);**
- **goto loop**;
- }

type

- Proces može primiti poruku odgovarujućeg tipa,
- postavljanjem **type** polja u formatu za **msgrcv**.

- Ako je **type=0**, **kernel nalazi prvu poruku** na **linkovanoj listi**.
- Ako je to **+integer**, **kernel vraća prvu poruku datog tipa**.

- Ako je **–integer**, kernal nalazi najniži tip svih poruka na MQ,
- obezbeđujući da je **manja ili jednaka**, **apsolutnoj vrednosti polja type**.
- Na prmer
  - MQ sadrži 3 poruke čiji su tipovi **3, 1 i 2,**
  - a proces hoće prijem poruke sa **-2,**
  - tada će uzeti poruku tipa 1.

- **Ako nijedna poruka** ne **zadovolji receive zahtev**,
- **kernel gura proces** na **spavanje**
- osim ako u flag-u nije **IPC_NOWAIT** bit setovan.

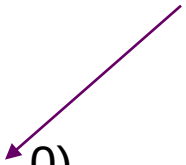
server/client example

- Analizirajmo programe **client proces** i **server proces**.
- Program **server proces**, prikazuje **strukturu servera** koji **obezbeđuje servis klijent procesima.**
  - **Na primer**, mogu se primati **zahtevi od klijentskih procesa** za **database informac**ije.
  - **Server proces** je **jedna od tačaka za pristup u database**.
- **Server kreira message strukturu**
  - postavljanjem IPC_CREAT flaga u **mssget SC** i
  - **prima sve poruke tipa =1** od **klijentskih procesa**.
- Zatim čita **text poruke**, nalazi **PID klijentskog procesa**, i
  - **setuje tip povratne poruke** za **klijentski proces** sa tim **PID-om,**
  - tako da **server proces prima poruke** od **svih klijentskih procesa tipa 1,**
  - a **klijent proces prima samo poruke** koje je **baš za njega poslao serverski** proces.
- Procesi u tom slučaju idu na na **isti MQ**.

server proces

- **server process**
- #include <sys/types.h>
- #include <sys/ipc.h>
- #include <sys/msg.h>
- #define MSGKEY 75
- **struct msgform**
- {
- **long mtype;**
- **char mtext[256];**
- **}msg;**
- main()
- {
- int i, pid, *pint;
- **extern cleanup();**
- for(i=0; i<20; i++) **signal(i, cleanup);**
- **msigid** = **msgget**(**MSGKEY**, 0777 | IPC_CREAT);

server proces

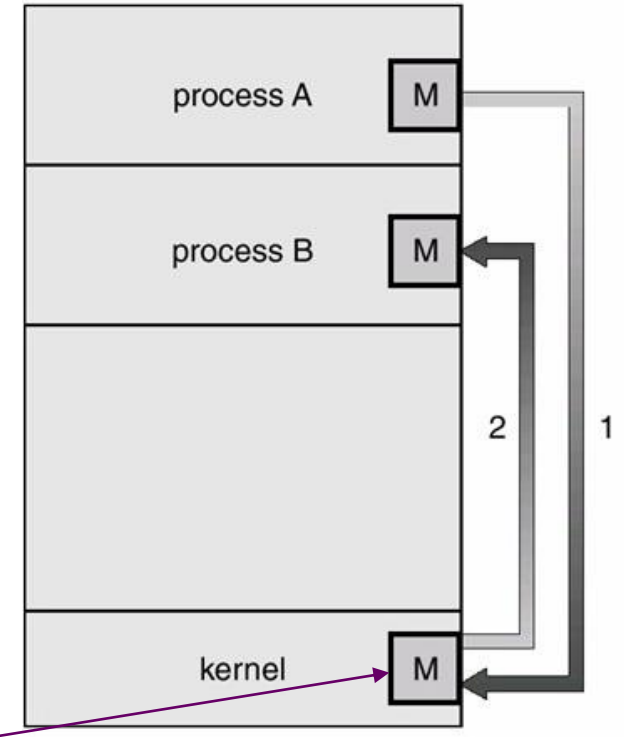


```
for(;;)
{
        msgrcv(msgid, &msg, 256, 1, 0)
        pint = (int *) msg.mtext;
        pid = *pint;
        printf("server. receive from pid %d", pid);

         msg.mtype = pid; /message for client*/
        *pint = getpid(); /*server PID*/
        msgsnd(msgid, &msg, sizeoff(int),0)
}
}
cleanup()
{
msgctl(msgid, IPC_RMID,0);
exit();
}
```

server/client example

- **poruke se formatiraju** u **type-data parovima.**
- **Type je prefiks koji dozvoljava procesim**a da **selektuju poruke posebnog tipa.**
- **Procesi mogu da ekstrakuju poruke partikularnog tipa iz MQ** u
- poretku u kome su stigle,
- a **kernel održava poredak**.
- Mada je moguće realizovati **message passing šemu** na **user-level sa FS**,
- **mesages su kernelske memorijske strukture**,
- **brže su i fleksibilniije za IPC.**

msgctl

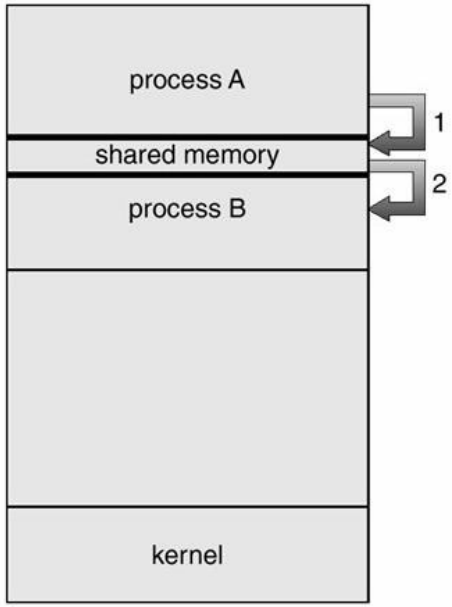


- proces može
  - zahtevati **upit za status MQ**,
  - da postavi status i
  - ukloni poruku iz MQ
- preko msgctl SC, čija je sintaksa:
- **msgctl(id, cmd, mstatbuf)**
- gde je
  - **id** message deskriptor,
  - **cmd** pecificira tip komande,
  - **mstatbuf** je adresa user data strukture koja sadrži **kontrolne parametre** ili **rezultat upita**.

- **Vratimo se na program serverski proces**,
- proces **hvata signal** i
- **poziva funkciju cleanup** da ukloni **MQ** iz sistema.

- Jedino ako primi **SIGKILL** koga ne može da uhvati, **MQ** će ostati u sistemu čak i kad **nema više poruka i procesa koji ukazuju na njega**.

Shared memory



- **Procesi mogu komunicirati direktno jedan sa drugim**
  - **deljenjem dela njihovog virtulenog adresnog prostora**,
  - a zatim **čitanjem i pisanjem u tu deljivu memoriju**.

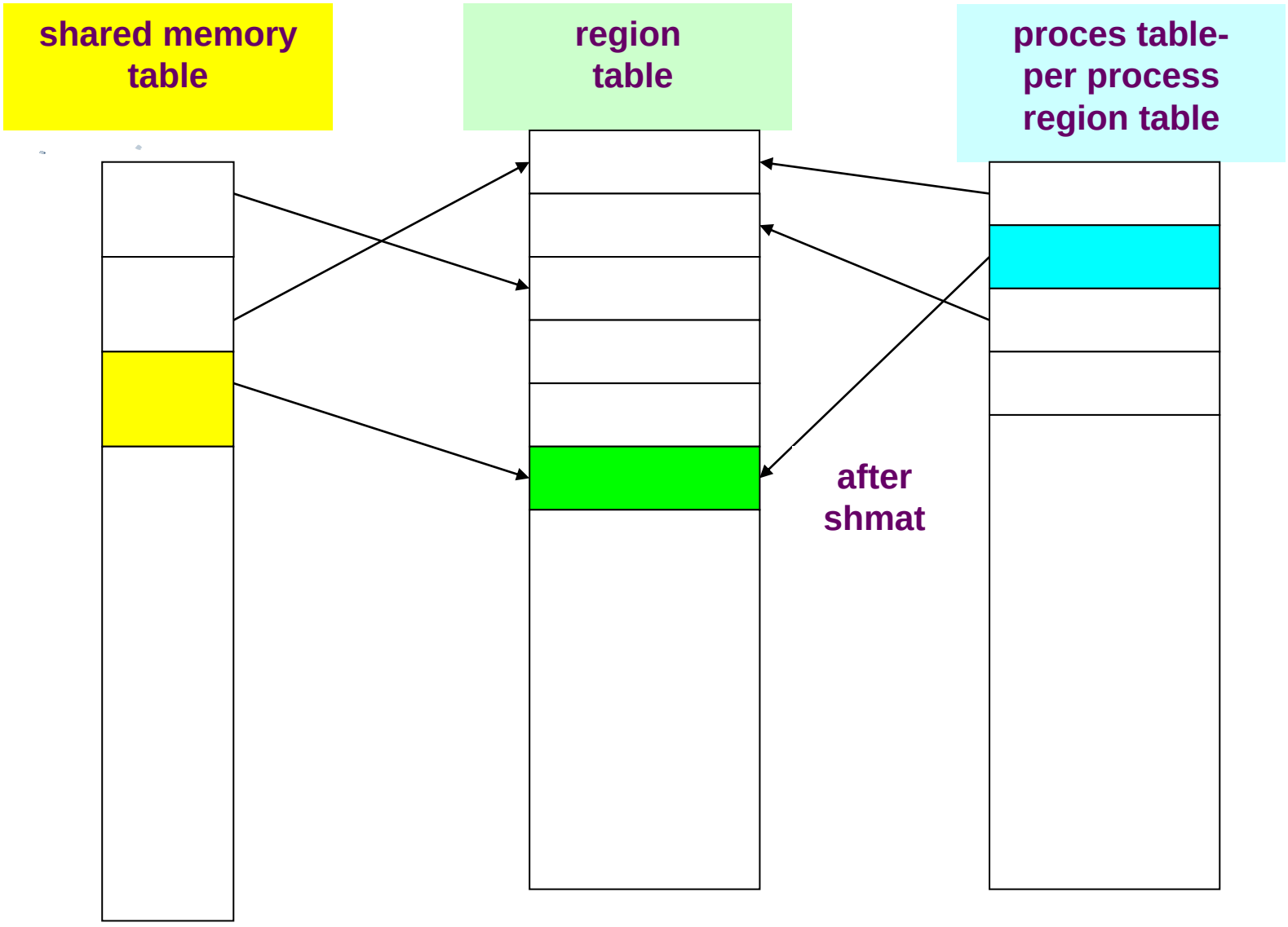
- **SC za SharedMemory** su **slični SC za messages**:
- **1. shmget:**
- **kreira novi region SM** ili **vraća jedan postojeći region SM**
- **2. shmat:**
- **logički attach-uje region** u **virtuelni adresni prostor procesa**
- **3. sgmdt:**
- **detach-uje** region iz **virtuelnog adresnog prostora procesa**
- **4. shmctl:**
- **manipuliše različitim parametrima** za **SM**

- **Procesi pristupaju SM istim memorijskim instrukcijama** kao i za **običnu memoriju**. Posle attaching-a SM regiona **ona postaje deo virtuelnog adresnog prostora procesa**, **nema posebnih SC** za **pristup SM**

shmget

- Sintaksa za shmget SC je
- **shmid = shmget(key, size, flag);**
- gde je
- **size** broj bajtova u regionu.
- Kernel pretražuje SMT (shared memory table) za **zadati key**,
- **ako nađe key** i prava su u redu, proces će **dobiti descriptor shmid**.
- Ako **ne nađe key**, a
  - **a user je setovao IPC_CREAT flag** da **kreira novi region**,
  - **kernel verifikuju** da je zadata veličina između **system-wide minimalne** i **maksimalne vrednosti**,
  - **alocira data region** preko **allocreg**.
- **Kernel**
  - čuva prava pristupa, **veličinu u pointer na ulaz** u **RT u SMT**
  - **postavlja flag** da ukazuje da **još nema memorije** koja je **dodeljena regionu**
- **Memorija se alocira (PT)** kada se **region attach-uje u proces.**

shmat

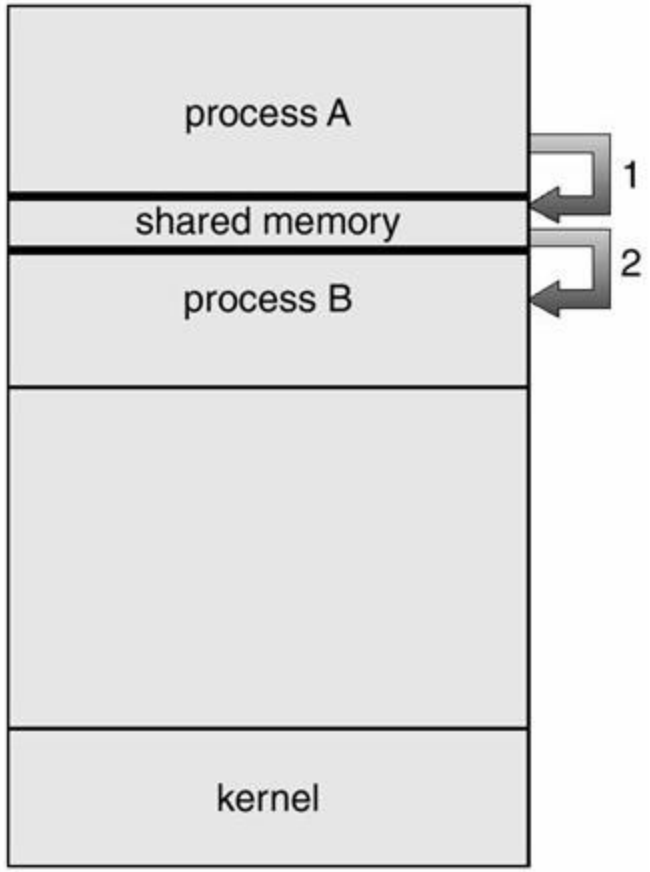


data structures for shared memory

shmget

- Kernel takođe **setuje flag u RT** ulazu
  - **da ukazuje** da se **region ne oslobađa**
  - **kada proces obavi exit.**

- Zato, **SM regioni ostaju netaknuti** čak i kad više **nema procesa vezanih za njih.**



shmat

- proces attach-uje SM region u svoj adresni prostor sa **shmat SC**:
-
- **virtaddr = shmat(id, addr, flags);**
- Parametar **id** je **vraćen od prethodnog shmget SC**,
  - **id identifikuje SM region**,
  - **addr** je **virtulena adresa** u koju user želi da attach-uje shared memoriju,
  - **flags** specificiraju da li region read-only i da li je kernel treba da zaokruži (**round-off**) user-specified adresu.
  - **virtaddr**, povratna vrednost, je virtulena adresa **gde kernel attach-uje region**, **n**e **mora da bude** = **addr**.
- Kada se izvršava **shmat SC**,
- **kernel proverava da li proces ima potrebna prava** za **pristup regionu**.
- **Potom** se **ispituje addr**, **ako je 0 -> kernel bira podesnu virtuelnu adresu.**

algorithm shmat

- **algorithm shmat** **/* attach shared memory */**
- **input:**
  - (1) shared memory descriptor
  - (2) virtual address to attach memory
  - (3) flags
- output: **virtual address where memory was attached**
- {
- **check validity** of **descriptors, permissions**;
- **if(user specified virtual address**)
- {
- **round off virtual address**, as **specified by flags**;
- **check legality** of **virtual address, size** of **region**;
- }
-

algorithm shmat

- else /* **user wants kernel to find good address***/
- {
- **kernel picks virtual address**: error if none available;
- **attach region to process address space** (algortihm **attachreg);**

- **if(region being attached** for the **first time**)
- **allocate page tables**, memory for region (alortigm **growreg**);
- return(virtual address where atached);
- }

shmat

- **SM region**
  - **ne sme da se preklapa sa drugim regionima** u procesovom virtual adresnom prostoru,
  - a **mora da se pažljivo bira** da **drugi regione ne porastu u SM region**.
- Na primer, proces može uvećati svoj **data region** sa brk SC, a novi **region se nastavlja na stari, pa se zato SM** region **ne dodeljuje iznad data regiona** ili **iznad stack regiona**. Ako stack **raste na gore**, povoljno je SM region postaviti **odmah ispod stack-a**.
- Kernel proverava da li SM region leži u adresnom prostoru procesa,
- a potom **attach-uje region** u proces.
- Ako prozvani proces **prvi attach-uje taj SM region**,
  - **kernel alocira neophodne tabele**,
  - **zatim pozove growreg**,
  - podešava polje ulaza u **SM tabeli** za "last time attached"
  - **vraća virtuelnu addresu** u kojoj je **attached region**

shmdt

- Proces **detach-uje SM region** iz adresnog prostora procesa preko
- **shmdt(addr)**
- gde je
  - **addr virtuelna adresa** koju je **prethodno vratio shmat SC**,
  - **virtulena adresa** se koristi jer **proces može imati više SM regiona u svom adresnom prostoru.**

- Kernel pretražuje procesov region attach-ovan na toj virtuelnoj adresi i skida ga sa detachreg algoritmom, a za to se **koristi SM tabela**.

- **Analizirajmo sledeći program**.
  - Proces **kreira 128K SM region**,
  - **obavi attach 2 puta u svoj space**, ali na različitim adresama.
  - Proces **upiše podatke u prvi SM region**, a čita iz **drugog.**

shmdt example: process #1

- **attaching shared memory twice to a process**
  - #include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
  - #define **SHMKEY 75**
  - #define **K** 1024
  - int **shmid;**
- main()
- {
- int i, *pint;
- char *addr1, *addr2;
- **extern char *shmat();**
- **extern cleanup();**
- **for(i=0; i<20; i++) signal(i, cleanup);**
-
- **shmid** = **shmget**(SHMKEY, 128*K, 0777 | IPC_CREAT);
-
- **addr1** = **shmat**(shmid,0,0);
- **addr2** = **shmat**(shmid,0,0);
- **printf("addr1 0x%x addr2 0x%x ", addr1, addr2);**

shmat example: process #1

```
        pint = (int *) addr1;
        for(i=0; i<256; i++) *pint++ = i;
        pint = (int *) addr1
        *pint = 256;

         pint = (int *) addr2;
        for(i=0; i<256; i++) printf("index %d value %d", i,*pint++ );
                pause();
}
cleanup()
{ shmctl(shmid, IPC_RMID,0); exit(); }
```

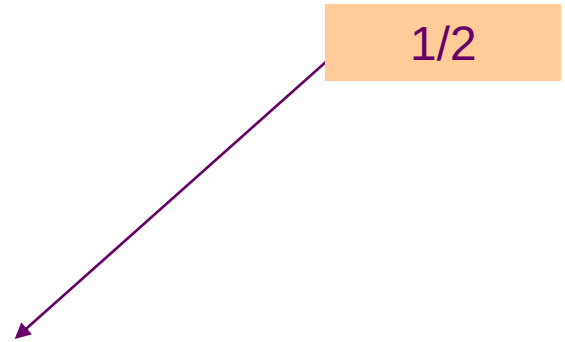
writing

reading

- **Sledeca strana**: Slika prikazuje **drugi proces** koji **attach-uje isti region** (ali uzima samo 64K, što pokazuje da proces može da da **attach-uje različite veličine regiona shared memorije).**
- **Drugi proces čeka dok prvi proces ne upiše non-zero vrednost** u **prvu reč SM regiona** i **tada čita shared memoriju**. **Prvi proces**, **pauzira** i daje šansu **drugom procesu** da se izvršava.
- Kada prvi **proces uhvati signal**, on **ukljanja SM region**

shmdt example: process #2

- **sharing memory between processes**
  - #include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
  - #define SHMKEY 75
  - #define K   1024
  - int shmid;




```
main()
{
int i, *pint;
char *addr;
extern char *shmat();
char  mtext[256];
        shmid = shmget(SHMKEY, 64*K, 0777);
        addr = shmat(shmid,0,0);

        pint = (int *) addr;
        while (*pint == 0)  ;
        for(i=0; i<256; i++) printf("%d", *pint++ );

}
```

shmctl

- Proces koristi shmctl SC, da traži **upit statusa**, i postavi parametre za SM region:
- **shmctl(id, cmd, shmstatbuf);** gde je
  - **id identifikuje** ulaz u SM tabelu,
  - **cmd specificira tip komande**,
  - **shmstatbuf** je **adresa user data strukture** koja sadrži statusne informacije ulaza u SM tabeli kada se pita ili postavlja status za taj SM region.
- Kernel tretira komande za upit statusa i promenu vlasništva **slično kao kod messages**.
- **Kada se ukljanja SM region,**
  - **kernel oslobađa ulaz i analizira RT ulaz**,
  - **ako nema drugih procesa** sa attach-ovanim tim SM regionom,
  - **oslobađa se RT ulaz** sa svim resursima, preko algoritma **freereg**.
  - Ako neki proces ima attached SM region (RC!=0), **kernel briše flag** koji ukazuje da **region ne treba da se oslobodi kada ga zadnji proces otkači**.
  - Procesi **koji još uvek koriste region, mogu to da rade**, ali drugi procesi ne mogu više da ga attach-uju (prvi proces je tražio njegovo brisanje).
  - Tek kada **svi procesi otkače region**, on se oslobađa-frees (**slično kao open pa unlink, odmah).**

Semaphores

- **SC semafor omogućava procesu** da **sinhronišu izvršavanje pomoću skupa atomskih operacija** na **skupu semafora**.
- **Pre implementacije semafora**, **proces je radio** sa **lock file**, treba da kreira **lock file sa creat SC**, ako želi da **lock-uje resurs**, a create će otkazati ako je datoteka već postoji i **proces će pretpostaviti** da je drugi **proces lock-ovao resurs**.
- **Glavni nedostatak** ove šeme je što proces ne zna **kada da pokuša ponovo**, a **lock datoteke mogu da ostanu u sistemu posle system crash-a**.

- **Semafor ima dve atomske operacije P i V.**
- **P (wait)** operacija dekrementira vrednost semafora ako je njegova vrednost veća od 0, a
- **V (signal)** operacija **inkrementira vrednost semafora**.
- **Kako su P i V atomske**, **samo jedna operacija može da se obavlja** u **jednom trenutku**.
- **P i V su atomske za kernel**, **nijedan drugi proces ne može ništa da radi sa semaforom dok drugi ne obavi operaciju**,
- a ako **ne može da je obavi**, proces spava.

Semaphores

- Semafor na **UNIX System V sastoji** se **od sledećih elemenata:**
  - **vrednost semafora**
  - **PID zadnjeg procesa** koji **manipuliše sa semaforom**
  - **broj procesa koji čekaju** da se **poveća vrednost semafora**
  - **broj procesa koji čekaju** da **vrednost semafora bude =0**

- System Calls za semafore su:
- **semget: za kreiranje i dobijanje pristupa skupu semafora**

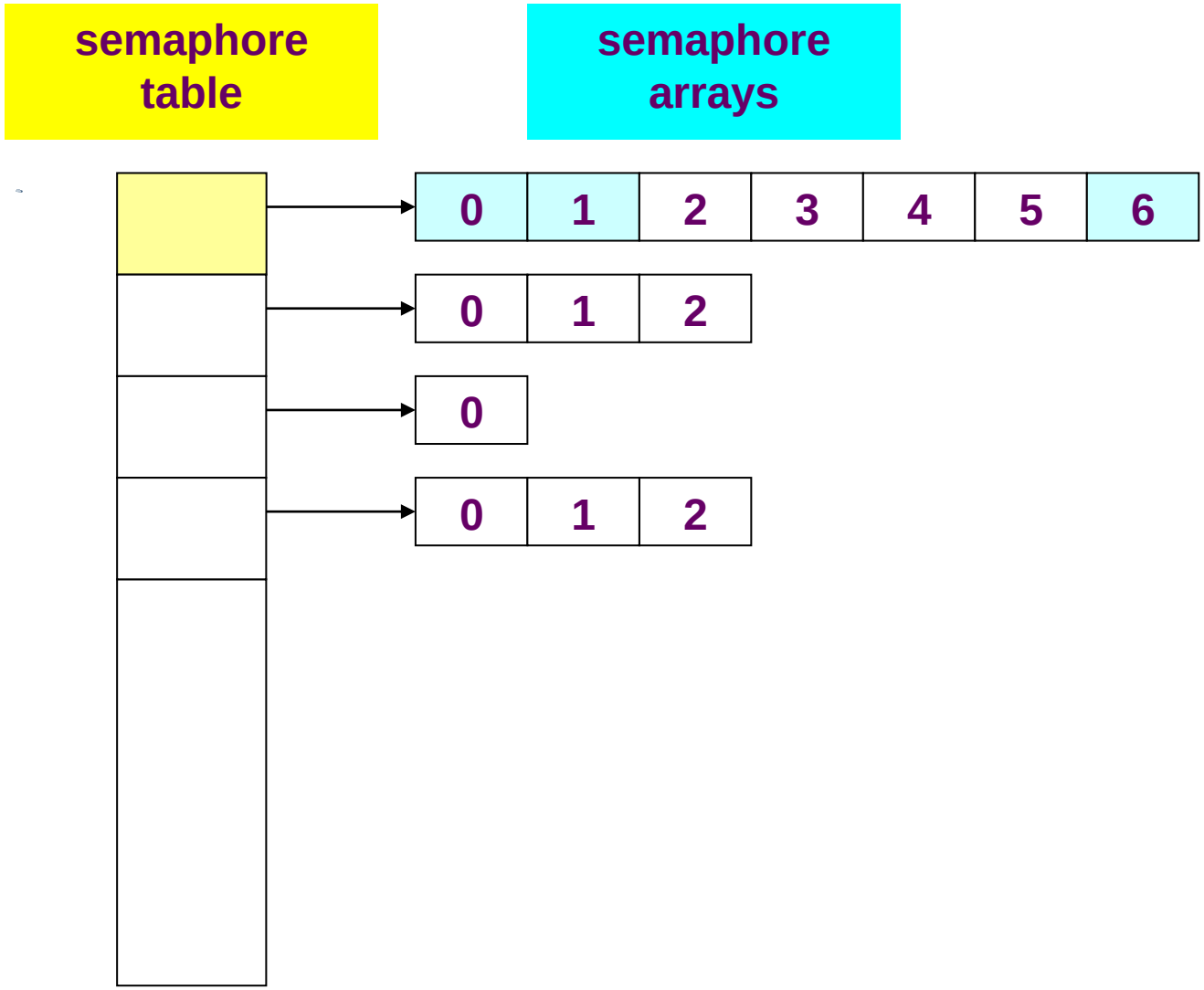
- **semctl:** obavlja **različite kontrolne operacije** nad semaforima

- **semop: manipuliše vrednostima** semafora

segmet

- **SC semget kreira polje** semafora:
- **id=semget(key, count, flag);**
- gde su
- **key**, **count** i **flag** kao kod **shared memorije**.
- **Kernel alocira ulaz** koji **ukazuje na polje semaforskih struktura** sa **count elementima**, kao na slici.

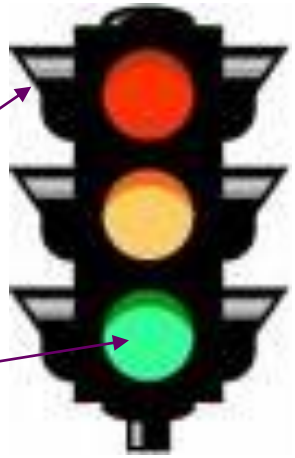
semaphores



data structures for semaphores

Semaphore table entry

- Ulaz u ST specificira, takođe
  - **broj semafora** u **polju**,
  - **vreme poslednjeg semop SC**,
  - **vreme poslednjeg semctl SC**.
- Proces manipuliše semaforom preko **semop SC**.
- **oldval = semop(id, oplist, count);**
- gde je
  - **id** deskriptor koji je vratio **semget**,
  - **oplist** je pointer na polje semaforskih operacija, a
  - **count** je **veličina polja**.
  - Povratna vrednost **oldval** je **vrednost poslednjeg semafora** nad kojim je operisano.



- **Format** za svaki element **opliste** je:
  - **broj semafora** koji identifikuje ulaz polja semafora nad kojim će se izvoditi operacija
  - **operacija**
  - **flags**
- Kernel čita polje semaforskih operacija=oplist, iz user adresnog prostora i verifikuje da li su semaforski **brojevi legalni** i **da li proces ima pravo da čita** ili da **menja semafor**.

semop (positive)

- **algorithm semop /*semaphore operations*/**
- input:
  - **(1) semaphor descriptor**
  - **(2) array of semaphor operation**
  - **(3) number of elements in array**
- **output: start value of last semphore operated on**
- {
- check legality of semaphore descriptor;
- **start:**
- **read array of semaphore operations from user to kernel space**;
- **check permissions for all semaphore operations**;
-
- **for(each semaphore operation in array**)
- {
- if(**semaphore operation is positive**)
- {
- **add "operation"** to semaphore value;
- **if(UNDO flag set** on **semaphore operation**) **update** process **undo structure**;
- **wakeup** all processes sleeping (**event semaphore value increase**);
- }

semop (negative)

- else if (**semaphore operation is negative**)
- {
- if("**operation" + semaphore value >= 0**)
- {
- **add "operation" to semaphore value**;
- if(UNDO flag set on semaphore operation) **update process undo structure**;

- if (**semaphore value 0**)
- **wakeup all processes sleeping** (event semaphore value **becomes 0;**
- continue;
- }
- **/*result is still negative*/**
- **reverse all semaphore operations already done this SC (previous** iterations);
- **if(flags specify not to sleep) return with error**;
- **sleep (event semaphore value increase)**;
- **goto start**; /* loop from begginning*/
- }
-

semop (zero)

- else **/* semaphore operation is zero */**
- {
- **if(semaphore value non 0)**
- {
- **reverse all semaphore oeprations already done this SC;**
- **if(flags specify not to sleep) return with error;**
- **sleep (event semaphore value == 0);**
- goto start; /* loop from begginning*/
- }
- }
- }/* for loop ends here*
- **/* semaphore operations all succeeded*/**
- **update timestamps, PID;**
- **return value of last semaphore operated on before call succeeded**;
- }

description

- **Ako kernel mora da ide na spavanje dok radi listu semaforskih operacija**, **on vraća vrednosti kao na početku SC**, pa **ide na spavanje** dok se **ne desi događaj**, pa restartuje SC. Zato što kernel čuva semaforske operacije u globalnom polju, on odatle čita ako mora da restartuje SC. Operacije su atomske, ili sve ili ništa.
- **Kernel menja vrednosti semafora u skladu sa vrednošću operacije**:
- -**ako je +**, to **inkremntira semafor** i **budi sve procese koji čekaju da vrednost semafora poraste.**
- -**ako je operacija 0**, **kernel proverava vrednost semafora**,
  - pa ako je 0, nastavlja sa drugim operacijama u polju,
  - u protivnom **inkrementira broj procesa** koji spavaju čekajući da semafor postane 0 i **ide na spavanje**.
- -**ako je -**,
  - **|operation| + semaphore value >= 0** kernel dodaje vrednost operacije-negativnu na vrednost semafora.
  - **Ako je rezultat 0**, kernel budi sve procese koji čekaju da rezultat postane 0.
  - **|operation| + semaphore value < 0**, kernel gura proces na spavanje, a događaj koji ga budi je kada se inkremntira vrednost semafora.
- Kada proces ode na spavanje u semafor operaciji, budi se na signal.

example

- Analizirajmo program za
- " **locking and unlocking semaphore operations**" i
- prepostavimo da ga nazovemo a.out i

- izvršimo a **3 puta na sledeći način:**
- **a.out &**
- **a.out a &**
- **a.out b &**

- **semget SC** u sledećem programu će kreirati semafor sa 2 elementa.

example

- **locking and unlocking semaphore operations**
- #include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
- #define **SEMKEY 75**
- int semid;
- unsigned int count;
- **/*definition of sembuf in file sys/sem.h**
- ***struct sembuf {**
- * **unsigned short sem_num;**
- * **short sem_op;**
- * **short sem_flg;**
- }*/
- **struct sembuf psembuf, vsembuf; /* ops for P and V*/**
- **main(argc, argv)**
- int argc;
- char *argv;
- char *argv[];
- {
- int i, **first, second;**
- **short initarray[2], outarray[2];**
- **extern cleanup();**

example

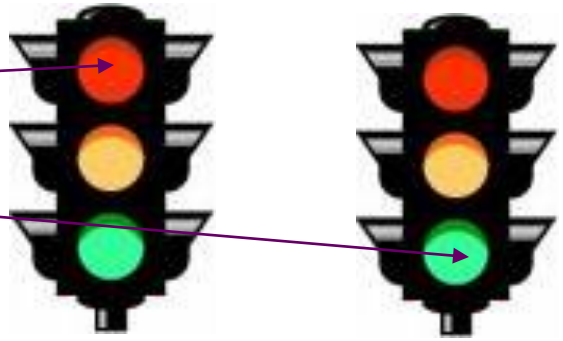
```
        if(argc == 1)
{
        for(i=0; i<20; i++) signal(i, cleanup);

        semid = semget(SEMKEY, 2, 0777| IPC_CREAT);

        initarray[0]= initarray[1]=1;
        semctl(semid, 2, SETALL, initarray);
        semctl(semid, 2, GETALL, outarray);
        printf("sem init vals %d %d", outarray[0], outarray[1]);

         pause(); /* sleep until awakned by signal*/
}
```

example

- else if (**argv[1][0] == a)**
- {
- **first =0;**
- **second=1**;
- }
- else **/*b*/**
- {
- **first =1;**
- **second=0;**
- }
-

example

- **semid = semget(SEMKEY, 2, 0777);**
- **psembuf.sem_op= -1;**
- psembuf.sem_flg= SEM_UNDO;
- **vsembuf.sem_op= 1;**
- vsembuf.sem_flg= SEM_UNDO;
-
- **for(count =0; ; count ++)** {
- psembuf.sem_num= first;
- **semop(semid, &psembuf,1);** **/*wait for first, #0 for a, #1 for b*/**
- psembuf.sem_num = second;
- **semop(semid, &psembuf,1);** **/*wait for second, #1 for a, #0 for b*/**
- **printf("proc %d count %d", getpid(), count);**
- vsembuf.sem_num= second; **/*signal for second, #1 for a, #0 for b*/**
- **semop(semid, &vsembuf,1);**
- vsembuf.sem_num= first;
- **semop(semid, &vsembuf,1); }** /*for*//**signal for first, #0 for a, #1 for b*/**
- }
- cleanup()
- { semctl(semid, IPC_RMID,0); exit(); }



description

- U **prvom slučaju** program se **pozove bez argumenata**, **proces kreira semaforski set sa 2 ulaza i inicijalizuje njihove vrednosti na 1.** Zatim **proces pauzira i spava, sve dok ga ne probudi signal, kada preko cleanup() briše semaforski set.**
- **Kada se program izvršava sa argumentom a**
  - **proces (A) radi 4 semaforske operacije u petlji,**
  - **wait #0, wait #1, signal #1, signal #0**
  - **dekremetira** se vrednost semafora **0, dekremetira se vrednost semafora 1**, izvršava se print naredba, a zatim se **inkrementira semafor 1,** pa semafor **0**. Proces ide na spavanje ako pokušava da dekrementira vrednost semafora koja je 0, tako da se semafor tada smatra da je **zaključan.**
  - **Kako su semafori postavljeni na 1**, i nema drugih procesa koji koriste semafor, proces A neće spavati nijedanput, i vrednost semafora će **oscilovati, između 0 i 1.**
- Kada se **program izvršava sa argumentom b,**
  - **proces (B) radi 4 semaforske operacije u petlji**,
  - **wait #1, wait#0, signal #1, signal #0**
  - dekremetira se vrednost semafora **0 i 1** u kontra poretku od procesa A.
- **Međutim ako A i B rade simultano**, situacija može da bude sledeća:
  - **proces A lockuje semafor 0**,
  - a **hoće da lock-uje semafor 1**,
  - ali je **proces B već lock-ovao semafor 1,**
  - a hoće da **lock-uje 0** koji je već locked.
- Oba procesa idu na spavanje i nikada se ne bude, **deadlock**, sve dok proces ne primi signal.

deadlock avoidance

- Da bi se **izbegavali ovakvi problemi**, procesi treba da **koriste sledeće strukture** i kod (za ovaj primer):
- **struct sembuf psembuf[2]; /* ops for P and V*/**
- **psembuf[0].sem_num = 0;**
- **psembuf[1].sem_num = 1;**
- **psembuf[0].sem_op = -1;**
- **psembuf[1].sem_op = -1;**

- **semop(semid, psembuf,2);**

- **psembuf** je polje semaforskih operacija **koje dekrementiraju semafore** 0 i 1 **istovremeno**.
- Ako bilo koja operacija **ne može da se izvrši**, proces spava sve dok se obe ne obave. Na primer, ako je vrednost semafora #0 = 1, a vrednost semafora #1 = 0, kernel će ostaviti ove vrednosti sve dok se ne dekrementiraju **obe vrednosti**.
- Proces može setovati **IPC_NOWAIT** flag, a ako **proces mora da čeka** na semaforu, trebalo bi da ide na spavanje, kernel će da se vrati iz SC sa error poruku. Zato je moguće implementirati uslovni semafor, gde proces **ne ide na spavanje ako ne može da dobije atomsku akciju**.

SEM_UNDO

- **Opasne siutacije** mogu da se **dogode**,
- ako proces obavi semaforsku operaciju,
- na primer lock-uje semafor (wait),
- i **obavi exit** a ne resetuju semaforsku vrednost (signal).

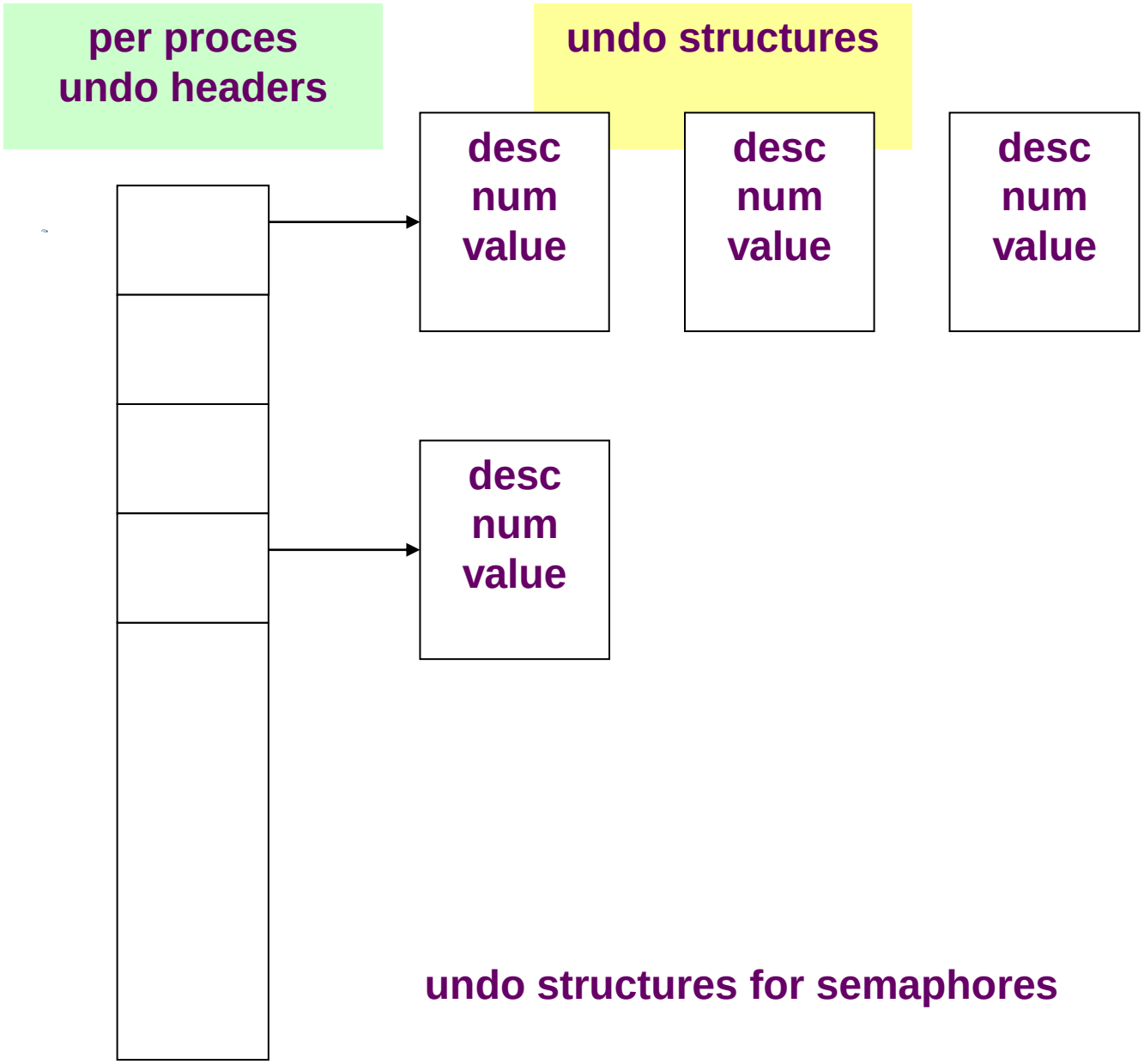
- Takve situacije su ili
  - programerska greška,
  - ili je signal ubio proces pre vremena.

- Ako u programu za "**locking and unlocking semaphore operations**", **proces dobije kill signal nakon što dekrementira semaforske vrednosti**, nema šanse da ih **inkrementira ponovo** jer kill signal ne može da se uhvati, a semafori ostaju locked.

- Da bi **se izbegavale takve situacije**, **procesi mogu** da **setuju SEM_UNDO flag** u **semop SC**, a kada se **okonča SC**, kernel **reverzira efekte** svake semaforske operacije koje je proces obavio.

SEM_UNDO

- Da bi to mogao da radi,
- **kernel ima tabelu** za svaki proces u sistemu,
- gde svaku ulaz ukazuje na **undo strukturu** i
- **svaki semafor** koji koristi proces **ima svoju undo strukturu**,
- koja je **polje tripleta** koga **čine:**
  - **semafor ID**
  - **semaforsku broj u skupu**
  - **vrednost**



undo structures for semaphores

sem_undo

- **Kernel alocira undo strukturu dinamički**,
- **kada proces izvršava prvi semop SC** sa **SEM_UNDO flagom**.

- **Na sledeći semop SC** sa **SEM_UNDO flagom**,
  - kernel pretražuje tabelu za undo strukturu za **semafor sa tim ID**
  - tada oduzima semaforsku operaciju od vrednosti u tabeli,
  - tako da **undo struktura sadrži negativan zbir svih semaforskih operacija**, koji su obavljene sa **SEM_UNDO**.

- Ako **undo-struktura ne postoji**, **kernel** je **kreira**, a kada **vrednost u undo-strukturi padne na 0**, kernel ukljanja tu undo-strukturu.

- Kada proces obavi **exit,** kernel poziva **specijalnu strukturu koja obavlja undo akciju na semaforu.**

SEM_UNDO

- Vratimo se opet na program za "**locking and unlocking semaphore operations**".
- Ako kernel kreira undo strukturu, svaku put kad proces dekrementira semaforsku vrednost,
- a ukljanja strukturu svaki put kad proces inkrementira semaforsku vrednost, zato što je tada vrednost u undo strukturi =0.

- Slika prikazuje izgled undo strukture za proces A.
- **wait #0 wait #1, signal #1, signal #0**
- **Posle prve operacije**, proces ima **triplet za semid =0 i vrednost =1.**
- Posle druge operacije, uvodi se novi triplet za semafor 1, sa vrednošću 1. Ako bi proces u tom trenutku iznenada završio, krenel bi dodao svakom semaforu vrednost 1 i tako im vratio vrednosti na 1.
- **U regularnom slučaju,**
  - krenel dekrementira podešenu vrednost za semafor 1 za vreme treće operacije, zato što inkremnetira semafor i taj triplet postaje 0 pa se ukljanja,
  - a isto se dešava sa tripletom za semafor 0 posle 4-te operacije.

- Polje operacija na semaforu, omogućava da se izbegava deadlock, ali undo šema je komlikovana. Problem deadlock-a može da se rešava na user-level.

semctl

- SC semctl sadrži brojne operacije za semafor:
- **semctl(id, number, cmd, arg)**

- arg se deklariše kao unija:
- **union semunion**
- {
- int val;
- struct semid_ds *semstat;
- unsigned short *array;
- }arg;

- Kernel interpretira **arg** **na bazi** **cmd**, kao kod SC(ioctl).
- Očekivane akcije su:
  - **setovanje ili čitanje kontrolnih parametara,**
  - **setovanje ili čitanje semaforskih vrednosti.**

- Na primer za **remove komandu**, IPC_RMID,
  - **kernel nalazi sve** procese koji imaju **undo strukturu** za semafore i
  - **ukljanjaju njihove triplete**,
  - zatim se reinicijalizuje semaforska data struktura i
  - **bude se svi uspavani** procesi koji **čekaju na neki događaj na semaforu**,
  - **koji kada nadju da semafor ID više ne postoji vraćaju error.**

General comments

- Postoje **sličnosti između FS i IPC mehanizama.**
  - **SC get (IPC) je sličan open SC (FS),**
  - a **"control" SC** koji ukljanjaju deskriptore iz sistema su slični kao **unlink SC**.
  - Jedino **close SC nema sličnu operaciju u IPC**, jer kernel ne vodi zapis o svim procesima koji koriste IPC.

- Proces može korisiti **IPC bez get SC**, ako **pogodi ID i ima dovoljno prava**. Kernel ne može obrisati nekorišćene IPC strukture automatski, zato što ne zna da li su i dalje potrebne. **Pogrešan proces može ostaviti nepotrebne IPC strukture**. Kernel može čuvati IPC stanje posle exit-a procesa u svojim strukturama, ali je bolje da to obavi u log datoteku.

- **IPC uvodi ključeve**, umesto datoteka, ali teško je proširiti takav IPC na **mrežu**, zato što treba da se **opišu različiti objekti na različitim mašinama**, tako da ovaj IPC važi za single mašinu.

- Korišćenjem **keys umesto filenames ubrzava se IPC**.

Network communications

- **Programi** kao što su **mail, ftp, remote-login**, koji hoće da komuniciraju sa drugim mašinama imaju **svoje ad-hoc** metode za uspostavljanje **konekcije i za razmenu podataka**. Na primer, **mail program čuva mail** u datoteci na /usr/mail/mjb za usera mjb. Kada user šalje mail na istu mašinu, mail program dodaje poruku na adresiranu datoteku koju drugi user otvara i čita. Da bi se poslao mail na **drugu mašinu**, mail program mora da pronađe odgovarajući mail datoteku na drugoj mašini. Kako **ne može direktno da pristupa, nego preko mreže**, postavljaju se **2 agenta na obe mašine** koji će im uspostaviti **komunikaciju.Lokalni proces** se zove **klijent** za **remote serverski proces**.

- Zato što UNIX kreira nove procese preko fork SC, **serverski proces mora postojati pre nego što clijentski proces uspostavi konekciju**. Bilo bi loše da remote-kernel kreira novi proces kada se po mreži pojavi zahtev.
- **Mnogo je bolje da serverski proces postoji unapred** i da se proverava da li ima **zahteva po mreži**. To obično radi **init proces**, koji **kreira serverski proces** koji čita **komunikacioni kanal** sve dok ne dobije neki **zahtev za servis** i **iza toga** se **poštuje neki protokol**.
- Klijentski i serverski programi tipično biraju mrežni uređaj i protokole.

Network communications

- Na primer, **uucp program** omogućava **ftp po mreži i rex** (remote execution of comands). **Klijent proces daje upit u bazi svog servera** za adesu i routing informacije (br. telefona), **otvara modem uređaj** (open SC), **radi write SC ili ioctl na otvorenom uređaju** i poziva udaljenu mašinu. Na udaljenoj mašini, ini će **izvršiti getty** na **telefonskoj liniji** kad se uspostavi **hardverska veza između modema**, pa se klijentski proces loguje, a dobija specijalan **shell uucico**.

- **Mrežne komunikacije izazivaju problem za UNIX** zato što poruke često uključuju i delova sa **podacima i kontrolne delove**. Kontrolne informacije obično imaju **adresu koj opisuje destinaciju poruke**, a **format adrese zavisi** od **protokola**, tako da bi proces morao da zna sve detalje protokola, a to ugrožava UNIX prinicp o nezaviosti tipa datoeteje-uređaja.

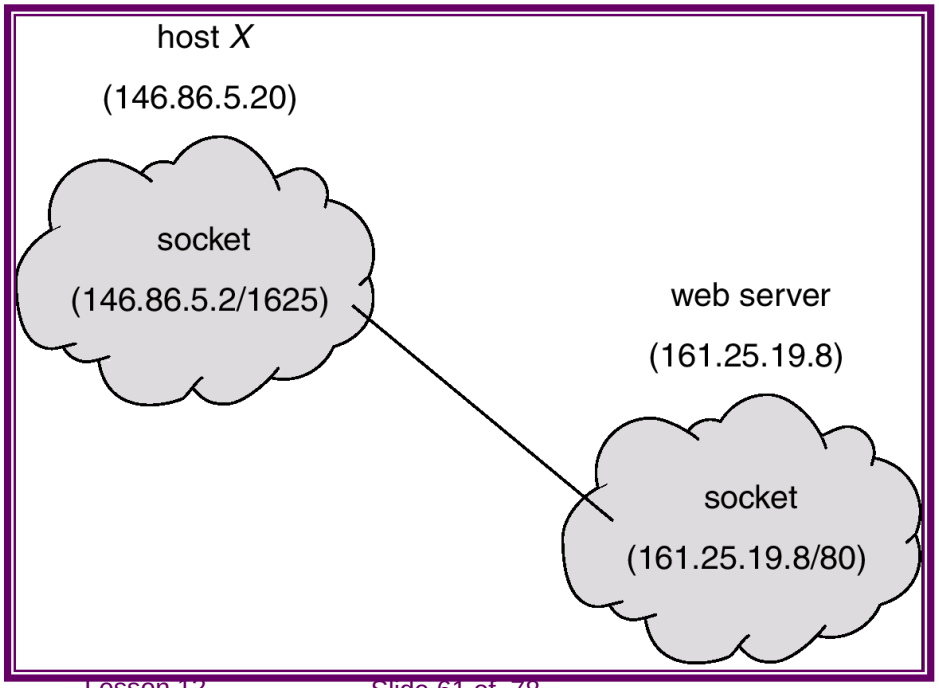
- **UNIX je stalno poboljšavao svoju mrežnu funckiju**, **streams** su doneli **elegantan metod za mrežnu podršku** zato što **protokol moduli** mogu **fleksibilno da se dodaju** u **stream**.
- Sada ćemo opisati BDS socket

Sockets

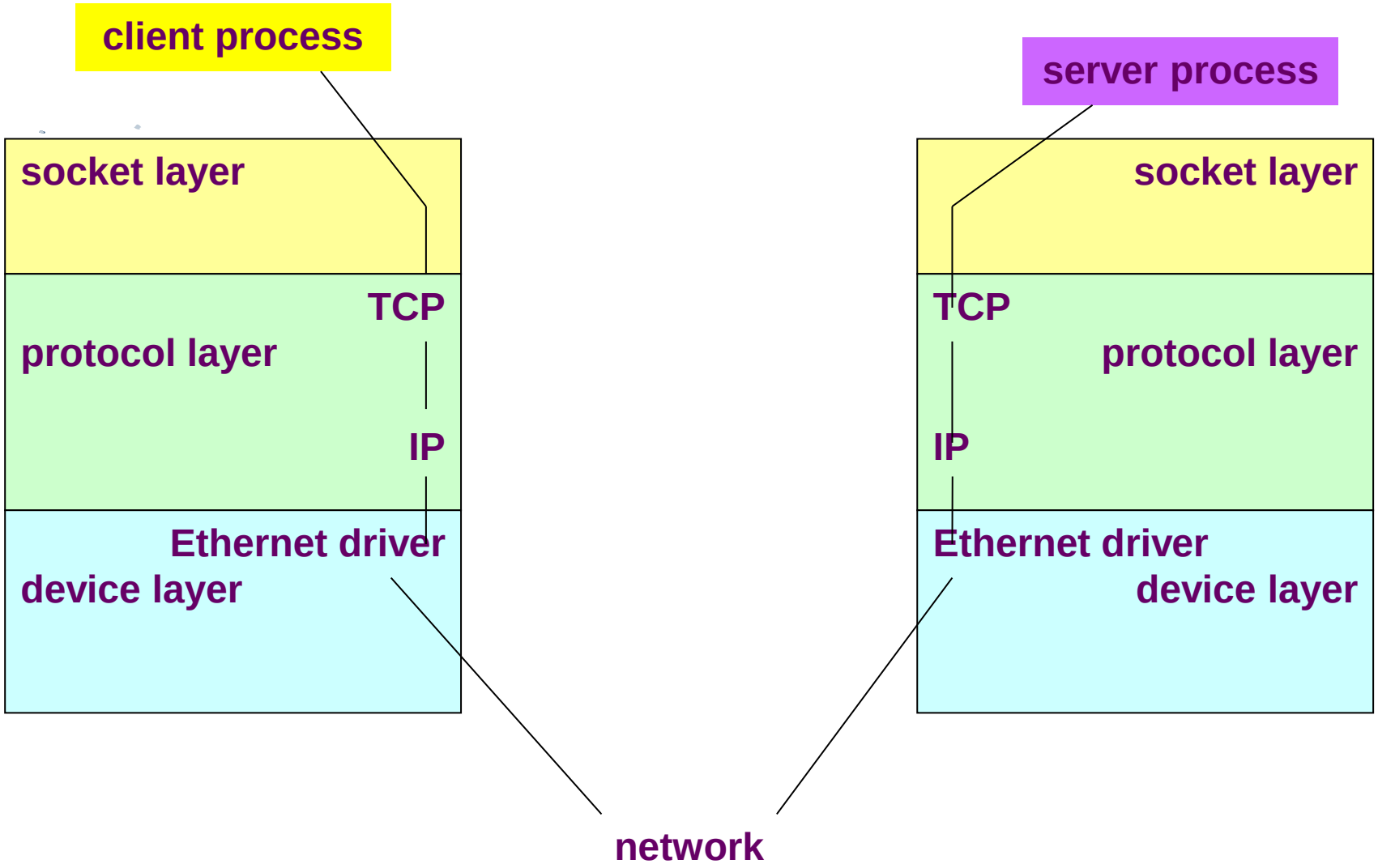
- **Procesi mogu komunicirati** po **mreži**,
- ali **način komunikacije zavisi** od **protokola i mrežnih uređaja**.

- **Da bi se obezbedila IPC preko raznih mrežnih protokola**,
- **BSD je obezbedio mehanizam** koji se **naziva socket**.

- Ovde ćemo objasniti **user-level aspekt socket mehanizma**.



socket



sockets model

Socket description

- **Kernelska socket struktura** sastoji se od **3 dela:**
  - **socket layer**
  - **protocol layer**
  - **device layer**

- **socket layer** obezbeđuje **interfejs između SC i nižih slojeva**,
- **protokol layer** sadrži **protokolske module korišćene u komunikaciji** (TCP, IP),
- **layer uređaja sadrži device drajvere** za **mrežne uređaje**.
- Legalna kombinacija protokola i drajvera se specificira prilikom konfiguracije sistema.
- **Procesi komuniciraju** koristeći **client-server model**,
- **server proces osluškuje** svoj socket (**listen**), koji je jedan od krajeva **two-komunikacione putanje.**
- **Klijent** proces komunicira sa **serverskim procesom** preko svog socket-a, koji je drugi kraj komunikacione putanje.
- **Kernel održava interne konekcije i rutira podatke** od **klijenta** do **servera.**

Socket description

- **Socket-i koji dele zajedničke komunikacione osobine** (naming, protocols address format) se **grupišu u domene**.

- **BSD uvodi 2 socket domena**:
  - **UNIX domen za procese koji komuniciraju** na **jednoj mašini**
  - **Internet domen** za procese **koji komuniciraju na mreži** koristeći DARPA komunikacione protokole.

- **Svaki socket** ima **tip-default protokol**
  - **datagram (udp)**
  - **virtual circuit. (tcp)**
- **VC dozvoljava, sekvenciranje, i pouzdan prenos podataka.**
- **Datagrami nemaju sekvenciranje, ni pouzdan** prenos, ali su jednostavniji za realizaciju od VC, zato što **nemaju skupe setup operacije**. Datagrami su podesni za **neke vrste komunikacija**.

- **Sistem ima default protokol**, za svaki domen-socket tip. Na primer TCP je VC, dok je UDP datagram service u Internet domenu.

Socket system calls

- **Socket mehanizam sadrži više System Calls**.

- SC socket uspostavlja end-point krajnu tačku komunikacionog linka:
- **sd=socket(format, type, protocol);**
  - **format** specificira komunikacioni domen (UNIX domain or Internet),
  - **type** ukazuje na tip komunikacije (VC or datagram) i
  - **protokol** opisujuje partikularni protokol koji kontroliše komunikaciju.

- Procesi koriste **socket deskriptor sd** u **drugim SC**.

- SC **close zatvara socket**.

- **SC bind udružuje ime** sa **socket deskriptorom**:
- **bind(sd, address, length);**
- gde je
  - **sd socket deskriptor**,
  - **address** ukazuje na **strukturu** koja specificira identifikatore za komunikacioni domen i protokol definisan u socket SC.
  - **length** je **dužina adresne strukture**, to je ime datoteke u UNIX domenu.

- **Server procesira bind adrese** u **socketima** i
- **postavlja njihova imena** da bi **identifikovao sebe kod klijentskih procesa**.

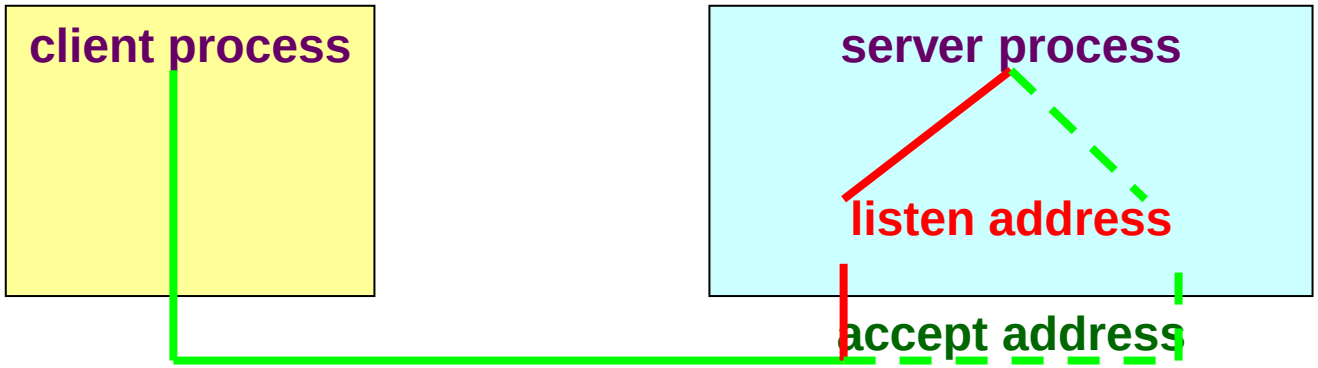
Socket system calls

- **SC connect omogućava kernelu** da **obavi konekciju u postojeći socket:**
- **connect(sd, address, length);**
- a **semantika je ista** kao za **bind SC**, ali je **adresa = adresa target socketa**, koji formira drugi kraj komunikacione linije.
- **Oba socketa moraju imati isti komunikacioni domen i protokol**, a kernel uređuje da se komunikacioni link postavi korektno. Ako je **tip = datagram**, connect SC informiše kernel o adresi koja će se koristiti za sledeći send SC preko socketa.
- **listen SC**.
- **listen SC specificira maximalnu veličinu queue**:
- **Kada server proces uređuje prijem konekcije preko VC**,
- kernel mora **ubaciti dolazeće zahteve u queue** pre nego što ih servisira.
-
- **listen(sd, qlength);**
- gde je sd socket deskriptor, a
- **qlength je maksimalni broj zahteva koji čekaju na obradu**.

Socket system calls

- **accept SC prima dolazeće zahteve** za **konekciju sa serverskim procesom:**
-
- **nsfd=accept(sd, address, addrlen);**
- gde je
  - **sd socket deskriptor**,
  - **address** ukazuje na user data polje koje kernel puni sa povratnim adresama **klijenata koji se povezuju**,
  - **addrlen** ukazuje na **dužinu address**.
- **Kada accept završi**, kernel prepisuje sadržinu addrlen sa brojem koji ukazuje na količinu prostora koju je uzelo polje address.
- **Accept vraća novi socket deskriptor**, **različit** od **socket deskrtiptora sd**.
- **Server može nastaviti da sluša na socketu**,
- **dok komunicira** sa **klijentskim procesom preko odvojenog komunikacionog kanala**, kao na slici:

accept SC



server accepting a call

Socket system calls

- **SC send i recv prenose podatke preko** konektovanog socket-a:
- **count = send(sd, msg, length, flags);**
- gde je
  - **sd socket** deskriptor,
  - **msg** je ukazivač na **poruku koja će biti poslata**,
  - **length je dužina poruke**,
  - **count** je broj bajtova koji je zaista poslat.
  - **flags** parametar može biti setovan na vrednost **SOF_OOB** da šalje podatke na **"out-of-band" način**,
  - koji znači da **data koji se šalju nisu deo regularne sekvence podataka** koja se šalje između procesa.
  - Na primer, za remote login, **<delete>** katakter se **šalje sa ovim flagom**.

Socket system calls

- Sintaksa za **recv SC** je
- **count = recv(sd, buf, length, flags);**
- gde je:
- **sd socket deskriptor**,
- **buf** je data polje za **dolazeće podatke,**
- **length je očekivana dužina poruke,**
- **count** je broj bajtova **koji je zaista primljen**.
- flags mogu biti setovani da **pokupe dolazeću poruku** i ispitaju joj sadžaj bez **uklanjanja iz queue**, ili na bazi "**out of data**" kao malopre.

- **Datagram** opcije za **recv SC**, su **sendto i recvfrom**, imaju dodatne parametre za adresiranje.

- **Procesi mogu da koriste** i **read i write SC** na **otvorenim socketima** umesto **send i recv**, a posle uspostave konekcije, **kao da su socket-i regularne datotke.**

Socket system calls

- **SC shutdown zatvara socket konekciju**:

- **shutdown(sd, mode);**

- gde **mode ukazuje da li se:**
  - sending kraju
  - receiving kraju
  - ili na oba kraja
- **više ne dozvoljava prenos podataka**.

- Ovaj **SC infomiše protokole da zatvore mrežnu komunikaciju**, ali **sd ostao netaknut**, **close SC će osloboditi socket deskriptor**

Socket system calls

- **SC getsockname uzima ime socketa** koji je **vezano preko prethodnog bind SC**
- **getsockname(sd, name, length);**

- **SC getsockopt** i **setsockopt**:
  - vraćaju ili postavljaju različite **opcije za socekt**,
  - u saglasnosti sa **domenom i protokolom**.

server process example

- Analizirajmo **sledeći program koji predstavlja serverski proces**.
- **proces kreira stream socket** u **UNIX sistem domenu** i **vezuje mu** ime **sockname** na njega.
- Zatim se pozove **listen SC** da **specificira interni queue** za dolazeće poruke i **ulazi u petlju**, čekajući dolazeće zahteve.
- **SC accept spava** sve dok **protokol ne objavi da je zahtev za konekciju upućen ka tom socketu.**
- Zatim, **accept SC vraća novi deskriptor za dolazeće zahteve**.
- **Serverski proces fork-uje proces da komunicira sa klijentskim procesom**: **roditelj i dete zatvaraju njihove deskriptore** tako da ne učestvuju u **komunikacionom saobraćaju drugog procesa**.
- **Proces dete obavlja svoju konverzaciju** sa **klijentskim procesom**, **završavajući posle povratka** iz **SC(read).**
- **Server proces ostaje u petlji** i **čeka na novi zahtev** za **konekciju** u **accept SC**.

server process example

- **server proces in the UNIX System Domain**
- #include <sys/types.h>
- #include <sys/socket.h>
- main()
- {
- int sd,ns;
- char buf[256];
- struct sockaddr sockaddr;
- int formlen;

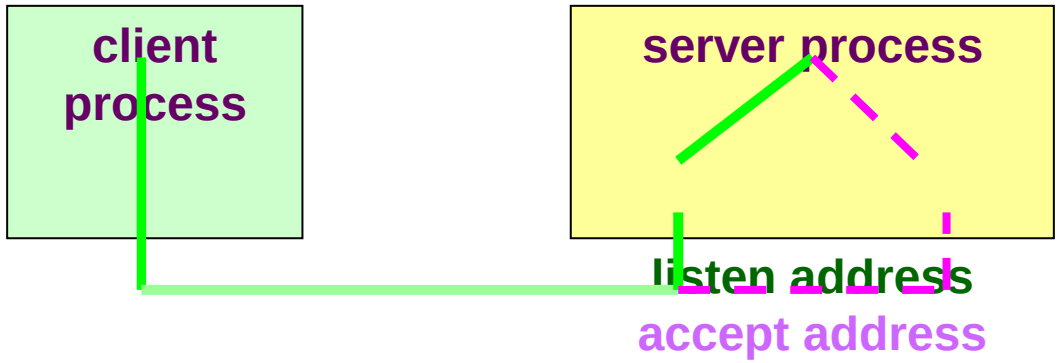
- **sd=socket(AF_UNIX, SOCK_STREAM, 0);**

- /***bind name-** null char is not part of name*/
- **bind**(**sd, "sockname",** sizeof("sockname"-1);

- **listen**(sd,1);
-

server process example

```
        for(;;)
        {
        ns=accept(sd, &sockaddr, &fromlen);
        if(fork() == 0)
                {
                /*child*/
                close(sd);
                read(ns, buf, sizeof(buf);
                printf("server read %s", buf);
                exit();
                }
        close(ns);
        }
}
```



server accepting a call

Client process

- Analizirajmo **klijent proces** koji sarađuje sa **serverskim procesom**.

- Klijent kreira socket u **istom domenu** **kao server proces** i
- šalje **connect** zahtev za ime **sockname**,
- koje je vezano za neki socket u serverskom procesu.

- Kada se **SC connect vrati**, klijent proces ima **VC ka serverskom procesu**.

- U ovom primeru, on upiše jednu poruku i izlazi(exit).

client proces in the UNIX System Domain

- **client proces in the UNIX System Domain**
- #include <sys/types.h>
- #include <sys/socket.h>
- main()
- {
- int sd,ns;
- char buf[256];
- struct sockaddr sockaddr;
- int formlen;

- **sd=socket(AF_UNIX, SOCK_STREAM, 0);**

- /*connect to name- null char is not part of name*/
- if(**connect**(sd, "sockname", sizeof("sockname"-1) = = -1) exit();

- **write**(sd, "hello!",6)
- }

Internet domain

- Ako server proces želi da **opslužuje procese na mreži**,
- **mora** da specificira **Internet domen u socket SC**, kao

- **socket(AF_INET, SOCK_STREAM, 0);**

- i obavi **bind mrežne** adrese **dobijene od name servera**.

- **BSD ima biblioteku za ove funkcije**.

- Slično, **drugi parametar u klijentovom connect SC treba da sadrži**
  - **adresnu informaciju**, potrebnu za identifikaciju **mašine na mreži** (routing adresu za slanje poruka na destination mašinu preko router mašine) i
  - **dodatne informacije** da identifikuju **partikularni socket** na **destination mašini**.

- **Ako server želi da osluškuje (listen) na mreži** i **lokalne procese,**
- **on treba da koristi 2 socketa**,
- a **select SC određuje** koji **klijent realizuje konekciju.**