BSD sockety

Jiří Ledvina

8. prosince 1993

Úvod

Operační systém UNIX obsahuje také prostředky pro komunikaci mezi procesy. Patří mezi ně posílání zpráv, semafory a sockety (schránky). Sockety umožňují realizovat komunikaci mezi procesy, rozmístěnými v jednotlivých uzlech sítě. To dovoluje vytvářet distribuovaný systém v prostředí lokálních i rozlehlých sítí.

Služby spojené s přenosem zpráv jsou zahrnuty do služeb jádra operačního systému. Z komunikačního hlediska se jedná o služby transportní úrovně.

Komunikaci pomocí socketů lze charakterizovat následovně:

- v jednotlivých uzlech sítě běží sekvenční procesy. Pro vzájemnou komunikaci a synchronizaci využívají mechanizmus posílání zpráv.
- komunikace probíhá mezi koncovými komunikačními body (porty). Port je využíván výlučným způsobem – proces jej má přidělen, využívá jej a uvolňuje.
- koncové komunikační body jsou jednoznačně identifikovatelné. Jejich číselná hodnota je dána číslem (adresou) uzlu a číslem portu. Celé spojení je identifikováno pěticí:
  - adresa cílového uzlu
  - číslo cílového portu
  - adresa zdrojového uzlu
  - číslo zdrojového portu
  - číslo použitého protokolu
- spojení procesu s portem se děje pomocí schránky (socketu). Socket je abstraktní datový typ, vytvářený dynamicky operačním systémem.
- příjem a vysílání zpráv se děje s využitím vnitřní vyrovnávací paměti. Na straně příjmu vznikají fronty, které jsou postupně zpracovávány procesy.
- pro komunikaci jsou k dispozici komunikační služby spojované (connection oriented – virtuální okruhy) a nespojované (connectionless – datagramy). Obojí dovolují realizovat duplexní přenos zpráv mezi dvěma koncovými komunikačními body.

- kromě služeb pro přenos zpráv poskytuje operační systém také služby pro transformaci adres a jmen, nastavení parametrů přenosu a pod.

Aparát socketů podporuje komunikaci nad různými komunikačními protokoly. Mezi ně patří Internet TCP/IP, XEROX NS a další. Následující text se omezuje pouze na popis týkající se komunikačního prostředí TCP/IP.

Systémová volání nad sockety jsou uspořádána podle do skupin podle významu. Nakonec je uvedeno 6 jednoduchých programů jako příklad použití jednotlivých funkcí. Příklady byly odladěny Pavlem Křižanovským pod operačním systémem ULTRIX, kde pracují bez problémů. Lze je též přeložit a spustit pod OS DYNIX s tím, že klient pracující s datagramovými službami z neznámého důvodu nepřijme odpověď.

Při ladění pod OS DYNIX je třeba při sestavování explicitně uvézt knihovny (v uvedeném pořadí) libsocket.a, libinet.a a libnsl.a, t.j. překládat příkazem

```
cc ..... -lsocket -linet -lnsl
```

1 Práce se jmény

Funkce pro práci se jmény dovolují pracovat jak s názvy hostitelských systémů, tak i se jmény služeb a protokolů. Konvertují symbolické jméno na adresu, nebo číselný kód služby, příp. protokolu. Všimněte si, že vrací odkaz na datovou strukturu, obsahující všechny informace, spojené s daným objektem.

Tyto funkce se volají zejména v úvodních částech programu, kdy je třeba připravit parametry příkazu *socket* nebo *bind* (viz kap. 3 a 4).

1.1 GET_HOST_BY_ADDR

Vyhledá informaci o hostitelském systému specifikovaném IP adresou. Pro TCP/IP je *atype* = AF_INET. Vrací ukazatel na strukturu uvedenou níže. Nulová adresa indikuje chybu. Informace o chybě je uložena v proměnné *h_errno*.

```
struct hostent *gethostbyaddr( addr, alen, atype );
char      *addr;  // Ukazatel na pole, obsahující adresu
                  // hosta (IP adresu).
int       alen;   // Délka adresy ve slabikách.
int       atype;  // Typ adresy, pro IP adresu AF_INET.
```

```
struct hostent {
  char *h_name;          // oficiální jméno hosta
  char *h_aliases[];     // seznam ostatních alias
  int  h_addr_type;      // typ adresy hosta
  int  h_length;         // délka adresy hosta
  char **h_addr_list;    // seznam adres hosta
  }
```

1.2 GET_HOST_BY_NAME

Vyhledá informaci o hostitelském systému, který je specifikován jménem. Vrací ukazatel na strukturu *hostent*. Nulová hodnota ukazatele indikuje chybu. Informace o ní je uložena v proměnné *h_errno*.

```
struct hostent *gethostbyname( name );
char      *name; // Adresa textového řetězce, obsahující
                 // jméno hostitelského systému.
```

1.3 GET_HOST_ID

Vrací 32 bitový identifikátor lokálního počítače. Běžně je to primární IP adresa počítače.

```
long gethostid();
```

1.4 GET_HOST_NAME

Vrací primární jméno lokálního počítače v textové podobě (max 64 znaků). Návratový kód 0 znamená úspěšné volání, −1 chybu. Kód chyby obsahuje globální proměnná *errno*.

```
int gethostname( name, namelen );
char      *name;   // Adresa pole do kterého má být jméno
                   // uloženo.
int       namelen; // Délka pole name.
```

1.5 GET_PEER_NAME

Dovoluje získat adresu vzdáleného konce spojeného socketu. Vrací 0 je-li volání úspěšné, jinak vrací −1. Chybový kód obsahuje globální proměnná *errno*.

```
int getpeername( socket, remaddr, addrlen );
int       socket;   // Popisovač socketu vytvořený
                    // funkcí socket.
sockadr   *remaddr; // Ukazatel na datovou strukturu
                    // sockaddr, obsahující adresu
                    // koncového bodu.
int       *addrlen; // Ukazatel na integer, obsahující
                    // při vyvolání délku zadané adresy.
                    // Po ukončení volání obsahuje délku
                    // aktuální adresy.
```

1.6 GET_PROTO_BY_NAME

Hledá oficiální kód přidělený protokolu. Vrací ukazatel na níže uvedenou datovou strukturu. Nulová hodnota ukazatele indikuje chybu, kód chyby je uložen v globální proměnné *errno*.

```
struct protoent *getprotobyname( name );
char      *name;    // Adresa řetězce obsahujícího jméno
                    // protokolu.
```

Datová struktura *protoent* má následující tvar:

```
struct protoent {
  char  *p_name;        // jméno protokolu
  char  **p_aliases;    // seznam alias protokolu
  int   p_proto;        // číslo protokolu
};
```

1.7 GET_SERV_BY_NAME

Vyhledá ze servisní databáze sítě informace dle zadaného jména služby a jména protokolu (např. *TCP*). Vrací ukazatel na níže uvedenou datovou strukturu. Nulová hodnota ukazatele indikuje chybu, kód chyby je uložen v globální proměnné *errno*.

```
struct servent *getservbyname( name, proto );
char      *name;        // jméno služby (např. "echo")
char      *proto;       // jméno protokolu (např. "udp")
```

Datová struktura *servent* má následující tvar:

```
struct servent {
  char  *s_name;       // jméno služby
  char **s_aliases;    // seznam alias
  int    s_port;       // číslo portu služby
  char  *s_proto;      // jméno použitého protokolu
}
```

1.8 GET_SOCK_NAME

Vrací jméno specifikovaného socketu. Vrací `0` je-li volání úspěšné, jinak -1. Chybový kód je uložen v globální proměnné *errno*.

```
int getsockname( socket, name, namelen );
int       socket;   // Popisovač socketu vytvořený
                    // funkcí socket.
char      *name;    // Adresa pole pro umístění
                    // jména socketu.
int       *namelen; // Délka vráceného řetězce.
```

1.9 SET_HOST_ID

Systémový manager spouští při inicializaci privilegovaný program, který volá funkci *sethostid* aby přiřadila počítači jednoznačný `32` bitový identifikátor. Běžně bývá tímto identifikátorem primární IP adresa počítače.

```
(void) sethostid( hostid );
int       hostid;    // IP adresa počítače
```

2 Konverzní a pomocné podprogramy

Konverzní funkce se používají pro transformaci zobrazení základních datových typů *int a long* hostitelského systému na zobrazení, používané sítí. Např. u počítačů typu IBM PC se předpokládá uložení slabik v pořadí nižší–vyšší. Jiné počítače ukládají data v pořadí vyšší–nižší slabika. Níže uvedené funkce uspořádají pořadí slabik tak, aby odpovídalo jak konvencím sítě, tak i konvencím použitého hostitelského počítače.

Ke konverzi pořadí slabik v proměnné daného typu slouží následující podprogramy:

```
u_long htonl( u_long hostlong );
u_short htons( u_short hostshort );
u_long ntohl( u_long netlong );
u_short ntohs( u_short netshort );
```

htonl – konvertuje host → síť, long int

htons – konvertuje host → síť, short int

ntohl – konvertuje síť → host, long int

ntohs – konvertuje síť → host, short int

Ke konverzi adres se používají funkce:

```
u_long inet_addr( char *ptr );
char *inet_ntoa( struct in_addr inaddr );
```

Funkce *inet_addr* slouží k převodu adresy zadané textově v desítkově tečkové notaci (`147.228.51.10`) na číselné vyjádření adresy jako `32` bitového čísla. Funkce *inet_ntoa* pak k opačnému převodu.

Ke práci s bloky slabik se používají funkce:

```
bcopy( char *src, char *dest, int nbytes );
bzero( char *dest, int nbytes );
int bcmp( char *ptr1, char *prt2, int nbytes );
```

Funkce *bcopy* kopíruje blok dat délky *nbytes* slabik. Funkce *bzero* nuluje blok paměti o délce *nbytes* slabik. Funkce *bcmp* slouží k porovnání dvou bloků dat. Vrací `0`, jsou-li oba řetězce identické.

Místo výše uvedených funkcí můžeme použít též následujících funkcí standardu ANSI:

```
memcpy( char *dest, char *src, int nbytes );
memset( char *dest, char value, int nbytes );
memcmp( char *ptr1, char *ptr2, int nbytes );
```

Ke zjištění délky tabulky deskriptorů programu lze použít funkci

```
int getdtablesize();
```

Funkce vrací počet položek tabulky. Položky se číslují od nuly.

S výhodou by ji bylo možno použít např. v programu (viz. příklad 9.5 ke zjištění počtu testovaných selektorů.

3 Vytvoření socketu

Následující funkce vytváří datovou strukturu, podobnou selektoru souboru, umožňující komunikaci mezi nezávislými procesy. Tyto procesy mohou být umístěny jak v jednom uzlu sítě, tak i v různých uzlech. Prostor pro *socket* (schránku) je vymezen v systémové datové oblasti. Uživateli je přístupný prostřednictvím různých funkcí, které dovolují nastavovat jeho parametry. Základní parametry jako typ použité rodiny síťového protokolu, vlastního protokolu a služby je nutno zadat při jeho vytváření. Další parametry, jako je adresa cílového uzlu, číslo portu a vlastnosti socketu se zadávají dodatečně (viz. kap. 5 a kap. 6.

3.1 SOCKET

Funkce *socket* vytváří socket pro použití v síťové komunikaci. Vrací číselný deskriptor socketu.

```
int socket( family, type, protocol );
int   family;   // Rodina protokolů.
int   type;     // Typ požadované služby.
int   protocol; // Číslo protokolu.
```

Type představuje typ služby. Pro Internet je SOCK_STREAM spojeno s protokolem TCP a SOCK_DGRAM s protokolem UDP). Kromě toho ještě existuje SOCK_RAW pro přímý přístup k rámci. Další typy jako SOCK_RDM pro spolehlivý přenos zpráv a SOCK_SEQPACKET pro uspořádaný přenos paketů nejsou v Internetu (TCP/IP) podporovány. Jednotlivé symbolické konstanty mají přiřazeny následující hodnoty.

```
#define SOCK_STREAM    1  /* stream socket */
#define SOCK_DGRAM     2  /* datagram socket */
#define SOCK_RAW       3  /* raw-protocol interface */
#define SOCK_RDM       4  /* reliably-delivered message */
#define SOCK_SEQPACKET 5  /* sequenced packet stream */
```

Family představuje tvar adresy dané rodiny protokolů. Pro Internet je použito označení AF_INET. Kódy ostatních rodin protokolů jsou uvedeny v následující tabulce.

```
#define AF_UNSPEC     0  /* unspecified */
#define AF_UNIX       1  /* local to host (pipes, portals) */
#define AF_INET       2  /* internetwork: UDP, TCP, etc. */
#define AF_IMPLINK    3  /* arpanet imp addresses */
#define AF_PUP        4  /* pup protocols: e.g. BSP */
#define AF_CHAOS      5  /* mit CHAOS protocols */
#define AF_NS         6  /* XEROX NS protocols */
#define AF_NBS        7  /* nbs protocols */
#define AF_ECMA       8  /* european computer manufacturers */
#define AF_DATAKIT    9  /* datakit protocols */
#define AF_CCITT     10  /* CCITT protocols, X.25 etc */
#define AF_SNA       11  /* IBM SNA */
#define AF_DECnet    12  /* DECnet */
#define AF_DLI       13  /* Direct data link interface */
#define AF_LAT       14  /* LAT */
#define AF_HYLINK    15  /* NSC Hyperchannel */
#define AF_APPLETALK 16  /* Apple talk */
#define AF_BSC       17  /* BISYNC 2780/3780 */
#define AF_DSS       18  /* Distributed system services */
#define AF_OSI       19  /* OSI Protocols */
#define AF_NETMAN    20  /* Phase V network management */
#define AF_X25       21  /* X25 protocols */
#define AF_MAX       22
```

Protocol je číslo použitého protokolu nebo nula, požadujeme-li implicitní protokol pro danou rodinu a typ. Pro Internet platí *PF_INET*. Přehled konstant pro ostatní protokoly je uveden v následující tabulce.

```
#define PF_UNSPEC    AF_UNSPEC
#define PF_UNIX      AF_UNIX
#define PF_INET      AF_INET
#define PF_IMPLINK   AF_IMPLINK
#define PF_PUP       AF_PUP
#define PF_CHAOS     AF_CHAOS
#define PF_NS        AF_NS
#define PF_NBS       AF_NBS
#define PF_ECMA      AF_ECMA
#define PF_DATAKIT   AF_DATAKIT
```

```
#define PF_CCITT     AF_CCITT
#define PF_SNA       AF_SNA
#define PF_DECnet    AF_DECnet
#define PF_DLI       AF_DLI
#define PF_LAT       AF_LAT
#define PF_HYLINK    AF_HYLINK
#define PF_APPLETALK AF_APPLETALK
#define PF_BSC       AF_BSC
#define PF_DSS       AF_DSS
#define PF_OSI       AF_OSI
#define PF_NETMAN    AF_NETMAN
#define PF_X25       AF_X25
#define PF_MAX       AF_MAX
```

Funkce *socket* vrací buď deskriptor socketu, nebo `-1` nastala-li chyba. V globální proměnné *errno* je uložen kód chyby.

Následující tabulka tab.1 shrnuje vztah mezi rodinou protokolů, typem protokolu a protokolem pro případ Internetu. Pro ostatní rodiny protokolů existují tabulky obdobné.

rodina	typ	protokol	akt. protokol
`AF_INET`	`SOCK_DGRAM`	`IPPROTO_UDP`	udp
`AF_INET`	`SOCK_STREAM`	`IPPROTO_TCP`	tcp
`AF_INET`	`SOCK_RAW`	`IPPROTO_ICMP`	icmp
`AF_INET`	`SOCK_RAW`	`IPPROTO_RAW`	raw

Tabulka 1: Kombinace rodiny, typu a protokolu

Z uvedeného je vidět, že programátor může využívat služeb protokolu UDP (datagramové služby – přenos oddělených zpráv) i služeb protokolu TCP (služby virtuálních okruhů – pro spolehlivý přenos většího objemu dat). Pokud má programátor privilegia superuživatele (UID = 0), může dokonce pracovat na úrovni protokolu ICMP, nebo přímo na základní úrovni, bez vazby na protokoly IP, UDP nebo TCP. Toho lze využít pro realizaci systémových komunikačních programů s využitím přehlednosti, poskytované programovacím jazykem C a knihovnami socketů.

4 Vytvoření a rušení spojení

K aktivnímu vytvoření spojení slouží funkce *connect*. Většinou je volána programem klienta po vytvoření *socketu*. Programátor musí definovat a naplnit i strukturu, obsahující adresu vzdálené stanice a číslo portu. Příklady jsou uvedeny v kap. 9.1 a 9.2.

K pasivnímu vytvoření spojení pomocí virtuálních okruhů slouží příkazy *bind* a *listen*. Příkaz *bind* naplní socket vzdálenou adresou a číslem lokálního portu. Příkaz *listen* převede *socket* do stavu pasivního čekání na požadavek navázání spojení. Vlastní čekání je realizováno příkazem *accept*. Příklady jsou uvedeny v kap. 9.3 a 9.3.

Nechceme-li pasivně čekat na navázání spojení (případ zpracování asynchronních událostí viz. příkladv kap. 9.5), můžeme nejprve testovat událost příkazem *select* a teprve poté až nastane, vyvolat příkaz *accept*. Tento postup dovoluje jedním procesem zpracovávat různé události, související s ukončením činnosti nad sockety, soubory a dalšími objekty, které mají přiřazeny deskriptory.

Realizujeme-li datagramové služby, používáme pouze příkaz *bind*. Spojení nenavazujeme, data očekáváme např. příkazem *recvfrom*, který navíc vrátí i adresu vzdálené stanice.

4.1 ACCEPT

Funkce *accept* slouží k akceptování dalšího spojení prostřednictvím pasivního socketu. Z fronty požadavků vybere další požadavek na vytvoření spojení, vytvoří nový socket pro další spojení a jeho deskriptor vrátí volajícímu procesu. Není-li ve frontě požadavek na vytvoření spojení, čeká na příchod tohoto požadavku. Funkci lze použít pouze pro virtuální spojení (TCP).

Uzavření spojení je spojeno se zrušením *socketu*. Realizuje se voláním funkce *close* nebo *shutdown*.

Služby jsou popsány v abecedním pořadí.

```
int accept( socket, addr, addrlen );
int        socket;    // Popisovač socketu vytvořený funkcí
                      // socket.
sockaddr   *addr;     // Odkaz na adresní strukturu. Funkce
                      // plní strukturu IP adresou a číslem
                      // portu vzdáleného uzlu.
```

```
int       *addrlen; // Odkaz na integer, který zpočátku
                    // specifikuje velikost sockaddr, po
                    // ukončení volání určuje počet slabik
                    // adresy.
```

Sockaddr je datový typ, který má následující strukturu

```
struct sockaddr {
  u_short sa_family;    // typ adresy
                        // (pro TCP/IP je to AF_INET)
  char    sa_data[14];  // adresa
}
```

4.2 BIND

Funkce *bind* spojuje lokální IP adresu a číslo portu protokolu se socketem. Je používán zejména procesy serverů pro vytváření všeobecně známých protokolových portů. Vrací `0` je-li vše v pořádku, jinak `−1`. Globální proměnná *errno* obsahuje chybový kód.

```
int bind( socket, localaddr, addrlen );
int       socket;     // Popisovač socketu vytvořený funkcí
                      // socket.
sockaddr  *localaddr; // Adresa struktury, která specifikuje
                      // IP adresu a číslo portu protokolu.
int       addrlen;    // Velikost adresní struktury ve
                      // slabikách.
```

4.3 CLOSE

Funkce *close* ukončuje komunikaci a ruší socket. Nepřečtená data jsou ztracena. Používá se při ukončení aplikace. Sdílí-li socket více procesů, je odstraněn teprve při ukončení posledního spojení. Vrací `0` je-li vše v pořádku, jinak `−1`. Kód chyby je uložen v globální proměnné *errno*.

```
int close( socket );
int       socket;  // Popisovač socketu vytvořený funkcí
                   // socket.
```

4.4 CONNECT

Funkce *connect* dovoluje specifikovat vzdálenou adresu a číslo portu pro daný socket. Je-li typ socketu TCP, vytvoří se spojení. Je-li typ socketu UDP, specifikuje pouze vzdálený uzel, ale nic nepřenáší. Vrací `0` je-li vše v pořádku, jinak `−1`. Kód chyby je uložen v globální proměnné *errno*.

```
retcode = connect( socket, addr, addrlen );
int       socket;  // Popisovač socketu vytvořený funkcí
                   // socket.
sockaddr_in *addr; // Koncový bod komunikace ve vzdáleném
                   // počítači (viz popis datové struktury).
int       addrlen; // Délka předchozího argumentu.
```

Adresa je umístěna do následující datové struktury. Kromě vlastní adresy uzlu obsahuje také číslo *portu* požadované služby. Typ adresy je určen konstantami. Pro TCP/IP je to AF_INET.

```
struct sockaddr_in {
  u_short sin_family;   // typ adresy
  u_short sin_port;     // protocol port number
  u_long  sin_addr;     // IP adresa
  char    sin_zero[8];  // nepoužito
}
```

4.5 LISTEN

Funkce *listen* převádí *socket* do pasivního režimu, kdy čeká na příchod požadavku na navázání spojení. Párovým příkazem ve vzdáleném počítači je příkaz *connect*. Příchozí požadavek je zařazen do fronty, kde čeká na aktivaci příkazem *accept*. Funkce *listen* (a *accept*) nejčastěji volají servery, *connect* naopak klienti. Parametr *queuelen* udává délku fronty nevyřízených požadavků. Funkci *listen* lze použít pouze pro sockety typu TCP. Vrací `0` je-li volání úspěšné, jinak vrací `−1`. Chybový kód je uložen v proměnné *errno*.

```
int listen( socket, queuelen );
int  socket     // Popisovač socketu vytvořený funkcí
                // socket.
int  queuelen   // Délka fronty nevyřízených požadavků.
                // Nastavuje se podle potřeby (běžně až 5).
```

4.6 SHUT_DOWN

Používá se pro duplexní *sockety* (TCP typu). Dovoluje postupné uzavírání spojení.

```
int shutdown( socket, direction );
int  socket;    // Popisovač socketu vytvořený funkcí
                // socket.
int  direction; // Směr uzavření spojení.
```

Parametr *direction* má následující význam:

0 – ukončení vstupu.
1 – ukončení výstupu.
2 – ukončení vstupu i výstupu.

Vrací 0, pokud je vše v pořádku, jinak −1. Kód chyby je umístěn do proměnné *errno*.

5 Přenos dat

5.1 WRITE

Funkce *write* dovoluje aplikaci přenášet data do vzdáleného počítače. Vrací počet odeslaných slabik nebo −1 nastala-li chyba. Kód chyby je uložen do globální proměnné *errno*.

```
int write( socket, buf, buflen );
int   socket;  // Popisovač socketu vytvořený funkcí
               // socket.
char  *buf;    // Adresa bufferu zprávy.
int   buflen;  // Délka zprávy.
```

5.2 READ

Funkce *read* slouží ke čtení informace ze socketu. Dovoluje blokované nebo neblokované volání. Při blokovaném volání se výpočet zastaví a čeká se na data. Při neblokovaném čtení se nečeká – nejsou-li data připravena, vrací chybový kód.

```
int read( socket, buff, buflen );
int   socket     // Popisovač socketu vytvořený funkcí
                 // socket.
char  *buff      // Adresa paměti pro uložení zprávy.
int   buflen     // Délka rezervovaného místa pro zprávu
                 // ve slabikách.
```

Funkce vrací následující hodnoty:

`0` – nalezení konce přenosu
-1– indikace chyby
`>0` – počet slabik v bufferu

Je-li indikována chyba, je v proměnné *errno* uložen chybový kód.

5.3 SEND

Funkce *send* pošle zprávu do jiné stanice. Vrací počet odeslaných slabik nebo -1 při chybě. Kód chyby je zaznamenán v proměnné *errno*.

```
int send( socket, msg, msglen, flags );
int   socket;  // Popisovač socketu vytvořený funkcí
               // socket.
char  *msg;    // Adresa zprávy.
int   msglen;  // Délka zprávy.
int   flags;   // Řídicí bity, které specifikují typ
```

Řídicí bity *flags* mohou být:

MSG_OOB – vyslání nebo příjem *out–of–band* dat. Jedná se o přednostně přenášená data po virtuálním spojení (SOCK_STREAM). Při vysílání (*send*, *sendto*, *sendmsg* se data tímto způsobem označí, při příjmu (*recv*, *recvfrom*, *recvmsg*) se jedná o pokyn přečíst přednostně přenášená data. Nejsou-li k dispozici, je hlášena chyba.

MSG_PEEK – používá se pouze pro čtení (příkazy *recv*, *recvfrom*, *recvmsg*). Umožňuje podívat se na data, která jsou připravena na čtení, aniž by byly vyjmuty ze vstupní fronty. (*look ahead*).

MSG_DONTROUTE – používá se pouze při vysílání. Nastavení příznaku způsobí přenost dat bez použití směrovacích tabulek síťové vrstvy.

5.4 RECV

Funkce *recv* přijme následující zprávu z UDP socketu. Vrací počet přijatých slabik. Pokud nastane chyba, vrací `−1` a kód chyby v proměnné *errno*.

```
int recv( socket, buffer, length, flags );
int    socket     // Popisovač socketu vytvořený funkcí
                  // socket.
char   *buffer    // Adresa paměti pro zprávu.
int    length     // Délka rezervované paměti pro zprávu.
int    flags      // Řídicí bity, specifikující způsob
                  // příjmu zprávy.
```

Význam řídicích bitů je uveden na str. 14.

5.5 SEND_TO

Funkce *send_to* posílá zprávu na cílovou adresu uvedenou v parametrech. Vrací počet odeslaných slabik nebo `−1` při chybě. Kód chyby je uložen do globální proměnné *errno*.

```
int sendto( socket, msg, msglen, flags, to, tolen );
int      socket; // Popisovač socketu vytvořený funkcí
                 // socket.
char     *msg;   // Adresa bufferu zprávy.
int      msglen; // Délka zprávy.
int      flags;  // Řídicí bity, které specifikují typ
                 // zprávy.
sockadr *to;     // Ukazatel na datovou strukturu sockaddr
                 // obsahující adresu koncového bodu.
int      tolen;  // Délka adresy koncového bodu.
```

Význam řídicích bitů je uveden na str. 14.

5.6 RECV_FROM

Přijme následující zprávu z UDP socketu a zaznamená adresu odesílatele do datové struktury *from*. Používá se zejména pro aplikace typu SERVER – KLIENT, kdy v nespojovaném režimu dovolí SERVERu zpracovat adresu odesílatele, a tím poslat na zprávu odpověď. Vrací počet slabik ve zprávě nebo `−1` při chybě. Kód chyby je uložen v proměnné *errno*.

```
int recvfrom( socket, buffer, buflen, flags, from, fromlen );
   int     socket   // Popisovač socketu vytvořený funkcí
                    // socket.
   char    *buffer  // Adresa paměti pro zprávu.
   int     buflen   // Délka rezervované paměti.
   int     flags    // Řídicí bity specifikující způsob příjmu
                    // zprávy.
   sockadr *from    // Ukazatel na datovou strukturu sockaddr
                    // obsahující adresu koncového bodu
                    // zdrojového uzlu zprávy.
   int     *fromlen // Délka bufferu při volání, délka adresy
                    // při ukončení volání.
```

Význam řídicích bitů je uveden na str. 14.

5.7 SEND_MSG

Funkce *send_msg* dovoluje odeslat zprávu, sestávající z několika samostatných částí. Adresy jednotlivých částí spolu s dalšími informacemi jsou uloženy ve struktuře *msghdr*.

```
int sendmsg( socket, msg, flags );
int   socket;  // Popisovač socketu vytvořený funkcí
               // socket.
struct msghdr *msg; // Adresa struktury obsahující zprávu.
int   flags;   // Řídicí bity, specifikující typ zprávy.
```

Význam řídicích bitů je uveden na str. 14.
Struktura *msghdr* má následující tvar:

```
struct msghdr {
  caddr_t  msg_name;    // volitelná adresa
  int      msg_namelen; // velikost adresy
  struct iovec *msg_iov;// scather/gather array
  int      msg_iovlen;  // počet elementú v msg_iov
  caddr_t  msg_accrights;   // práva poslat/přijmout
  int      msg_accrghtslen; // délka předchozího pole
}
```

Struktura *iovec* obsahuje dvojice, kde *iovec_base* je adresa vyrovnávací paměti a *iov_len* je její délka.

```
struct iovec {
  caddr_t  iov_base;   // počáteční adresa bufferu
  int      iov_len;    // velikost bufferu ve slabikách
}
```

Pomocí *msg_accrights* lze přenášet mezi procesy v jednom uzlu přístupová práva k souborům. Např. lze takto předat číslo deskriptoru souboru, vytvořeného jedním procesem druhému procesu.

Funkce vrací počet odeslaných slabik nebo `-1` při výskytu chyby. Kód chyby je uložen do globální proměnné *errno*.

5.8 RECV_MSG

Přijme další zprávu z UDP socketu a umístí ji do struktury typu *msghdr*. Dovoluje přijímat zprávy, sestávající z několika samostatných částí, které jsou ukládány podle adres, uvedených v položce *iovec* výše uvedené struktury.

Vrací počet přijatých slabik ve zprávě nebo `-1` při chybě. Kód chyby obsahuje proměnná *errno*.

```
int recvmsg( socket, msg, flags );
int  socket        // Popisovač socketu vytvořený funkcí
                   // socket.
struct msghdr *msg // Adresa datové struktury pro příjem
                   // zprávy.
int  flags         // Řídicí bity, určení způsobu
                   // příjmu zprávy.
```

Význam řídicích bitů je uveden na straně 14. Datová struktura *msghdr* je popsána na str.16.

6 Řídicí operace nad sockety

Dále uvedené funkce dovolují nastavovat parametry otevřených socketů. Při podrobnějším studiu zjistíte, že stejného efektu lze dosáhnout různými způsoby. Ve většině jednoduchých aplikací není třeba parametry socketu nastavovat.

6.1 FCNTL

Systémové volání *fcntl* lze použít ke změně parametrů již otevřeného socketu.

```
int fcntl( fd, cmd, arg );
int  fd;    // deskriptor socketu
int  cmd;   // příkaz
int  arg;   // parametr příkazu
```

Příkazy povolené pro sockety jsou shrnuty v tabulce tab. 2.

příkaz (cmd)	parametr (par)
F_SETOWN	UID GID
F_GETOWN	UID GID
F_GETFL	viz tab. 2
F_SETFL	viz tab. 2

Tabulka 2: Tabulka příkazů *fcntl*

Význam jednotlivých příkazů (*cmd*), uvedených v tab. 2 je následující:

F_SETOWN – dovoluje nastavení UID nebo GID procesu nebo skupiny procesů, kterým bude předán signál SIGIO nebo SIGURG při příjmu socketu s popisovačem *fd*. Argumentem *arg* může být buď kladné číslo (pak představuje UID), nebo záporné číslo (pak v absolutní hodnotě představuje GID).

F_GETOWN – dovoluje procesu zjistit UID a GID, jejichž hodnoty vrací jako výsledek volání funkce. Kódování je stejné jako v předchozím případě. Hodnota -1 znamená chybu.

F_GETFL – pomocí příkazu lze testovat nastavení příznakových bitů souboru (socketu).

F_SETFL – pomocí příkazu lze nastavovat příznakové bity souboru (socketu).

Argumenty příkazu F_GETFL a F_SETFL jsou uvedeny v tab. 3.

Význam argumentů je následující:

FAPPEND – každý zápis je proveden na konec souboru.

argument (arg)	význam argumentu
FAPPEND	přidá zápis na konec souboru
FASYNC	signalizuje skupině procesů připravenost dat
FCREAT	pokud neexistuje, vytvoří nový soubor
FEXCL	hlásí chybu, pokud již soubor existuje
FNDELAY	povolení neblokované operace
FTRUNC	zkracení souboru na nulovou délku

Tabulka 3: Tabulka argumentů F_GETFL a F_SETFL

FCREAT – pokud neexistuje, je soubor založen.

FEXCL – jestliže soubor existuje, hlásí chybu.

Pro manipulaci se sockety mají význam zejména následující argumenty.

FASYNC – tato volba dovoluje přijímat asynchronní v/v signály.

FNDELAY – nastavuje operace nad socketem do režimu "*neblokované*", t.j. pokud nemůže být operace nad socketem ukončena okamžitě, není provedena vůbec. Nastane-li tato situace, vrací systém v proměnné *errno* chybové hlášení EWOULDBLOCK. Nastavení má vliv na systémová volání *accept*, *connect*, *read*, *readv*, *recv*, *recvfrom*, *recvmsg*, *send*, *sendto*, *sendmsg*, *write* a *writev*. Při použití příkazu *connect* pro navázání virtuálního spojení skončí příkaz okamžitě s návratovým kódem EINPROGRESS. Při provádění výstupních operací v "neblokovaném" režimu se provede pro sockety typu SOCK_STREAM částečný zápis. Pro datagramové sockety se buď zaplní daty celý socket najednou, nebo se nezapíše vůbec nic.

FTRUNC – zkrátí soubor na nulovou délku.

6.2 IOCTL

Pomocí funkce *ioctl* lze měnit parametry otevřeného socketu (obdoba funkce *fcntl*, *setsockopt* a *getsockopt*).

```
int ioctl( fd, request, arg );
int    fd;        // popisovač socketu
```

```
u_long request;  // požadavek
char  *arg;      // argument
```

Požadavky jsou rozděleny do čtyř kategorií, podle typu deskriptoru a požadované operace. Typ parametru *arg*, definovaný v proceduře jako *char* ∗, se liší podle zadaného příkazu. Situaci ilustruje následující tabulka (tab. 4).

V následující části jsou vysvětleny položky tab.4.

FIOCLEX – nastaví příznak *close–on–exec* podle hodnoty nejnižšího bitu *arg*. Je-li příznak nastaven, je soubor uzavírán při systémovém volání *exec*.

FIONCLEX – nuluje příznakový bit *close–on–exec*.

FIONBIO – nastavuje/nuluje příznakový bit "*neblokované i/o*" podle hodnoty nejnižšího bitu *arg*.

FIOASYNC – nuluje/nastavuje příznakový bit dovolující příjem signálu SIGIO.

FIONREAD – vrací v argumentu počet slabik, které je možné číst ze souboru, příslušnému *fd*.

FIOSETOWN – nastavuje UID nebo GID specifikovaného socketu. Toto nastavení dovoluje přijímat procesem nebo skupinou procesů asynchronní signály SIGIO a SIGURG.

FIOGETOWN – čti UID nebo GID socketu.

SIOCATMARK – slouží k indikaci nastavení ukazovátka na začátek dat OOB (*out–of–band*). Je-li na řadě příjem uvedených dat, vrací nenulu.

SIOCSPGRP – je ekvivalentní FIOSETOWN.

SIOGPGRP – je ekvivalentní FIOGETOWN.

SIOCADDRT – přidání položky do směrovací tabulky.

SIODELRT – mazání položky ze směrovací tabulky.

SIOCGIFADDR – vrací adresu interface.

SIOCSIFADDR – nastavuje adresu interface.

kategorie	požadavek	význam	datový typ
file	`FIOCLEX`	výhradný přístup k fd	
	`FIONCLEX`	uvolnění přístupu k fd	
	`FIONBIO`	povol/zakaž neblokové i/o	int
	`FIOASYNC`	povol/zakaž asynchronní i/o	int
	`FIONREAD`	vrací počet slabik připravených ke čtení	int
	`FIOSETOWN`	nastav vlastníka	int
	`FIOGETOWN`	čti vlastníka	int
socket	`SIOCATMARK`	indikace dat OOB	int
	`SIOCSPGRP`	nastav skupinu procesů	int
	`SIOGPGRP`	čti skupinu procesů	int
routing	`SIOCADDRT`	přidej cestu	struct rtentry
	`SIODELRT`	zruš cestu	struct rtentry
interface	`SIOCSIFADDR`	nastav adresu interface	struct ifreq
	`SIOCGIFADDR`	čti adresu interface	struct ifreq
	`SIOCSIFFLAGS`	nastav přízn. interface	struct ifreq
	`SIOCGIFFLAGS`	čti příznaky interface	struct ifreq
	`SIOCGIFCONF`	čti seznam příznaků	struct ifconf
	`SIOCSIFDSTADDR`	nastav síťovou adresu	struct ifreq
	`SIOCGIFDSTADDR`	čti síťovou adresu	struct ifreq
	`SIOCGIFBRDADDR`	čti broadcast adresu	struct ifreq
	`SIOCSIFBRDADDR`	nastav bcast adresu	struct ifreq
	`SIOCGIFNETMASK`	čti masku síťové adresy	struct ifreq
	`SIOCSIFNETMASK`	nastav masku adresy	struct ifreq
	`SIOCGIFMETRIC`	čti metriku směrování	struct ifreq
	`SIOCSIFMETRIC`	nastav metr. směrování	struct ifreq
	`SIOCSARP`	nastav položku ARP	struct arpreq
	`SIOCGARP`	čti položku ARP	struct arpreq
	`SIOCDARP`	vymaž položku ARP	struct arpreq

Tabulka 4: Tabulka parametrů *ioctl*

SIOCSIFFLAGS – nastavuje příznaky interface. Tyto příznaky indikují např. podporu broadcast, dvoubodový spoj, nečinnost interface, ...

SIOCGIFFLAGS – čtení příznaků interface.

SIOCGIFCONF – čte celou konfiguraci interface do struktury *ifreq*.

SIOCSIFDSTADDR – nastavuje koncovou adresu interface.

SIOCGIFDSTADDR – čte koncovou adresu interface.

SIOCSIFBRDADDR – nastavuje broadcast adresu interface.

SIOCGIFBRDADDR – čte broadcast adresu interface.

SIOCSIFNETMASK – nastavuje masku síťové adresy.

SIOCGIFNETMASK – čte masku síťové adresy.

SIOCGIFMETRIC – čte metriku směrování interface.

SIOCSIFMETRIC – nastavuje metriku směrování pro interface.

SIOCSARP – nastavuje položku ARP.

SIOCGARP – čte položku ARP.

SIOCDARP – maže položku ARP.

6.3 GET_SOCK_OPT

Funkce *getsockopt* dovoluje získat hodnotu parametru socketu nebo protokolu, který je se socketem spojen. Parametr *level* určuje úroveň protokolu (IP, TCP, SOCKET), na které mají být modifikovány parametry. Vlastní typ parametru se zadává v *opt*. Adresa hodnoty parametru je uložena v *optval*. Protože typ parametru závisí na hodnotě *level* i *opt*, je typ *optval* specifikován univerzálně jako *char* $*$. Délka *optval* je uvedena v *optlen*.

Funkce vrací `0` je-li volání úspěšné, jinak vrací $-$`1`. Kód chyby je uložen v globální proměnné *errno*.

```
int getsockopt( socket, level, opt, optval, optlen );
int       socket;  // Popisovač socketu vytvořený funkcí
                   // socket.
int       level;   // Číslo určující úroveň protokolu.
```

```
int       opt;     // Číslo identifikující parametr.
char      *optval; // Adresa bufferu pro uložení parametru.
int       *optlen; // Délka pole optval při vyvolání funkce,
                   // skutečná délka po ukončení volání.
```

Volitelné parametry socketu (*opt*) jsou shrnuty v tabulce tab. 5.

úroveň protokolu	jméno parametru	význam parametru	datový typ
IPPROTO_IP	IP_OPTIONS	parametry IP záhlaví	int
IPPROTO_TCP	TCP_MAXSEG	max. velikost TCP segmentu	int
	TCP_NODELAY	vysílání bez zpoždění	int
SOL_SOCKET	SO_BROADCAST	povolení vysílání bcast	int
	SO_DEBUG	záznam trasování	int
	SO_DONTROUTE	nepoužij směrování	int
	SO_ERROR	vrací stav a nuluje chybu	int
	SO_KEEPALIVE	udržení spojení	int
	SO_LINGER	pozdržení ukončení spojení	struct linger
	SO_OOBINLINE	uvolnění přijatých OOB dat (out–of–band) viz. kap. 5.3	int
	SO_RCVBUF	velikost paměti pro příjem socketu	int
	SO_SNDBUF	velikost paměti pro vyslání socketu	int
	SO_REUSEADDR	povolení znovupoužití lokální adresy	int
	SO_TYPE	čti typ socketu	int

Tabulka 5: Tabulka parametrů socketu

Datová struktura *linger* má následující tvar:

```
struct linger {
  int    l_onoff;   // nula -- vypnuto, nenula -- zapnuto
```

```
    int    l_linger;  // čas v sekundách
  }
```

V následujícím textu jsou popsány parametry socketů, které lze nastavit pomocí funkce *setsockopt* a číst pomocí funkce *getsockopt*. Navíc byly vybrány pouze ty, které souvisí s implementací socketů v prostředí Internetu (TCP/IP).

IPPROTO_IP – dovoluje zasahovat do struktury paketu na úrovni IP.

- **IP_OPTIONS** – dovoluje zapsat volitené parametry do záhlaví IP paketu. Funkce slouží i ke čtení těchto parametrů. Její použití vyžaduje znalost struktury IP.

IPPROTO_TCP – dovoluje zasahovat do struktury segmentu na úrovni TCP.

- **TCP_MAXSEG** – vrací maximální délku segmentu použitelnou pro socket. Pro 4.3 BSD sockety bývá nastavena na 1024 slabik. Hodnotu nelze měnit, pouze číst.
- **TCP_NODELAY** – používá se pro služby virtuálních okruhů. Dovoluje změnit chování TCP vrstvy tak, že při vysílání krátkých zpráv nečeká na potvzení předchozí zprávy, ale vyšle zprávu okamžitě.

SOL_SOCKET – dovoluje měnit parametry, související s přenosem socketů.

- **SO_BROADCAST** – povoluje/zakazuje procesu vysílání zpráv typu broadcast. Tato volba je účinná v sítích, které vysílání zpráv se všeobecnou adresou podporují na úrovni technického vybavení.
- **SO_DEBUG** – povoluje/zakazuje ukládání ladicí informace, související s vysíláním a příjmem socketů na úrovni jádra systému.
- **SO_DONTROUTE** – specifikuje, že vysílané zprávy nemají používat zabudovaný mechanizmus směrování. Zpráva je směrována podle síťové části adresy.
- **SO_ERROR** – vrací volajícímu obsah proměnné *so_error*, která obsahuje chybový kód pro sockety. Proměnná je poté jádrem nulována.
- **SO_KEEPALIVE** – povoluje periodické přenosy na připojeném socketu pokud nejsou vyměňována jiná data. Jestliže druhá strana neodpovídá na tyto zprávy, je spojení považováno za přerušené a v proměnné *so_error* je to indikováno chybou ETIMEDOUT.

SO_LINGER – tato volba říká, co udělat se zprávami, které nebyly odeslány v okamžiku volání funkce *close*. Implicitně se *close* ukončí okamžitě a systém se pokusí doručit všechna nedoručená data. Je-li volba nastavena, závisí akce na obsahu proměnné typu *struct linger*. Jestliže je položka *l_linger* nulová, jsou neodeslaná data v okamžiku volání *close* ztracena. Je-li nenulová, odešlou se všechna data.

SO_OOBINLINE – nastavení příznaku specifikuje, že upřednostněná data (*out–of–band data*) budou umístěna v normální vstupní frontě. Při jejich čtení není třeba použít příznak MSG_OOB (viz. str. 14).

SO_RCVBUF – specifikuje velikost přijímací fronty pro sockety.

SO_SNDBUF – specifikuje velikost vysílací fronty pro sockety.

SO_REUSEADDR – dává systému příkaz aby znovupoužil v lokální adrese totéž číslo portu.

SO_TYPE – vrací typ socketu (SOCK_STREAM nebo SOCK_DGRAM).

6.4 SET_SOCK_OPT

Funkce *setsockopt* dovoluje aplikaci měnit parametry socketu nebo protokolů s ním spojeným.

```
int setsockopt( socket, level, opt, optval, optlen );
int   socket;   // Popisovač socketu vytvořený funkcí
                // socket.
int   level;    // Úroveň protokolu.
int   opt;      // Číselné označení parametru (viz. níže).
char  *optval;  // Adresa uložení parametru.
int   optlen;   // Délka parametru.
```

Parametry *level* a *opt* jsou uvedeny v tab. 3 na straně 23.

Vrací 0, je-li vše v pořádku, jinak −1. Kód chyby obsahuje globální proměnná *errno*.

7 Práce s časem

Funkce patří mezi doplňkové, se *sockety* souvisí pouze potud, že ji používají některé demonstrační programy. Zde je uvedena pro úplnost.

7.1 GET_TIME_OF_DAY

Funkce *gettimeofday* vybere informaci o běžném čase a datumu podle lokální časové zóny. Vrací `0` je-li volání úspěšné, jinak vrací `-1`. Chybový kód je uložen v proměnné *errno*. Informace je uložena v následujících datových strukturách:

```
int gettimeofday( tm, tmzone );
struct timeval *tm;      // Adresy níže uvedených datových
struct timezone *tmzone; // struktur.
```

Datové struktury *timeval* a *timezone* jsou definovány následovně:

```
struct timeval {
  long tv_sec;          // počet sekund od 1.1.1970
  long tv_usec;         // počet us běžné sekundy
}
struct timezone {
  int  tz_minuteswest; // počet minut západně od
                       // Greenwichského poledníku
  int  tz_dsttime;     // typ aplikované korekce
}
```

8 Multiplexní zpracování asynchronních událostí

Funkce *select* je opět funkce univerzální, dovolující realizovat v programmech asynchronní zpracování událostí, souvisejících s ukončením operací nad deskriptory socketů, souborů a pod. Zde je uvedena proto, že je použita v příkladu 9.5.

8.1 SELECT

Funkce *select* umožňuje multiplexní zpracování asynchronních V/V. Povolí procesu čekat na připravenost jednoho z deskriptorů souboru specifikované množiny. Množina deskriptorů je zadána bitově. Parametr *numfds* určuje počet deskriptorů, které je třeba testovat. Parametry *refds*, *wrfds* a *exfds* představují odkazy na bitové pole masek jednotlivých deskriptorů. Příkaz *select* akceptuje událost pouze v případě, že je odpovídající bit v masce pro danou událost nastaven.

Volající může též určit maximální dobu čekání. Je-li hodnota parametru *timeval* nenulová, skončí příkaz nejdéle po vyčerpání nastaveného časového kvanta. Je-li hodnota časového kvanta rovna nule, skončí okamžitě. Tento režim je vhodný pro čtení stavu selektorů metodou výběru (*pooling*). Nulová hodnota odkazu indikuje čekání bez časového omezení.

```
int select( numfds, refds, wrfds, exfds, time );
int       numfds; // Počet deskriptorů souborů v množině.
fd_set    *refds; // Deskriptory souborů pro vstup.
fd_set    *wrfds; // Deskriptory souborů pro výstup.
fd_set    *exfds; // Deskriptory souborů pro výjimky.
struct timeval *time; //  Maximální doba čekání,
                  // odkaz na strukturu obsahující nulu
                  // nebo nulový odkaz.
```

Datový typ *timeval* má následující strukturu:

```
struct timeval {
  long tv_sec;          // počet sekund od 1.1.1970
  long tv_usec;         // počet us běžné sekundy
}
```

Funkce vrací počet připravených deskriptorů nebo −1 v případě chyby. Chybový kód je pak uložen v proměnné *errno*.

Pro manipulaci s deskriptory souborů jsou z důvodu kompatibility k dispozici následující makra:

```
FD_ZERO( fd_set *fdset );
FD_SET( int fd, fd_set *fdset );
FD_CLR( int fd, fd_set *fdset );
FD_ISSET( int fd, fd_set *fdset );
```

jejich význam je následující:

FD_ZERO – nuluje celou množinu příznaků deskriptorů.

FD_SET – nastavuje příznak v masce deskriptoru.

FD_CLR – nuluje příznak v masce deskriptoru.

FD_ISSET – testuje připravenost deskriptoru.

Maximální počet deskriptorů je závislý na implementaci. Bývá omezen na 32.

9 Příklady

9.1 UDP klient

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <string.h>
#include <stdio.h>

#define MY_PORT 2222
#define BUFF_LEN 40
#define  MSG "Hi !"

int main(int argc, char **argv)
{
  int udp_socket;
  char buff[BUFF_LEN];
  struct sockaddr_in sin;
  struct hostent *p_hent;

/*parametry prikazove radky*/
  if(argc == 1) {
    printf("\nUsage : %s hostname\n", argv[0]);
    exit(1);
  }

/*naplneni struktury sin*/
  sin.sin_family = AF_INET;
  sin.sin_port = htons((u_short)MY_PORT);
  if((p_hent = gethostbyname(argv[1])) != NULL)
    memcpy( (char *)&(sin.sin_addr), p_hent->h_addr,
                                        p_hent->h_length);
  else {
    fprintf(stderr,
```

```
        "\nCLIENT ERROR : Cannot find host %s\n", argv[1]);
    exit(1);
  }

/*alokace socketu*/
  if((udp_socket = socket(PF_INET, SOCK_DGRAM,
                                     IPPROTO_UDP)) < 0) {
    fprintf(stderr,
              "\nCLIENT ERROR : Cannot allocate socket\n");
    exit(1);
  }
/*connect socketu*/
  if(connect(udp_socket, (struct sockaddr *)&sin,
                                      sizeof(sin)) < 0) {
    fprintf(stderr,
        "\nCLIENT ERROR : Cannot connect socket to host %s \
                              port %d\n", argv[1], MY_PORT);
    exit(1);
  }

  write(udp_socket, MSG, sizeof(MSG));
  read(udp_socket, buff, BUFF_LEN);
  printf("\nREPPLY FROM HOST %s: %s\n", argv[1], buff);
  close(udp_socket);
}
```

9.2 TCP klient

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <string.h>
#include <stdio.h>

#define MY_PORT 2222
```

```
#define BUFF_LEN 40

int main(int argc, char **argv)
{
  int tcp_socket;
  char buff[BUFF_LEN];
  struct sockaddr_in sin;
  struct hostent *p_hent;

/*parametry prikazove radky*/
  if(argc == 1) {
    printf("\nUsage : %s hostname\n", argv[0]);
    exit(1);
  }

/*naplneni struktury sin*/
  sin.sin_family = AF_INET;
  sin.sin_port = htons((u_short)MY_PORT);
  if((p_hent = gethostbyname(argv[1])) != NULL)
    memcpy( (char *)&(sin.sin_addr), p_hent->h_addr,
                                      p_hent->h_length);
  else {
    fprintf(stderr,
        "\nCLIENT ERROR : Cannot find host %s\n", argv[1]);
    exit(1);
  }

/*alokace socketu*/
  if((tcp_socket = socket(PF_INET, SOCK_STREAM,
                                IPPROTO_TCP)) < 0) {
    fprintf(stderr,
               "\nCLIENT ERROR : Cannot allocate socket\n");
    exit(1);
  }
/*connect socketu*/
  if(connect(tcp_socket, (struct sockaddr *)&sin,
                                      sizeof(sin)) < 0) {
```

```
    fprintf(stderr,]
          "\nCLIENT ERROR : Cannot connect socket to host %s \
                   port %d\n", argv[1], MY_PORT);
    exit(1);
  }
/*cteni ze socketu*/
  read(tcp_socket, buff, BUFF_LEN);
  printf("\nREPPLY FROM HOST %s: %s\n", argv[1], buff);
  close(tcp_socket);
}
```

9.3 UDP server

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

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

#define BUFF_LEN 40
#define MY_PORT 2222

int main()
{
  int sock;
  char buff[BUFF_LEN];
  int alen;
  struct sockaddr_in sin;

/*naplneni struktury sin*/
  sin.sin_family = AF_INET;
  sin.sin_port = htons((u_short)MY_PORT);
  sin.sin_addr.s_addr = INADDR_ANY;
```

```
/*alokace socketu*/
  if((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
    fprintf(stderr,
             "\nUDP_SERVER ERROR : Cannot allocate socket");
    exit(1);
  }
/*bind socketu*/
  if(bind(sock, (struct sockaddr_in *)&sin, sizeof(sin)) < 0) {
    fprintf(stderr,"\nUDP_SERVER ERROR : Cannot bind socket");
    exit(1);
  }

  while(1) {
    alen = sizeof(sin);
    recvfrom(sock, buff, BUFF_LEN, 0, (struct sock_addr *)
                  &sin, &alen);

    strcpy(buff, "UDP_SERVER sends you many greetings");

    if(sendto(sock, buff, BUFF_LEN, 0, (struct sock_addr *)
                  &sin, sizeof(sin)) < 0)
      perror("SERVER ERROR :");
  }
}
```

9.4 TCP server

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

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

#define QLEN 5
```

```
#define BUFF_LEN 40
#define MY_PORT 2222

int post()
{
  union wait status;
  while(wait(&status, WNOHANG, (struct rusage *)0) >= 0)
    ;
}


int main()
{
  int master_socket, slave_socket;
  char buff[BUFF_LEN];
  int alen;
  struct sockaddr_in sin;

/*naplneni struktury sin*/
  sin.sin_family = AF_INET;
  sin.sin_port = htons((u_short)MY_PORT);
  sin.sin_addr.s_addr = INADDR_ANY;

/*alokace socketu*/
  if((master_socket = socket(PF_INET, SOCK_STREAM,
                                      IPPROTO_TCP)) < 0) {
    fprintf(stderr,
             "\nTCP_SERVER ERROR : Cannot allocate socket");
    exit(1);
  }
/*bind socketu*/
  if(bind(master_socket, (struct sockaddr_in *)&sin,
                                        sizeof(sin)) < 0) {
    fprintf(stderr,"\nTCP_SERVER ERROR : Cannot bind socket");
    exit(1);
  }
/*listen socketu*/
```

```
  if(listen(master_socket, QLEN) < 0) {
    fprintf(stderr,
        "\nTCP_SERVER ERROR : Cannot listen socket on port %d",
           MY_PORT);
    exit(1);
  }

  alen = sizeof(sin);
  strcpy(buff, "TCP_SERVER sends you many greetings");
/*  signal(SIGCLD, SIG_IGN);*/
  while(1) {
    slave_socket = accept(master_socket, (struct sock_addr *)
                             &sin, &alen);
    /*vytvoreni dalsiho procesu*/
    if(fork() != 0) /*otec*/
      close(slave_socket);
    else { /*syn*/
      close(master_socket);
      write(slave_socket, buff, BUFF_LEN);
      close(slave_socket);
      exit(0);
    }
  }
}
```

9.5 Použití příkazu select

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/select.h>
#include <sys/time.h>

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

```
#define QLEN 5
#define BUFF_LEN 40
#define MY_PORT 2222

int main()
{
  int tcp_socket, udp_socket;
  char buff[BUFF_LEN];
  int alen;
  struct sockaddr_in sin;
  int nfds;
  fd_set rfds;

/*naplneni struktury sin*/
  sin.sin_family = AF_INET;
  sin.sin_port = htons((u_short)MY_PORT);
  sin.sin_addr.s_addr = INADDR_ANY;

/*alokace tcp socketu*/
  if((tcp_socket = socket(PF_INET, SOCK_STREAM,
                                 IPPROTO_TCP)) < 0) {
    fprintf(stderr,
            "\nSERVER ERROR : Cannot allocate tcp socket");
    exit(1);
  }
/*bind tcp socketu*/
  if(bind(tcp_socket, (struct sockaddr_in *)&sin,
                                     sizeof(sin)) < 0) {
    fprintf(stderr,
                 "\nSERVER ERROR : Cannot bind tcp socket");
    exit(1);
  }
/*listen tcp socketu*/
  if(listen(tcp_socket, QLEN) < 0) {
    fprintf(stderr,
      "\nSERVER ERROR : Cannot listen tcp socket on port %d",
                        MY_PORT);
```

```
    exit(1);
  }

/*alokace udp socketu*/
  if((udp_socket = socket(PF_INET, SOCK_DGRAM,
                                            IPPROTO_UDP)) < 0) {
    fprintf(stderr,
               "\nSERVER ERROR : Cannot allocate udp socket");
    exit(1);
  }
/*bind udp socketu*/
  if(bind(udp_socket, (struct sockaddr_in *)&sin,
                                             sizeof(sin)) < 0) {
    fprintf(stderr,"\nSERVER ERROR : Cannot bind udp socket");
    exit(1);
  }
/*pocet deskriptoru, ktere se maji prohlizet*/
  nfds = (tcp_socket > udp_socket ? tcp_socket:udp_socket)+1;
/*vynulovani masky read udalosti*/
  FD_ZERO(&rfds);

  alen = sizeof(sin);
  strcpy(buff, "TCP/UDP SERVER sends you many greetings");
  while(1) {
    FD_SET(tcp_socket, &rfds);
    FD_SET(udp_socket, &rfds);

    if(select(nfds, &rfds, (fd_set *)0, (fd_set *)0,
       (struct timeval *)0) < 0) {
      fprintf(stderr,"\nSERVER ERROR : select()");
      exit(1);

    }
    if (FD_ISSET(tcp_socket, &rfds)) {
      int slave_socket;

      slave_socket = accept(tcp_socket, (struct sock_addr *)
```

```
                             &sin, &alen);
      write(slave_socket, buff, BUFF_LEN);
      close(slave_socket);
    }
    if (FD_ISSET(udp_socket, &rfds)) {
      recvfrom(udp_socket, buff, BUFF_LEN, 0,
                              (struct sock_addr *) &sin, &alen);
      strcpy(buff, "TCP/UDP SERVER sends you many greetings");
      if(sendto(udp_socket, buff, BUFF_LEN, 0,
                   (struct sock_addr *) &sin, sizeof(sin)) < 0)
        perror("SERVER ERROR :");
    }
  }
}
```

9.6 Klient TCP echo

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <string.h>
#include <stdio.h>

#define QLEN 5
#define ECHO_PORT 7
#define BUFF_LEN 40

int main(int argc, char **argv)
{
  int tcp_socket;
  char buff[BUFF_LEN] = "Zdravim te !";
  struct sockaddr_in sin;
  struct hostent *p_hent;

/*parametry prikazove radky*/
```

```
  if(argc == 1) {
    printf("\nUsage : %s hostname\n", argv[0]);
    exit(1);
  }

/*naplneni struktury sin*/
  sin.sin_family = AF_INET;
  sin.sin_port = htons((u_short)ECHO_PORT);
  if((p_hent = gethostbyname(argv[1])) != NULL)
    memcpy( (char *)&(sin.sin_addr), p_hent->h_addr,
             p_hent->h_length);
  else {
    fprintf(stderr,
        "\nECHO CLIENT ERROR : Cannot find host %s\n",
        argv[1]);
    exit(1);
  }

/*alokace socketu*/
  if((tcp_socket = socket(PF_INET, SOCK_STREAM,
         IPPROTO_TCP)) < 0) {
    fprintf(stderr,
         "\nECHO CLIENT ERROR : Cannot allocate socket\n");
    exit(1);
  }
/*connect socketu*/
  if(connect(tcp_socket, (struct sockaddr *)&sin,
             sizeof(sin)) < 0) {
    fprintf(stderr,
         "\nECHO CLIENT ERROR : Cannot connect socket to  \
             host %s port %d\n", argv[1], ECHO_PORT);
    exit(1);
  }
/*zapis do socketu*/
  write(tcp_socket, buff, BUFF_LEN);
/*cteni ze socketu*/
  read(tcp_socket, buff, BUFF_LEN);
```

```
  printf("\nREPPLY FROM HOST %s: %s\n", argv[1], buff);
  close(tcp_socket);
}
```

Obsah

Seznam tabulek