ELEC462001 System Programming

Ch11. Connecting to Processes Near and Far Servers and Sockets

Prof. Seokin Hong

Kyungpook National University

Fall 2018

Chapter Summary

- client/server programming using pipes and sockets
- Interprocess communication and client/server design.
- The idea and techniques of socket programming

Common Interface for different types of sources

- **Disk/Device File**
  - Use open() to connect, and use **read()** and **write()** to transfer data.

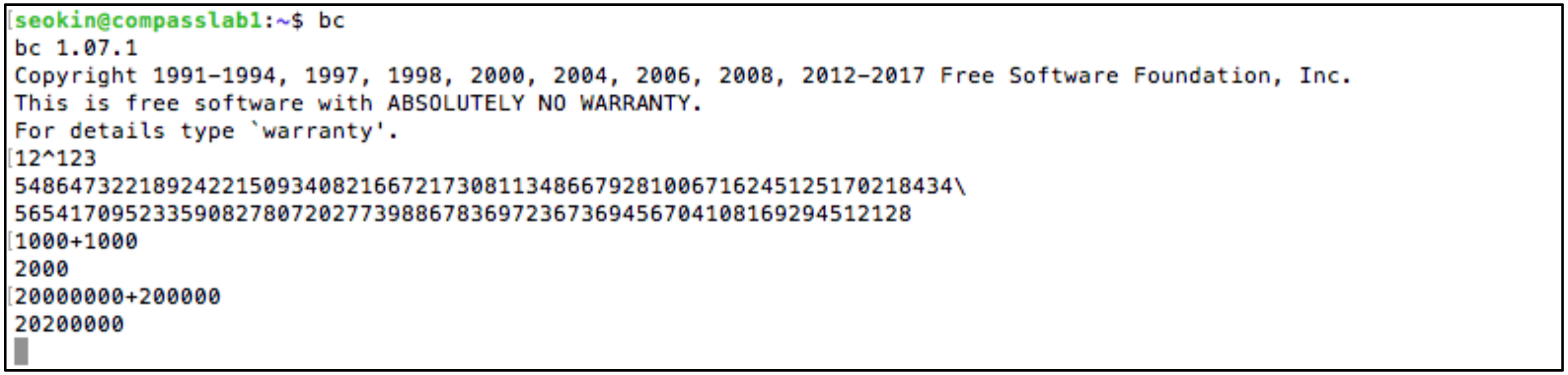
- **Pipes**
  - Use pipe() to create, use fork() to share, and use **read()** and **write()** to transfer data

- **Sockets**
  - Use socket(), listen(), and connect() to connect, and use **read()** and **write()** to transfer data.

Use **read()** and **write()** system call to transfer data to/from processes

bc: A UNIX CALCULATOR

- Bc can handle very long number:



```
seokin@compasslab1:~$ bc
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
12^123
548647322189242215093408216672173081134866792810067162451251702184344\
56541709523359082780720277398867836972367369456704108169294512128
1000+1000
2000
20000000+200000
20200000
```

- But bc Is Not a calculator!

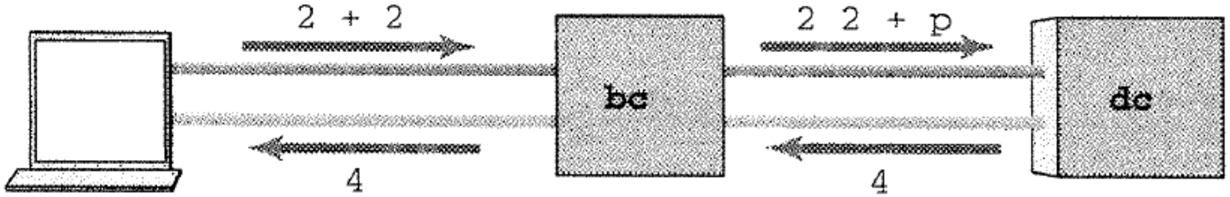


FIGURE 11.3

bc and dc as coroutines.

bc: A UNIX CALCULATOR

- Ideas from bc
  - **Client/Server Model**
    - dc provides a service (calculation)
    - bc provides a user interface AND uses the service
    - bc is called a client of dc.
  - **Bidirectional Communication**
    - Using pipes, you need two pipes
  - **Persistent Service (coroutines)**
    - Both processes continue to run
    - Control passes from one to the other as each completed its part of the job
    - bc has the job of parsing input and printing the computation results
    - dc has the job of computing

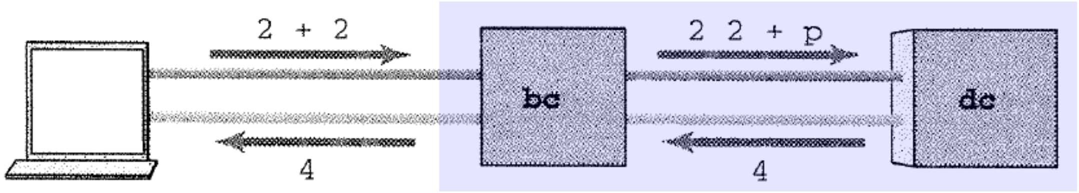
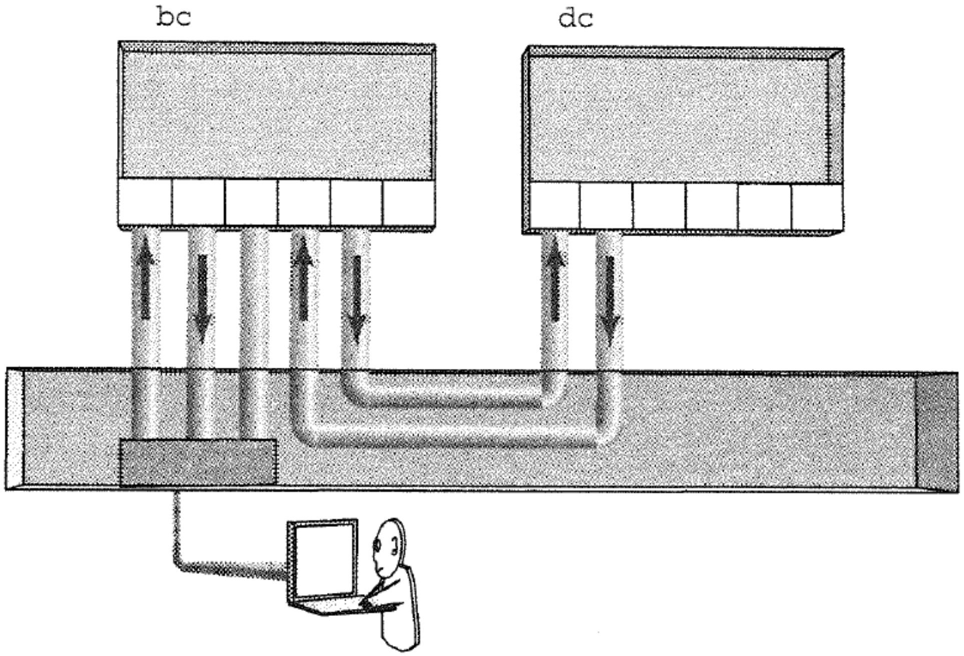


FIGURE 11.3

bc and dc as coroutines.

Coding bc: pipe, fork, dup, exec



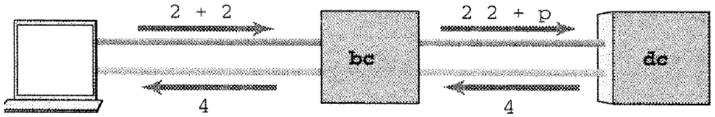


FIGURE 11.4

bc, dc, and kernel.

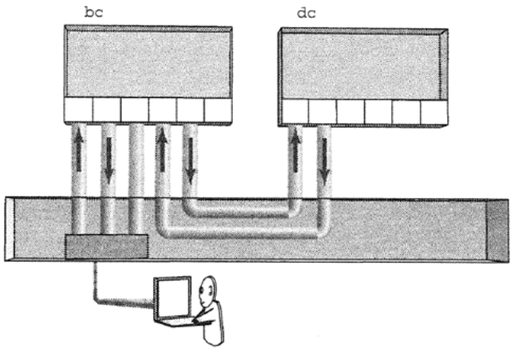
(a) Create two pipes.

(b) Create a process to run dc.

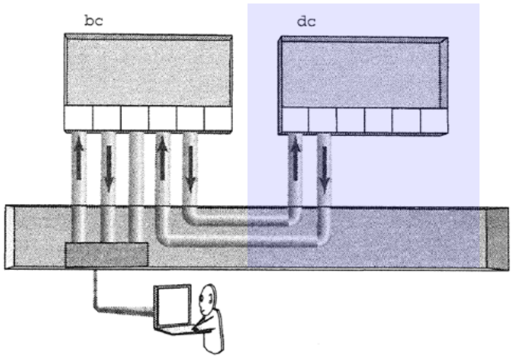
(c) In the new process, redirect stdin and stdout to the pipes, then exec dc.

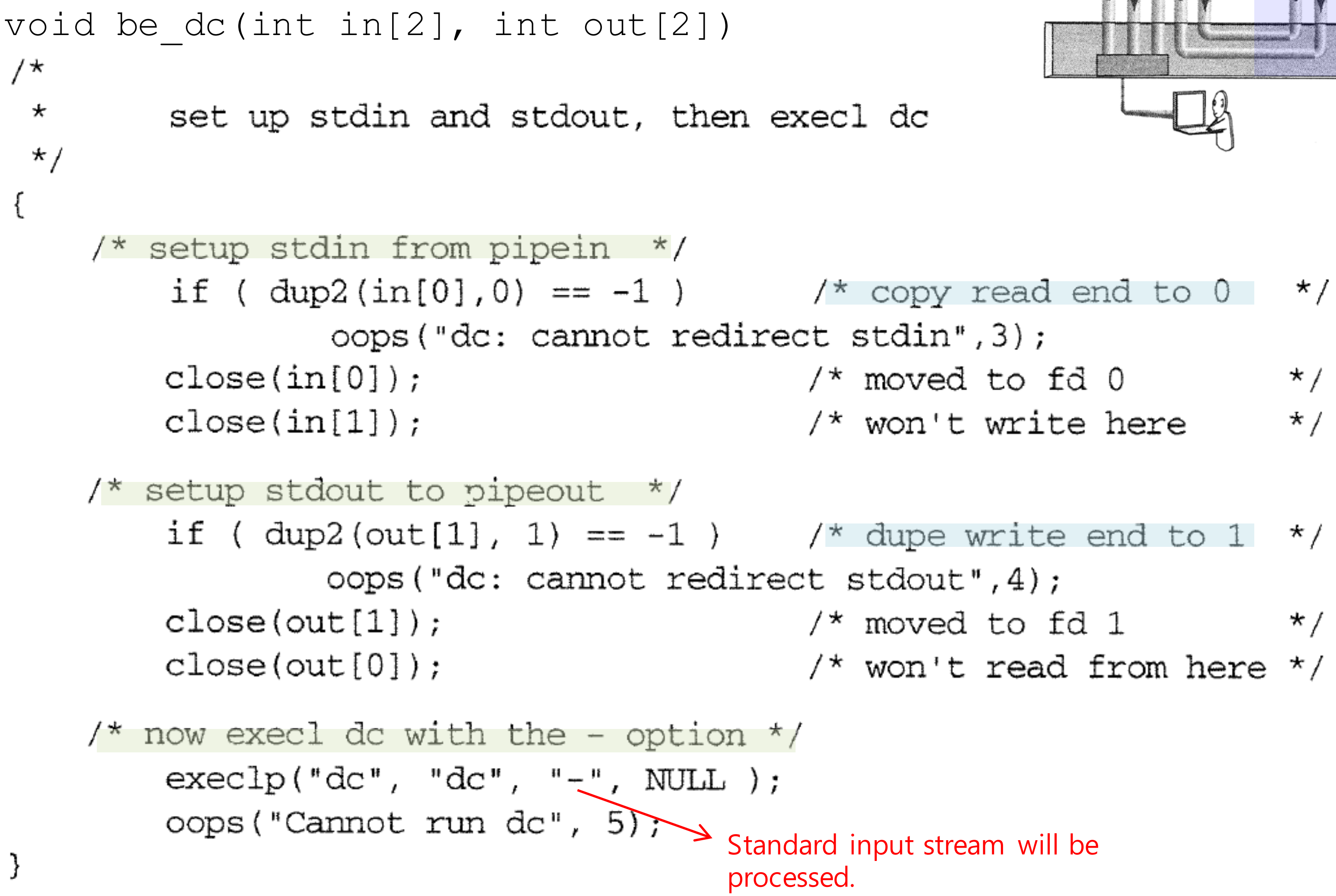
(d) In the parent, read and parse user input, write commands to dc, read response from dc, and send response to user.

```
/**     tinybc.c        * a tiny calculator that uses dc to do its work
 **                     * demonstrates bidirectional pipes
 **                     * input looks like number op number which
 **                       tinybc converts into number \n number \n op \n p
 **                       and passes result back to stdout
 **
 **          +-----------+                  +-----------+
 **     stdin  >0            >== pipetodc ====>            |
 **            |  tinybc |                  |   dc -    |
 **     stdout <1            <== pipefromdc ==<            |
 **          +-----------+                  +-----------+
 **
 **                * program outline
 **                        a. get two pipes
 **                        b. fork (get another process)
 **                        c. in the dc-to-be process,
 **                                connect stdin and out to pipes
 **                                then execl dc
 **                        d. in the tinybc-process, no plumbing to do
 **                                just talk to human via normal i/o
 **                                 and send stuff via pipe
 **                        e. then close pipe and dc dies
 **                 * note: does not handle multiline answers
 **/
```



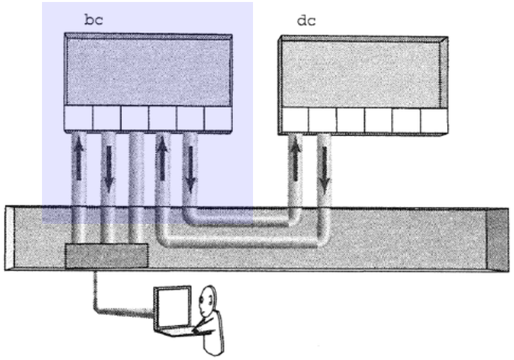
```
#include        <stdio.h>
#include        <unistd.h>
#include        <stdlib.h>
#define         oops(m, x){perror(m); exit(x);}

void be_dc(int in[2], int out[2]);
void be_bc(int todc[2], int fromdc[2]);
void fatal(char mess[]);

main()
{
        int     pid, todc[2], fromdc[2];         /* equipment    */

        /* make two pipes */

        if ( pipe(todc) == -1 || pipe(fromdc) == -1 )              ①
                oops("pipe failed", 1);

        /* get a process for user interface */

        if ( (pid = fork()) == -1 )
                oops("cannot fork", 2);                            ②
        if ( pid == 0 )                          /* child is dc  */
                be_dc(todc, fromdc);
        else {
                be_bc(todc, fromdc);             /* parent is ui */
                wait(NULL);                      /* wait for child */
        }
}
```



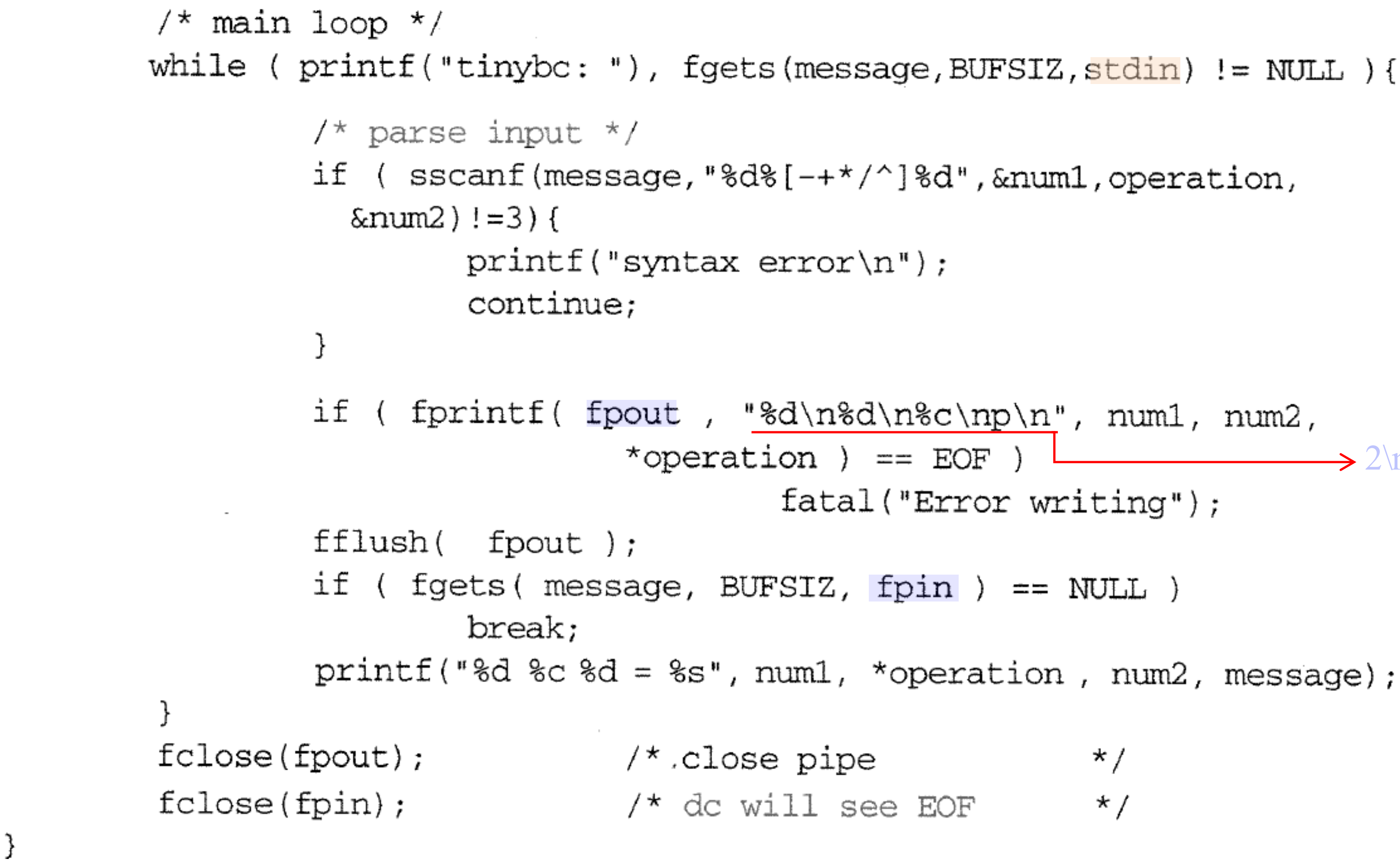


```
void be_dc(int in[2], int out[2])
/*
 *      set up stdin and stdout, then execl dc
 */
{
    /* setup stdin from pipein  */
        if ( dup2(in[0],0) == -1 )        /* copy read end to 0    */
                oops("dc: cannot redirect stdin",3);
        close(in[0]);                     /* moved to fd 0         */
        close(in[1]);                     /* won't write here      */

    /* setup stdout to pipeout  */
        if ( dup2(out[1], 1) == -1 )      /* dupe write end to 1   */
                oops("dc: cannot redirect stdout",4);
        close(out[1]);                    /* moved to fd 1         */
        close(out[0]);                    /* won't read from here  */

    /* now execl dc with the - option */
        execlp("dc", "dc", "-", NULL );
        oops("Cannot run dc", 5);
}
```

Standard input stream will be processed.

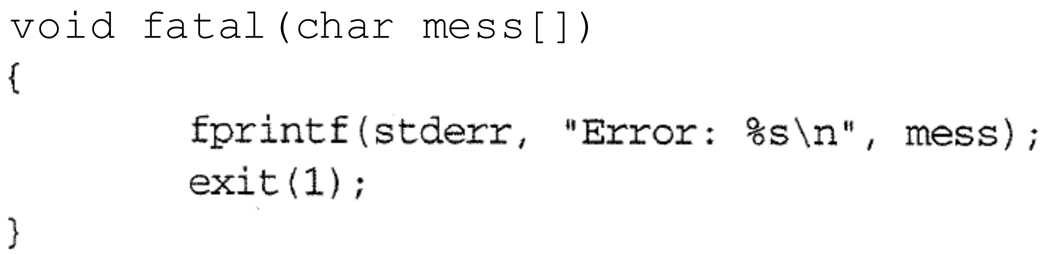


```
void be_bc(int todc[2], int fromdc[2])
/*
 *      read from stdin and convert into to RPN, send down pipe
 *      then read from other pipe and print to user
 *      Uses fdopen() to convert a file descriptor to a stream
 */
{
        int     num1, num2;
        char    operation[BUFSIZ], message[BUFSIZ], *fgets();
        FILE    *fpout, *fpin, *fdopen();

        /* setup */
        close(todc[0]);                   /* won't read from pipe to dc  */
        close(fromdc[1]);                 /* won't write to pipe from dc */ ④

        fpout = fdopen( todc[1],   "w" );      /* convert file desc-  */
        fpin  = fdopen( fromdc[0], "r" );      /* riptors to streams  */ ⑤
        if ( fpout == NULL || fpin == NULL )
                fatal("Error converting pipes to streams");
```

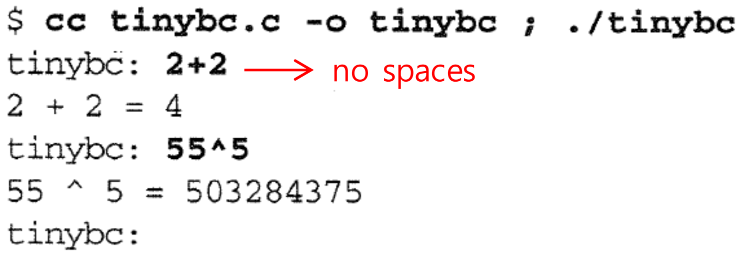


```
    /* main loop */
   while ( printf("tinybc: "), fgets(message,BUFSIZ,stdin) != NULL ){

           /* parse input */
           if ( sscanf(message,"%d%[-+*/^]%d",&num1,operation,
             &num2)!=3){
                    printf("syntax error\n");
                    continue;
           }

           if ( fprintf( fpout , "%d\n%d\n%c\np\n", num1, num2,
                             *operation ) == EOF )
                                     fatal("Error writing");
           fflush(   fpout );
           if ( fgets( message, BUFSIZ, fpin ) == NULL )
                    break;
           printf("%d %c %d = %s", num1, *operation , num2, message);  // stdout
    }
    fclose(fpout);              /* close pipe          */
    fclose(fpin);               /* dc will see EOF     */
}
```

2\n2\n+\np\n



```
void fatal(char mess[])
{
          fprintf(stderr, "Error: %s\n", mess);
          exit(1);
}
```



```
$ cc tinybc.c -o tinybc ; ./tinybc
tinybc: 2+2 → no spaces
2 + 2 = 4
tinybc: 55^5
55 ^ 5 = 503284375
tinybc:
```

```
…
tynybc: ctrl+D
 $
```

`fdopen`: Making File Descriptors Look like Files

- `fopen:` file name → FILE *

- `fdopen:` file descriptor → FILE *
  - you can use **standard**, **buffered I/O** operations;
  - **`tinybc.c` uses `fprintf` and `fgets` to send data through the pipes to `dc`.**

Contents

- bc: A Unix Calculator
- popen: Making Process Look like Files
- Sockets: Connecting to Remote Processes

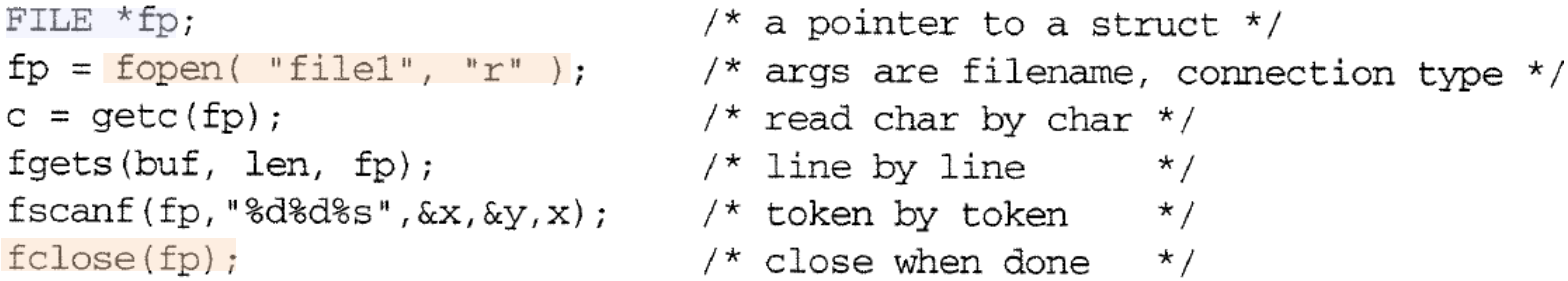
popen: MAKING PROCESSES LOOK LIKE FILE

- We examine the popen library function.

- We see what popen does and how popen works, and then we write our own version.

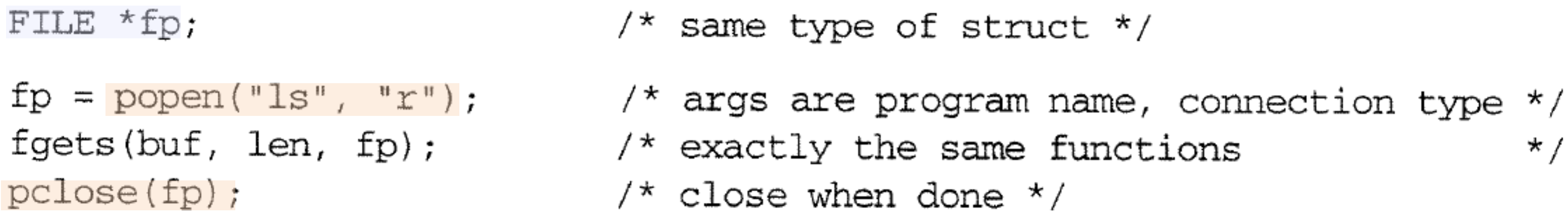
What popen Does

- fopen() opens a buffered connection to a file:



```
FILE *fp;                       /* a pointer to a struct */
fp = fopen( "file1", "r" );     /* args are filename, connection type */
c = getc(fp);                   /* read char by char */
fgets(buf, len, fp);            /* line by line      */
fscanf(fp,"%d%d%s",&x,&y,x);    /* token by token    */
fclose(fp);                     /* close when done   */
```

- popen() opens a buffered connection to a process:



```
FILE *fp;                    /* same type of struct */

fp = popen("ls", "r");       /* args are program name, connection type */
fgets(buf, len, fp);         /* exactly the same functions             */
pclose(fp);                  /* close when done */
```

What popen Does

- Similarities between popen and fopen.

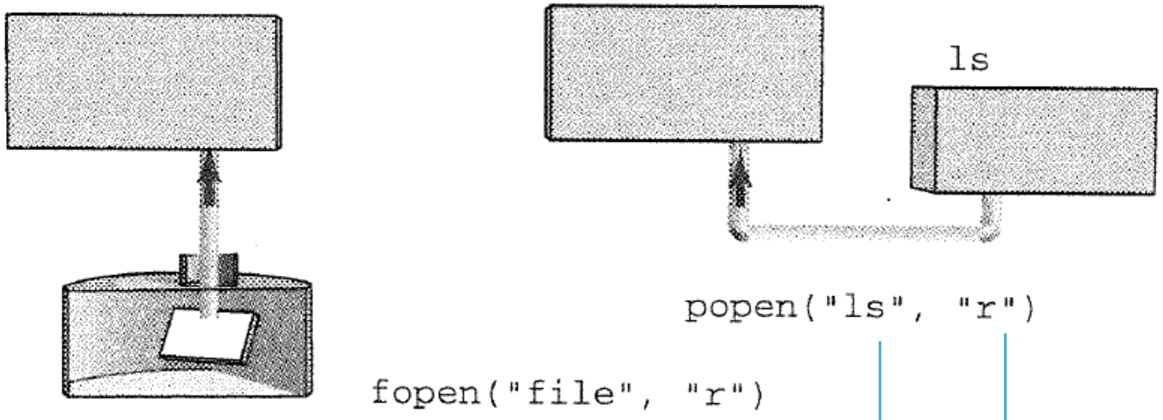


FIGURE 11.5

fopen and popen.

It can be either "r" or "w", but never "a".

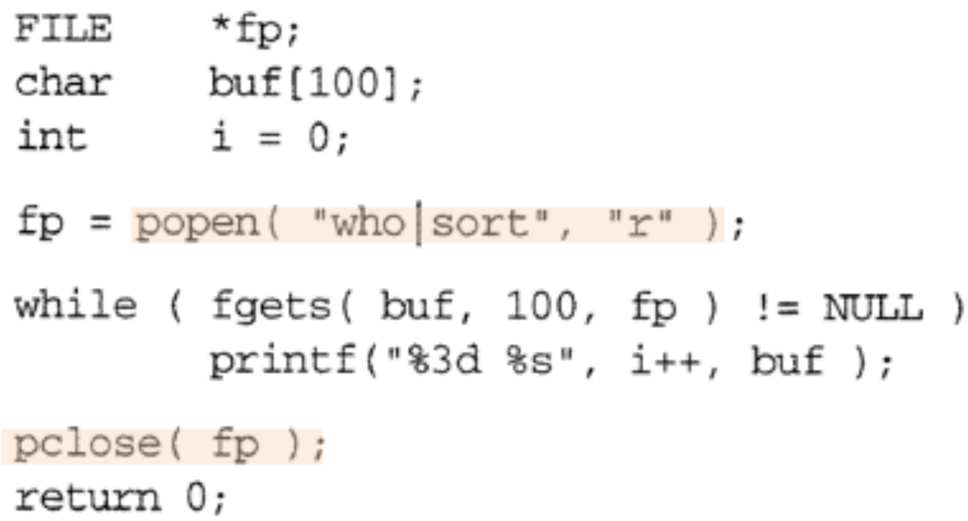
It can be any shell command.

```
/* popendemo.c
 *      demonstrates how to open a program for standard i/o
 *      important points:
 *              1. popen() returns a FILE *, just like fopen()
 *              2. the FILE * it returns can be read/written
 *                 with all the standard functions
 *              3. you need to use pclose() when done
 */
#include        <stdio.h>
#include        <stdlib.h>

int main()
{
        FILE    *fp;
        char    buf[100];
        int     i = 0;

        fp = popen( "who|sort", "r" );          /* open the command  */

        while ( fgets( buf, 100, fp ) != NULL ) /* read from command */
                printf("%3d %s", i++, buf );    /* print data        */

        pclose( fp );                           /* IMPORTANT!        */
        return 0;
}
```

- `pclose` is Required
  - A process needs to be waited for, or it becomes a *zombie*.
  - **`pclose` calls `wait`.**

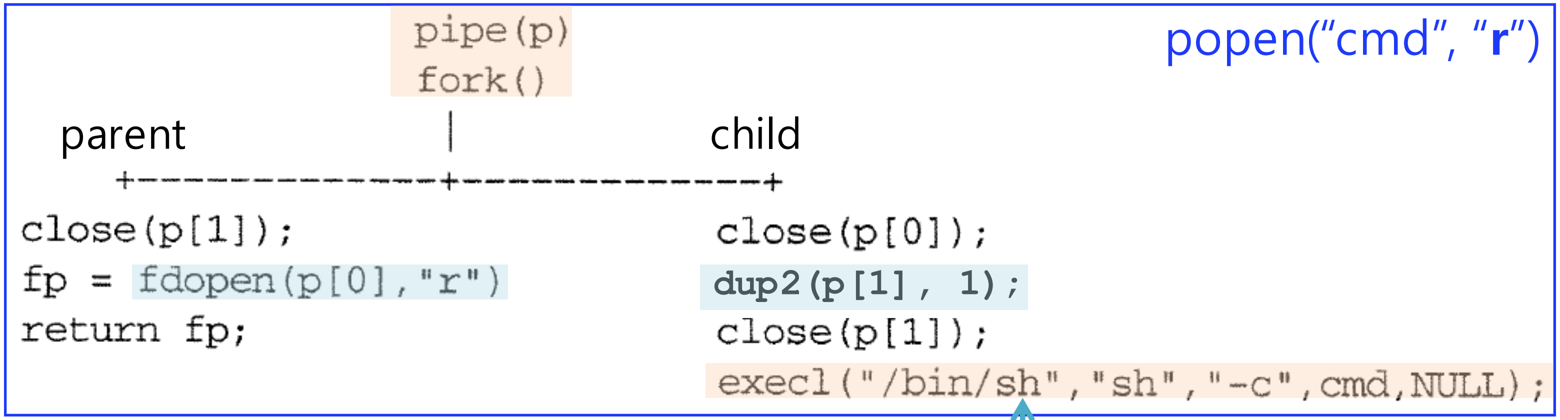
Writing `popen`: Using `fdopen`

- How does `popen` work?
  - o `popen` **runs** a program and **returns** a connection to the standard input or standard output of that program.



```
FILE    *fp;
char    buf[100];
int     i = 0;

fp = popen( "who|sort", "r" );

while ( fgets( buf, 100, fp ) != NULL )
        printf("%3d %s", i++, buf );

pclose( fp );
return 0;
```

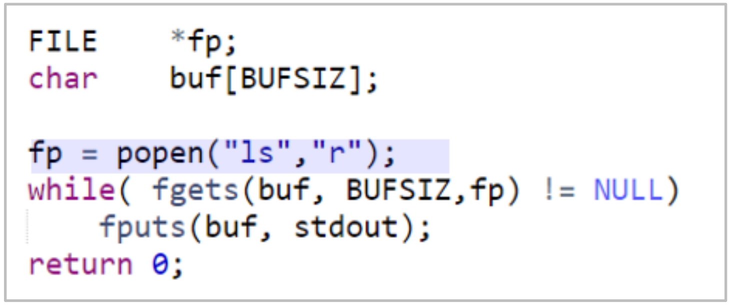
▪ Writing `popen`: Using `fdopen`



popen("cmd", "**r**")

```
               pipe(p)
               fork()
   parent         |            child
      +-----------+-----------+
close(p[1]);                  close(p[0]);
fp = fdopen(p[0],"r")         dup2(p[1], 1);
return fp;                    close(p[1]);
                              execl("/bin/sh","sh","-c",cmd,NULL);
```

The only program that can run any shell command is the shell itself



```
FILE    *fp;
char    buf[BUFSIZ];

fp = popen("ls","r");
while( fgets(buf, BUFSIZ,fp) != NULL)
    fputs(buf, stdout);
return 0;
```

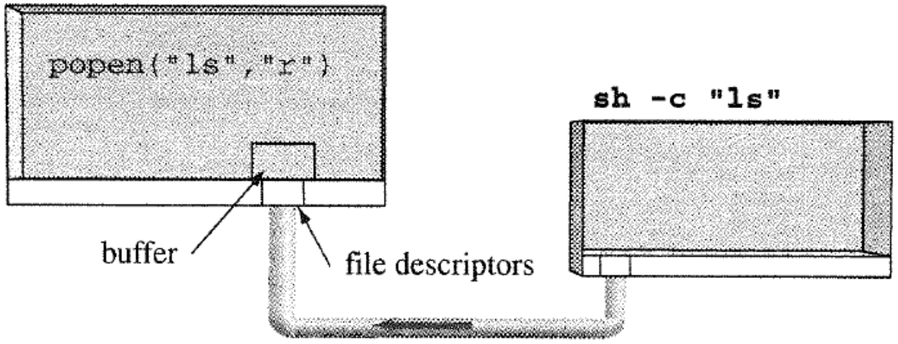
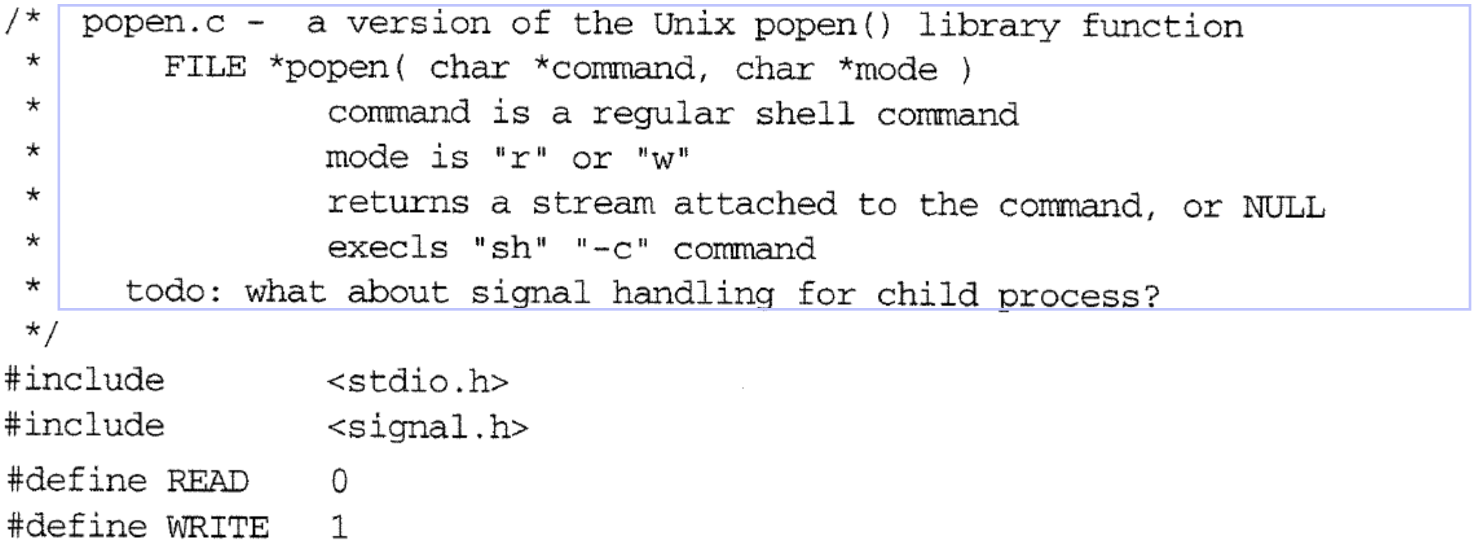


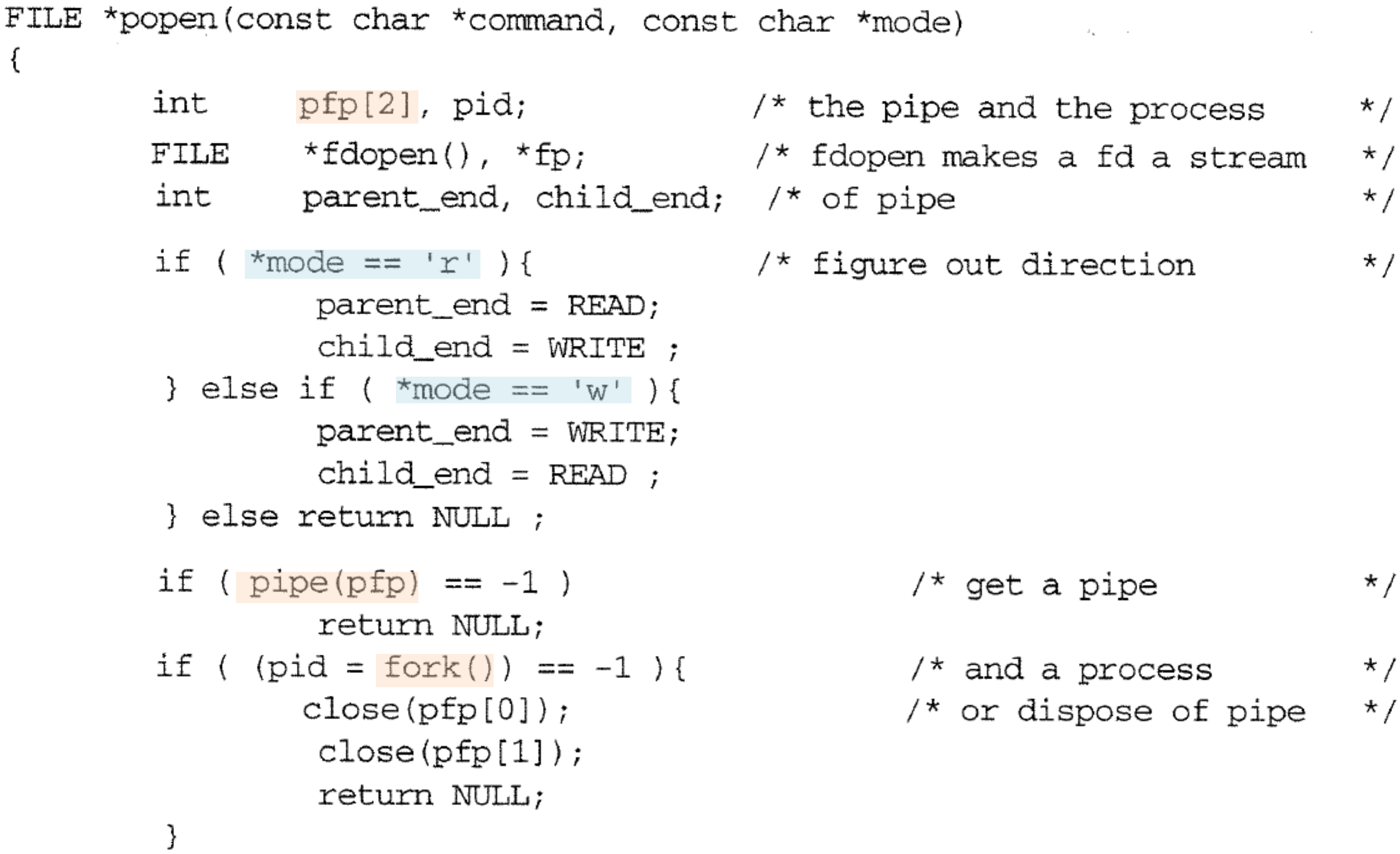
FIGURE 11.6

Reading from a shell command.



```
/*  popen.c -  a version of the Unix popen() library function
 *      FILE *popen( char *command, char *mode )
 *              command is a regular shell command
 *              mode is "r" or "w"
 *              returns a stream attached to the command, or NULL
 *              execls "sh" "-c" command
 *    todo: what about signal handling for child process?
 */
#include        <stdio.h>
#include        <signal.h>
#define READ    0
#define WRITE   1
```

```
const char* given_command = "who | sort";
const char* given_mode    = "r";


void main(){

    FILE* fp = popen(given_command, given_mode);
    char buf[100];
    int i=0;

    while(fgets(buf,100, fp)!=NULL)
        printf("%3d %s", i++, buf);
    pclose(fp);

    return 0;
}
```



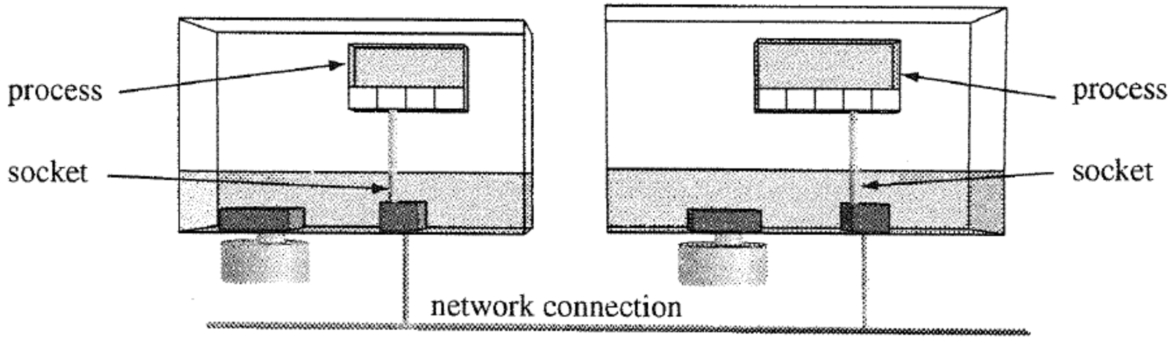
```
FILE *popen(const char *command, const char *mode)
{
        int     pfp[2], pid;            /* the pipe and the process     */
        FILE    *fdopen(), *fp;         /* fdopen makes a fd a stream   */
        int     parent_end, child_end;  /* of pipe                      */

        if ( *mode == 'r' ){            /* figure out direction         */
                parent_end = READ;
                child_end = WRITE ;
        } else if ( *mode == 'w' ){
                parent_end = WRITE;
                child_end = READ ;
        } else return NULL ;

        if ( pipe(pfp) == -1 )                  /* get a pipe           */
                return NULL;
        if ( (pid = fork()) == -1 ){            /* and a process        */
                close(pfp[0]);                  /* or dispose of pipe   */
                close(pfp[1]);
                return NULL;
        }
        /* --------------- parent code here ------------------- */
        /*   need to close one end and fdopen other end         */

        if ( pid > 0 ){
                if (close( pfp[child_end] ) == -1 )
                        return NULL;
               return fdopen( pfp[parent_end] , mode );   /* same mode */
        }

        /* --------------- child code here --------------------- */
        /*   need to redirect stdin or stdout then exec the cmd  */

       if ( close(pfp[parent_end]) == -1 )      /* close the other end  */
               exit(1);                         /* do NOT return        */

        if ( dup2(pfp[child_end], child_end) == -1 )
                exit(1);

       if ( close(pfp[child_end]) == -1 )        /* done with this one   */
                exit(1);

                                                /* all set to run cmd   */
        execl( "/bin/sh", "sh", "-c", command, NULL );
        exit(1);
}
```

Contents

- bc: A Unix Calculator
- popen: Making Process Look like Files
- Sockets: Connecting to Remote Processes

Sockets

- Limitations of pipes
  - It can only connect related processes : ex) parent/child
  - It can only connect processes on the same computer
- Unix provides another method of inter-process communication : sockets
- Sockets allow processes to create pipe-like connections to unrelated processes and even to process on other computers



Important Concepts in Socket Programming

- **Client and server**
  - In Unix terms, server is a program, not a computer
  - A server process wait for a request processes it and loops back to take the next request but a client does not loop.
- **Hostname and port:**
  - A server on the internet is a process running on a computer
  - The computer is called the host, and it has a hostname and a port number.
- **Address family**
  - Like phone-number or zip code, a host has an address.
- **Protocol**
  - A set of rules that governs interaction between client and server.

Example: Time Server and Client

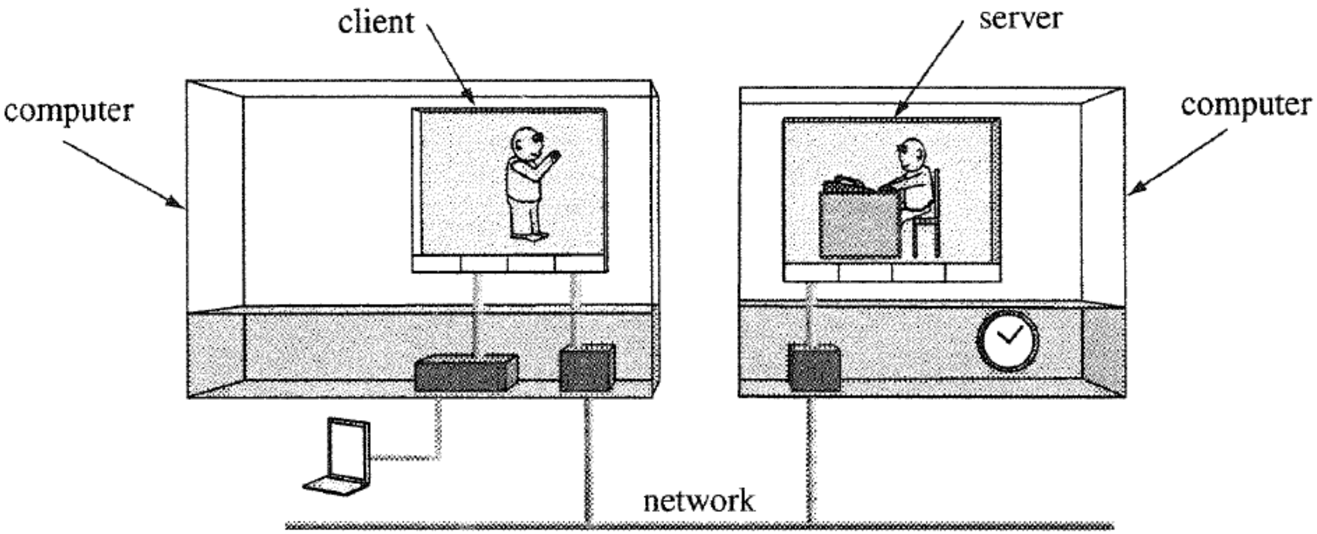
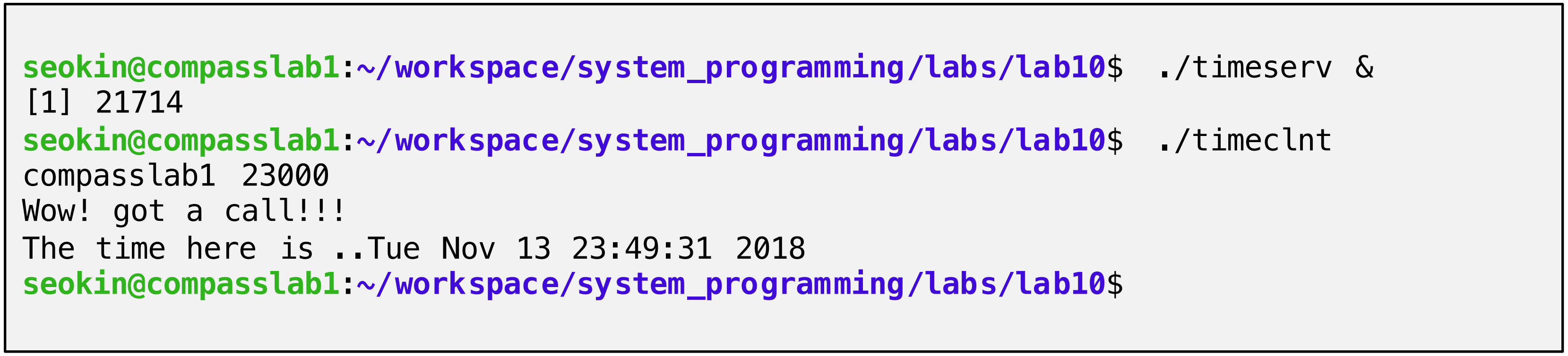


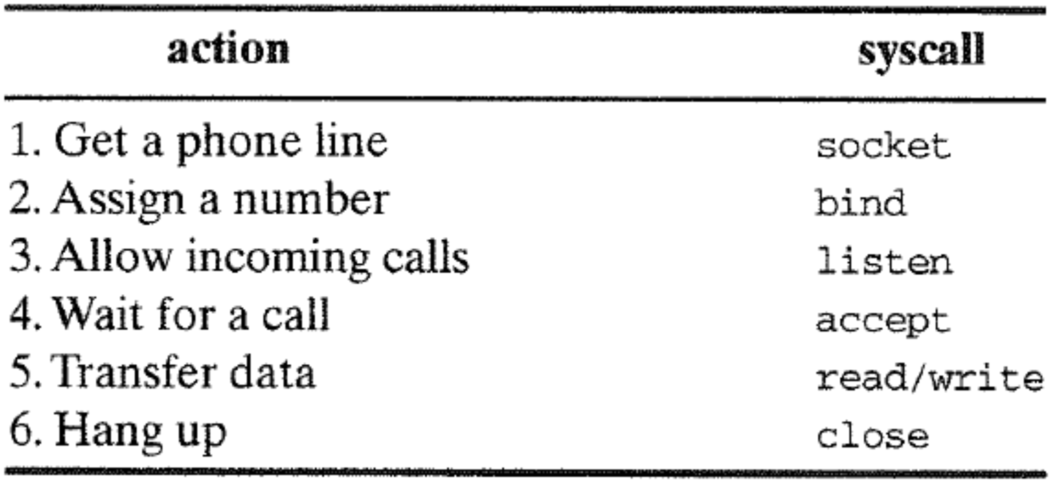
FIGURE 11.9

Processes on different computers.

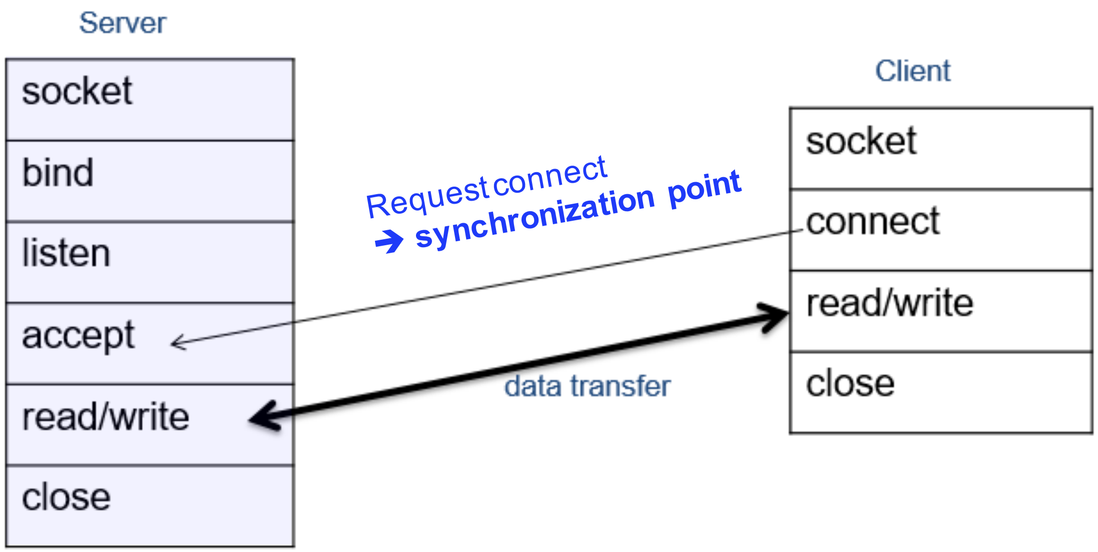


```
seokin@compasslab1:~/workspace/system_programming/labs/lab10$  ./timeserv &
[1] 21714
seokin@compasslab1:~/workspace/system_programming/labs/lab10$  ./timeclnt
compasslab1 23000
Wow! got a call!!!
The time here is ..Tue Nov 13 23:49:31 2018
seokin@compasslab1:~/workspace/system_programming/labs/lab10$
```

Writing `timeserv.c`: A Time Server



action	syscall
1. Get a phone line	`socket`
2. Assign a number	`bind`
3. Allow incoming calls	`listen`
4. Wait for a call	`accept`
5. Transfer data	`read/write`
6. Hang up	`close`



Server

socket

bind

listen

accept

read/write

close

Client

socket

connect

read/write

close

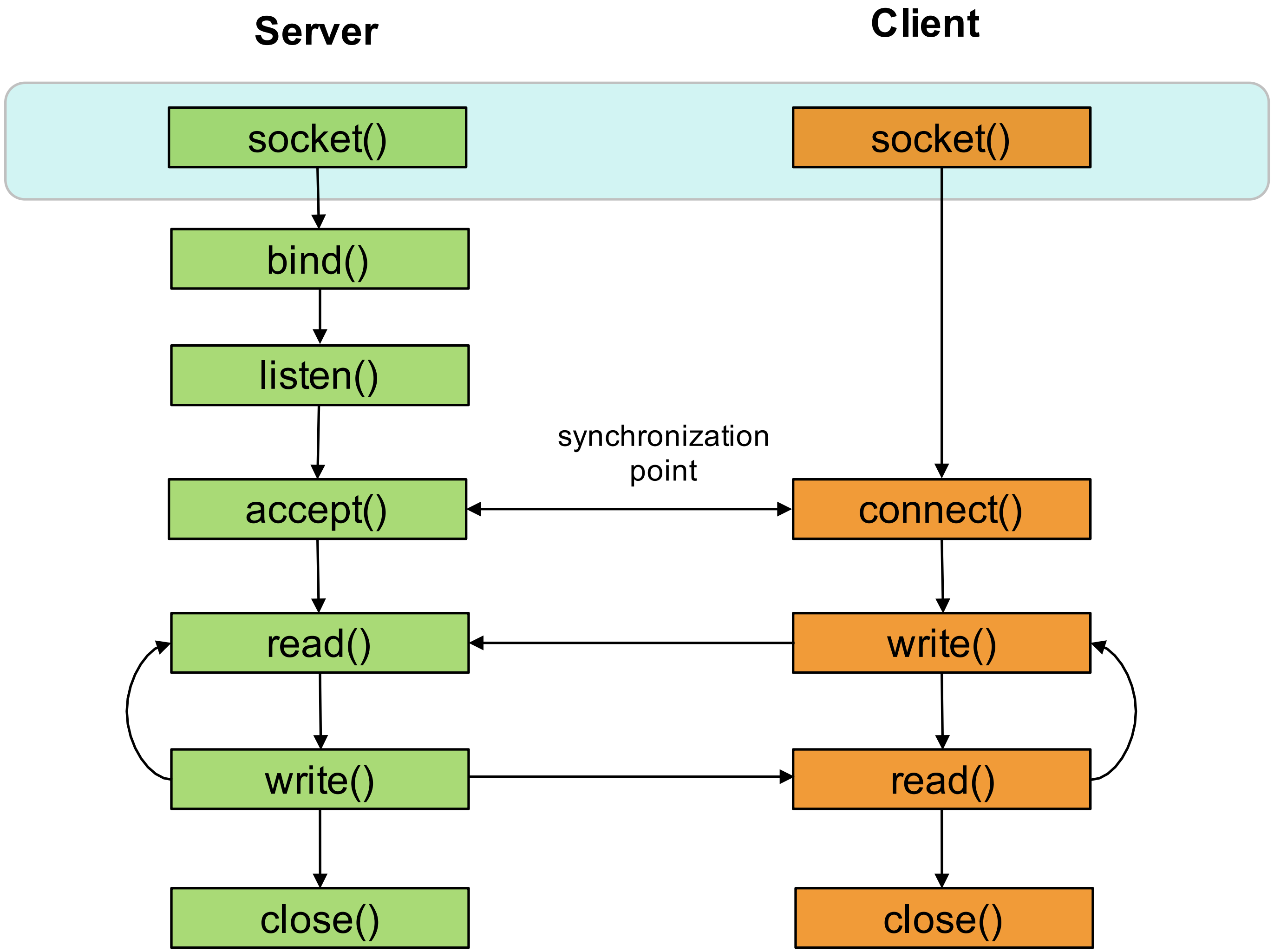
Request connect

→ **synchronization point**

data transfer

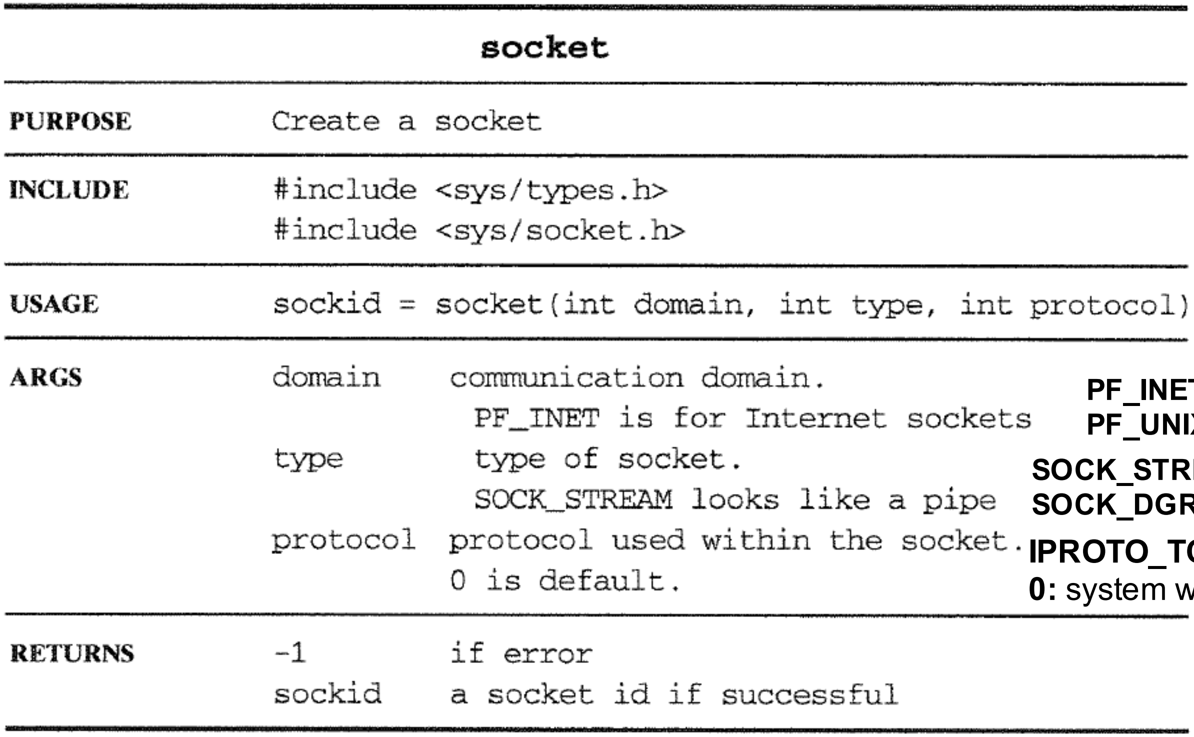
Writing `timeserv.c`: A Time Server

- Step 1: Ask kernel for a socket



Writing `timeserv.c`: A Time Server

- Step 1: Ask kernel for a socket



socket		
PURPOSE	Create a socket	

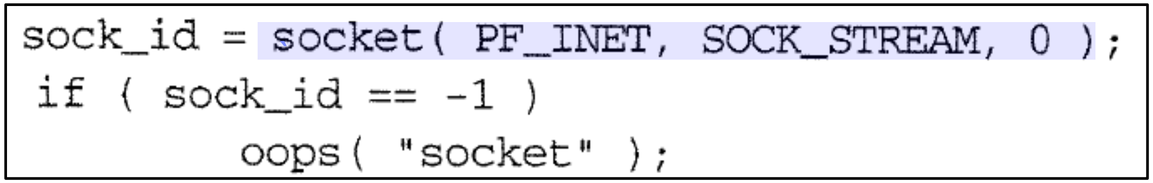
INCLUDE	#include <sys/types.h>
	#include <sys/socket.h>	

USAGE	sockid = socket(int domain, int type, int protocol)	
ARGS	domain	communication domain. PF_INET is for Internet sockets
	type	type of socket. SOCK_STREAM looks like a pipe
	protocol	protocol used within the socket. 0 is default.
RETURNS	-1	if error
	sockid	a socket id if successful

PF_INET : IPv4 protocols (internet addresses)
PF_UNIX : local communication

SOCK_STREAM : reliable, 2-way, connection-based service
SOCK_DGRAM : unreliable, connectionless

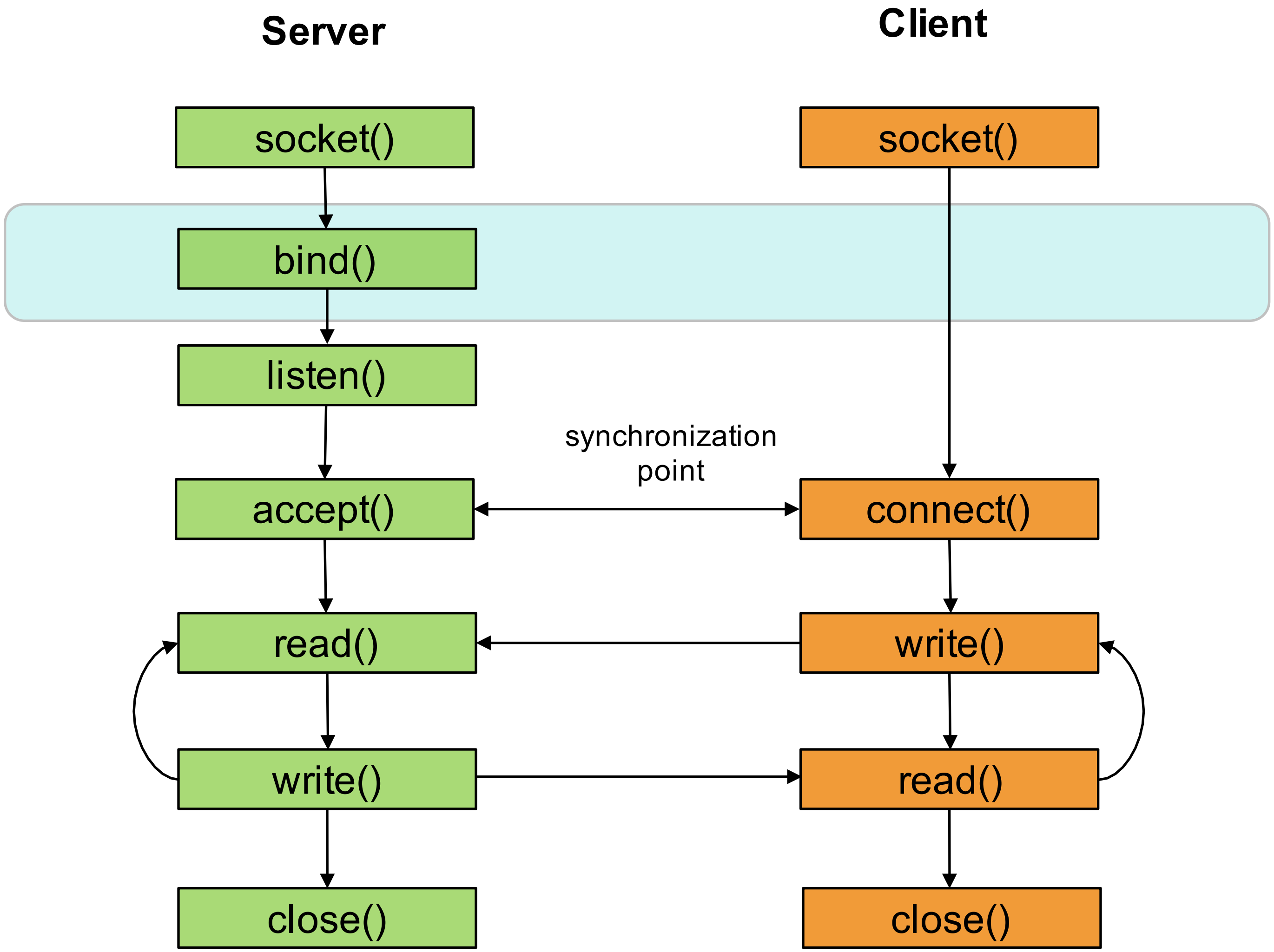
IPROTO_TCP, IPPROTO_UDP
0: system will determine the protocol



```
sock_id = socket( PF_INET, SOCK_STREAM, 0 );
 if ( sock_id == -1 )
          oops( "socket" );
```

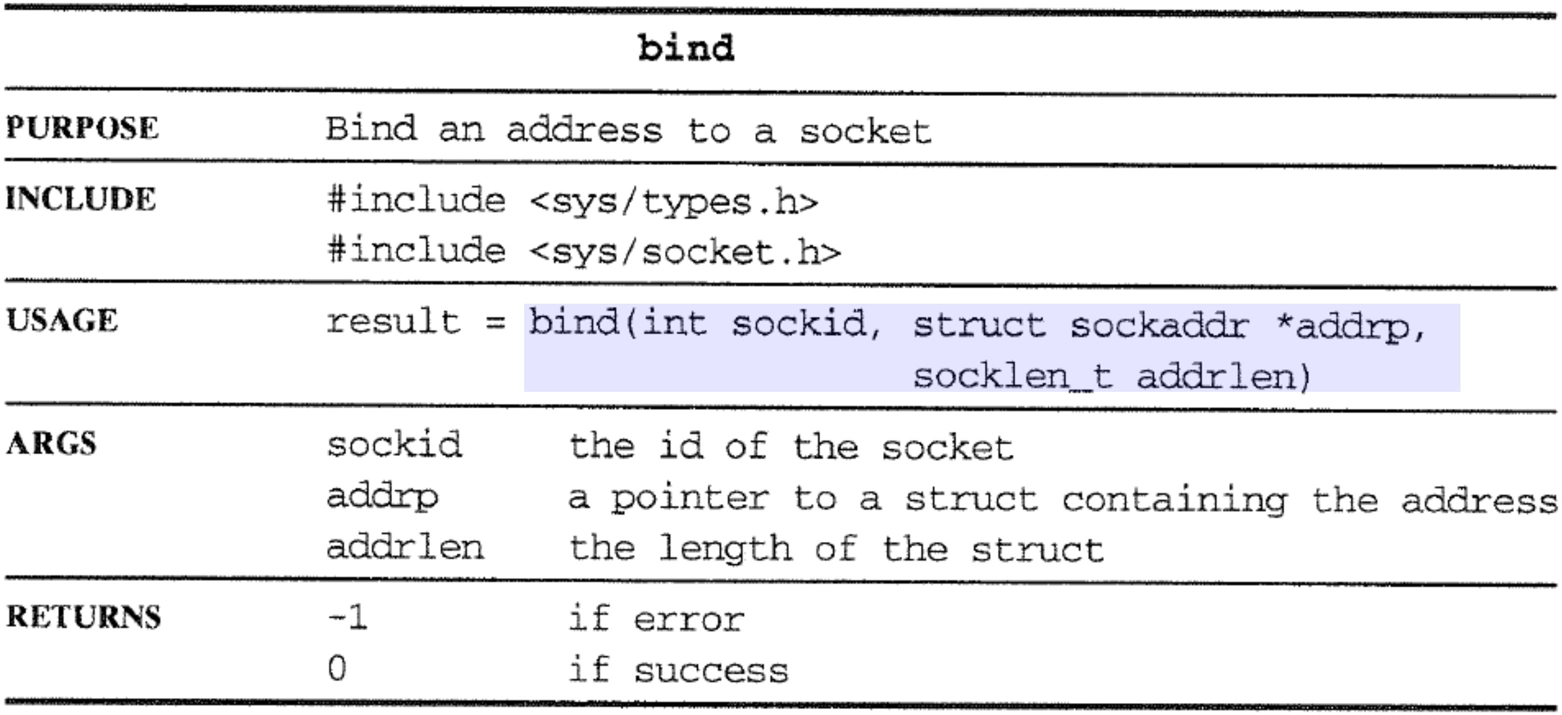
Writing `timeserv.c`: A Time Server

- Step 2: Bind address to socket; Address is host, port.



Writing `timeserv.c`: A Time Server

- Step 2: Bind address to socket; Address is host, port.

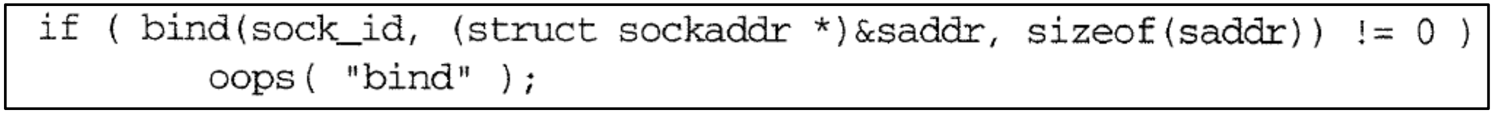


bind

PURPOSE	Bind an address to a socket	

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

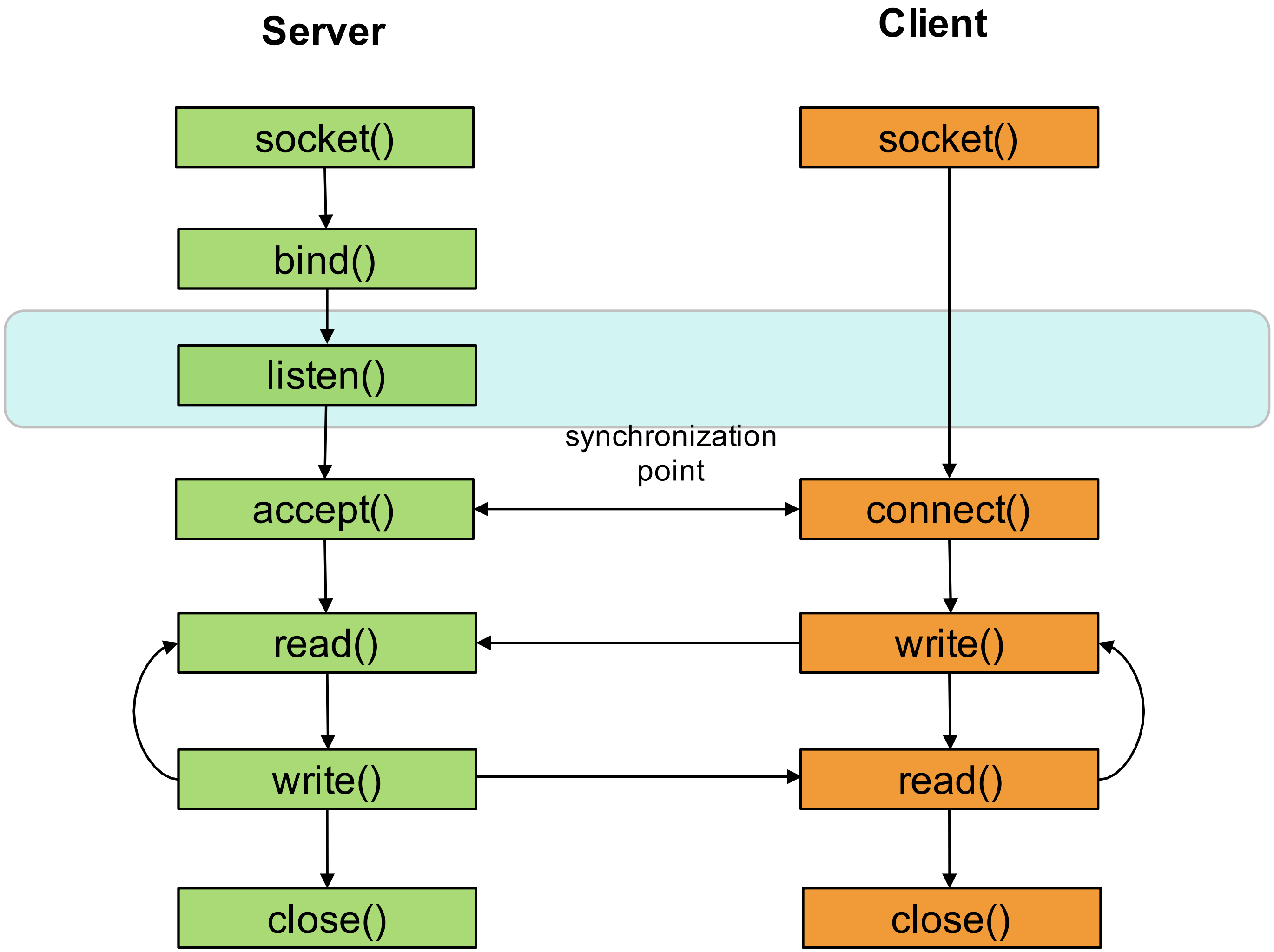
USAGE	result = bind(int sockid, struct sockaddr *addrp, socklen_t addrlen)	
ARGS	sockid	the id of the socket
	addrp	a pointer to a struct containing the address
	addrlen	the length of the struct
RETURNS	-1	if error
	0	if success



```
 if ( bind(sock_id, (struct sockaddr *)&saddr, sizeof(saddr)) != 0 )
         oops( "bind" );
```

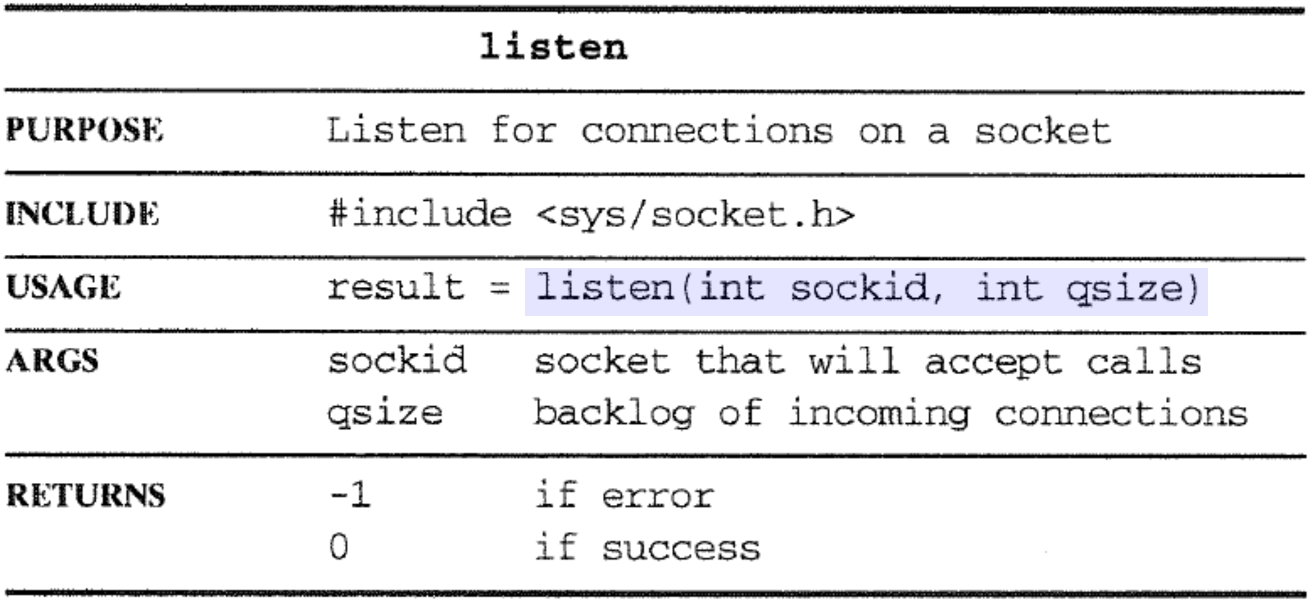
Writing `timeserv.c`: A Time Server

- Step 3: Allow incoming calls on socket



Writing `timeserv.c`: A Time Server

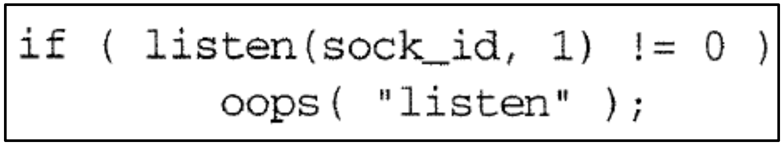
- Step 3: Allow incoming calls on socket



listen

PURPOSE	Listen for connections on a socket	
INCLUDE	#include <sys/socket.h>	
USAGE	result = listen(int sockid, int qsize)	
ARGS	sockid qsize	socket that will accept calls backlog of incoming connections

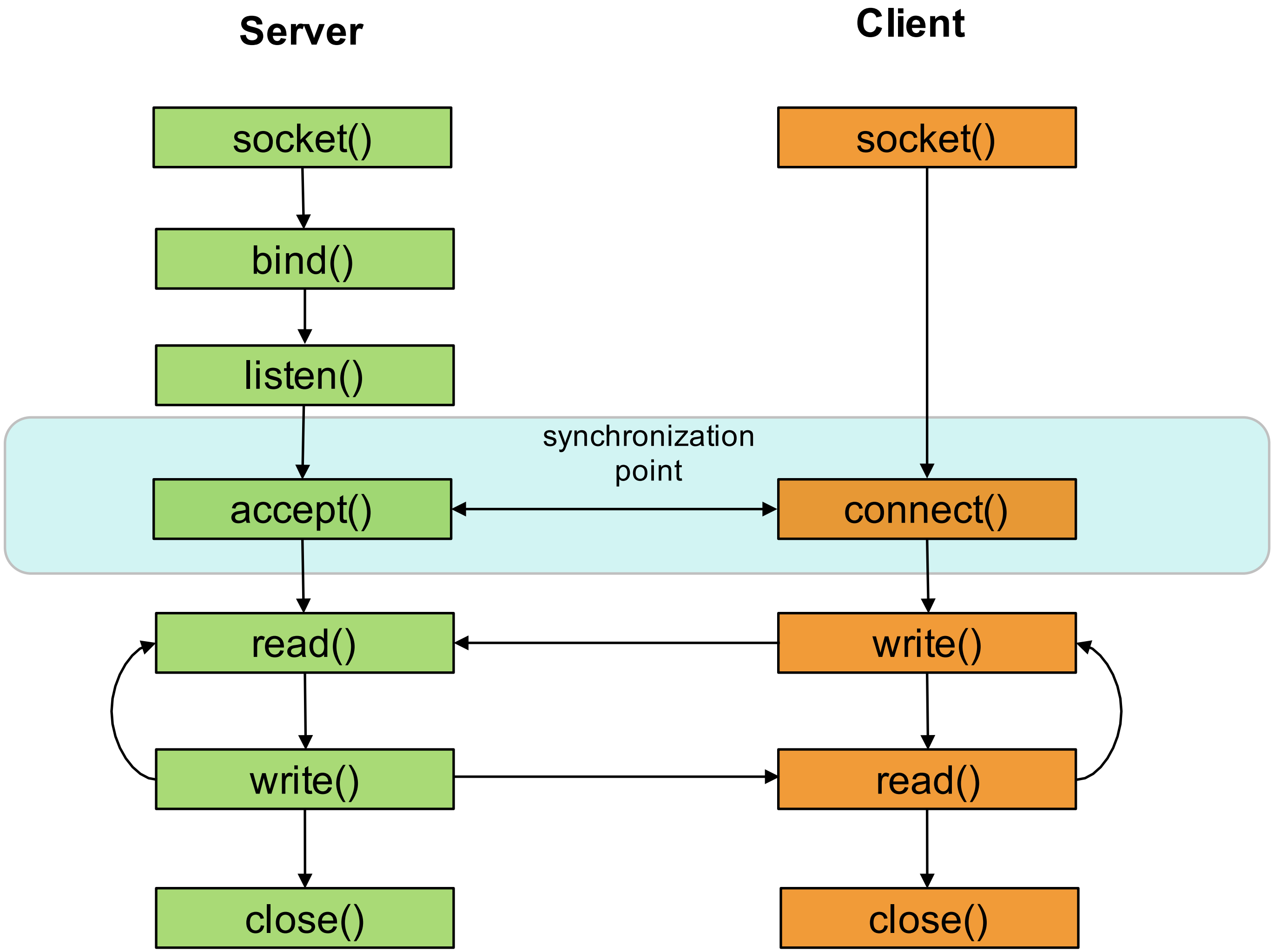
RETURNS	-1
	0
		if error
		if success



```
if ( listen(sock_id, 1) != 0 )
        oops( "listen" );
```

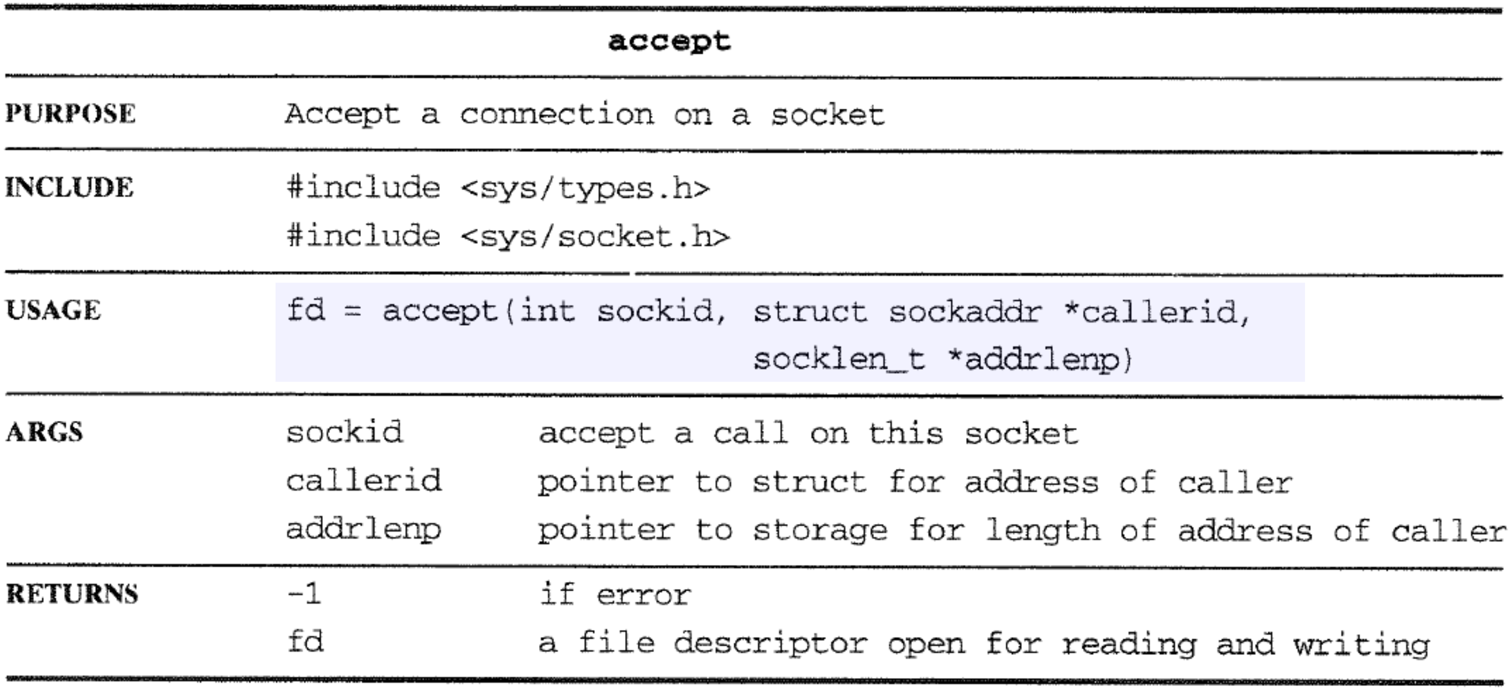
Writing `timeserv.c`: A Time Server

- Step 4: Wait For/Accept a connection



Writing `timeserv.c`: A Time Server

- Step 4: Wait For/Accept a connection

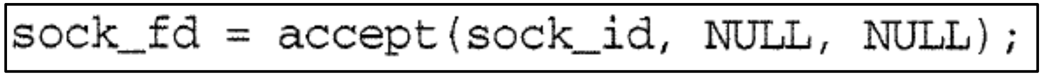


accept

PURPOSE	Accept a connection on a socket	

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

USAGE	fd = accept(int sockid, struct sockaddr *callerid, socklen_t *addrlenp)	
ARGS	sockid	accept a call on this socket
	callerid	pointer to struct for address of caller
	addrlenp	pointer to storage for length of address of caller
RETURNS	-1	if error
	fd	a file descriptor open for reading and writing

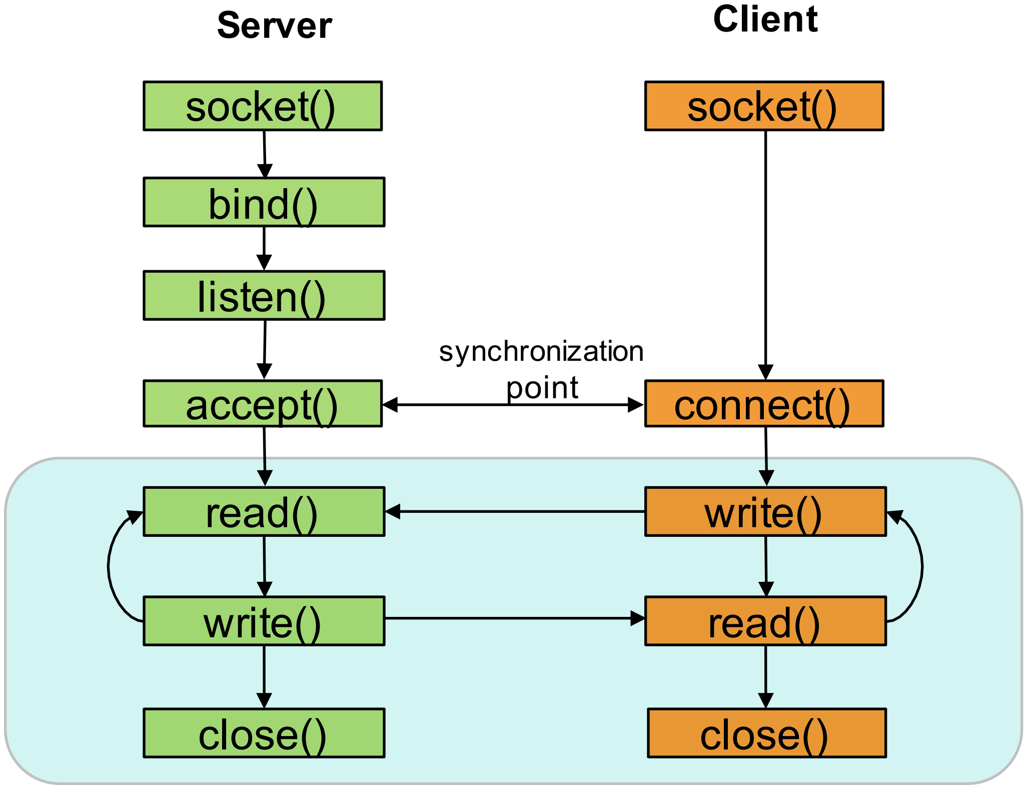


```
sock_fd = accept(sock_id, NULL, NULL);
```

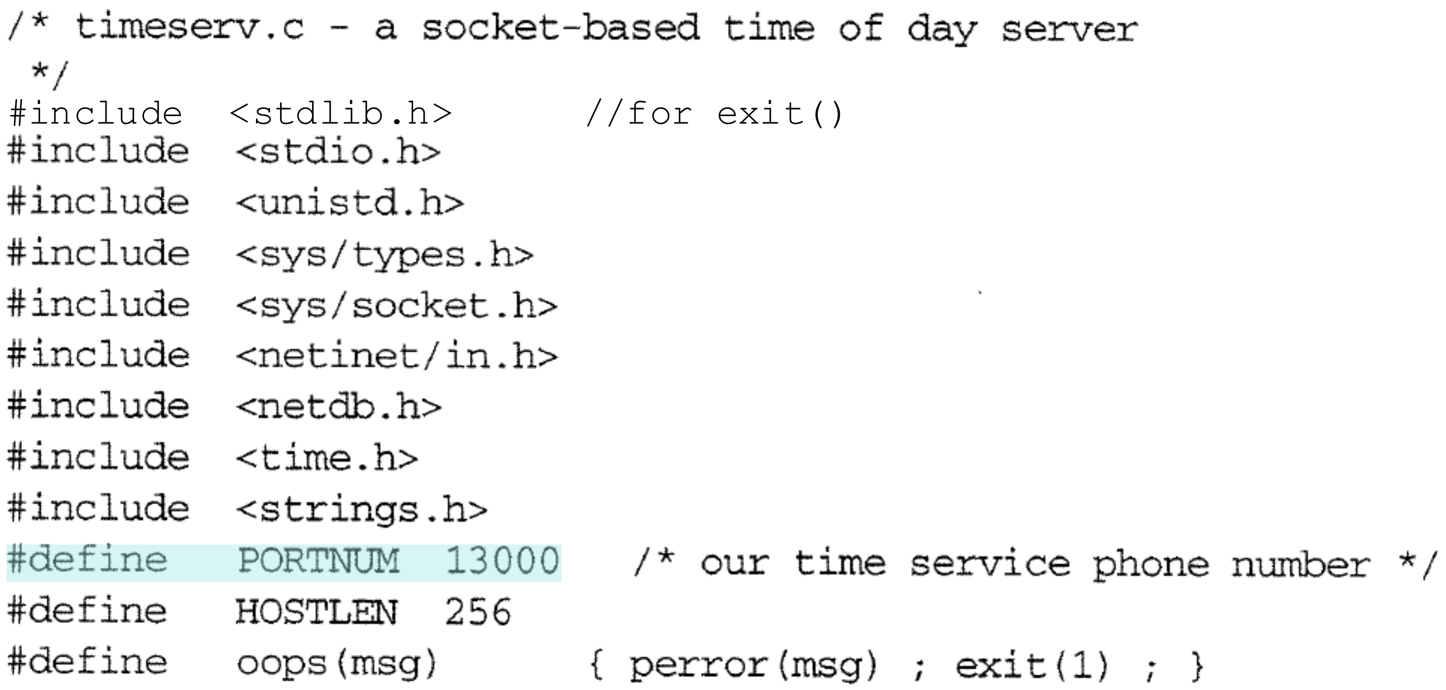
Writing `timeserv.c`: A Time Server

- Step 5: Transfer Data

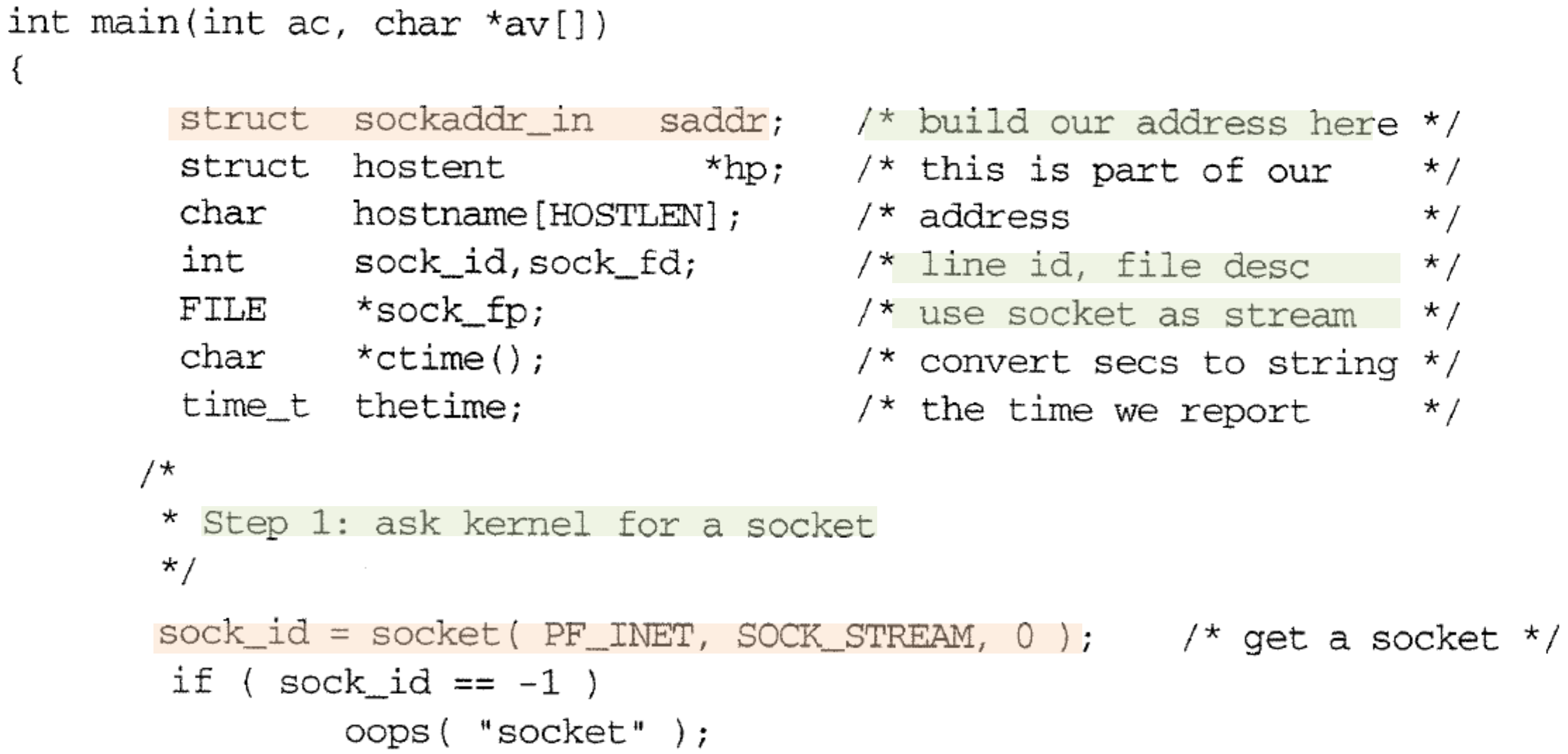
- Step 6: Close Connection



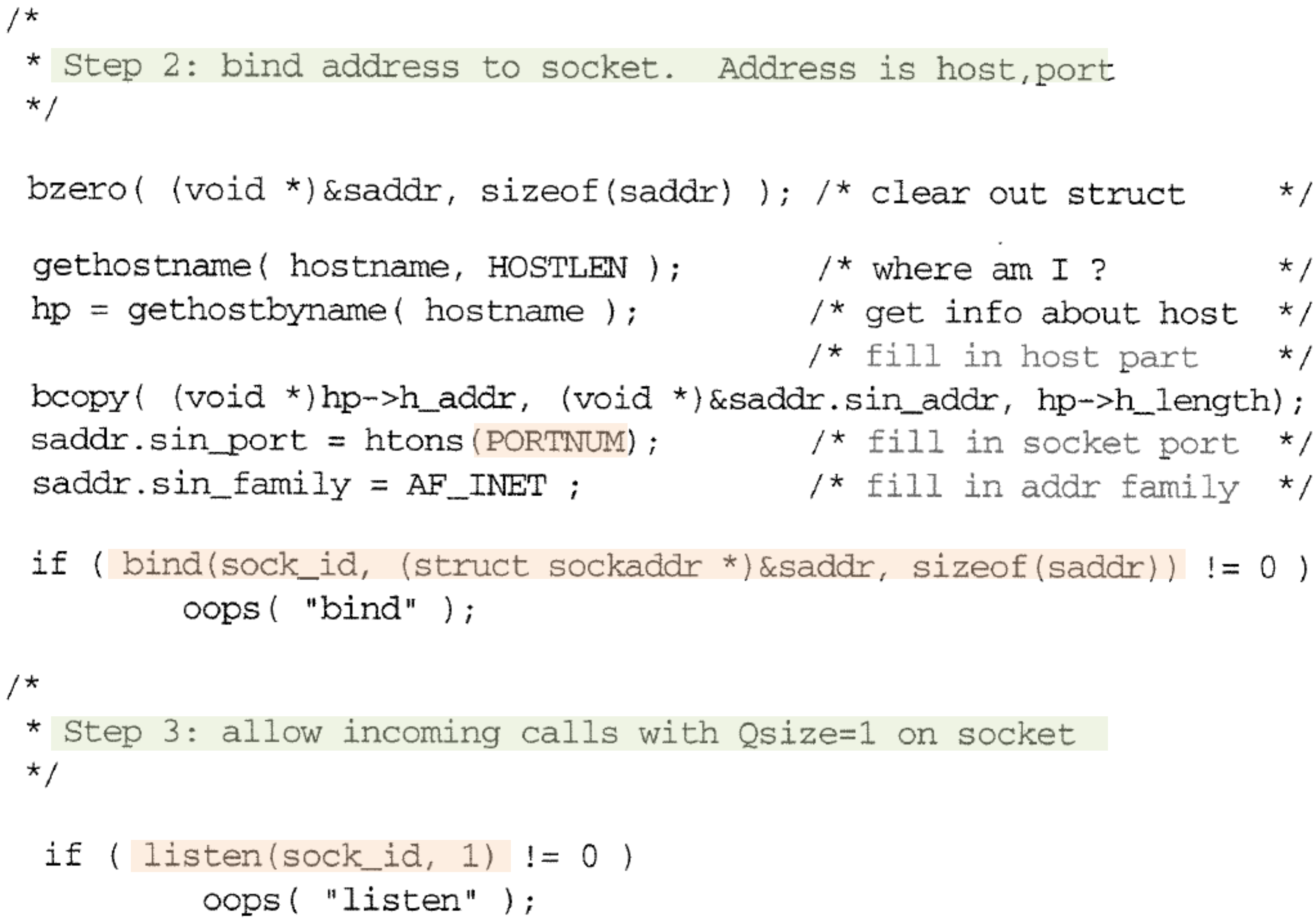
Writing `timeserv.c`: A Time Server



```
/* timeserv.c - a socket-based time of day server
 */
#include  <stdlib.h>      //for exit()
#include  <stdio.h>
#include  <unistd.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>
#include  <netdb.h>
#include  <time.h>
#include  <strings.h>
#define   PORTNUM  13000   /* our time service phone number */
#define   HOSTLEN  256
#define   oops(msg)      { perror(msg) ; exit(1) ; }
```



```
int main(int ac, char *av[])
{
        struct  sockaddr_in   saddr;    /* build our address here */
        struct  hostent          *hp;   /* this is part of our    */
        char    hostname[HOSTLEN];      /* address                */
        int     sock_id,sock_fd;        /* line id, file desc     */
        FILE    *sock_fp;               /* use socket as stream   */
        char    *ctime();               /* convert secs to string */
        time_t  thetime;                /* the time we report     */

       /*
        * Step 1: ask kernel for a socket
        */

       sock_id = socket( PF_INET, SOCK_STREAM, 0 );    /* get a socket */
        if ( sock_id == -1 )
                oops( "socket" );
```



```
/*
 * Step 2: bind address to socket.  Address is host,port
 */

 bzero( (void *)&saddr, sizeof(saddr) ); /* clear out struct     */

 gethostname( hostname, HOSTLEN );        /* where am I ?         */
 hp = gethostbyname( hostname );          /* get info about host  */
                                          /* fill in host part    */
 bcopy( (void *)hp->h_addr, (void *)&saddr.sin_addr, hp->h_length);
 saddr.sin_port = htons(PORTNUM);         /* fill in socket port  */
 saddr.sin_family = AF_INET ;             /* fill in addr family  */

 if ( bind(sock_id, (struct sockaddr *)&saddr, sizeof(saddr)) != 0 )
        oops( "bind" );

/*
 * Step 3: allow incoming calls with Qsize=1 on socket
 */

  if ( listen(sock_id, 1) != 0 )
        oops( "listen" );
```

* htons(): translate a short integer from host byte order to network byte order

```
      /*
       * main loop: accept(), write(), close()
       */

        while ( 1 ){
              sock_fd = accept(sock_id, NULL, NULL); /* wait for call */
               printf("Wow! got a call!\n");
              if ( sock_fd == -1 )
                      oops( "accept" );        /* error getting calls  */

              sock_fp = fdopen(sock_fd,"w");   /* we'll write to the   */
              if ( sock_fp == NULL )           /* socket as a stream   */
                      oops( "fdopen" );        /* unless we can't      */

              thetime = time(NULL);            /* get time             */
                                               /* and convert to strng */
              fprintf( sock_fp, "The time here is .." );
              fprintf( sock_fp, "%s", ctime(&thetime) );
              fclose( sock_fp );               /* release connection   */
        }
}
```

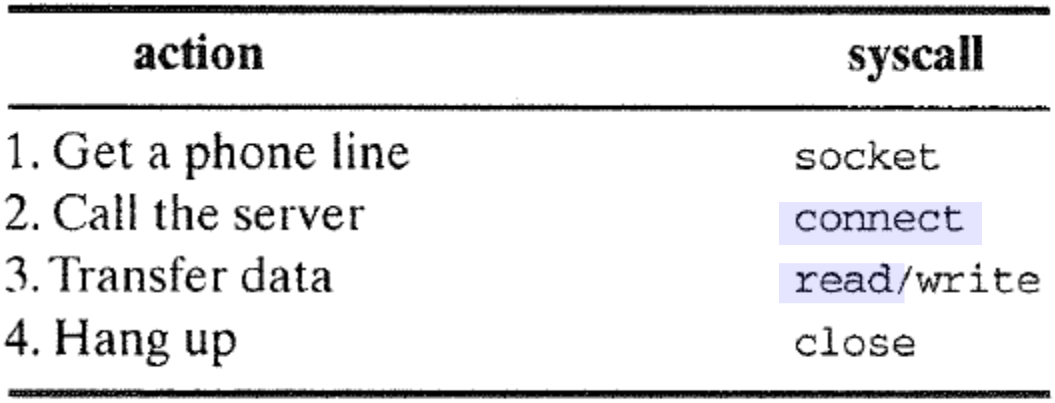
```
struct hostent {
   char *h_name; /* official name of host */
   char **h_aliases; /* alias list */
   int h_addrtype; /* host address type */
   int h_length; /* length of address */
   char **h_addr_list; /* list of addresses */
};
#define h_addr h_addr_list[0] /* for backward compatibility */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

```
struct sockaddr {
  unsigned short sa_family; /* Address family (e.g. AF_INET) */
  char sa_data[14]; /* Family-specific address information */
}
```

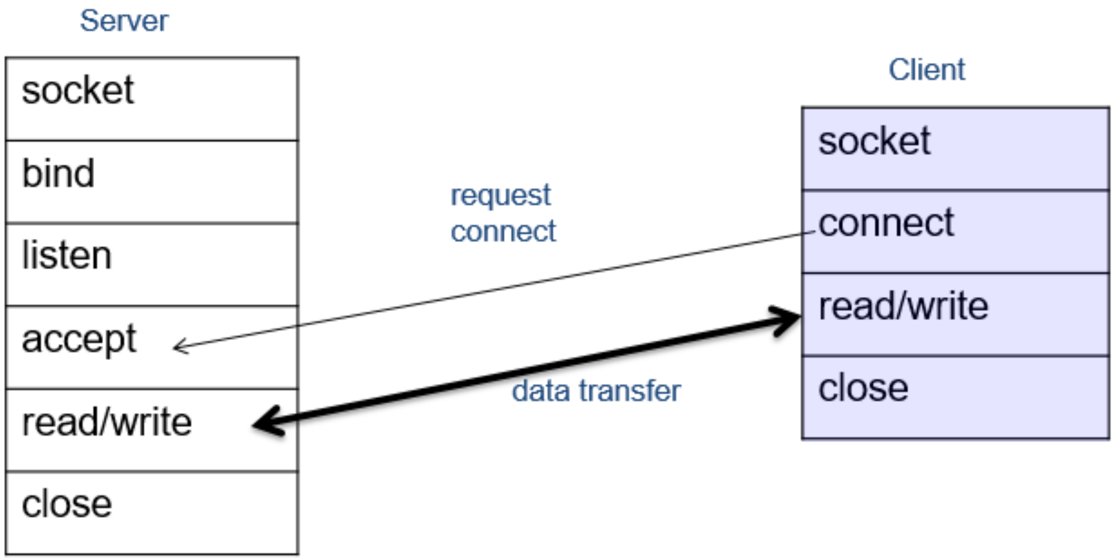
```
struct in_addr {
  unsigned long s_addr; /* Internet address (32 bits) */
}
```

```
struct sockaddr_in {
  unsigned short sin_family; /* Internet protocol (AF_INET) */
  unsigned short sin_port; /* Address port (16 bits) */
  struct in_addr sin_addr; /* Internet address (32 bits) */
  char sin_zero[8]; /* Not used */
}
```

Writing `timeclnt.c`: a Time Client



action	syscall
1. Get a phone line	`socket`
2. Call the server	`connect`
3. Transfer data	`read/write`
4. Hang up	`close`



Server

socket

bind

listen

accept

read/write

close

request connect

data transfer

Client

socket

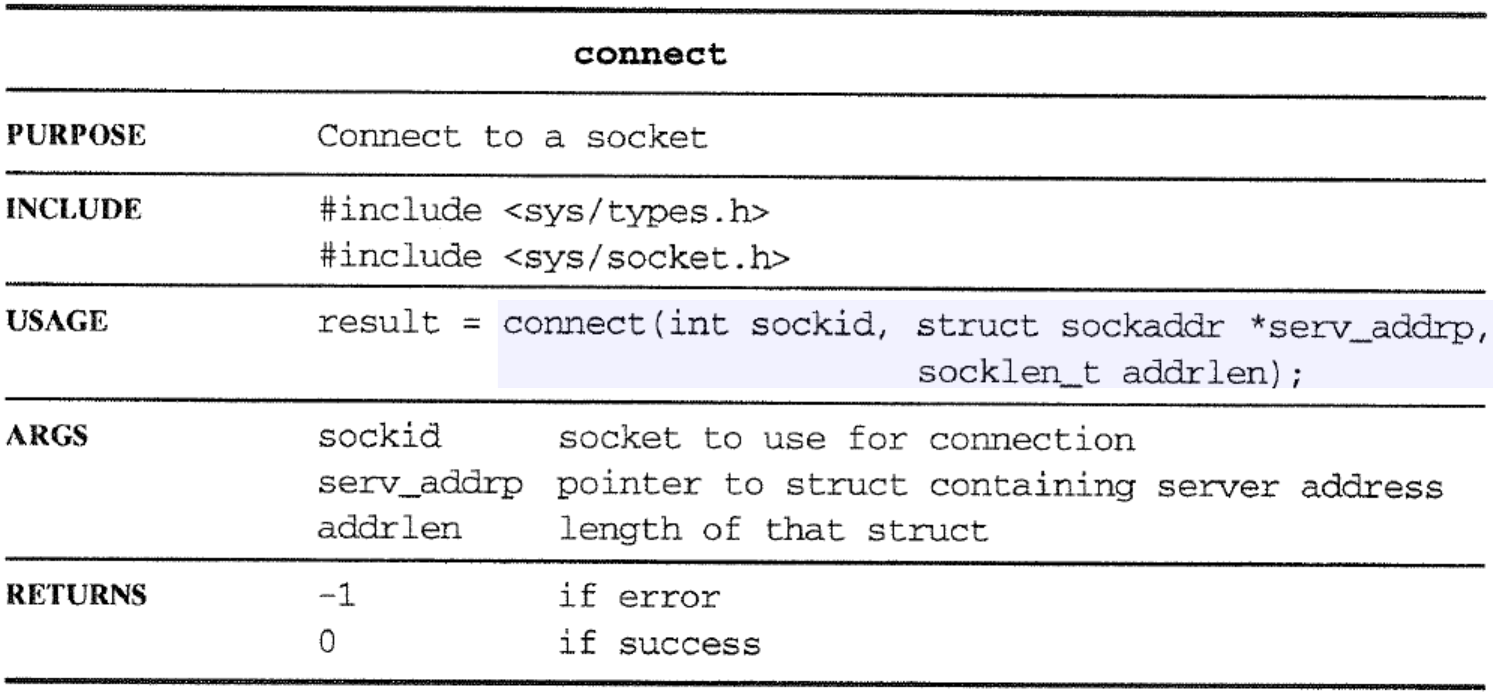
connect

read/write

close

Writing `timeclnt.c`: a Time Client

- Establish a connection



connect

PURPOSE	Connect to a socket	

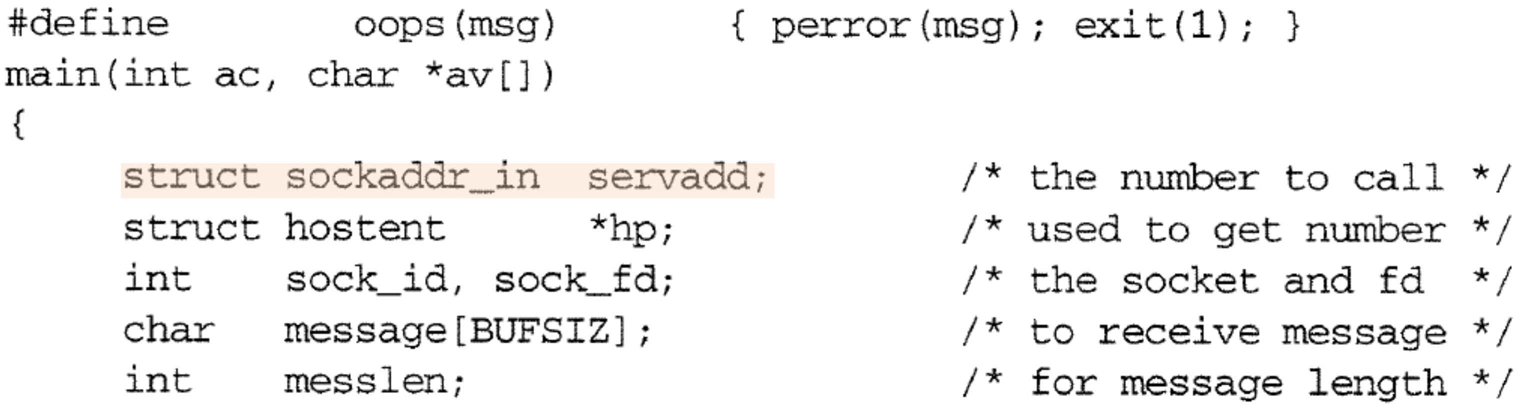
INCLUDE	#include <sys/types.h>
	#include <sys/socket.h>	

USAGE	result = connect(int sockid, struct sockaddr *serv_addrp, socklen_t addrlen);	
ARGS	sockid serv_addrp addrlen	socket to use for connection pointer to struct containing server address length of that struct

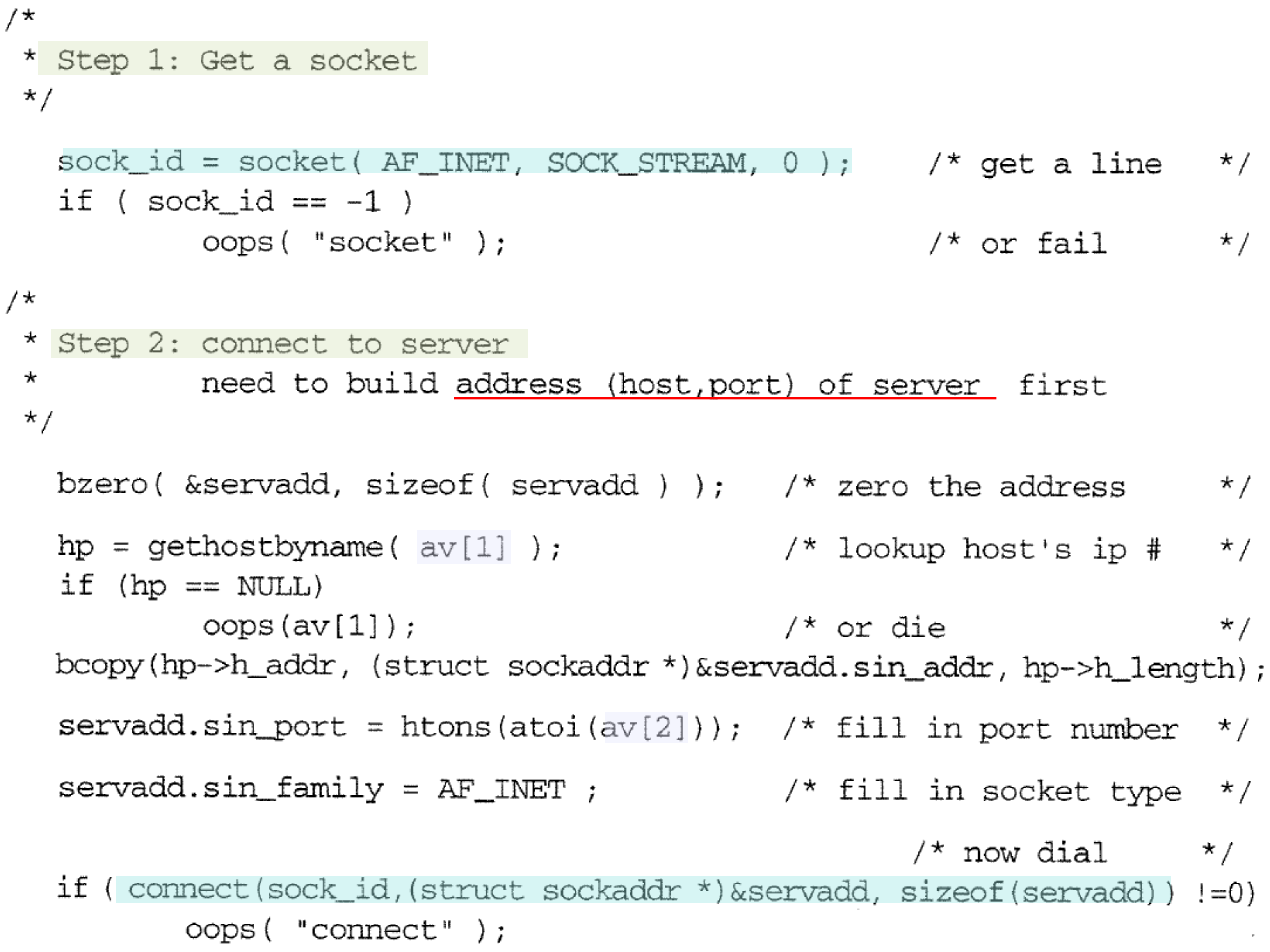
RETURNS	-1
	0
		if error
		if success

```
/* timeclnt.c - a client for timeserv.c
 *                usage: timeclnt hostname portnumber
 */
#include       <stdio.h>
#include       <sys/types.h>
#include       <sys/socket.h>
#include       <netinet/in.h>
#include       <netdb.h>
```

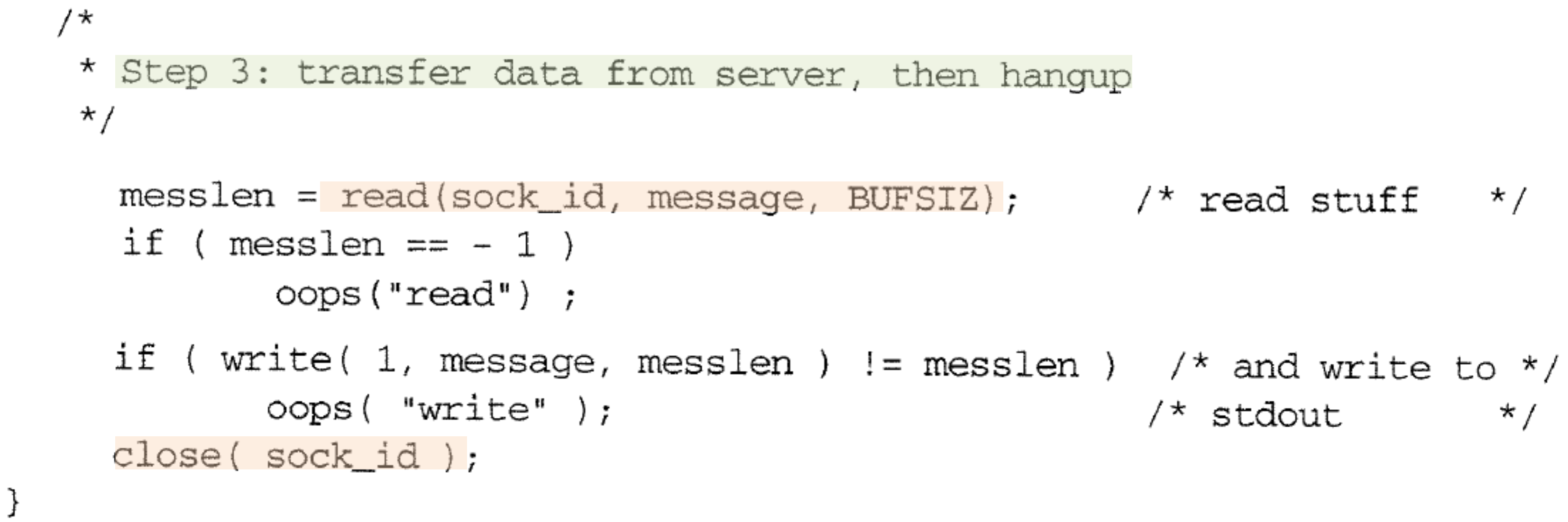
```
#include        <stdlib.h>
#include        <strings.h>
```



```
#define        oops(msg)         { perror(msg); exit(1); }
main(int ac, char *av[])
{
     struct sockaddr_in  servadd;         /* the number to call */
     struct hostent      *hp;             /* used to get number */
     int    sock_id, sock_fd;             /* the socket and fd  */
     char   message[BUFSIZ];              /* to receive message */
     int    messlen;                      /* for message length */
```



```
/*
 * Step 1: Get a socket
 */

  sock_id = socket( AF_INET, SOCK_STREAM, 0 );      /* get a line     */
  if ( sock_id == -1 )
          oops( "socket" );                         /* or fail        */

/*
 * Step 2: connect to server
 *        need to build address (host,port) of server  first
 */

  bzero( &servadd, sizeof( servadd ) );    /* zero the address       */

  hp = gethostbyname( av[1] );             /* lookup host's ip #     */
  if (hp == NULL)
          oops(av[1]);                     /* or die                 */
  bcopy(hp->h_addr, (struct sockaddr *)&servadd.sin_addr, hp->h_length);

  servadd.sin_port = htons(atoi(av[2]));   /* fill in port number    */

  servadd.sin_family = AF_INET ;           /* fill in socket type    */

                                                 /* now dial      */
  if ( connect(sock_id,(struct sockaddr *)&servadd, sizeof(servadd)) !=0)
         oops( "connect" );
```



```
    /*
     * Step 3: transfer data from server, then hangup
     */

      messlen = read(sock_id, message, BUFSIZ);      /* read stuff    */
      if ( messlen == - 1 )
              oops("read") ;
      if ( write( 1, message, messlen ) != messlen )  /* and write to */
              oops( "write" );                        /* stdout       */
      close( sock_id );
}
```