System Programming
(ELEC462)

Connecting to Processes Near and Far

Dukyun Nam

HPC Lab@KNU

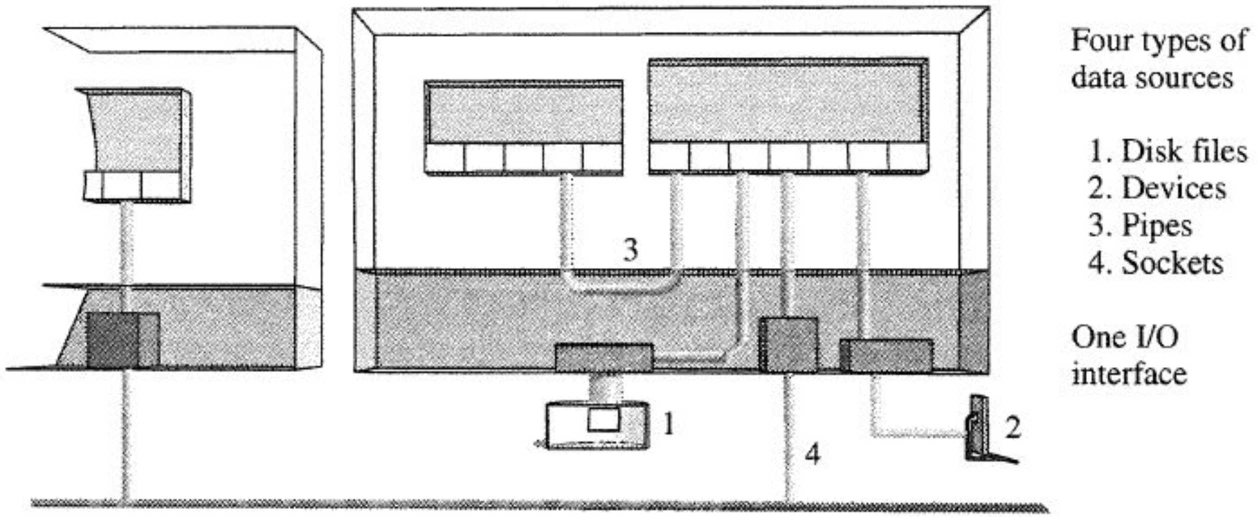
Contents

Introduction

- Ideas and Skills
  - The client/server model
  - Using pipes for two-way communication
  - Coroutines
  - The file/process similarity
  - Sockets: Why, What, How?
  - Network services
  - Using sockets for client/server programs
- System Calls and Functions
  - `fdopen`, `popen`, `socket`
  - `bind`, `listen`, `accept`, `connect`

Four Types of Data Sources

- Unix presents one interface, even though data come from different types of sources
  - (1)/(2) Disk/device files
    - Use `open` to connect
    - Use `read` and `write` to transfer data
  - (3) Pipes
    - Use `pipe` to create
    - Use `fork` to share
    - Use `read` and `write` to transfer data
  - (4) Sockets
    - Use `socket`, `listen`, and `connect` to connect
    - Use `read` and `write` to transfer data
    - Basis on a *client-server model*



< One interface, different sources >

`bc`: A Unix/Linux Basic Calculator

- `bc` has variables, loops, and functions
  - Can handle very long numbers
    - The trailing backslashes indicate continuation

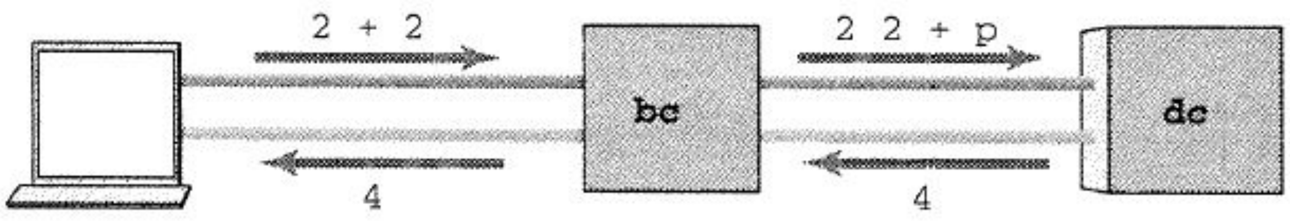
```
dynam@DESKTOP-Q4IJBP7:~/lab12$ bc
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
17^123
22142024630120207359320573764236957523345603216987331732240497016947\
29282299663749675090635587202539117092799463206393818799003722068558\
0536286573569713
2+2
4
2*2+2
6
2+2/2
3
```

bc: A Unix/Linux Basic Calculator (cont.)

- But bc is NOT a real calculator but actually runs dc
  - dc: a stack-based (desktop) calculator requiring the user to

    (i) enter both values and then

    (ii) specify the operation
    - e.g., 2 + 2 = 4



< bc and dc as coroutines >

  - How bc works:
    - Reads the expression from stdin and parses out the values and the operation
    - Sends via a pipe the sequence commands, “2”, “2”, “+”, and “p” to dc
    - Later reads the result through the pipe it attached to stdout of dc
    - Forwards that message to the user

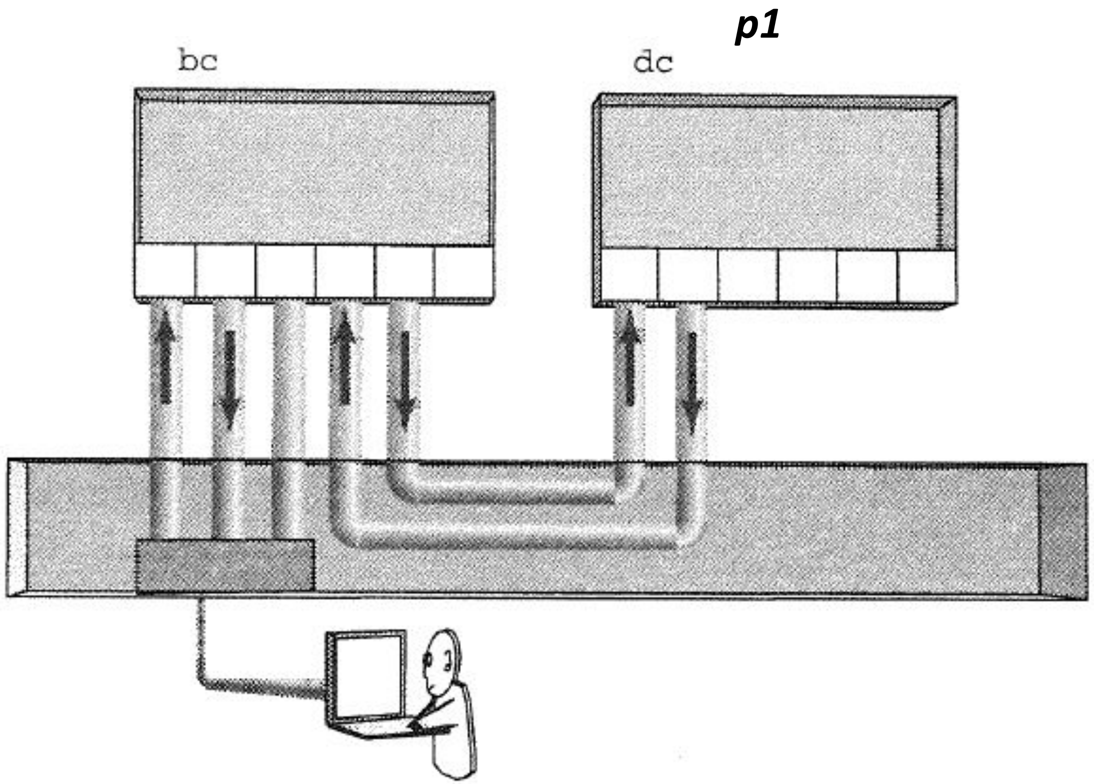
```
dynam@DESKTOP-Q4IJBP7:~/lab12$ echo '2 + 2' | bc
4
dynam@DESKTOP-Q4IJBP7:~/lab12$ echo '2 2 + p' | dc
4
```

  - How dc works:
    - Stacks up the received two values, applies the “+” operation,

      and then prints to stdout the value on the top of the stack

Coding `bc`: `pipe`, `fork`, `dup`, `exec`

- Data connections in the kernel <u>from user to `bc`</u> and <u>`bc` to `dc`</u>
- Guidelines
  - Create `todc` / `fromdc`
  - Create a process *p1* to run `dc` (via `fork`)
    - `bc`: will run in the parent
  - In *p1*, redirect to `stdin` and `stdout` to the pipes, and then `exec dc`
  - In the parent (`bc`),
    - Read and parse user input,
    - Write commands to `dc`
    - Read response from `dc`
    - Send response to user



< `bc`, `dc` and kernel >

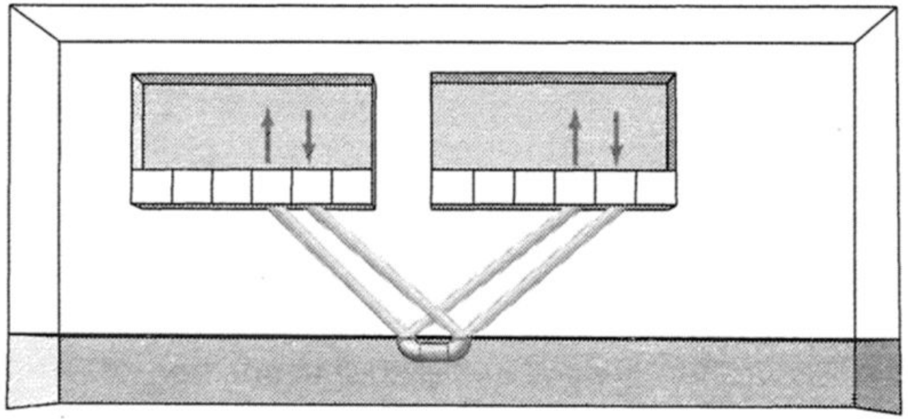
Tiny bc

- A simple version of `bc`
  - Uses `sscanf` to parse and speaks with `dc` through two pipes
- Program outline
  - a. Get two pipes
  - b. Fork (get another process)
  - c. In the child process to be `dc`, connect `stdin` and `out` to pipes then `execl dc`
  - d. The parent (`tinybc`) receives input and sends it via `pipe`
  - e. Then close `pipe`, and `dc` dies

```
          +-----------+                      +----------+
stdin  >0                >== pipetodc ====>             |
          |   tinybc    |                    |   dc -   |
stdout <1                <== pipefromdc ==<             |
          +-----------+                      +----------+
```

[Remind] Using `fork` to Share a Pipe

- Sharing a pipe: Interprocess data flow
    - Parent/child can *write* bytes to the *writing end* of the pipe
    - Parent/child can *read* bytes from the *reading end* of the pipe



- A pipe is "most effective" when one process writes data and the other processes reads the data on the same host
    - Of course, processes can read and write together

Writing tiny bc

- tinybc.c

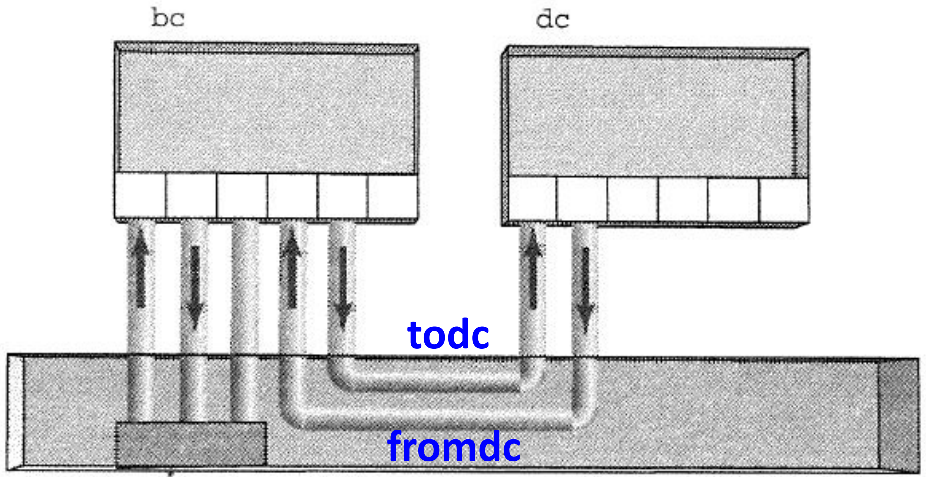
```
/**     tinybc.c        * a tiny calculator that uses dc to do its work
 **                     * demonstrates bidirectional pipes
 **                     * input looks like number op number which
 **                       tinybc converts into number \n number \n op \n p
 **                       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
 **/
```

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

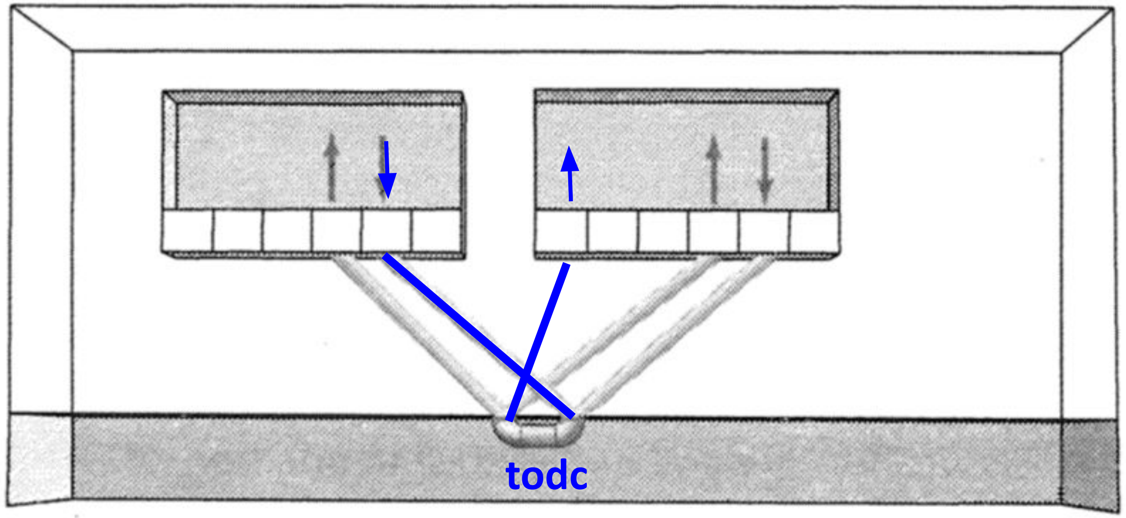
```
#include        <stdio.h>
#include        <stdlib.h>
#include        <unistd.h>
#include        <sys/wait.h>

#define oops(m,x)       { perror(m); exit(x); }

void be_dc(int *, int *);
void be_bc(int *, int *);
void fatal( char * );

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

        /* make two pipes */

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

        /* get a process for user interface */

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

        return 0;
}
```

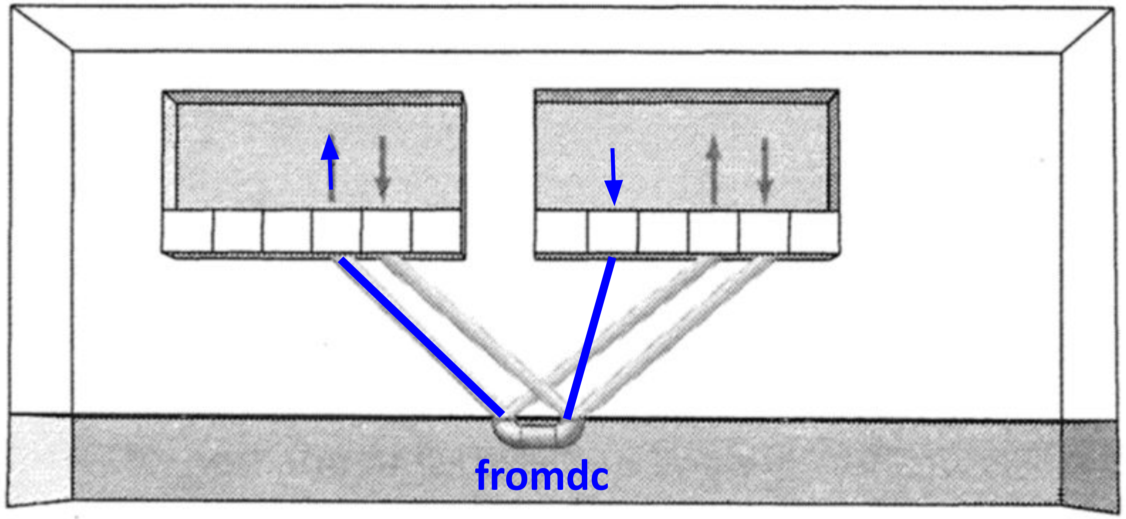
Writing tiny `bc` (cont.)

- `tinybc.c - be_bc`



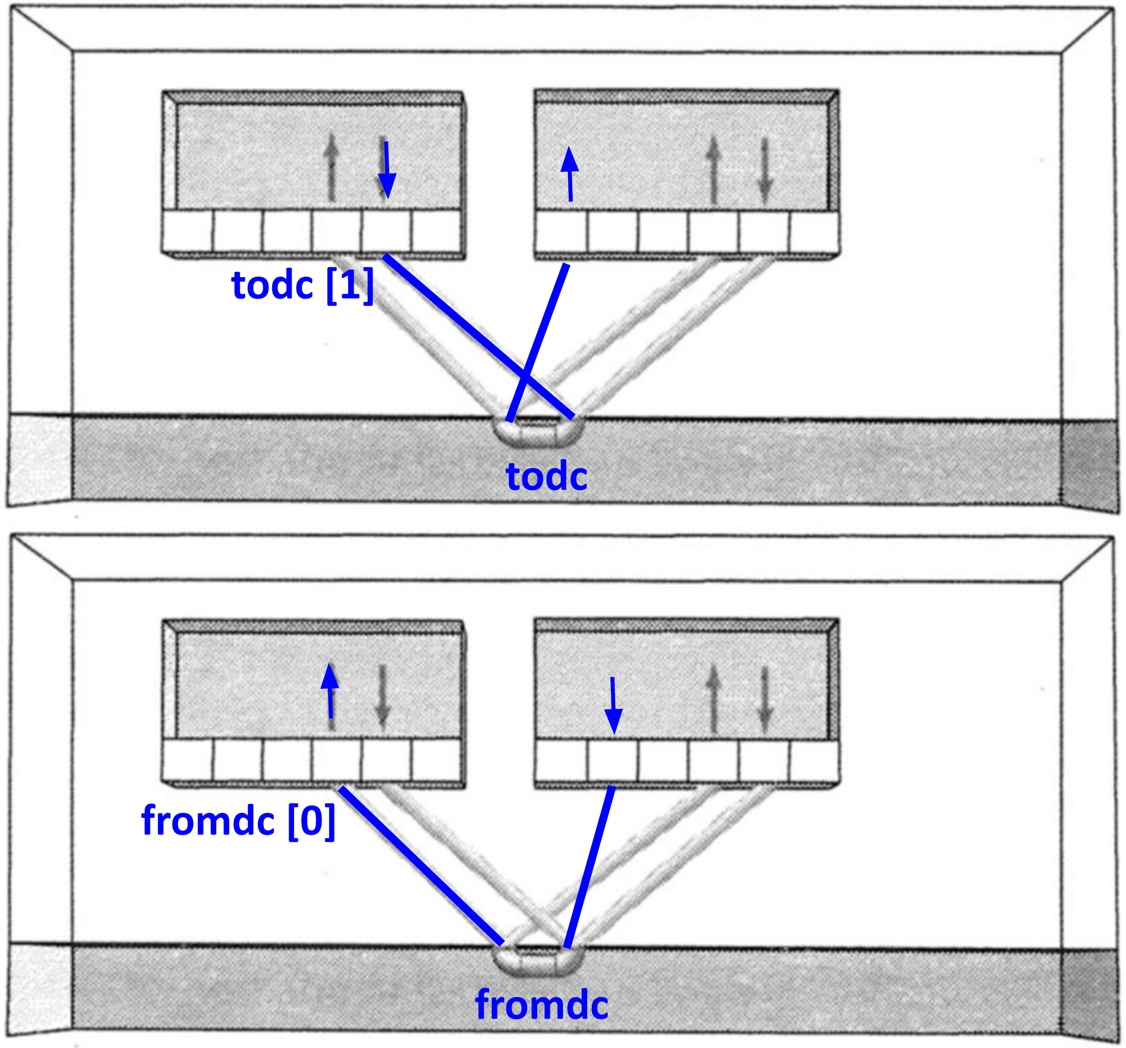

```
int todc[2], fromdc[2];
pipe(todc); pipe(fromdc);
fork()

close(todc[0])
close(fromdc[1])
readline from user
convert
write(todc[1],cmd,len)
read(fromdc[0],reply)
printf to user
```





Writing tiny `bc` (cont.)

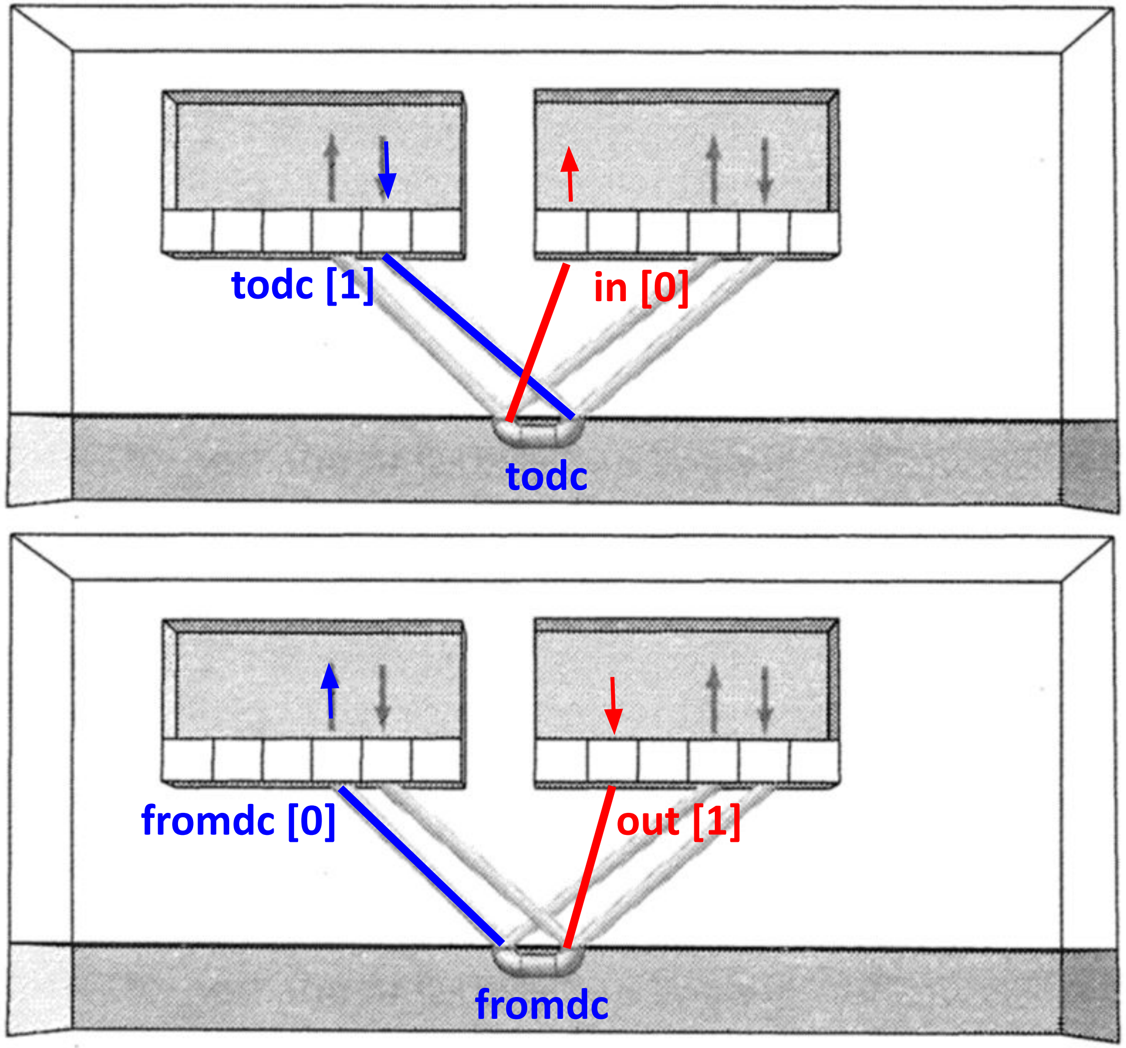
- `tinybc.c - be_bc`




```c
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 */

        fpout = fdopen( todc[1],    "w" );      /* convert file desc-  */
        fpin  = fdopen( fromdc[0],  "r" );      /* riptors to streams  */
        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);
        }
        fclose(fpout);          /* close pipe           */
        fclose(fpin);           /* dc will see EOF      */
}
```

Writing tiny `bc` (cont.)

- `tinybc.c - be_dc`




```c
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);
}
```

Writing tiny `bc` (cont.)

- Execution

```
dynam@DESKTOP-Q4IJBP7:~/lab12$ ./tinybc
tinybc: 2+2
2 + 2 = 4
tinybc: 2^10
2 ^ 10 = 1024
tinybc: 123*456
123 * 456 = 56088
tinybc: 123/123
123 / 123 = 1
tinybc: dynam@DESKTOP-Q4IJBP7:~/lab12$
```

`fdopen`: Making File Descriptors Look Like Files

- `fdopen`: a library function
  - Works like `fopen`, returning a `FILE*`
  - Takes a *file descriptor* not a filename as argument
  - Used when
    - You have a file descriptor but no filename
      - c.f., `fopen` if you know a filename
    - You want to convert the pipe connection into a `FILE*`
      - So you can use standard, buffered I/O operations
      - Notice how the `tinybc.c` code uses
        - `fprintf` and `fgets` to send data through the pipes to dc
  - Makes a remote access feel even more like a file

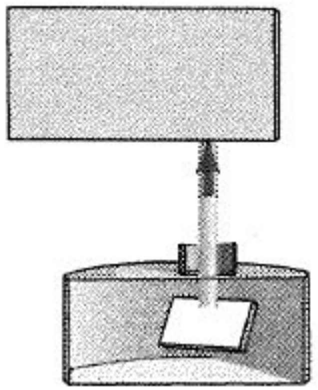
Lessons from `bc/dc`

- Client/Server Model
  - `bc/dc`: an example of the client/server model of a program design
- Bi-directional Communication
  - Requires one process to communicate with both `stdin` and `stdout` of another process
    - Traditionally, pipes can carry data in a unidirectional way
- Persistent service
  - Recall that on a shell each command creates a new process
  - `bc` keeps a single `dc` process running
  - `bc` uses that same instance of `dc` over and over again:
    - Sends `dc` commands in response to each line of user input
  - The `bc/dc` pair is treated as ***coroutines***: c.f., subroutines applied to function calls
    - Both "continue to run," but control passes from one to another
    - e.g., parsing and printing => `bc`, computing => `dc`

popen: Making Processes Look Like Files

- fopen: a library function
    - Opens a buffered connection to a *file*

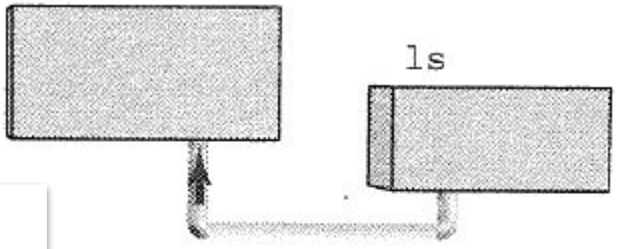
```
FILE *fp;                         /* a pointer to a struct */
fp = fopen( "file1", "r" );       /* args are filename, connection type */
c = getc(fp);                     /* read char by char */
fgets(buf, len, fp);              /* line by line      */
fscanf(fp,"%d%d%s",&x,&y,x);      /* token by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



fopen("file", "r")

- popen: a library function
    - 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 */
```



popen("ls", "r")

What `popen` Does: Use Case

- Using `popen` to obtain a sorted list of current users
  - By the command of "`who | sort`"

```
/* 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;
}
```

What `popen` Does: Use Case (cont.)

- `pclose` is required when `popen` gets invoked
  - NOT `fclose`
  - A callee process needs to be `wait`ed for
  - The callee process becomes a zombie process unless being awaited…
    - So its parent needs to retrieve its exit value
  - `pclose` calls `wait`

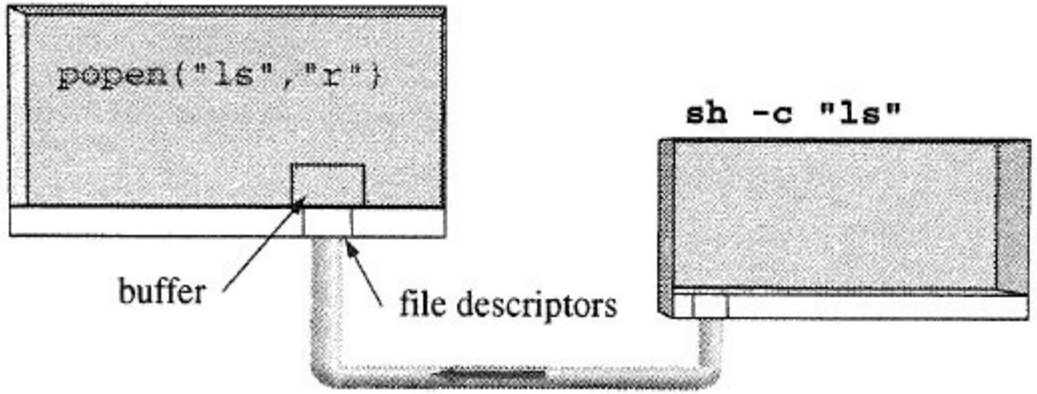
How Does `popen` Work? How to Write It?

- `popen`
  - Runs a program in a new process: use `fork`
  - Returns a connection to `stdin` or `stdout` of that program
    - Use `pipe`: for the connection
    - Use `dup2()`: for the redirection
    - Use `fdopen`: to make a file descriptor (fd) into a buffered stream
    - Use `exec`: to run any shell command in that process
      - `/bin/sh`: can execute any program on the shell
      - The “`-c`” option: tells the shell to run a command and then exit.
      - e.g., “`sh -c “who | sort”`”: what does this do?

How Does popen Work? How to Write It? (cont.)

- popen (Cont'd)
  - Flow chart and illustration for writing popen

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



< Reading from a shell command >

How Does `popen` Work? How to Write It? (cont.)

- `popen.c`

```
/*  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        <stdlib.h>
#include        <unistd.h>
#include        <signal.h>

#define READ    0
#define WRITE   1

int main()
{
        FILE    *fp;
        char    buf[BUFSIZ];

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

        return 0;
}
```

How Does popen Work? How to Write It? (cont.)

- popen.c (Cont'd)

```c
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;
        }
```

```c
        /* --------------- 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);
}
```

Access to Data: Files, APIs, and Servers

- Method 1: Getting data (directly) from *files*
  - By reading from a file: `who` for the `utmp` file
  - Not a perfect solution, as a client needs to know a file format and specific names in structures

Access to Data: Files, APIs, and Servers (cont.)

- Method 2: Getting data from *functions*
  - A library function can hide all the details behind a standard function interface
  - Application programming interface (API)-based information services are not always a right solution
  - Two methods for using system library functions
    - *Static* library (or static linking) including actual function code
      - but potentially containing a bug or using out-of-date file formats
    - *Dynamic* library (or shared libraries)
      - not always installed on a system or version error

Access to Data: Files, APIs, and Servers (cont.)

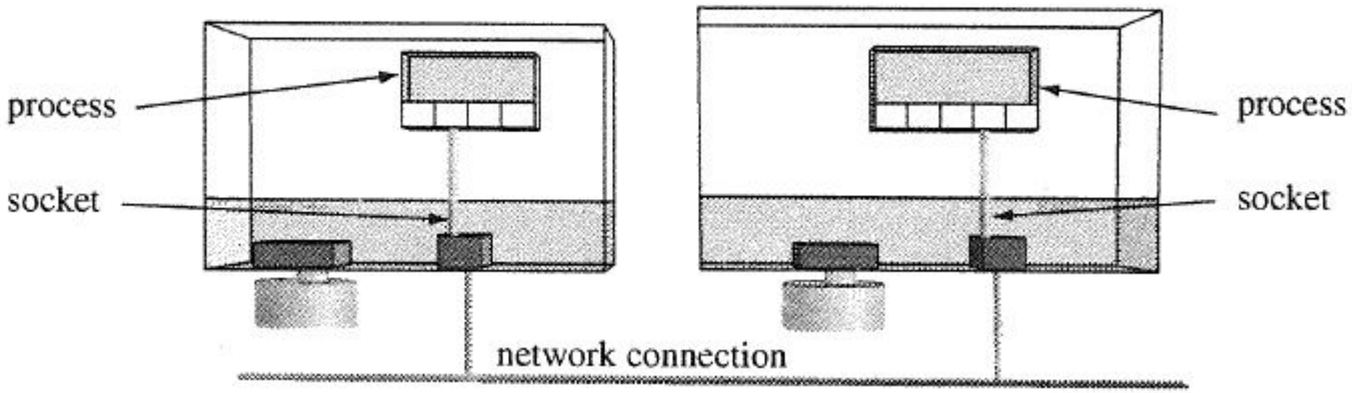
- Method 3: Getting data from processes; the `bc`/`dc` example
  - May require a network connection
  - Good for a client-server model
    - Server program can be written in any language: C, C++, Java, Perl, Python …
  - A server can be at a different machine from a client machine
    - How to connect to a process on a different machine?
    - Solution: IP address and port #
- What mechanism to allow us to connect to a process on a different computer?

Recall) What Pipes Can and Cannot Do

- Pros
  - Simple, easy, less complicated, no need of network
  - Allowing processes to send data to other processes as easily as they send data to files
- Cons
  - Created in one process and shared by calling `fork`
    - Can only connect related processes
  - Can only connect process on the SAME machine
    - What if you want to send your data to another remote host?
- Linux provides another method of IPC for remote connection: **Sockets!**

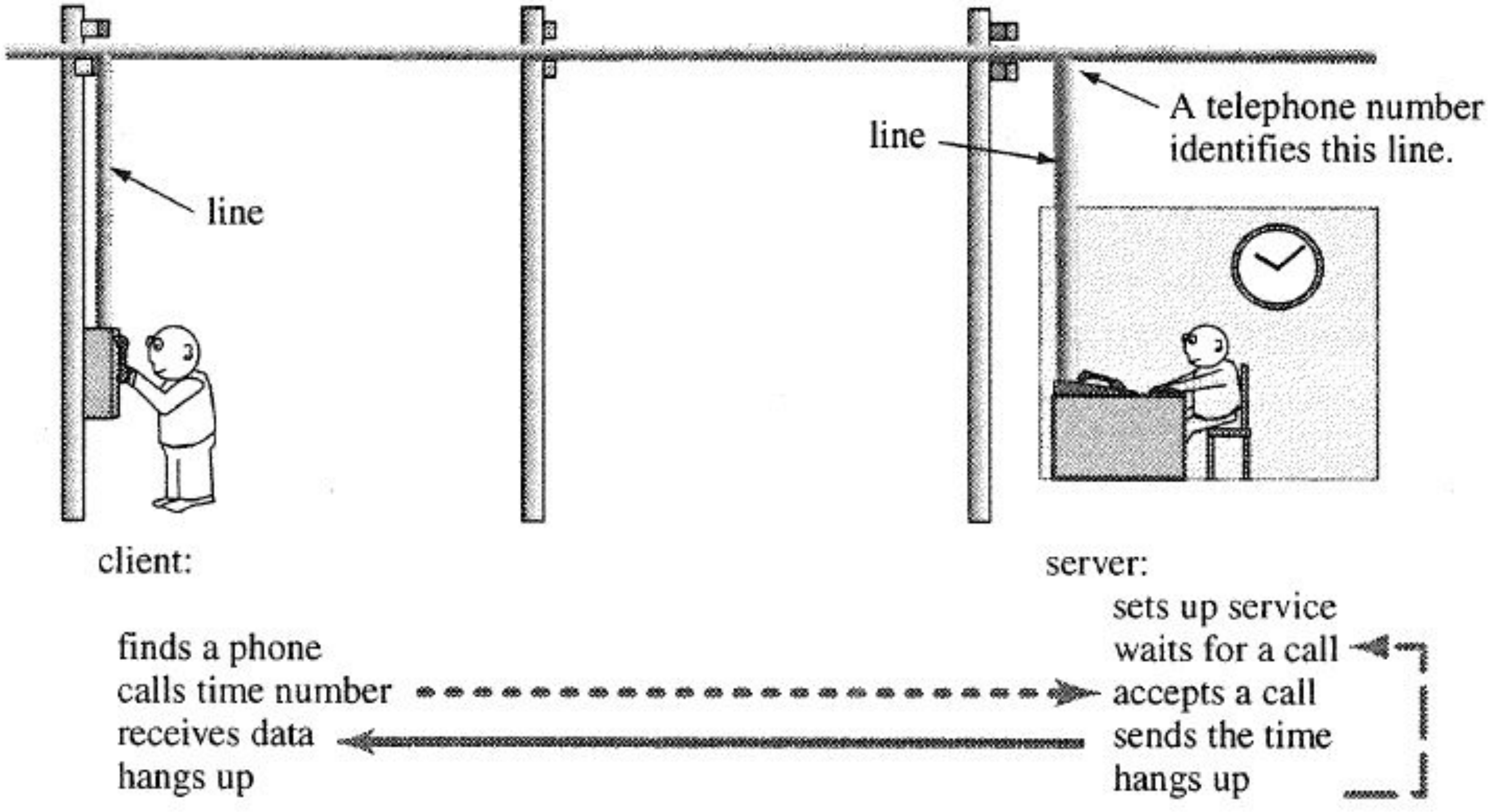
Sockets: Connecting to Remote Processes

- Allow processes to create pipelike connections to
  - Not only unrelated processes but also ones on other machines
- We'll study the basic ideas of sockets
- We'll see how to use sockets to connect clients and servers on different machines



< Connecting to a remote process >

An Analogy: “At the Tone, the Time Will Be…”

- 

<A time service >

Four Important Concepts

- 1. Client and Server
  - Server: a program (rather than a machine) that provides “services”
    - Its process waits for a request, processes that request, and then loops back to take the next request
  - Client: a program (rather than a machine) that requests “services”
    - Connects to and exchanges some data with the server, and then continues (its own task) and later terminates
      - Note that it does not loop
- 2. Protocol
  - The rules of interaction between the client and the server
  - In the time service, the protocol is simple:
    - If the client calls, the server answers, sends the time and then hangs up

Four Important Concepts (cont.)

- 3. Hostname and port
  - Host (identified by Internet Protocol (IP) address)
    - A server on the Internet
    - A running process on a machine, or host
    - Has its assigned name (hostname) and a port number
      - These two determines a server (address), or an end of communication.
      - e.g., `cse.knu.ac.kr`: `cse` as hostname, `80`: port number (hidden)
- 4. Address family
  - A group (or, set) of different addresses for indicating a service
    - Telephone + ext. number (for telephone): maybe denoted as `AF_PHONE`
    - Street address + zip code (postal code) (for mailing): maybe denoted as `AF_MAIL`
    - Longitude + latitude (for GPS): maybe denoted as `AF_GLOBAL`
  - IP address + port number (for network connection): `AF_INET`

Lists of Services: Well-Known Ports

- 119 for emergences, 112 for spy, 114 for phone, …
- How can we know what services available on my machine?

```
dynam@DESKTOP-Q4IJBP7:~/lab12$ more /etc/services
# Network services, Internet style
#
# Note that it is presently the policy of IANA to assign a single well-known
# port number for both TCP and UDP; hence, officially ports have two entries
# even if the protocol doesn't support UDP operations.
#
# Updated from https://www.iana.org/assignments/service-names-port-numbers/servic
e-names-port-numbers.xhtml .
#
# New ports will be added on request if they have been officially assigned
# by IANA and used in the real-world or are needed by a debian package.
# If you need a huge list of used numbers please install the nmap package.

tcpmux          1/tcp                           # TCP port service multiplexer
echo            7/tcp
echo            7/udp
discard         9/tcp           sink null
discard         9/udp           sink null
systat          11/tcp          users
daytime         13/tcp
daytime         13/udp
netstat         15/tcp
qotd            17/tcp          quote
chargen         19/tcp          ttytst source
chargen         19/udp          ttytst source
ftp-data        20/tcp
ftp             21/tcp
fsp             21/udp          fspd
ssh             22/tcp                          # SSH Remote Login Protocol
telnet          23/tcp
smtp            25/tcp          mail
time            37/tcp          timserver
time            37/udp          timserver
whois           43/tcp          nicname
tacacs          49/tcp                          # Login Host Protocol (TACACS)
tacacs          49/udp
domain          53/tcp                          # Domain Name Server
domain          53/udp
bootps          67/udp
bootpc          68/udp
tftp            69/udp
gopher          70/tcp                          # Internet Gopher
finger          79/tcp
http            80/tcp          www             # WorldWideWeb HTTP
kerberos        88/tcp          kerberos5 krb5 kerberos-sec     # Kerberos v5
```

How Do We Write Time Server and Time Client?

- Six steps for our telephone-based time server

Action	System call
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`

- Four steps for our telephone-based time client

Action	System call
1. Get a phone line	`socket`
2. Call the server	**`connect`**
3. Transfer data	`read/write`
4. Hang up	`close`

Working Principle of a Time Server

- Step 1: Ask kernel for a socket
  - A **socket**: a place from which calls can be made and a place to which calls can be directed
- Step 2: Bind address to a socket.
  - Address is *hostname* and *port*
- Step 3: Allow incoming calls with queue size=1 on socket
  - A server accepts incoming calls.
- Step 4: Wait for/Accept a Call.
  - Once the socket is created, assigned an address, and set up receive incoming calls, then the program is ready to go!
- Steps 5 and 6: Transfer Data and then Hang Up

Step 1: Ask kernel for a socket

- `socket` creates an endpoint (종단점) for communication and returns an identifier for that socket
    - Various sorts of communication systems, each called *domain* (e.g., Internet)
    - The *type* of a socket specifies the type of data flow
        - `SOCK_STREAM`: a bidirectional type (like TCP)
        - `SOCK_DGRAM`: connectionless (like UDP)
    - *Protocol* used within the network code in the kernel
        - c.f., `/etc/protocols`

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

Step 2: Bind address to a socket

- `bind` assigns a network address to a socket
    - The Internet address family (`AF_INET`) uses host and port
        - Port 13000 will be used; port 13 reserved for the *real* time server

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

Step 3: Allow Incoming calls with Queue size=1 on Socket

- `listen` asks the kernel to allow the specified socket to receive incoming calls.
  - Applied to `SOCK_STREAM` (not to `SOCK_DGRAM`)
  - Queue for incoming calls
  - Queue size=1 means a queue of one call
    - Maximum queue size depends on the socket implementation

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

Step 4: Wait for / Accept a Call

- `accept` suspends the current process until an incoming connection on the specified socket is established
  - The socket has an address, consisting of a hostname and port number
  - Returns a file descriptor (*fd*) opened for reading and writing
    - *fd*: a connection to a file descriptor in the calling process

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

The Remaining Steps

- Step 5: Transfer Data
  - The fd returned by `accept` is a regular file descriptor.
  - Use `fdopen` to make the *fd* into a buffered data stream for `fprintf`
- Step 6: Close Connection
  - The *fd* returned by accept should be closed with `close`
  - When one process closes one end,
    - The other end will see EOF for a data read (as seen in pipes)

A Time Server: timeserv.c

•

Action	System call
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

```
/* timeserv.c - a socket-based time of day server
 */

#include  <stdio.h>
#include  <stdlib.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     */
```

A Time Server: `timeserv.c` (cont.)

-

Action	System call
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`

```
/*
 * 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" );
```

A Time Server: `timeserv.c` (cont.)

Action	System call
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`

```
      /*
       * 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   */
       }
}
```

Working Principle of a Time Client

- Step 1: Ask Kernel for a Socket
  - Needs a socket to connect to the network
  - Like a client needing a phone line in the phone network
- Step 2: Connect to Server
  - Uses the `connect` system call
- Step 3: Transfer Data
  - Reads one line from the server through the connected socket
- Step 4: Hang Up
  - `closes` the file descriptor for the connected socket and exits

More Details about Step 2

- connect attempts to connect the socket specified by `sockid` to the socket address pointed by `serv_addrp`
  - If the attempt succeeds, `result` will get 0.
    - `sockid` now then becomes a fd open for reading and writing
    - Data written into are sent to and read from the socket's ends

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	socket to use for connection
	serv_addrp	pointer to struct containing server address
	addrlen	length of that struct
RETURNS	-1	if error
	0	if success

A Time Client: `timeclnt.c`

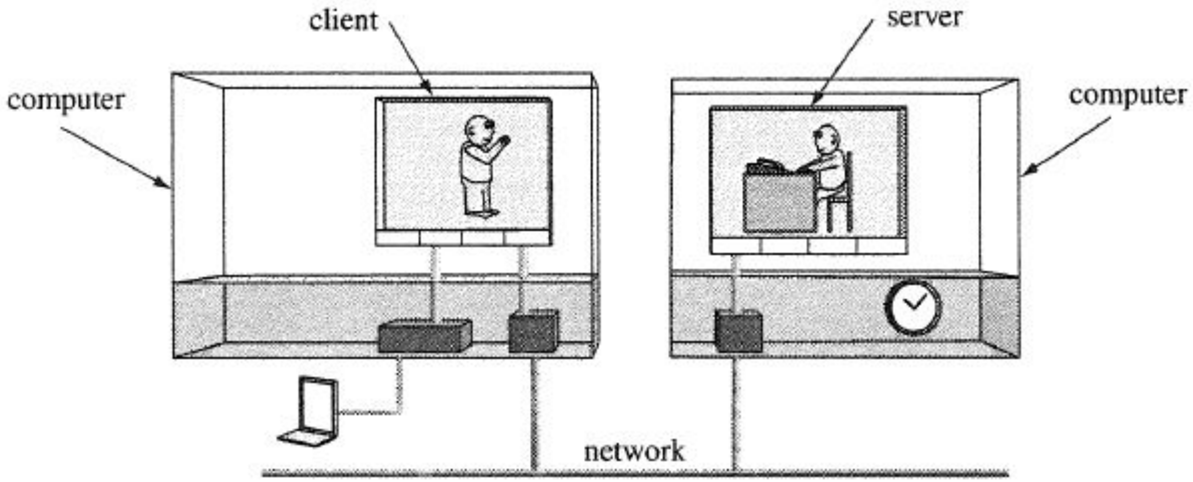
Action	System call
1. Get a phone line	`socket`
2. Call the server	**`connect`**
3. Transfer data	`read/write`
4. Hang up	`close`

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

#define         oops(msg)       { perror(msg); exit(1); }

int 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 */
```

```c
  /*
   * 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 );

        return 0;
}
```

Testing `timeserv` and `timeclnt`

- The server process runs on one machine
- A client process on another machine connects to the server over the network
- Then the server sends data to client by write
- The client receives that message by read



< Processes on different computers >

Testing `timeserv` and `timeclnt` (cont.)

- Execution
  - Server
    - $ `./timeserv &`

```
dynam@DESKTOP-Q4IJBP7:~/lab12$ ./timeserv &
[1] 53
dynam@DESKTOP-Q4IJBP7:~/lab12$ Wow! got a call!
```

  - Client
    - $ `./timeclnt localhost 13000`

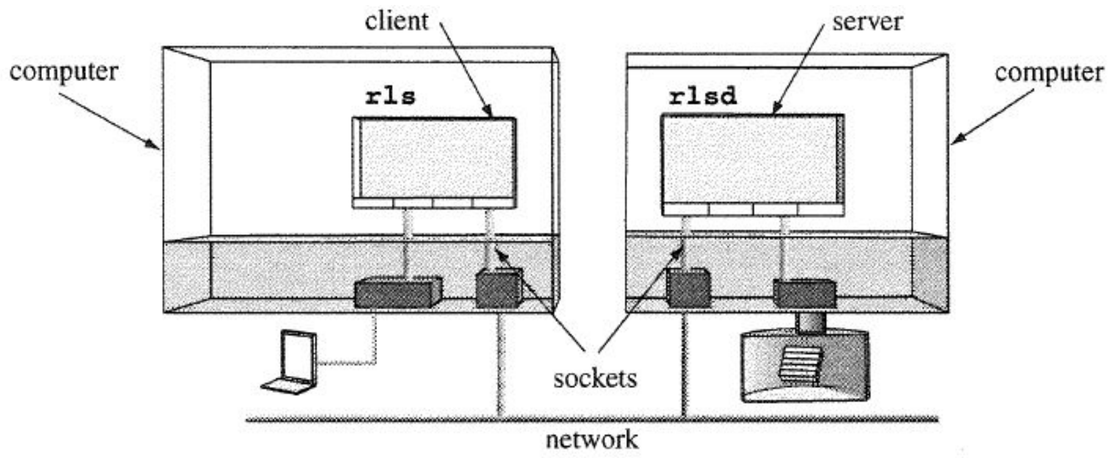
```
dynam@DESKTOP-Q4IJBP7:~/lab12$ ./timeclnt DESKTOP-Q4IJBP7.localdomain 13000
The time here is ..Thu Nov 24 02:32:57 2022
dynam@DESKTOP-Q4IJBP7:~/lab12$
```

Example of Another Server and Client: Remote `ls`

- Listing files on a remote computer
    - Could log in to that machine and run `ls`
        - e.g., `$./rls 155.230.100.100 /home/`*`username`*
    - `rls` needs a server process running on the other machine
        - To receive the request, do the work, and return the result
    - The server runs on one computer
    - A client on another computer connects to the server
        - Sends the name of a requesting directory: e.g., '`/home/`*`username`*'
    - The server sends back to the client a list of the files in that requested directory
    - The client displays the list by writing to `stdout`
- This two-process system really provides access to directories on a different machine!

Remote `ls`

- Three things to implement the remote `ls` system
  - 1) A protocol
    - Consists a request and a reply between a client and a server program
  - 2) A client program
    - Sends a single-line containing the name of a requested directory
    - Reads the list of files line by line until the server closes the connection
  - 3) A server program
    - The server opens and reads that directory and sends back to the client the list of files
    - When closing the connection, it generates an EOF condition



< A `remote ls` system >

Remote `ls` (cont.)

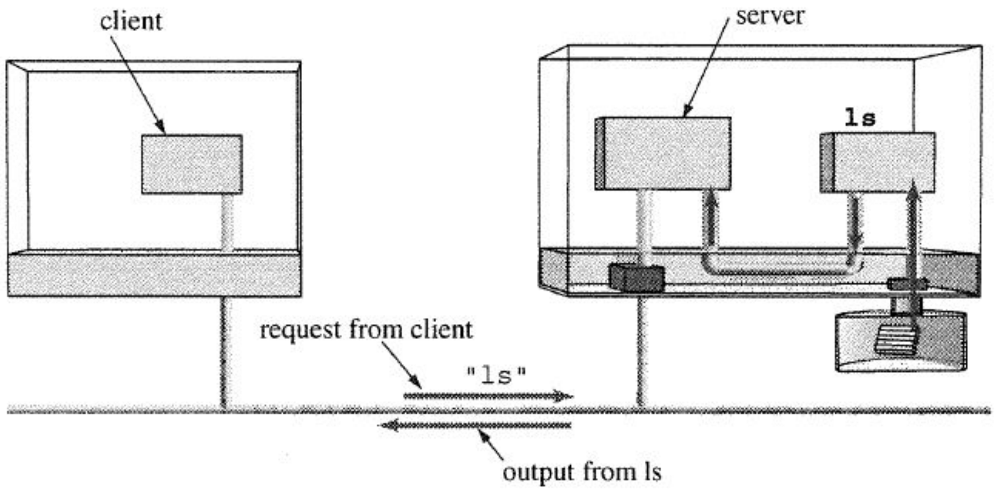
- The client: `rls`
  - Differences of this client from the time client
    - 1. Writing the directory name into the socket
    - 2. Entering a loop, copying data from the socket to `stdout` until EOF
      - The loop uses a standard buffer size for efficiency
    - 3. Using `write` and `read` for data transfer (with the server)

Remote `ls` (cont.)

- The server: `rlsd`
  - Has to get a socket
    - Bind and listen, and then accept a call.
  - Read then the name of a requested directory from the socket
  - Lists the contents of that directory
    - Use `popen` to read the output from the regular version of `ls`
- Notes
  - The server uses standard buffered streams for reading and writing
    - Use `fgets` to read the directory name from the client
  - After `popen`, it transfers data using `getc`/`putc` for a file copy
    - Actually copying data from one process to a process on another machine

Remote `ls` (cont.)

- Additional notes
  - The string the program receives
    - Does not overflow the input buffer
    - Does not overflow the buffer for the command
    - Doesn't allow special characters in the directory name
  - `popen`: indeed too risky for network services
    - Because it passes a string to a shell
    - It's a *poor* idea to write any server-passing strings to a shell!
    - Two reasons of why to use this example
      - For showing another use of `popen`
      - For alerting you guys to this danger



< Using `popen("ls")` to list remote directories >

Writing remote `ls`: `rlsd.c`

```c
/* rlsd.c - a remote ls server
 */

#include  <stdio.h>
#include  <stdlib.h>
#include  <unistd.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>
#include  <netdb.h>
#include  <time.h>
#include  <ctype.h>
#include  <strings.h>

#define   PORTNUM  15000   /* our remote ls server port */
#define   HOSTLEN  256
#define   oops(msg)        { perror(msg) ; exit(1) ; }

void sanitize(char * );

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_fpi, *sock_fpo;    /* streams for in and out */
        FILE     *pipe_fp;                /* use popen to run ls    */
        char     dirname[BUFSIZ];         /* from client            */
        char     command[BUFSIZ];         /* for popen()            */
        int      dirlen, c;
```

```c
    /** 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   */
      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" );
```

Writing remote `ls`: `rlsd.c` (cont.)

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

     while ( 1 ){
             sock_fd = accept(sock_id, NULL, NULL); /* wait for call */
             if ( sock_fd == -1 )
                     oops("accept");

             /* open reading direction as buffered stream */
             if( (sock_fpi = fdopen(sock_fd,"r")) == NULL )
                     oops("fdopen reading");

             if ( fgets(dirname, BUFSIZ-5, sock_fpi) == NULL )
                     oops("reading dirname");
             sanitize(dirname);

             /* open writing direction as buffered stream */
             if ( (sock_fpo = fdopen(sock_fd,"w")) == NULL )
                     oops("fdopen writing");

             sprintf(command,"ls %s", dirname);
             if ( (pipe_fp = popen(command, "r")) == NULL )
                     oops("popen");

             /* transfer data from ls to socket */
             while( (c = getc(pipe_fp)) != EOF )
                     putc(c, sock_fpo);
             pclose(pipe_fp);
             fclose(sock_fpo);
             fclose(sock_fpi);
    }

     return 0;
}
```

```c
void sanitize(char *str)
/*
 * it would be very bad if someone passed us an dirname like
 * "; rm *"  and we naively created a command  "ls ; rm *"
 *
 * so..we remove everything but slashes and alphanumerics
 * There are nicer solutions, see exercises
 */
{
        char *src, *dest;

        for ( src = dest = str ; *src ; src++ )
                if ( *src == '/' || isalnum(*src) )
                        *dest++ = *src;
        *dest = '\0';
}
```

Writing remote `ls`: `rls.c`

```c
/* rls.c - a client for a remote directory listing service
 *         usage: rls hostname directory
 */
#include        <stdio.h>
#include        <stdlib.h>
#include        <unistd.h>
#include        <sys/types.h>
#include        <sys/socket.h>
#include        <netinet/in.h>
#include        <netdb.h>
#include        <string.h>

#define         oops(msg)       { perror(msg); exit(1); }
#define          PORTNUM         15000

int 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    buffer[BUFSIZ];             /* to receive message */
        int     n_read;                     /* for message length */

        if ( ac != 3 ) exit(1);

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

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

```c
     /** Step 2: connect to server **/

        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(PORTNUM);      /* fill in port number  */
        servadd.sin_family = AF_INET ;          /* fill in socket type  */

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

     /** Step 3: send directory name, then read back results **/


        if ( write(sock_id, av[2], strlen(av[2])) == -1)
                oops("write");
        if ( write(sock_id, "\n", 1) == -1 )
                oops("write");

        while( (n_read = read(sock_id, buffer, BUFSIZ)) > 0 )
                if ( write(1, buffer, n_read) == -1 )
                        oops("write");
        close( sock_id );

        return 0;
}
```

Execution of Remote `ls`

```
dynam@DESKTOP-Q4IJBP7:~/lab12$ ./rlsd &
[1] 193
dynam@DESKTOP-Q4IJBP7:~/lab12$ ./rls DESKTOP-Q4IJBP7.localdomain /home/dynam
GoogleDrive
a.out
demodir
elec462
```

Software Daemons

- Many Linux server programs ending in the letter 'd'
  - e.g., `httpd`, `inetd`, `syslogd`, `atd`, `ntpd`, `sshd`, …
  - '`d`' stands for daemon
    - Daemon: a supernatural helper floating around waiting you to help out
      - Provides a variety of services and performs system maintenance
        - Alerting, flushing, logging, printing, accepting network connections, …
  - Most daemon processes get started at the boot-up time
    - `/etc/rcX.d`, where *X* depends on system
      - Starts these servers in the background for providing services, with being detached from any terminals

Summary

- Some programs are written as separate processes for data transfer
    - A server process responsible for processing or data delivery in the CS model
- A CS system consists of a communication system and a protocol
        - Protocol: a set of rules for the structure of a conversation
    - Clients/servers can communicate through pipes or sockets
- `popen` can make any shell command into a server program.
    - Makes access to the server look like having access to buffered files

Summary (cont.)

- A pipe: a connected pair of fds
  - Socket: an unconnected communication endpoint (potential fd)
    - A client creates a comm. link by connecting its socket to a server socket
- Connections between sockets from one machine to another.
  - Each socket is identified by an *IP address* and a *port number*
- Connections to pipes and sockets use fds
  - Fds provide programs with a single interface for communication with different objects: files, devices, and other processes