# Operacinės sistemos P175B304

8 paskaita (praktinė)

Doc. Ingrida Lagzdinytė-Budnikė

2014-04-28

# Paskaitos turinys

- Berkeley soketų API
- Serverių tipai ir galimos jų realizacijos
- Kliento-serverio darbo algoritmai TCP, UDP atvejais

# API

- Taikomųjų programų sąsaja (API) tai sąsaja, kurią teikia operacinė sistema. Ji leidžia taikomosioms programoms kreiptis tam tikrų paslaugų atlikimo į žemiau esančius pagal OSI arba TCP/IP modelį lygius.

- API- tai rinkinys procedūrų bei įrankių, skirtų programinėms aplikacijoms kurti.

- API pavyzdžiai:
  - UNIX-tipo operacinės sistemos teikia : Berkeley Sockets, System V TLI.
  - Microsoft Windows turi į  Soketų API panašią  API Winsock.

# Berkeley Soketai

- Berkeley Soketai tai **originali tinklinė sąsaja**, pirmi bandymai susiję su UNIX 4.2 BSD – kaip “pipe” mechanizmo išplėtimas.
- Tarp-procesinio tinklinio bendravimo pradžia galima laikyti UNIX 4.3 BSD , kuri turi posistemę, skirtą tarp-procesiniam bendravimui tinkle.
- Tai Berkeley Soketai  sukurti 1983m.

- Šios API tikslas – leisti vartotojams kurti aplikacijas, kurios bendrauja tarpusavyje komunikuodamos tinkle. Vartotojui nereikia rūpintis tokiais klausimais:
  - **Kaip veikia tinklai,**
  - **Kaip siunčiami duomenys tinklu,**
  - **Kaip duomenys paruošiami persiuntimui tinklu**.

# Komunikavimo nusakymas

- Soketas yra tam tikra **abstrakcija**, atitinkanti **galinį** komunikavimo tašką.
- Jį apibrėžia IP adresas ir prievado (porto) numeris.
- Komunikavimas vyksta tarp dviejų procesų, taigi yra nusakomas dviem galiniais taškais.
- Pavyzdžiui kliento-serverio komunikacijose duomenys keliauja tarp dviejų soketų.

# Veiksmai su soketais

- Visuma operacijų, kurios gali būti vykdomos su soketais sudaro Soketų API .
  - **kontrolės operacijos:**
    - tai prievado numerio surišimas su soketu, ryšio inicijavimas arba priėmimas sokete arba soketo sukūrimas, suardymas.
  - **duomenų perdavimo operacijos:**
    - duomenų rašymas / skaitymas per soketą.
  - **Statusą nusakančios operacijos:**
    - Tokios kaip kad IP adreso susijusio su soketu suradimas.

# Taikomosios programos veiksmai vykdomi su soketais:

- Sukurti soketą ir surišti jį su tam tikru adresu.
- Sudaryti sujungimus ir priimti susijungimus naudojant sukurtą soketą.
- Siųsti ir priimti duomenis per sukurtą soketą.
- Nutraukti soketų operacijas.

# Soketo sukūrimas

- Operacinė sistema sukurdama soketą **grąžina** sveiką skaičių. Soketas egzistuoja tam tikrame **domene**, yra tam tikro **tipo** ir su juo yra surišama tam tikra **protokolų šeima.** Jis sukuriamas naudojant kreipinį ***socket():***

```
int s = socket(domain, type, protocol);
```

  - s: soketo deskriptorius, sveikas skaičius (kaip ir failų-atveju);
  - domain: komunikacinis domenas, pavyzdžiui, AF_INET (IPv4 protokolui) AF_UNIX , AF_INET6 ;
  - type: komunikacijų tipas, kuris gali būti, pavyzdžiui, toks:
    - SOCK_STREAM: patikimas, 2-krypčių sujungimu pagrįstas susijungimas;
    - SOCK_DGRAM: nepatikimas, be sujungimo.
    - SOCK_RAW : teikia priėjimą prie vidinio tinklinio protokolo. Prieinamas tik “root” vartotojui.
  - protocol: nusakomas protokolas,– paprastai nurodomas 0

# Domenas – adresų šeima

Pasirinktas domenas nusako atitinkamą, sąsajoje naudojamą adresų šeimą:

**/usr/include/sys/socket.h** faile nurodomos šios dažniausiai naudojamos adresų šeimos:

**AF_UNIX**

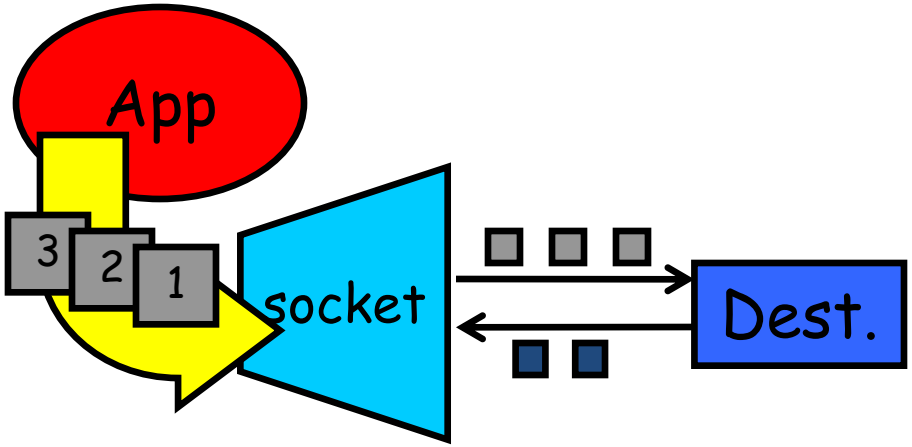
Žymi UNIX operacinėje sistemoje įprastus kelio vardus.

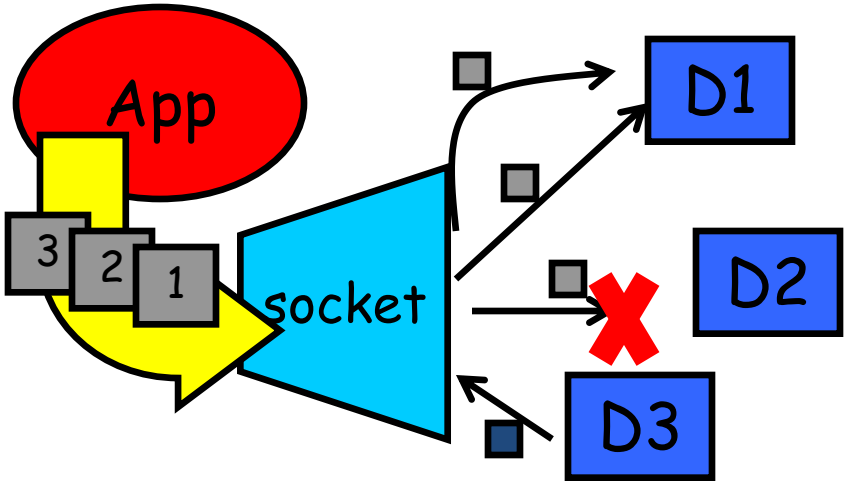
**AF_INET**

Žymi ARPA Interneto adresus.

# Du pagrindiniai soketų tipai

- SOCK_STREAM
  - TCP
  - Patikimas pristatymas
  - Garantuota pristatymo tvarka
  - Orientuotas į susijungimą
  - dvipusis
- SOCK_DGRAM
  - UDP
  - Nepatikimas pristatymas
  - Negarantuota eilės tvarka
  - Neorientuotas į susijungimus
  - Gali siųsti arba priimti



# Pavyzdys

```
#include <sys/types.h>
#include <sys/socket.h>

s1 = socket(AF_UNIX, SOCK_DGRAM,0);

s2 = socket(AF_INET, SOCK_STREAM, 0);
```

Jei protokolas nėra nusakomas ( 0 ), tai sistema parinks tinkamą protokolą, iš tų kurie tinka pagal komunikacinį domeną bei soketo tipą.

# struct sockaddr

- Bendra:

```
struct sockaddr {
        u_short sa_family;
        char sa_data[14];
};
```

  - sa_family
    - Nusako, kuri adresų šeima yra naudojama.
    - Apsprendžia, galinį komunikavimo tašką.

# Soketų adresų uždavimas (sockaddr_un)

```
struct sockaddr_un
{ unsigned char sun_len;    /* length including null */
  sa_family_t sun_family; /* AF_UNIX */
  char sun_path[104];      /* pathname */ };
```

# Soketų adresų uždavimas (sockaddr_in)

Soketas neturi savyje informacijos apie prievadų numerius, lokalių mašinų ar nutolusių mašinų IP adresus.

Struktūra, taikoma protokolui IPv4 yra ***sockaddr_in***:

```
struct sockaddr_in {
short int sin_family;              // Adresų šeima
unsigned short int sin_port;       // Prievado numeris
struct in_addr sin_addr;           // Internetinis adresas
unsigned char sin_zero[8];  }      // skirtas išlyginimui
```

šios struktūros iki dydžio naudojamo standartinėje struktūroje **sockaddr**

Šioje struktūroje naudojama adreso struktūra: struct **in_addr** turi tik vieną įrašą: **sin_addr**; /* 32-bitų IPv4 adresas, kuris turi būti užduodamas prisilaikant tinklinės baitų orientacijos */

# Pastabos

Savojo IP adreso ir porto uždavimas gali būti automatizuotas:

```
my_addr.sin_port = 0; // choose an unused port at random
my_addr.sin_addr.s_addr = INADDR_ANY; // use my IP address
```

Nustatydami *my_addr.sin_port* lygų nuliui, pranešate `bind()` parinkti laisvą portą.

Nustatant *my_addr.sin_addr.s_addr* į `INADDR_ANY`,

liepiate automatiškai įrašyti mašinos IP adresą (kurioje šis procesas sukasi). `INADDR_ANY` nereikia užduoti, kad būtų "Network Byte Order"! : `INADDR_ANY` realiai yra nuliai

# Su soketų adresais surištos funkcijos

- Keturios su soketais surištos funkcijos perduoda soketo adreso struktūrą iš proceso operacinės sistemos branduoliui:

  `bind(), connect(), sendto(), ir sendmsg()`

- Penkios su soketais surištos funkcijos perduoda soketo adreso struktūrą iš branduolio– procesui:

  `accept(), recvfrom(), recvmsg(), getpeername(), ir getsockname(),`

- Visose jose argumentų sąraše yra nurodoma perduodama adreso struktūra bei jos ilgis.

# Soketo surišimas su adresu

- Funkcija `bind()` soketui priskiria lokalų adresą.
- Interneto protokolų atveju, adresą nusako kombinacija 32 bitų IPv4 adreso (arba 128 bitų IPv6 adreso), kartu su 16 bitų TCP arba UDP prievado numeriu.

```
int bind(int sockfd, const struct sockaddr *name,
    socklen_t namelen);
```

int sockfd – soketo deskriptorius,
const struct sockaddr *name – soketo adreso struktūra,
socklen_t namelin – adreso struktūros ilgis.

**TCP klientams soketo surišimas su adresu nėra būtinas.**
**Operacinė sistema pati vykdo šį surišimą vykdydama connect() kreipinį .**

# Soketo sukūrimas ir surišimas su adresu (pavyzdys)

```
int   main(int argc,char **argv)
{      int z;        struct sockaddr_in  adr_inet  ;/* AF_INET */
    int len_inet;                  /* Length  */
      int sck_inet;                   /* Socket */
 /* Sukurti Soketa */
      sck_inet = socket(AF_INET,SOCK_STREAM,0);

 if ( sck_inet == -1 )         {  printf ("soketo sukurt nepavyko \n");
    exit(1) }
    /* pildoma adresine struktura */
        memset(&adr_inet,0,sizeof adr_inet);
        adr_inet.sin_family = AF_INET;
        adr_inet.sin_port = htons(9000);
        adr_inet.sin_addr.s_addr = inet_addr("193.219.33.100");
        len_inet = sizeof adr_inet;
   /* surisimas su soketu */
        z = bind(sck_inet,(struct sockaddr *)&adr_inet, len_inet);
    if ( z == -1 )
       {   printf("bind funkcija nepavyko \n"); exit(1) }
...
  }
```

# Funkcija `gethostbyname()`.

Ši funkcija ima ASCII eilutę, turinčią kompiuterio domeno vardą ir grąžina adresą `hostent` struktūroje, kur tarp kitų laukų yra ir kompiuterio IP adresas dvejetainiame pavidale.

- *`hostent`* struktūra yra aprašoma faile ***netdb.h***:
- ```
  struct hostent {
   char *h_name;       /* oficialus kompiuterio vardas  */
   char **h_aliases;   /* kiti hosto vardo sinonimai */
   int h_addrtype;     /* adreso tipas */
   int h_length;       /* adreso ilgis */
   char **h_addr_list; /* kompiuterio dvejetainių IP
  adresų sąrašas */
         };
  ```
- Taigi po kreipinio į funkciją gethostbyname() hosto tinklinį adresą galima rasti lauke **`h_addr_list[0]`**
  ```

# Adresų ir prievadų nustatymas

Problema:

- Skirtingos mašinos / operacinės sistemos naudoja skirtingą žodžių rikiavimą saugodamos reikšmes savo atmintyje.
- Galimi rikiavimai:
  - little-endian: kai žemiausio svorio baitai eina pirmi,
  - big-endian: kai aukščiausio svorio baitai eina pirmi.
- Šios mašinos gali komunikuoti viena su kita tinkle
- Perduodant adresus duomenų pakete tiek adresas, tiek prievadas turi būti užduotas naudojant **tinklinio rikiavimo** – big-endian būdą, todėl užpildant adresų struktūras reikia atlikti duomenų konvertavimą, kuriam atlikti naudojamos funkcijos:

```
htons() -- "Host to Network Short"
htonl() -- "Host to Network Long"
ntohs() -- "Network to Host Short"
ntohl() -- "Network to Host Long"
```

# Sujungimo sudarymas

- Susijungimas yra įkuriamas iš kliento pusės naudojant funkciją `connect()`, kurioje nurodomas serverio adresas:

```
int status = connect(sock, &name, namelen);
```

  - status: 0 jei susijungimas sukuriamas, -1 priešingu atveju.
  - sock: integer, kliento soketas, kuris naudojamas susijungimui.
  - name: struct sockaddr: serverio pusės soketą nusakanti adresinė struktūra.
  - namelen: integer, sizeof(name) serverio adresinės struktūros ilgis.

# Serverio veiksmai

- Serveris yra pasyvus komunikacijų dalyvis. Serverio procese, sukūrus soketą ir surišus jį su adreso struktūra, toliau jis turi laukti ateinančių kliento užklausų, kurioms atėjus reikia jas priimti ir aptarnauti.
- TCP atveju serveris laukia ateinančių užklausų susijungimui – klauso numatytame prievade, tai realizuojama naudojant funkciją - **`listen()`**, o jų sulaukęs jas priima naudodamas funkciją - **`accept()`**.

# Listen() ir accept ()

Funkcija `listen()`

- **`int listen(int sockfd, int backlog);`**
  - **sockfd** yra soketo deskriptorius iš **socket()** kreipinio.
  - **backlog** leistinų susijungimų kiekis įėjimo eilėje.

Funkcija `accept()`

- **`int s = accept(sock, &name, &namelen);`**
  - s: integer, naujas soketas duomenų priėmimui-perdavimui,
  - sock: integer, originalus soketas (kuriame serveris klauso)
  - name: struct sockaddr, kliento soketo adreso struktūra,
  - namelen: sizeof(name): adreso struktūros dydis
- Priimtas kliento susijungimas bus aptarnaujamas TCP protokolo atveju per soketą s, sukurtą veikiant accept().

# Serverio proceso programos fragmentas

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define MYPORT 3490 // prievadas klientu aptarnavimui
#define BACKLOG 10 // kiek „kabančiu“ susijungimu gales buti eileje

main()
{
int sockfd, new_fd;   // sock_fd-sock klausymui, new_fd-sock klientu aptarn.
struct sockaddr_in my_addr;      // serverio adreso informacija
struct sockaddr_in their_addr;   // kliento adreso informacija
int sin_size;

sockfd = socket(AF_INET, SOCK_STREAM, 0);    //reikia tikrint klaidas !

my_addr.sin_family = AF_INET;                // host byte order
my_addr.sin_port = htons(MYPORT);            // short, network byte order
my_addr.sin_addr.s_addr = INADDR_ANY;        // parinkt vietini IP
memset(&(my_addr.sin_zero), '\0', 8);        // užpildyt 0-liais

bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

listen(sockfd, BACKLOG); sin_size = sizeof(struct sockaddr_in);

new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size); . . .
```

# Duomenų siuntimas / priėmimas SOCK_STREAM soketo atveju

SOCK_STREAM tipo informacijos siuntimui per soketą galima naudoti funkciją `send()`, arba įprastą, veiksmuose su failais naudojamą funkciją `write()`:

```
int count = send(sock, &buf, len, flags);
```

count: kiek baitų yra perduota (-1 klaidos atveju),

buf: char[], buferis, kuriame yra perduodami duomenys,

len: integer, buferio ilgis, kuris turi būti perduodamas (baitais)

flags: integer, speciali opcija, paprastai tiesiog 0.

```
int count = recv(sock, &buf,  len, flags);
```

count: kiek baitų yra gauta (-1 jei klaida),

buf: void[], saugo priimtus baitus,

len: max buf dydis

flags: integer, speciali opcija, paprastai tiesiog 0.

# Duomenų siuntimas / priėmimas SOCK_DGRAM soketo atveju

Esant soketams SOCK_DGRAM susijungimas nėra įkuriamas, todėl kartu su siunčiama informacija reikia nurodyti ir kam ji yra siunčiama:

```
int count = sendto(sock, &buf, len, flags,
  &addr, addrlen);
```

count, sock, buf, len, flags: tas pats kaip ir send() atveju,

addr: struct sockaddr, gavėjo adreso struktūra

addrlen: sizeof(addr)

```
int count = recvfrom(sock, &buf,  len, flags,
  &name, &namelen);
```

count, sock, buf, len, flags: tas pats kaip ir recv() atveju,

name: struct sockaddr, šaltinio (siuntėjo) adresas,

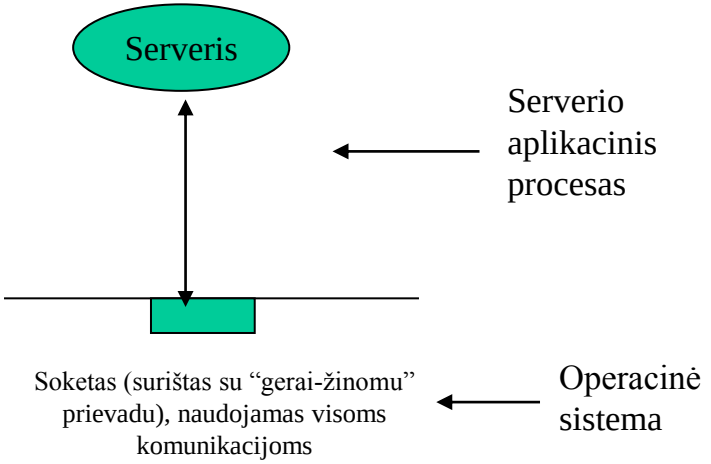
namelen: sizeof(name): struktūros ilgis.

# Galimos serverių realizacijos

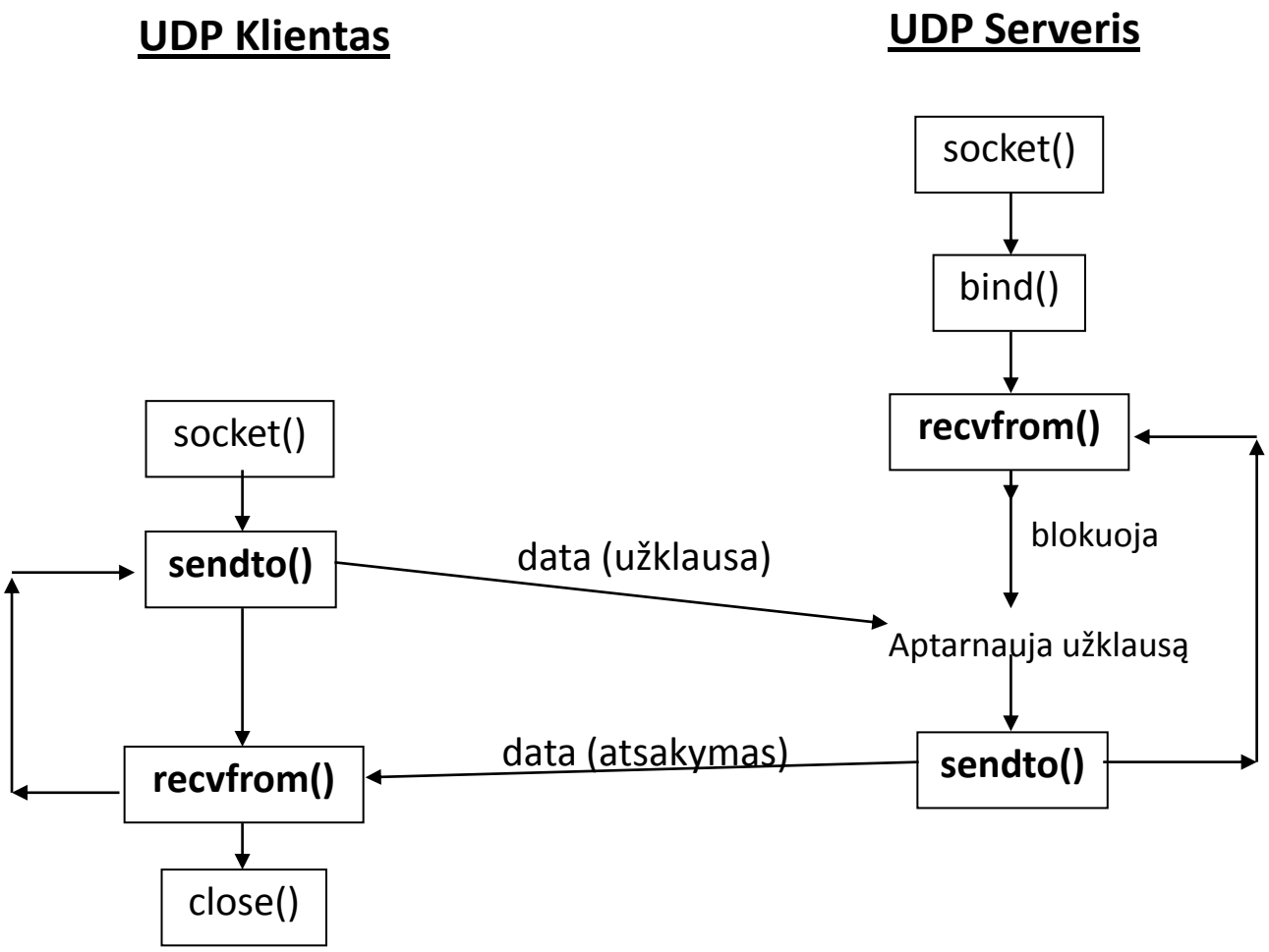
| | |
|---|---|
| Iteratyvus, be susijungimo | Iteratyvus, su susijungimu |
| Konkuruojantis, be susijungimo | Konkuruojantis, su susijungimu |

# Į susijungimą neorientuotas serveris

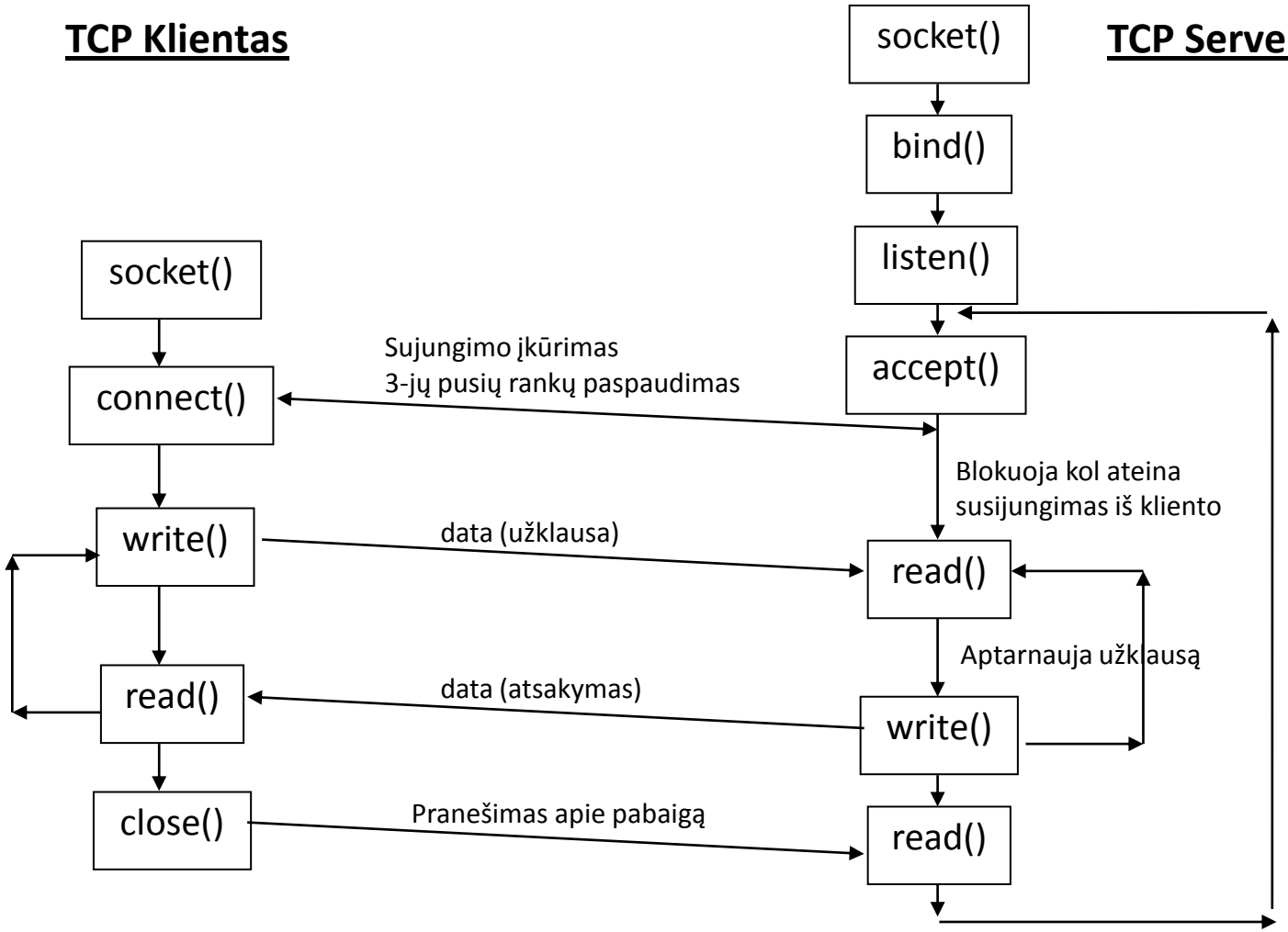
- Šio tipo serveriai komunikacijoms su klientu naudoja SOCK_DRAM tipo soketus ir remiasi UDP protokolu.
- Komunikuojama su klientu naudojant tą patį soketą, kuriame ir klausomasi, kuriame yra laukiama ateinančių kliento užklausų.



# Kliento - serverio darbo algoritmas (UDP atveju)



# TCP kliento -serverio iteratyvaus aptarnavimo algoritmas



# Serverių aptarnavimo parinkimas

| **Serveriai** | **Iteratyvaus aptarnavimo** | **Konkurencinio aptarnavimo** |
|---|---|---|
| Aptarnauja | Vieną užklausą vienu metu. | Daug užklausų vienu metu. |
| Kada reiktų naudoti: | Kai yra garantija, kad užklausa bus apdorota per mažą laiko intervalą. | Tais atvejais, kai negalima apriboti užklausos įvykdymo laiką. |
| Privalumai: | Triviali realizacija.<br>Lengva sutvarkyti kreipinius (pvz jei yra ryšiai su duomenų baze). | Individualios klientų užklausos gali būti bet kokio sudėtingumo ir trukmės. |
| Problemos: | Serveris yra blokuojamas kol turi reikalų su užklausa. Jei užklausos aptarnavimas užtrunka ilgiau nei leistina, joks kitas klientas negauna serviso. | Sudėtingumas susijęs su konkurenciniu aptarnavimu |

# Konkurencinio aptarnavimo serverio projektavimo alternatyvos

- *Vienas* vaiko *procesas kiekvienam* klientui aptarnauti.
- *Viena gija* ( thread ) *kiekvienam* klientui aptarnauti.
- Iš anksto sukurti (preforking) *keli vaikų procesai*.
- Iš anksto sukurtos (prethreaded) *kelios gijos*.
- Realizacija *viename procese* (naudojant sisteminį kreipinį `select()` kelių soketų priežiūrai.)

# Konkurencinio aptarnavimo serverių algoritmai

- Pagrindinis serverio procesas dar yra vadinamas „master“ procesu, o klientus aptarnaujantys procesai (gijos) vadinamos „slave“ procesais.
- Į susijungimą orientuoti serveriai konkurenciniam aptarnavimui užtikrinti sukuria naują procesą ***kiekvienam naujam susijungimui***.
- Į susijungimą neorientuoti serveriai konkurenciniam aptarnavimui užtikrinti sukuria naują procesą ***kiekvienai naujai užklausai***.
- Master procesas atidaro soketą gerai žinomame prievade, laukia ateinančių klientų užklausų ir sukuria “slave” procesą kiekvienos užklausos aptarnavimui.
- Master procesas niekad nekomunikuoja tiesiai su klientu.
- Kiekvienas slave procesas vykdo komunikacijas su vienu klientu ir baigiasi po šio kliento užklausos (UDP atveju) arba po pilno kliento aptarnavimo(TCP atveju).

# Konkurencinio aptarnavimo į susijungimą neorientuoto serverio darbo algoritmas

- *Master:* Sukurti soketą ir surišti jį su *gerai-žinomu* adresu.
- *Master:* Ciklišk ai kviesti `recvfrom()` sekančios klientų užklausos priėmimui ir gavus kliento užklausą sukurti slave procesą (naudojant `fork()`) kliento užklausos aptarnavimui.

- *Slave:* priimti kliento užklausą, formuluoti atsakymą bei jį siųsti klientui naudojant `sendto()`.
- *Slave:* Uždaryti susijungimą ir pasibaigti.

# Konkurencinio aptarnavimo orientuoto į susijungimą serverio darbo algoritmas

- *Master:* Sukurti soketą, jį surišti su *gerai-žinomu adresu..*
- *Master:* Soketas paliekamas **pasyvioje** būsenoje, laukdamas susijungimo **listen().**
- *Master:* Ciklišk*ai* vykdyti **accept()** priimant naujas užklausas iš kliento, priėmus sukurti naują soketą bei naują slave procesą kliento aptarnavimui.

- *Slave:* **Gauti** susijungimo užklausą*: **soketą** susijungimui bei kliento adresą*.
- *Slave: Komunikuoti* su klientu pasinaudojant susijungimu*: **skaityti** užklausas ir **siųsti** atgal atsakymus.*
- *Slave: Uždaryti susijungimą ir pasibaigti. Slave procesas pasibaigia pilnai aptarnavęs vieną klientą*

# TCP konkurencinio aptarnavimo serverio programos fragmentas

```
int sockfd, newsockfd;
 if ( (sockfd = socket( ... )) < 0) /* create socket */
.....
if ( bind(sockfd,... ) < 0) /* bind socket */
.....
if ( listen(sockfd,5) < 0) /* announce we're ready */
.....
while (1==1) { /* loop forever */
  newsockfd = accept(sockfd, ...); /* wait for client */
  if ( (pid = fork()) == 0) {
     /* child code begins here */
     close(sockfd); /* child doesn't wait for client */
     ......
     /* child does work here, communicating
         with client using the newsockfd */
     .......
     exit(0); /* child dies here */
     }
  /* parent continues execution below */
  close(newsockfd); /* parent won't communicate with */
  /* client - that's childsplay! */
  } /* end of while */
```

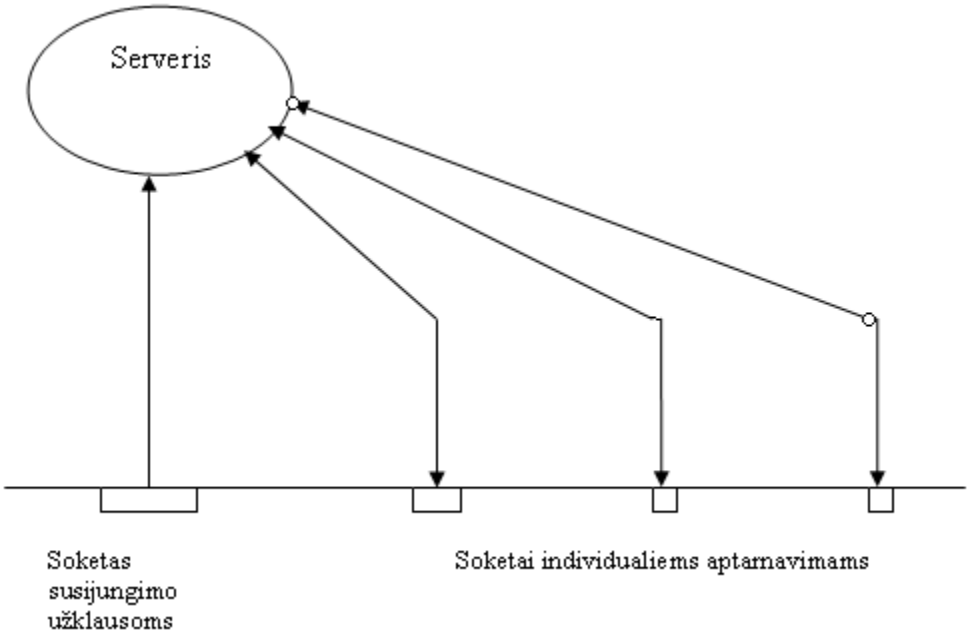
# Problemos, kurios iškyla vykdant klientų aptarnavimą

- Klientas susijungia bet neatsiunčia užklausų, todėl serveris blokuojamas ties `read()` kreipiniu.
- Klientas siunčia užklausą, bet negali nuskaityti atsakymo. Serveris blokuojasi ties `write()` kreipiniu.
- Gali susidaryti mirties taško situacija, jei programa ar grupė programų negali vykdyti jokių veiksmų, nes yra užblokuotos – laukia įvykio, kuris niekad neįvyks. Serverio atveju, tai gali susilpninti jo galimybes atsakyti į užklausas.

# Konkurencija naudojant vieną procesą.

Į susijungimus-orientuoto serverio proceso struktūra, kai konkurencija realizuojama viename procese, valdančiame daug soketų.

Soketai prižiūrimi naudojant funkciją `select()`.



# `Select()` kreipinys

- **`select()`** kreipinys leidžia valdyti keletą soketų vienu metu.
- Jis gali informuoti OS apie soketus, kuriais jis domisi, o aplikacinei programai gali pranešti, kurie soketai yra pasiruošę rašymui, skaitymui ar yra “pakibę” dėl klaidų.
- Aplikacija gali atsakinėti keliems soketams. Ji nėra blokuojama kuriame nors sokete laukdama įvykio.

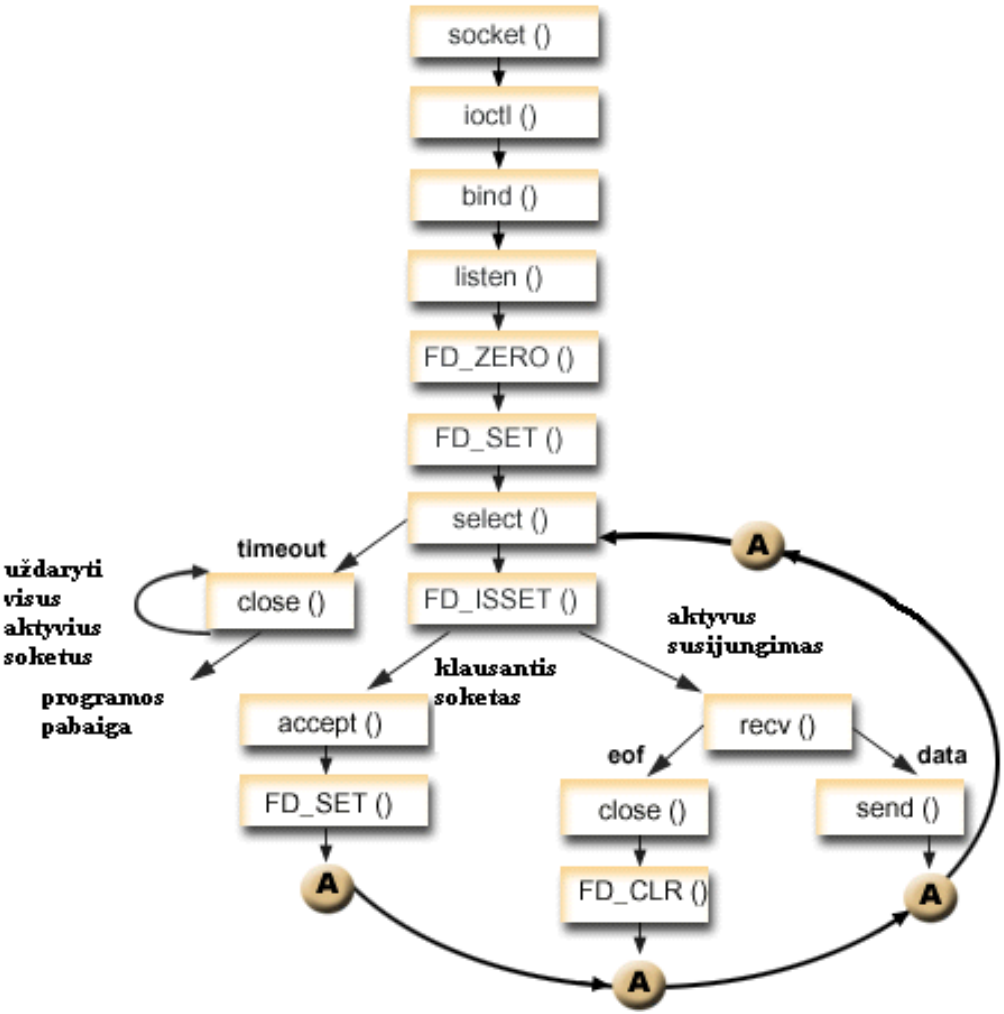
**`select()`** kreipinys:

- **`int select(int maxfd, fd_set *readset, fd_set *writeset, fd_set *exceptset, struct timeval *timeout);`**

Susijusios funkcijos:

- **FD_SET(fd, &fdset)** - prideda fd į soketų rinkinį.
- **FD_CLR(fd, &fdset)** - išvalo fd iš soketų rinkinio.
- **FD_ISSET(fd, &fdset)** – tikrina, ar fd yra nustatytas rinkinyje.
- **FD_ZERO(&fdset)** - išvalo failų deskriptorių rinkinį.

# Algoritmas naudojant select()



# Serverio algoritmas, realizuojant veiksmus viename procese.

- Sukurti soketą ir surišti jį su adresu. Pridėti šį soketą prie sąrašo soketų, kurie gali būti naudojami įvedimui - išvedimui.
- Naudoti `select()` laukiant įvedimo- išvedimo viename iš sąraše nurodytų soketų.
- Jei originalus soketas pasirengęs I/O, naudoti `accept()` sekančio susijungimo gavimui ir pridėti šį soketą prie sąrašo soketų, kuriuose galimas I/O.
- Jei bet kuris kitas(ne originalus) soketas yra pasiruošęs skaitymui iš jo, naudoti **read()** sekančiai užklausai priimti, suformuoti atsakymą ir panaudoti **write()** atsakymo nusiuntimui atgal.
- Tęsti `select()` veiksmus...

# Supaprastintas `select()` panaudojimo pavyzdys

```
#include <sys/types.h>
main(){
fd_set readset;
int nready;

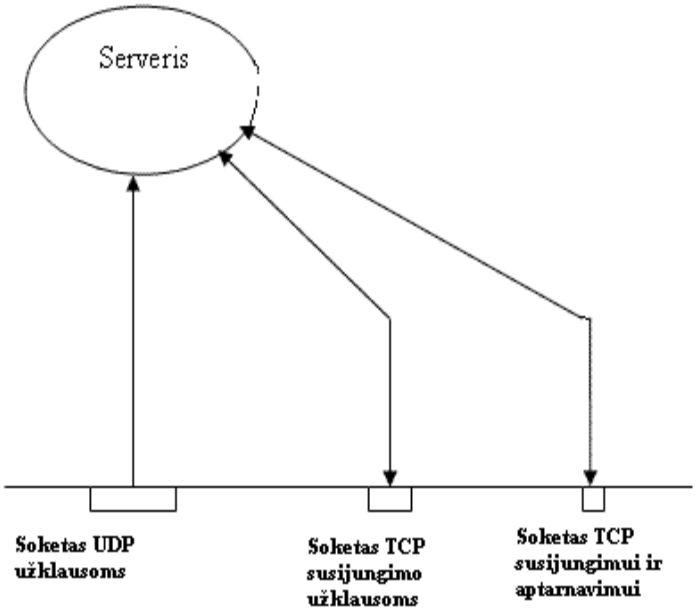
  /* open, bind, connect/accept etc. sockets
sd1, sd2, sd3 */
/* tell OS sockets we're interested in */
FD_ZERO(&readset);
FD_SET(sd1,&readset);
FD_SET(sd2,&readset);
FD_SET(sd3,&readset);

  /* call select, returning when something's ready */
nready = select(64, &readset, NULL, NULL, NULL);
/* read from appropriate socket */
if (FD_ISSET(sd1, &readset))
..... /* do i/o from sd1 */
else if (FD_ISSET(sd2, &readset))
..... /* do i/o from sd2 */
else if (FD_ISSET(sd3, &readset))
..... /* do i/o from sd3 */
```

# Daugiaprotokoliai (TCP,UDP) serveriai

- Galimi atvejai, kai tas pats aplikacijos algoritmas realizuojamas naudojant tiek TCP, tiek UDP protokolus. Tai gali būt realizuojama tiek atskiruose, dviejuose serverio procesuose, tiek tame pačiame, viename procese.

- Projektuojant serverius galima ir dar sudėtingesnė realizacija, kai serveris ne tik kad teikia kažkurią paslaugą pagal abu protokolus, bet dar kartu teikia kelias įvairaus pobūdžio paslaugas.



# Ačiū už jūsų dėmesį