H.O.#4
COMP 362
Gary Chan
Fall 2003

Network Programming

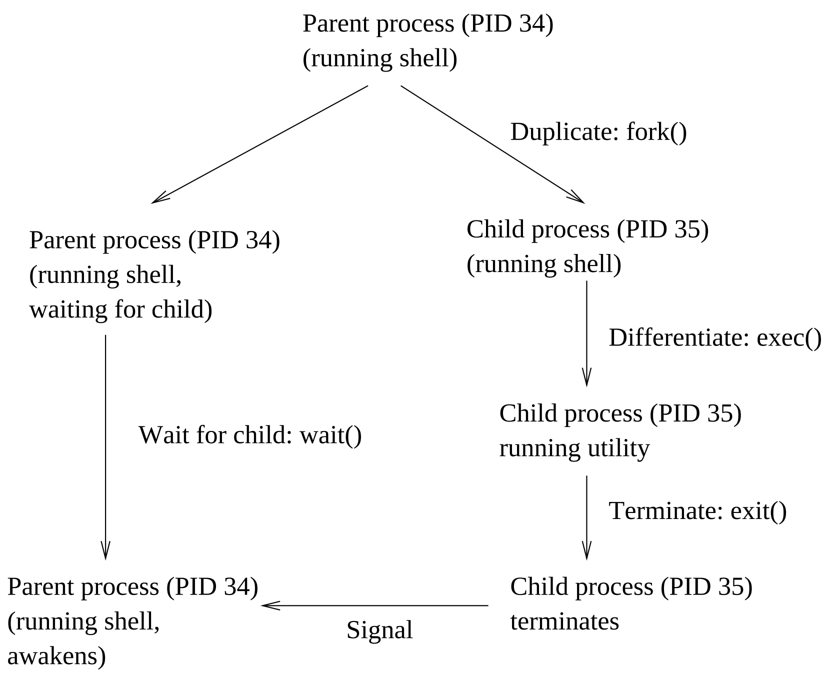
- Process fundamentals
  - Process management: forking and zombies
  - Signals
  - External data representation (XDR)
- Interprocess communications
  - Unnamed pipes
  - Named pipes (FIFO)
- TCP socket programming
  - Client programming
  - Server programming
- UDP socket programming
- Advanced topics
  - I/O multiplexing
  - Getting and setting socket options
  - Broadcasting and multicasting

Programming Fundamentals: Process Management

- When UNIX is first started, there is only one visible process in the system, the “init” process with PID 1
  - To create more processes, “init” duplicates itself
  - “init” is the ancestor of all subsequent processes
- When a process duplicates, the parent and child processes are identical in every way except their PID
  - The child’s code, data, and stack are a copy of the parent’s, and they continue to execute the same code
  - A child process may, however, replace its code with that of another executable file, thereby differentiating itself from its parent
  - When a child process terminates, its death is communicated to its parent so that the parent may take some appropriate action

An Example

- A shell executing a utility in the foreground
- When the child dies, the parent process awakens and presents the user with another shell prompt
- Important calls: fork, exit, wait, exec, getpid, getppid, etc.



Zombie Process

- A process that terminates cannot leave the system until its parent accepts its return code
- If its parent is already dead, it will be adopted by “init” which always accept its children’s return code
- However, if its parent is alive but never execute a wait(), the process’s return code will never be accepted and the process will remain a zombie which costs space in the process table

Function Calls

- `int fork()`
  - Duplicate a process
  - On success, returns the PID of the child to the parent process, and returns 0 to the child process
  - On failure, returns -1 to the parent process and no child is created
- `int getpid(), int getppid()`
  - Returns a process's id and parent process's id numbers respectively
  - Always succeed
- `int exit(int` *status*`)`
  - When a child process terminates, it sends its parent a SIGCHLD signal and waits for its termination code *status* to be accepted
  - A process that is waiting for its parent to accept its return code is called a zombie process
  - A parent accepts a child's termiation code by executing wait()
  - All terminating process's children are orphaned and adopted by "init" in the kernal by setting their PPID to 1
  - exit() never returns any value

- `int wait(int *` *status*`)`
  - Causes a process to suspend until one of its children terminates
  - On success, returns the pid of the child that terminated and places a status code into *status*
    - If the rightmost byte of status is zero, the leftmost byte contains the low eight bits of the value returned by the child's exit()/return() (To get the exit code, use `status>>8`)

```
+--------------------+
| Exit code |    0   |
+--------------------+
```

    - If the rightmost byte is non-zero, the rightmost *seven* bits are equal to the number of the signal that caused the child to terminate; if the child produced a core dump, the remaining bit of the rightmost byte is set to 1

```
+------------------------+
|           |  Non-zero  |
+------------------------+
```

  - If a process executes a wait() but has no child, wait() returns immediately with -1
  - If a process executes a wait() and one or more of its children are already zombies, wait() returns immediately with the status of one of the zombies

Program Outline for Forking

```
int pid, child_id, status;

while(1){
  if( (pid = fork()) == 0 ) /* child process */
    do_childish_things();
  else /* parent process */
    child_id = wait( &status );  /* parent process stops */
}
```

- The signal SIGCHLD is given whenever a child dies

Signals

- Dealing with unexpected or unpredictable events, i.e., interrupts
  - The death of a child process
  - An alarm clock "rings"
  - A termination request from a user
  - A suspend request from a user, etc.
- When such an event occurs, a uniquely numbered signal (1–31) corresponding to the event is sent to the process
- A programmer may arrange for a particular signal to be ignored or to be processed with signal handler
- Some common signals:
  - SIGINT (2) — interrupt (e.g., `^C` or `^Z`)
  - SIGCHLD (20) — child status has changed
  - SIGALRM (14) — alarm clock
  - SIGSTP (18) — `^Z`
  - SIGPIPE (13) — write on a pipe or other socket with no one to read it
  - SIGURG (16) — urgent condition present on socket
  - SIGCONT (19) — continue after stop

Alarm Signal

`unsigned int alarm(unsigned int` *count*`)`

- Instructs the kernel to send the SIGALRM signal to the calling process after *count* seconds. If an alarm had already been scheduled, it is overwriten. If *count* is zero, any pending alarm requests are cancelled
- Returns the number of seconds that remain until the alarm signal is sent

Signal Handler

`void (*signal(int` *sigCode*`, void (*`*func*`)()))()`

- Specify the action taken when a particular signal is received
- *sigCode* specifies the number of the signal that is to be reprogrammed, and *func* may be:
  - `SIG_IGN`: the specified signal should be ignored and discarded
  - `SIG_DFL`: the kernal's default handler should be used
  - an address of a user-defined function
- The signal numbers are in `/usr/include/signal.h`.
- With the exception of SIGCHLD, signals may not be stacked (i.e., if a process is sleeping and three identical signals are sent to it, only one signal is actually processed)
- Returns the previous *func* value associated with *sigCode* on success; otherwise it returns -1

```
old_handler = signal( SIGALRM, new_handler );
...
signal( SIGALRM, old_handler )
```

Signal Handling

`int setitimer(int` *which*`, const struct itimerval *` *value*`, struct itimerval *` *ovalue*`)`

- Set alarm periodically: *which*= `ITIMER_REAL`
- Sets the specified interval timer to the time values specified by *value* and returns the previous value in *oldvalue*. *value->it_value* will run down and a SIGALRM signal will be sent when it hits 0. The interval timer will then be loaded with the time value given in *value->it_interval*.

```
struct itimerval{
  struct timeval it_interval;  /* timer interval */
  struct timeval it_value;    /* current value -- running down unti
                         it hits zero, and then loads it_interval
};
struct timeval{
  long tv_sec;  /* seconds */
  long tv_usec; /* microseconds = 10^(-6)*/
};
```

- If *it_interval* is 0, the timer will fire once.

An Example on the Use of `setitimer`

```
/* The following program fires off alarms 5 times with sub-second
   interval by using setitimer */

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

void alarmhandler( void ){
  printf("Alarm just fires!\n");
  signal( SIGALRM, alarmhandler ); /* re-establish alarm handler */
  return;
}

main(){

  struct itimerval it, oit;
  int i;

  signal( SIGALRM, alarmhandler );

  it.it_interval.tv_sec = 0; /* fires once */
  it.it_interval.tv_usec = 0; /* fires once */
  it.it_value.tv_sec = 0; /* 0 seconds */
  it.it_value.tv_usec = 200; /* fires off after 200 microseconds*/

  for( i = 0; i < 5; i++ ){
    setitimer( ITIMER_REAL, &it, &oit );
    pause();
  }
  return;
}
```

Other Signal Handling

- `int kill(int` *pid*`, int` *sigCode*`)`
  - Sends a signal with value *sigCode* to the process with PID *pid*.
  - Returns 0 on success, and -1 otherwise
- `int pause()`
  - Suspends the calling process and returns when the calling process receives a signal

```
void alarmhandler( void ){
  return;
}

main(){

  signal( SIGALRM, alarmhandler );
  alarm( 3 );
  pause();
  printf("Just exited\n");
  return;
}
```


An Example on Handling SIGCHLD

- Limit the run-time of a command at the command line

```
hyperion:~> a.out 5 ls
a.out     limit.c   limit.c~
 Child 19976 terminated within 5 seconds
hyperion:~> a.out 4 sleep 100
Child 19978 exceeded limit and is being killed
```

Pipes

- Allow two or more processes to send information to each other
- Commonly used within shells to connect the standard output of one utility to the standard input of another
- A pipe automatically buffers the output of the writer and suspends the writer if the pipe gets too full
- If a pipe empties, the reader is suspended until some more output becomes available
- Pipes
  - Unnamed pipes
  - Named pipes (FIFO)

Unnamed Pipes

- A unidirectional communication link
  - Buffer size up to 4K (BSD) or 40K (System V)
- Each end of a pipe has an associated file descriptor
  - The write end may be written to using write()
  - The read end may be read from using read()
  - Close the pipe using close()
- Usually used for communication between a parent process and its child, with one process writing and the other reading
  1. The parent process creates an unnamed pipe by calling `pipe()`
  2. The parent process forks
  3. The writer closes its read end of the pipe, and the designated reader closes its write end of the pipe
  4. The processes communicate by using write() and read() calls
  5. Each process closes its active pipe descriptor when it's finished with it
- Bidirectional communication is only possible by using two pipes

pipe()

```
int pipe(int fd[]);

int fd[2];
pipe( fd );
```

- Creates an unnamed pipe and returns two file descriptors
  - The descriptor for the "read" end is stored in *fd[0]*
  - The descriptor for the "write" end is stored in *fd[1]*
- If a process reads from a pipe whose write end has been closed, the read() returns a 0
- If a process reads from an empty pipe whose write end is still open, it sleeps until some input becomes available (blocks for input)
- If a process writes to a pipe whose read end has been closed, the write fails and the writer is sent a SIGPIPE signal. The default action of this signal is to terminate the receiver.
- Returns 0 on success; return -1 otherwise

An Example

- A program implementing UNIX "pipe" (i.e., |): take two command line arguments and pipe the output of the first command into the input of the second command
  - Neither programs are invoked with options

```
cssu5:~> ls
connect*        pipe.ps         reader.c        writer.c
connect.c       reader*         writer*
cssu5:~> who
gchan      console      Feb 23 19:10    (:0)
gchan      pts/4        Feb 23 19:10    (:0.0)
gchan      pts/6        Feb 23 19:10    (:0.0)
gchan      pts/5        Feb 23 19:10    (:0.0)
gchan      pts/3        Feb 23 19:10    (:0.0)
gchan      pts/13       Feb 23 19:10    (:0.0)
gchan      pts/12       Feb 23 19:10    (:0.0)
gchan      pts/11       Feb 23 19:10    (:0.0)
gchan      pts/10       Feb 23 19:10    (:0.0)
gchan      pts/9        Feb 23 19:10    (:0.0)
gchan      pts/7        Feb 23 19:10    (:0.0)
gchan      pts/8        Feb 23 19:10    (:0.0)
cssu5:~> connect ls wc
       7       7      58
cssu5:~> connect who wc
      12      72     526
cssu5:~>
```

Named Pipes

- Often referred to as FIFO
- Have a name that exists in the file system
- May be used by unrelated processes
- Exist until explicitly deleted
- Larger buffer capacity, typically about 40 KB
- Created by mknod() system call
  - Specify `S_IFIFO` as the file mode

```
mknod( ‘‘myPipe’’, S_IFIFO, NULL );
chmod( ‘‘myPipe’’, 0660 );  /* chmod ug+rw */
```


Named Pipes (Cont.)

- A special file is added into the file system
  - Open a named pipe using open()
  - write() adds data at the start of the FIFO queue
  - read() removes data from the end of the FIFO queue
  - Close the named pipe with close()
  - Remove the pipe using unlink()
- A named pipe is intended for use as a unidirectional link
- If a process tries to open a named pipe for read-only and no process currently has it open for writing, the reader will wait until a process opens it for writing
- If a process tries to open a named pipe for write-only and no process currently has it open for reading, the writer will wait until a process opens it for reading
- Named pipes will not work across a network

An Example

- A single reader process is executed, which creates a named pipe. It then reads and displays NULL-terminated lines from the pipe until the pipe is closed by all of the writing processes.
- One or more writer processes are executed, each of which opens the named pipe and sends two messages to it. If the pipe does not exist when a writer tries to open it, the writer retries every second until it succeeds. When all of a writer's messages are sent, the writer closes the pipe and exits.

```
cssu5:~> reader& writer& writer& writer& writer&
[3] 21601
[4] 21602
[5] 21603
[6] 21604
[7] 21605
Hello from PID 21602
Hello from PID 21603
Hello from PID 21604
Hello from PID 21605
Hello from PID 21602
Hello from PID 21603
Hello from PID 21604
Hello from PID 21605
[4]    Exit 1                       writer
[5]    Exit 1                       writer
[6]    Exit 1                       writer
[7]    Exit 1                       writer
[3]    Exit 1                       reader
```


File Locking: lockf()

- In concurrent programming, we may have multiple processes concurrently accessing a file. If the file is in write mode, this will cause consistency problems.
- File locking can be used to ensure that only one process can be accessing the file at any time.
- One method of implementing file locking is to use the `lockf()` system call:

  ```
  int lockf(int fileid, int command, int size);
  ```

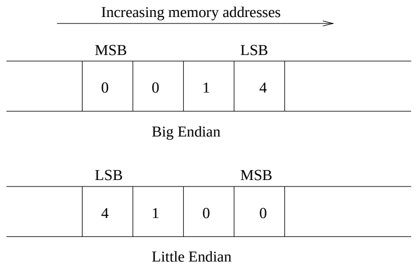
  - `fileid` is the integral open file descriptor (may be obtained from a file stream descriptor `fd` using `fileno(fd)` system call)
  - `command` is the control value to specify whether we need to lock or unlock the file
  - `size` is the amount of the file in bytes to be locked. For your purposes you can set this to 0, which means that the whole file should be locked
  - The `lockf` system call returns a 0 on success and -1 on failure

lockf(): Usage Example

```
/* find the file number corresponding to the file
 * descriptor
 */
fileid = fileno(fd);

/* lock the database file for exclusive write access */
if (lockf(fileid, F_LOCK, 0) < 0)
   errmess("Server: Error locking the database file");

fprintf(fd,"%d %s\n", number, name);
fflush(fd);

/* unlock the database file */
if (lockf(fileid, F_ULOCK, 0) < 0)
   errmess("Server: Error unlocking the database file");

fclose(fd);
```


External Data Representation (XDR)

- Each computer provides its own definition for the representation of data
  - E.g., representating 260 as a 32-bit binary integer (1 0 0 0 0 0 1 0 0)

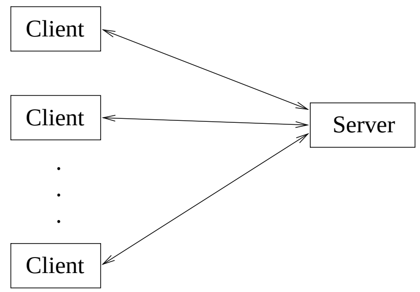


- Communication between two hosts must agree on the exact representation for all data sent across the channel
  - Network byte order
  - Both ends perform conversion to their machine format
  - The standard, machine-independent representation is known as the external data representation
  - Big-endian order

What is Network Programming?

- Writing of computer programs that can work with other programs across a network
- Client-Server programs
  - Web browsers (client) and web servers
  - FTP client and server
  - Telnet
  - etc.
- Client-Server use application program interface (API) to set up communication
  - Socket

Client-Server Communication



- Client
  - Usually connect to a server one at a time
- Server
  - Connected to multiple clients at a time
  - Iterative server: queues the clients and processes them one at a time
  - Concurrent server: Processes the clients concurrently (by making a copy of itself, i.e., forking)
- Clients and server communicate using “sockets”

What is Socket?

- A data structure with fields essential of establishing a connection between a client and a server
- Allow processes to talk with each other even they are on different machines
- Setting the fields in the TCP/IP packet headers (set them in network byte order)
  - protocol, source / destination IP addresses, source / destination port number
  - Options
- BSD sockets, Berkeley sockets (heritage from Berkeley Unix)
- IPv4 and IPv6 sockets
- IP address
  - Binary
  - Dotted-decimal
  - Dotted-alphanumeric (or simply host name)

Connection-Oriented vs. Connectionless Services in Socket Programming

- Connection-oriented services
  - Client explicitly connects to the server
  - An end-to-end connection has to be initially set up before data is transferred
  - Reliable communication (through ACK or retransmission)
  - Flow control
  - E.g., TCP (Transmission Control Protocol)
    - An example of stream sockets
- Connectionless services
  - Client and server do not have to have a connection before sending data
  - Unreliable services
  - No flow control (packets may be lost)
  - E.g., UDP (User Datagram Protocol)
    - An example of datagram sockets

Port Numbers

- A 16-bit integer (0 – 65535)
- A server has well-defined long-lasting port-number for the access of its service
- A client has ephemeral (short-lived) port number
  - Automatically assigned by TCP or UDP
- Port numbers for services
  - Well-known ports 0–1023: controlled and assigned by IANA (Internet Assigned Numbers Authority)
  - Registered ports 1024 – 49151: for application usage
  - Dynamic or private ports 49152 – 65535: not generally used

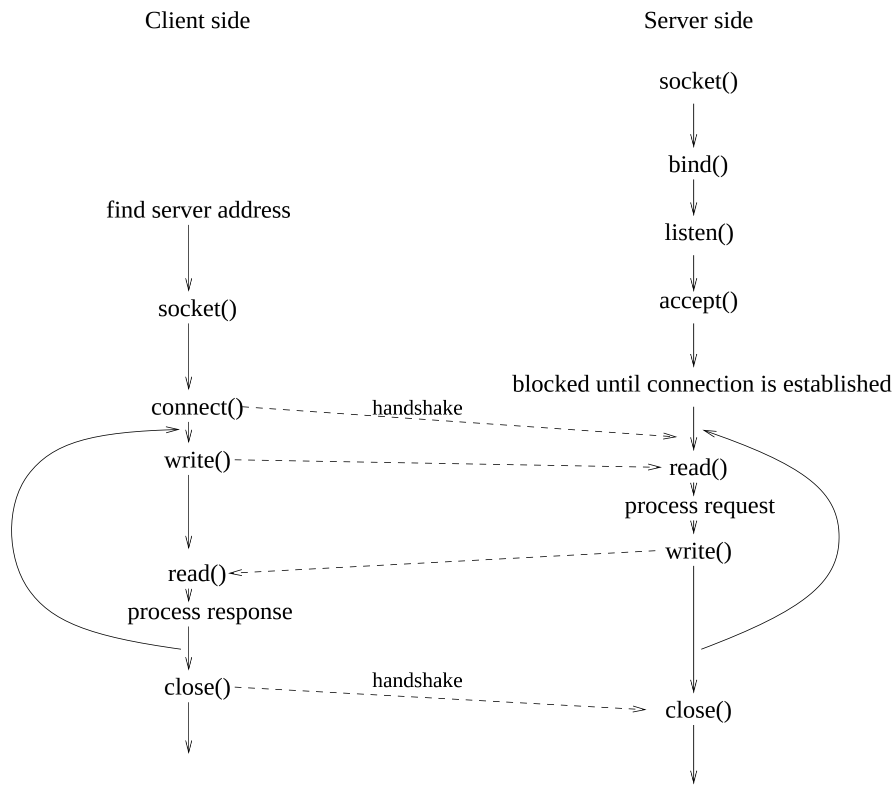
Well-Known Port Numbers for Well-Known Services (from /etc/services)

```
#
# Network services, Internet style
#
echo            7/tcp
echo            7/udp
daytime         13/tcp
daytime         13/udp
netstat         15/tcp
ftp-data        20/tcp
ftp             21/tcp
telnet          23/tcp
smtp            25/tcp          mail
time            37/tcp          timserver
time            37/udp          timserver
name            42/udp          nameserver
whois           43/tcp          nicname
hostnames       101/tcp         hostname
#
# Host specific functions
#
tftp            69/udp
finger          79/tcp
```

TCP/IP Connection

- Uniquely identifying every TCP connection is a 4 tuples defining the 2 endpoints of the connection
  - Local IP address
  - Local TCP port
  - Remote IP address
  - Remote TCP port
- Client's TCP ports are ephermeral port

Time-Space Diagram for Connection-Oriented Socket Programming



Connection-Oriented Socket Programming: Client Side

1. Create a socket characterized by Domain, Type, and Protocol
   - `socket()`⇒`sktfd`
   - Domain (Unix, Internet, Xerox network system)
   - Type (reliable or unreliable data communication)
   - Protocol (obtained by calling `getprotobyname`)
2. Use `struct sockaddr_in` to fill in the server's information
   - `sin_family = AF_INET`
   - `sin_port = htons( COMM_PORT )`
   - `sin_addr = htonl( IPsvrAddr )`
     – e.g., 32-bits IPv4 address (using `gethostname`, `gethostbyname` or `gethostbyaddr`) and port number
3. `connect` to the server using the `struct` and `sktfd`
   - If successful, the `sktfd` can be used for R/W
   - Send requests and get server response
   - `Close` connection

socket()

`int socket(int` *domain*`, int` *type*`, int` *protocol*`)`

- Create an unnamed socket of the specified domain, type and protocol
- Domain (Address family)
  – `AF_UNIX`: Clients and server must be on the same machine (<sys/un.h>)
  – `AF_INET`: Internet IPv4 applications (<netinet/in.h>, <arpa/inet.h>, <netdb.h>)
  – `AF_INET6`: Internet IPv6 applications (<netinet/in.h>, <arpa/inet.h>, <netdb.h>)
  – `AF_NS`: Xerox network system
- Type
  – `SOCK_STREAM`: sequenced, reliable, two-way connnection based
  – `SOCK_DGRAM`: connectionless, unreliable
- Protocol
  – Specify the low-level implementation details of a socket
  – May be obtained by calling `getprotobyname`
    ∗ Almost always set to `0` to get the default
- Return a file descriptor for R/W on success; -1 otherwise
- Note: the file descriptor has a read end and a write end
- E.g., `int sktfd = socket( AF_INET, SOCK_STREAM, 0 )`

Generic Socket Address Structure (in <netinet/in.h>)

```
#ifndef _SA_FAMILY_T
#define _SA_FAMILY_T
typedef unsigned short sa_family_t;
#endif

#ifndef _IN_PORT_T
#define _IN_PORT_T
typedef unsigned short in_port_t;
#endif

/*
 * Structure used by kernel to store most
 * addresses.
 */
struct sockaddr {
sa_family_t sa_family;
/* address family */
char sa_data[14];
/* up to 14 bytes of direct address */
};
```

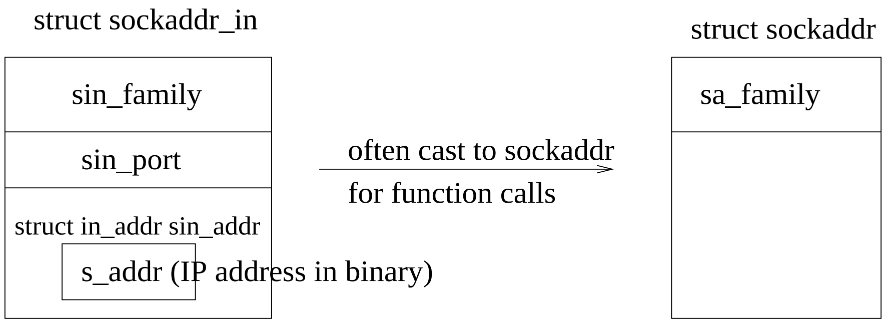

Internet Socket Address Structure

- Socket for IPv4: `struct socketaddr_in`

```
/*
 * Socket address, internet style.
 */
struct sockaddr_in {
sa_family_t sin_family;
in_port_t sin_port;
struct in_addr sin_addr;
#if !defined(_XPG4_2) || defined(__EXTENSIONS__)
char sin_zero[8];
#else
unsigned char sin_zero[8];
#endif
/* !defined(_XPG4_2) || defined(__EXTENSIONS__) */
};
```

- Socket for IPv6: `struct socketaddr_in6`
- Socket for Unix: `struct socketaddr_un`

Casting struct sockaddr_in to sockaddr in Function Calls



Connection-Oriented Socket Programming: Server Side

1. Create a (listening) socket characterized by Domain, Type, and Protocol (as in the client)
   - `socket()`⇒`lfd`
2. Fill in a `struct sockaddr_in`
   - `sin_family = AF_INET`
   - `sin_port = htons( COMM_PORT )`
   - `sin_addr.s_addr = htonl( INADDR_ANY )`
3. "`Bind`" it with `lfd`
4. `Listen` to the socket port
   - To make `lfd` a listening socket
5. `Accept` connection from a client
   - ⇒ `rw_sktfd` for read/write or input/output
   - Process request
   - `Close` connection

Reading Data from a Socket: read() and recv()

- `int read( int` *sockfd*`, char *` *buff*`, int` *buflen*`)`
  - A client or server uses `read()` to obtain input from a socket
  - Blocking read
  - *sockfd* is the socket descriptor created by the `socket()` call
  - *buff* points to the array of characters to hold the input
  - *buflen* is the size of the buffer array
  - Returns 0 if it detects an end-of-file condition on the socket, the number of bytes read if it obtains input, and -1 on error
- `int recv( int` *sockfd*`, char *` *buff*`, int` *buflen*`, int` *flag*`)`
  - Same as above
  - *flag* is generally set to 0

Writing Data to a Socket: write() and send()

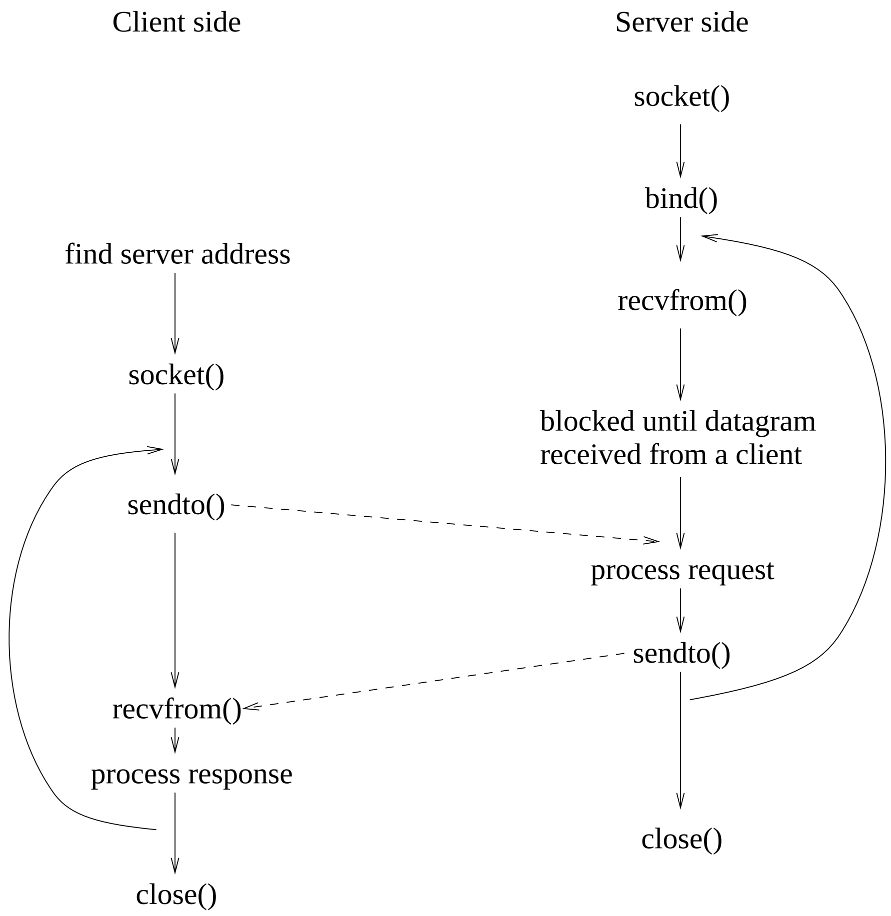
- `int write( int` *sockfd*`, char *` *buff*`, int` *buflen*`)`
  - A client or server uses `write()` to send data to a socket
  - Arguments defined same as `read()`
  - Returns the number of bytes transferred if successful and -1 on error
- `int send( int` *sockfd*`, char *` *buff*`, int` *buflen*`, int` *flags*`)`
  - Same as `write()`
  - *flags* is set to 0

Concurrent TCP Servers (Outline)

```
main(){
signal( SIGCHLD, sig_chld );
...
listenfd = socket();... bind();... listen();
for( ; ; ){
  connfd = accept( listenfd, ... );
  if( (pid = fork() ) == 0 ){
    close( listenfd );
    do_child_things( connfd );
    close( connfd );
    exit( 0 );
  }
  close( connfd );
}

void sig_chld(int signo){

  pid_t   pid;
  int           stat;

  while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
  /* -1: wait for the first child to terminate
   WNOHANG: process blocked if there are children */
    printf("child %d terminated\n", pid);
  return;
```

- Concurrent UDP server: Sect. 20.7, Stevens UNP v.1

Connectionless Services



Receiving and Sending Messages: recvfrom() and sendto()

- `ssize_t recvfrom(int` *sockfd*`, void *` *buff*`, size_t` *nbytes*`, int` *flags*`, struct sockaddr *` *from*`, socklen_t *` *addrlen*`)`
  - Extracts the next message that arrives at a socket and records the sender's address (enabling the caller to send a reply)
  - The first three arguments are used similar to `read()`
  - *flags* is set to 0
  - *from* points to the socket structure holding the sender's address
  - *addrlen* is the size of the structure pointed to by *from*
  - Return the size of message received; -1 if unsuccessful
- `ssize_t sendto(int` *sockfd*`, void *` *buff*`, size_t` *nbytes*`, int` *flags*`, const struct sockaddr *` *to*`, socklen_t` *addrlen*`)`
  - Sends a message by taking the destination address from a structure
  - The first three arguments are used similar to `read()`
  - *flags* is set to 0
  - *to* points to the socket structure holding the destination's address
  - *addrlen* is the size of the structure *to* in bytes
  - Return the size of message sent; -1 if unsuccessful

Send and Receive a "Packet" of Many Fields

```
/* At the sending side */
struct packet{
  char packet_data[MAX_PKT_SIZE];
  int chksm;
};
main(){
  struct packet apacket;
  char * pkt_ptr;
.
.
.
  apacket.packet_data = "This is data embedded into a packet\0";
  apacket.chksm = CHKSM;  /* put checksum here */

  pkt_ptr = (char *) &apacket;
  /* send pkt_ptr as if it is a character string */
  write( sktfd, pkt_ptr, sizeof( struct packet ) );
.
.
.
/* At the receiving end */
  char rvr_string[MAX_PKT_SIZE];
  struct packet * rvr_pkt;
.
.
.
  /* read all the characters of the packet */
  read( sktfd, rvr_string, sizeof( struct packet ) );
  /* cast back to a packet to access the fields */
  rvr_pkt = (struct packet *) rvr_string;

  /* rvr_pkt -> chksm is then the checksum field;
     rvr_pkt -> packet_data is then the data portion */
}
```

Lack of Reliability in UDP

- UDP does not provide any reliability and flow control
- Adding reliability in UDP (Sect. 20.5, Stevens UNP v.1)
  - Timeout and retransmission
  - Sequencing
  - Error detection
  - Window-flow mechanism

```
static sigjmp_buf jmpbuf;
signal(SIGALRM, sig_alrm);

while( transmission_not_ended() ){
  newpack(); /* initialize a new packet */

 sendagain:
  sendmsg(); /* send the packet */
  alarm( rtt_start() ); /* calc timeout value & start timer */
  if (sigsetjmp(jmpbuf, 1) != 0) {
    if (rtt_timeout() < 0) /* retransmitted enough? */
      give_up();
    goto sendagain;
  }
  do {
    n = Recvmsg(); /* waiting for a ACK with correct seq. # */
  } while (wrong sequence #);

  alarm(0); /* the packet successfully sent:
   stop SIGALRM timer */
  rtt_stop(); /* update estimate of rtt */
  process_reply();
}

static void sig_alrm(int signo){
  siglongjmp(jmpbuf, 1); }
```


When to Use UDP?

- UDP currently support broadcasting and multicasting
- Light-weighted and simple to use
- No connection setup or teardown — good for sending small packets
- There may be no point to do error recovery for some data (e.g., voice and video applications)

I/O Multiplexing

`int select( int` *numfds*`, fd_set *` *refds*`, fd_set * ` *wrfds*`, fd_set *` *exfds*`, struct timeval *` *time*`)`

- Provides asynchronous I/O by permitting a single process to wait for the first of any file descriptors in a specified set to become ready. The caller can also specify a maximum timeout for the wait.
- *numfds*: number of file descriptors in the set
- *refds, wrfds, exfds*: address of file descriptors for read, write and exceptions, respectively
- *time*: maximum time to wait or 0 (i.e, poll); if it is a null pointer, `select` blocks indefinitely until a descriptor is "selectable"
- Returns the number of ready file descriptors if successful, 0 if the time limit was reached, and -1 to indicate an error

An Example

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

fd_set read_template;
struct timeval wait;

for(;;){
  wait.tv_sec = 1;  /* one second */
  wait.tv_uec = 0;  /* one second */

  FD_ZERO( &read_template);
  FD_SET( s1, &read_template);
  FD_SET( s2, &read_template);
  nb = select( FD_SETSIZE, &read_template, (fd_set *) 0, \
  (fd_set *) 0, &wait);
  if( nb <= 0 )
    error;
  if( FD_ISSET( s1, &read_template ) )
    process( s1 );
  if( FD_ISSET( s2, &read_template ) )
    process( s2 );
}
```

Getting and Setting Socket's Options: getsockopt() and setsockopt()

- We may get and set some socket attributes and the option fields in an IP or TCP packet
- `int getsockopt(int` *sockfd*`, int` *level*`, int` *optname*`, void *` *optval*`, socklen_t *` *optlen*`)`
  - Get the socket option of the open descriptor *sockfd*
  - *optval* is pointing to a variable into which the value of the option is stored
  - The *level* and *optname* are given in Stevens UNPv1 p.179.
  - Return 0 on success; -1 otherwise
- `int setsockopt(int` *sockfd*`, int` *level*`, int` *optname*`, const void *` *optval*`, socklen_t *` *optlen*`)`
  - Set the socket option of the open descriptor *sockfd*
  - *optval* is pointing to a variable from which the new value of the option is fetched
  - Return 0 on success; -1 otherwise
  - An example on how to set a socket option

```
int on = 1;
setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
```


Some Option Fields

- (level) SOL_SOCKET (Generic socket)
  - (optname) SO_BROADCAST, SO_ERROR, SO_KEEPALIVE, SO_RCVBUF SO_SNDBUF, SO_TYPE, SO_USELOOPBACK, SO_KEEPALIVE, SO_RCVTIMEO, SO_SNDTIMEO, SO_REUSEADDR, SO_REUSEPORT
- IPPROTO_IP (IPv4 Socket)
  - IP_OPTIONS, IP_TOS, IP_TTL
  - IP_ADD_MEMBERSHIP, IP_DROP_MEMBERSHIP, IP_MULTICAST_TTL, IP_MULTICAST_LOOP
- IPPROTO_IPV6 (IPv6 Socket)
  - IPV6_ADDRFORM, IPV6_HOPLIMIT, IPV6_UNICAST_HOPS
  - IPV6_ADD_MEMBERSHIP, IPV6_DROP_MEMBERSHIP, IPV6_MULTICAST_TTL, IPV6_MULTICAST_LOOP
- IPPROTO_TCP (TCP)
  - TCP_KEEPALIVE, TCP_MAXSEG, TCP_NODELAY, TCP_MAXRT
- Stevens UNP chapters 7 and 24

Unicast, Broadcast, and Multicast

- Unicast: a process addressing to exactly one other process in the network
  - Packet addressed to a single interface
- Broadcast: a process addressing to all other processes in the network
  - Packet addressed to all interfaces
  - The broadcast extend subject to router's willingness to forward packets. Generally limited to a LAN
- Multicast: a process addressing to a subset of processes in the network
  - Packet addressed to a set of interfaces
  - Can span a LAN or WAN
  - Joining/Leaving a group

Support of Multicast and Broadcast

- Multicast is supported by IPv6, but optional in IPv4
- Broadcasting support is no longer provided by IPv6 (use multicast instead for such purpose)
- Broadcasting and multicasting requires UDP (they don't work with TCP yet)

Four Types of Broadcast Addresses

Denote an IPv4 address as {netid, subnetid, hostid} (let a field with all 1's be -1)

- Directed to all hosts in a subnet: {netid, subnetid, -1}
- Directed to all subnets in a network: {netid, -1, -1}
- Directed to everyone in the world: {-1, -1, -1}
  - Almost never be forwarded by a router

```
setsockopt( sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof( on ) );
sendto( sockfd, ... );
```


IPv4 Multicasting

- Class D address (The starting four IP address bits are 1110), in the range 224.0.0.0 through 239.255.255.255, are the multicast address
  - The low-order 28 bits form the multicast group ID (or group address)
  - A host joins a multicast group in order to receive multicast packets
- 224.0.0.1 is the all-hosts group. All multicast-capable hosts on a subnet must join this group
- 224.0.0.2 is the all-routers group. All multicast routers on a subnet must join this group

Multicast Socket Programming

1. Create a UDP socket
2. Set some multicast socket options (set by `setsockopt()`)
   - IP_ADD_MEMBERSHIP (IPV6_ADD_MEMBERSHIP)
     - Join a multicast group
   - IP_DROP_MEMBERSHIP (IPV6_DROP_MEMBERSHIP)
     - Leave a multicast group
   - IP_MULTICAST_IF (IPV6_MULTICAST_IF)
     - Specify the interface for outgoing multicast
   - IP_MULTICAST_TTL (IPV6_MULTICAST_TTL)
     - Specify TTL for outgoing multicast packets
   - IP_MULTICAST_LOOP (IPV6_MULTICAST_LOOP)
     - Enable or disable loopback of outgoing multicast packets (for the sending host to process its own loopback packets)
3. Following communication similar to what we have covered

Some Socket Functions

(You can get help on most of these functions by typing `man <function>` in Unix)

Server Side Function: accept()

`int accept(int` *fd*`, struct sockaddr *` *address*`, int *` *addressLen*`)`

- Listen to the socket descriptor *fd* and waited till a client connection request is received.
- Create an unnamed socket with the same attributes as the original named server socket, connects it to the client's socket, and returns a new file descriptor that may be used for communication with the client. The original named server socket may be used to accept more conections.
- The *address* is filled with the information of the client, and the *addressLen* is the size of it.
- Return a new file descriptor on success; -1 otherwise.

Server Side Function: bind()

`int bind(int` *fd*`, struct sockaddr *` *address*`, int` *addressLen*`)`

- Associate the unnamed socket pointed to by *address* with the file descriptor *fd*. *addressLen* is the size of the address structure.
- Return 0 on success; -1 otherwise

Server Side Function: listen()

`int listen(int` *fd*`, int` *queueLength*`)`

- Specify the maximum number of pending connections (given by *queueLength*) on a socket with descriptor *fd*
- A client's request is declined if the queue is full

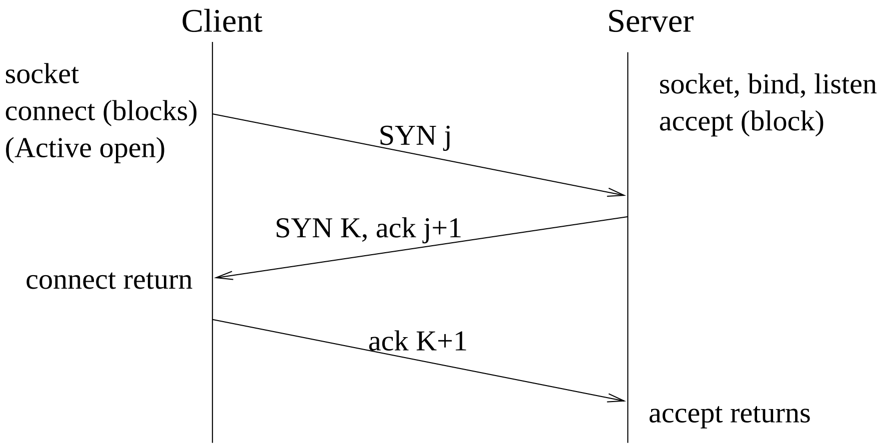
connect()

`int connect(int` *fd*`, struct sockaddr *` *address*`, int` *addressLen*`)`

- `Connect()` connects to the server socket whose address is contained within a structure pointed to by *address*.
- To connect to a server's socket, a client must first fill a structure with the address of the server's socket and then use `connect()`
- For a socket in the `AF_INET` domain, a pointer to a `sockaddr_in` structure must be cast to a `(sockaddr *)` and passed in as *address* (*addressLen* is the size of the address structure). The structure has to be set as follows:
  - `sin_family`: `AF_INET`
  - `sin_port`: the port number of the Internet socket
  - `sin_addr`: a structure of type in_addr that holds the Internet address
  - `sin_zero`: unused
- If successful, *fd* can be used to communicate with the server's socket
- Return 0 on success; -1 otherwise

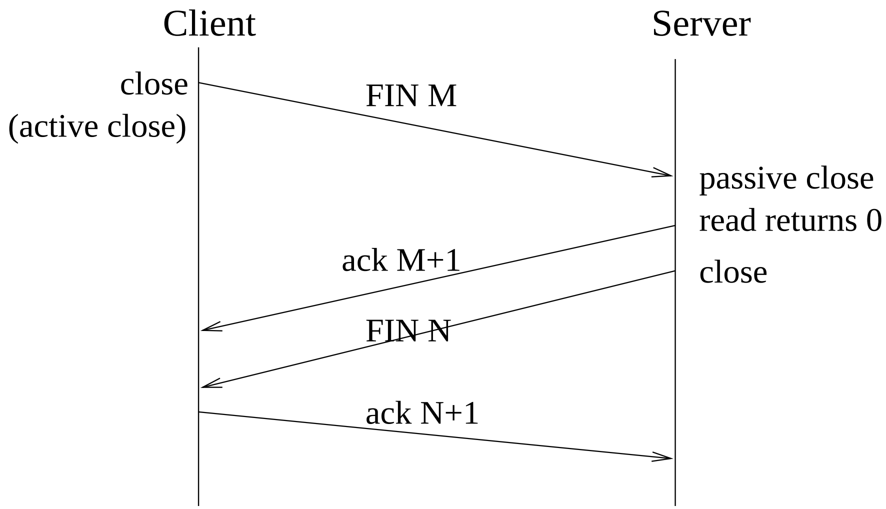
3-Way Handshakes at Connection Set-up

- Server does a passive open (using `socket, bind, listen` and `accept`)
- Client issues an active open by calling `connect()`
- 3-way handshake



Connection Termination: close()

- `int close(int` *fd*`)`
  - Close the file associated with *fd*
  - Return -1 on failure
- One application calls `close()` to perform an active close
- The other end after receiving FIN performs a passive close
- Connection is half-close
- Example: Client performs active close



gethostname()

`int gethostname(char * name, int nameLen)`

- Set the pointer *name* of length *nameLen* to a null-terminated string equal to the local host's name
- Return 0 on success, -1 otherwise

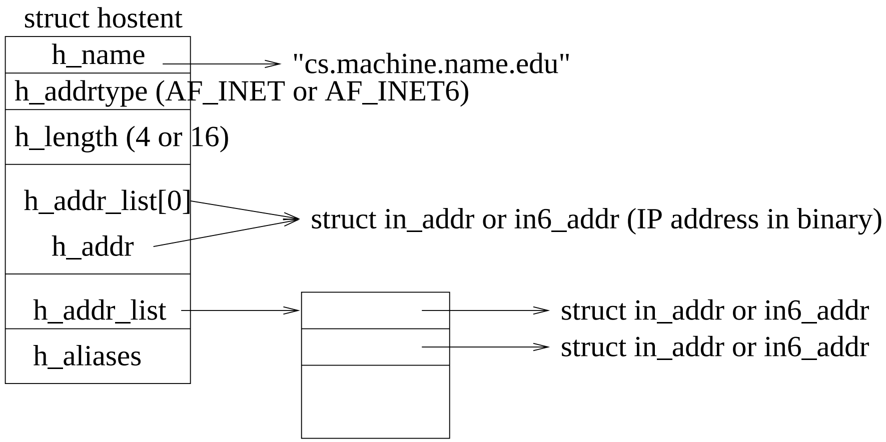
Getting Host Information: gethostbyname() and gethostbyaddr

- `struct hostent * gethostbyname(char * name)`
  - Search the files `/etc/hosts` or `/etc/resolv.conf` and return a pointer to a hostent structure that describes the file entry associated with the string *name*
  - The field of interest is normally `h_addr`
  - Return NULL if not found
- `struct hostent * gethostbyaddr(const char * addr, size_t len, int family)`
  - Take a binary IP address and find the hostname corresponding to that address
  - The field of interest is normally `h_addr`
  - The *addr* is a pointer to an `in_addr` or `in6_addr` structure containing the IPv4 or IPv6 addresses
  - The *family* is either `AF_INET` or `AF_INET6`
  - The *len* is 4 for IPv4, and 16 for IPv6
  - Return NULL on failure
  - The field of interest is normally `h_name`

struct hostent

```
struct hostent {
char *h_name; /* official name of host */
char **h_aliases; /* alias list */
int h_addrtype; /* host address type */
int h_length; /* length of address */
char **h_addr_list;
/* list of addresses from name server */
#define h_addr h_addr_list[0]
/* address, for backward compatiblity */
};
```

- We usually are interested in the field `h_addr`, which contains the host's associated IP binary address in the field `s_addr`.



Getting Information Given a Socket Descriptor: getpeername() and getsockname()

- `getpeername` returns the remote information (via a socket structure) associated with a socket descriptor; `getsockname` returns the local information associated with a socket descriptor
  - Local/Remote port number
  - Local/Remote protocol number
  - Local/Remote family
  - Local/Remote IP binary address
- `int getpeername(int` *clientfd*`, struct sockaddr *` *address*`, socketlen_t *` *addressLen*`)`
  - Retrieve the peer address of the socket descriptor *clientfd* and store it in the stucture pointed by *address* and the length of the address in the object pointed by *addressLen*
  - Return 0 on success; -1 otherwise
- `int getsockname(int` *sockfd*`, struct sockaddr *` *localaddr*`, socketlen_t *` *addrlen*`)`
  - Similar to above but obtain local information

Getting Service Information: getservbyname() and getservbyport()

- `struct servent * getservbyname(const char *` *servname*`, const char *` *protoname*`)`
  - Look up the port number given a service name pointed to by *servname* and a protocol name pointed to by *protoname*
  - Return NULL on error

```
struct servent {
char *s_name; /* official service name */
char **s_aliases; /* alias list */
int s_port; /* port #, network byte order */
char *s_proto; /* protocol to use */
};
```


Getting Service Information (Cont.): getservbyname() and getservbyport()

- `struct servent * getservbyport(int` *port*`, const char *` *protoname*`)`
  - Look up a service given its port name *port* and the protocol name pointed to by *protoname*
  - Return NULL on error

```
/* Get the protocol and port numbers */

main(){

  struct protoent * pptr;
  struct servent * sp;

  if( (pptr = getprotobyname( "tcp" )) == NULL )
    perror("TCP: getprotobyname");
  printf("TCP protocol number = %d\n", pptr -> p_proto );
  if( (pptr = getprotobyname( "udp" )) == NULL )
    perror("UDP: getprotobyname");
  printf("UDP protocol number = %d\n", pptr -> p_proto );

  if( (sp = getservbyname( "ftp", "tcp" )) == NULL )
    perror("ftp/tcp getservbyname");
  printf("ftp/tcp port number = %d\n", ntohs( sp -> s_port ));
  if( (sp = getservbyname( "daytime", "udp" )) == NULL )
    perror("daytime/udp getservbyname");
  printf("daytime/udp port number = %d\n", ntohs( sp -> s_port ));

  return 0;
}
```

Getting Protocol Information: getprotobyname() and getprotobynumber()

- `struct protoent * getprotobyname(const char *` *strptr* `)`
  - Get a protocol's official integer value from its name pointed to by *strptr*
  - Return NULL on error
- `struct protoent * getprotobynumber(int` *pnumber* `)`
  - Get a protocol's official name by its number given by *pnumber*
  - Return NULL on error

```
struct protoent {
char *p_name; /* official protocol name */
char **p_aliases; /* alias list */
int p_proto; /* protocol # */
};
```


htonl(), htons(), ntohl(), and ntohs()

`unsigned long htonl( unsigned long` *hostLong*`)`
`unsigned short htons( unsigned short` *hostShort*`)`
`unsigned long ntohl( unsigned long` *networkLong*`)`
`unsigned short ntohs( unsigned short` *networkShort*`)`

- Convert between a host-format number and a network-format number.
- long: 32-bit
- short: 16-bit

Manipulating Internet Addresses (for IPv4): inet_aton(), inet_addr(), and inet_ntoa()

- `int inet_aton(const char *` *strptr*`, struct in_addr *` *addrptr*`)`
  - Convert a dotted-decimal string pointed to by *strptr* to IPv4 binary address pointed by *addrptr*
  - Return 1 on success; 0 otherwise
- `unsigned long inet_addr(char *` *string*`)`
  - Return the 32-bit IP address that corresponds to the A.B.C.D format *string*
  - Return the constant `INADDR_NONE` (typically all 1's) on error
  - Slight problem: the dotted-decimal string 255.255.255.255 (the IPv4 limited broadcast address) cannot be handled by this function
- `char * inet_ntoa(struct in_addr` *inaddr*`)`
  - Take a structure of type `in_addr` (a 32-bit IP binary network byte ordered address) and returns a pointer to a string that describes the address in the dot-decimal notation

Manipulating Internet Addresses (for IPv4 & IPv6): inet_pton(), and inet_ntop()

- p stands for presentation (dotted-decimal notation); n stands for numeric (network byte order)
- `int inet_pton(int` *family*`, const char *` *strptr*`, void *` *addrptr*`)`
  - Convert the presentation format pointed to by *strptr* to network byte order pointed to by *addrptr*
  - family: `AF_INET` or `AF_INET6`
  - Return 1 on success; -1 on error
- `const char * inet_ntop(int` *family*`, const void *` *addrptr*`, char *` *strptr*`, size_t` *len*`)`
  - Convert from numeric to presentation
  - `len` is the size of the char string
    - For IPv4, *len* should be at least 16
    - For IPv6, *len* should be at least 46
  - Return pointer to result on success; NULL otherwise