I/O Connecting to Processes Near and Far

류은경
ekryu@knu.ac.kr

Objectives

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`

Connecting to Processes Near and Far

11.2 Introductory Metaphor : A Beverage Interface

11.3 bc: A Unix Calculator

11.4 popen: Making Process Look like Files

- **Imagine a vending machine** :

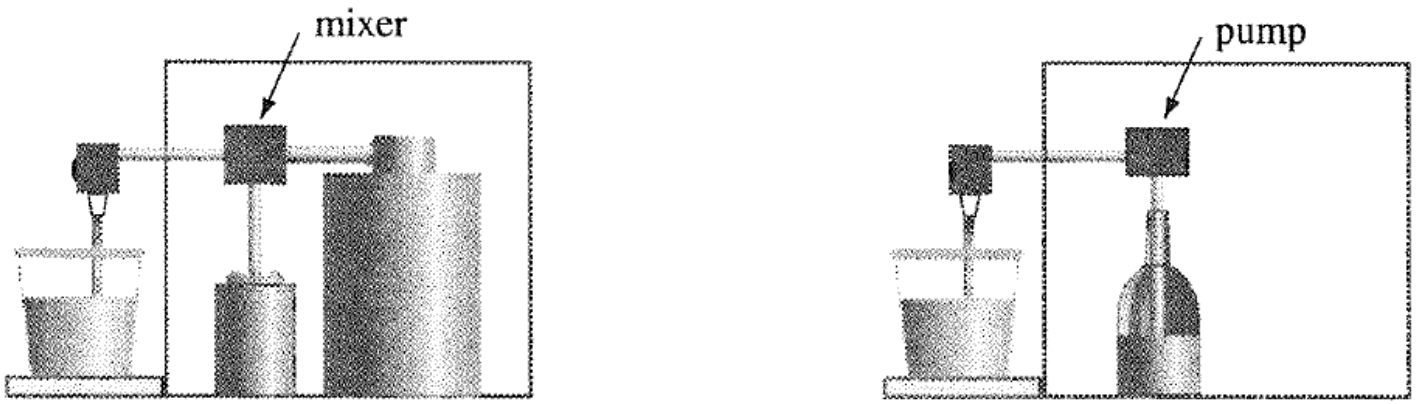
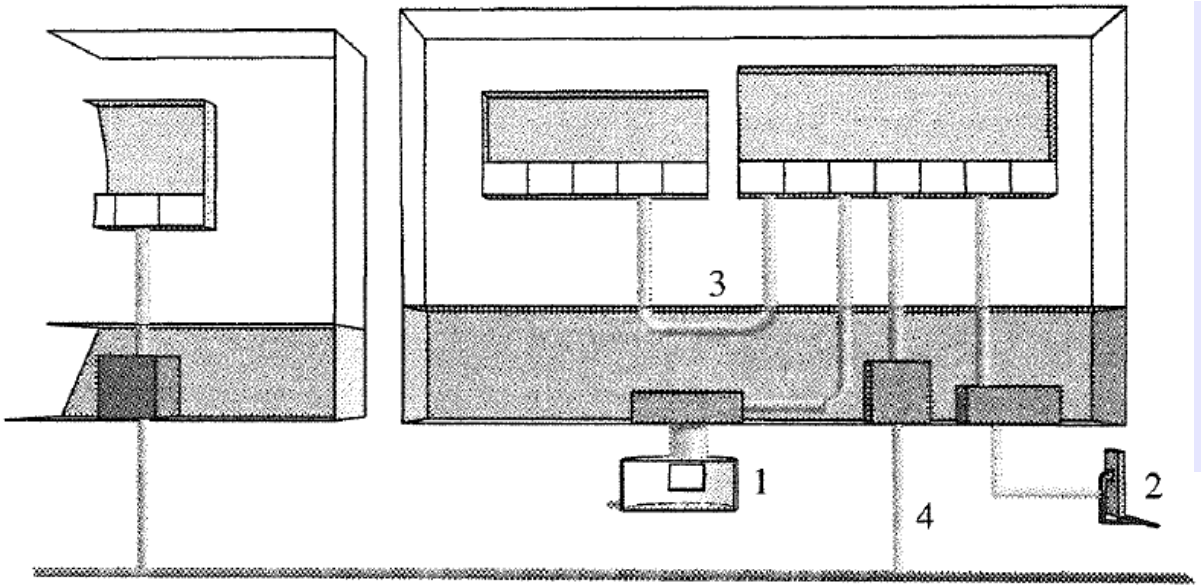


FIGURE 11.1

Dynamically generated or static beverage?

- **Unix presents one I/O interface even though data come from different types of sources : …**



Four types of data sources

1. Disk files
2. Devices
3. Pipes
4. Sockets

One I/O interface

FIGURE 11.2

One interface, different sources.

Connecting to Processes Near and Far

11.2 Introductory Metaphor : A Beverage Interface

11.3 bc: A Unix Calculator

11.4 popen: Making Process Look like Files

bc: A Unix Calculator

- **bc is** also **a programming language with c-like syntax**

```
root@DESKTOP-K4MA2V5: ~
root@DESKTOP-K4MA2V5:~# bc
bc 1.06.95
Copyright 1991-1994, 1997,
This is free software with
For details type `warranty'
x=3
if(x==3)
{
y=x*3;
}
y
9
root@DESKTOP-K4MA2V5:~#
```

- **bc handles very long numbers:**

```
$ bc
17^123
221420246301202073593205737642369575233456032169873317322404970169 47\
292822996637496750906355872025391170927994632063938187990037220685 58\
0536286573569713
```

- **But `bc` Is Not a Calculator**

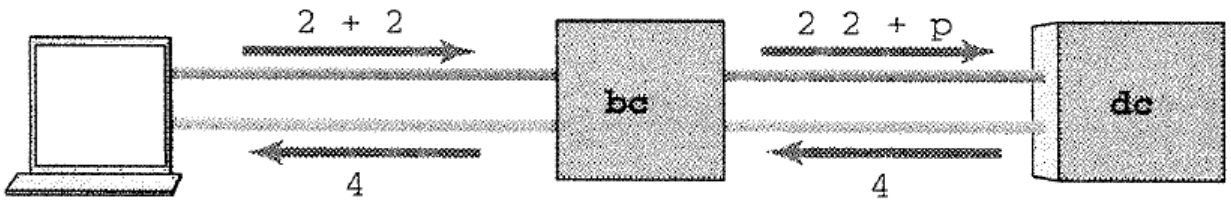


FIGURE 11.3

`bc` and `dc` as coroutines.

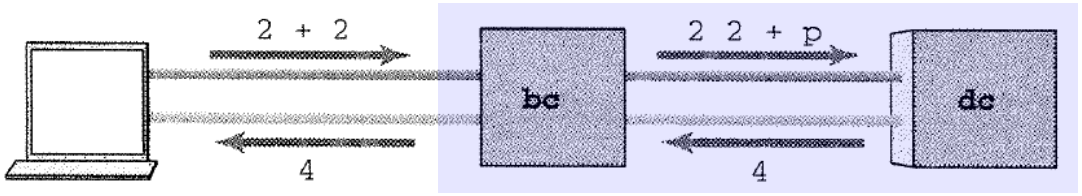


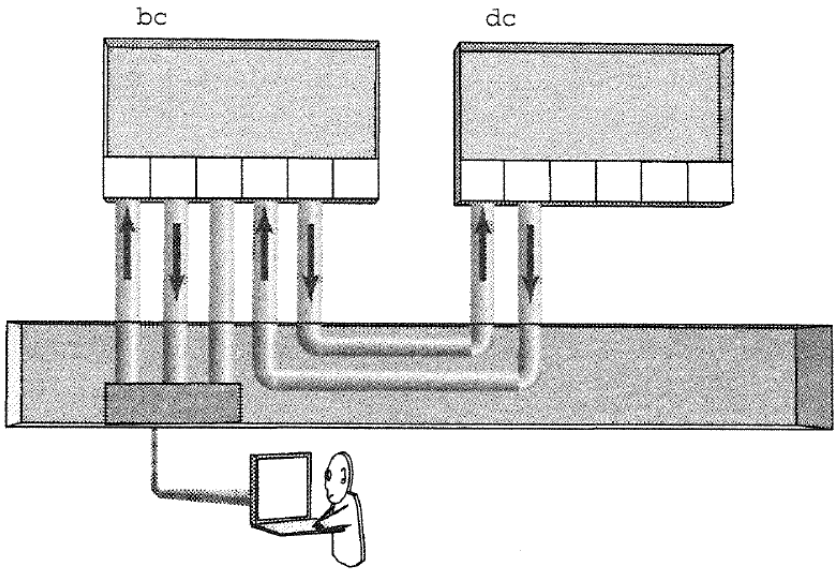
FIGURE 11.3

bc and dc as coroutines.

- **Ideas from `bc`**

1. **Client/Server Model**
   - `dc` provides a **service (server)**
   - `bc` provides a user interface AND uses the service **(client)**
2. **Bidirectional Communication**
   - Using pipes, you need two pipes
3. **Persistent Service**
   - `bc` program keeps a single `dc` process running.

11.3.1 Coding bc: pipe, fork, dup, exec



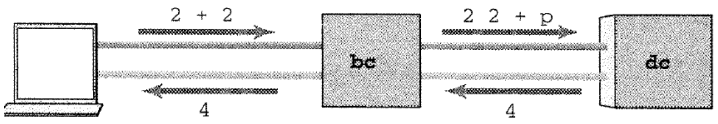


FIGURE 11.4

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.

```
$ cc tinybc.c -o tinybc ; ./tinybc
tinybc: 2+2
2 + 2 = 4
tinybc: 55^5
55 ^ 5 = 503284375
tinybc:
```

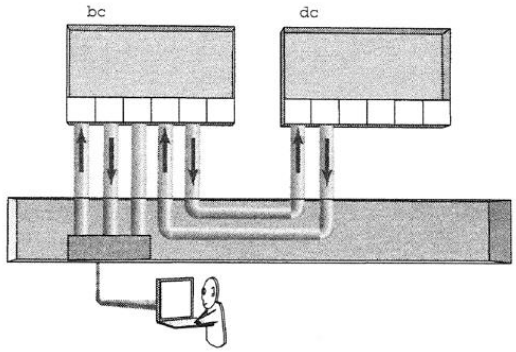
no spaces

…

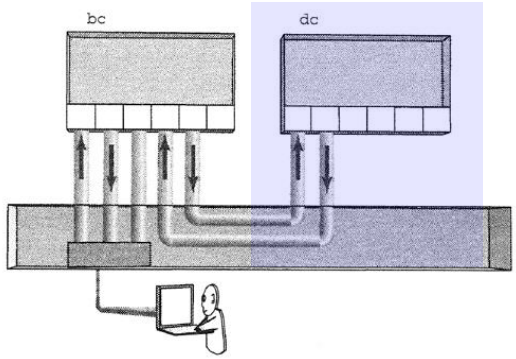
tynibc: ctrl+D

$

```
/**      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
 **/
```



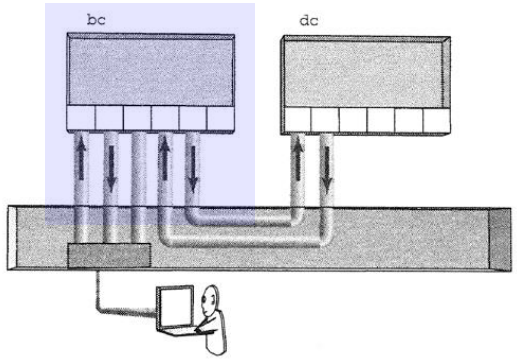

```
#include        <stdio.h>
#include        <unistd.h>
#include        <stdlib.h>
#define         oops(m, x){perror(m); exit(x);}

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 )                             ②
                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 */
        }
}
```




```
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);
}
```

③

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");
```

https://www.gnu.org/software/libc/manual/html_node/Streams-and-File-Descriptors.html

Streams and File Descriptors

```
/* 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     */
}
```

Input ex.
2+2

⑥

2
2
+
p

⑦

// stdout

⑧

```
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:

…
tynibc: ctrl+D
$
```

11.3.3 `fdopen`: Making File Descriptors Look like Files

- **`fopen`: file name → FILE ***

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

Connecting to Processes Near and Far

11.2 Introductory Metaphor : A Beverage Interface

11.3 bc: A Unix Calculator

11.4 popen: Making Process Look like Files

11.4.1 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 */
```

- **Similarities between `popen` and `fopen`.**

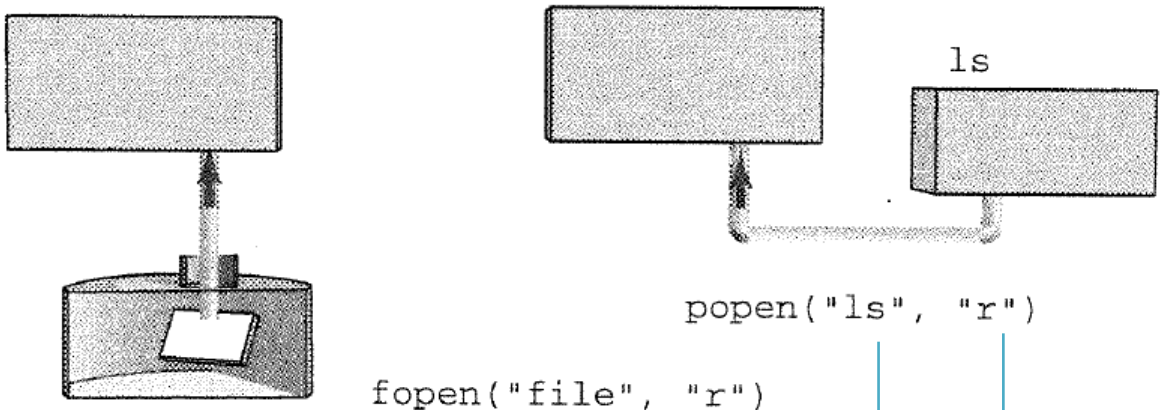


FIGURE 11.5

fopen and popen.

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

It can be any shell command.

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

11.4.2 Writing popen

- **How does popen work? …**

```
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;
```

- **How** do we **write popen?**

- **Writing `popen` :**

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;
```

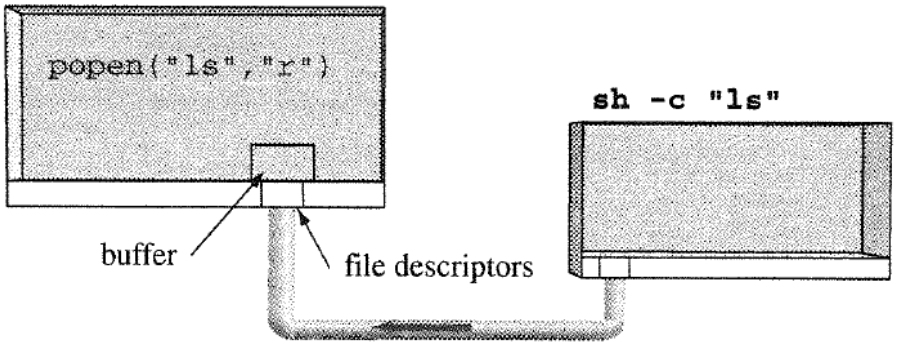
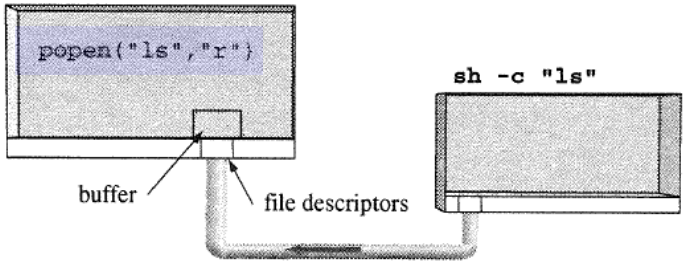


FIGURE 11.6

Reading from a shell command.

```
root@DESKTOP-K4MA2V5:~# ./popen
DESKTOP-K4MA2V5
a
a.out
a.txt
data
demodir
fido.7
file1.txt
file2.txt
file3
listing
lsout
outputfile
pipe
pipe.c
popen
popen.c
popen_ex3
popen_ex3.c
popendemo.c
rls
rls.c
rls2.c
rlsd
rlsd.c
sort_test
timeclnt
timeclnt.c
timeserv
timeserv.c
unlist
us
w
x
root@DESKTOP-K4MA2V5:~#
```

```
/*  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
```




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

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

```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;
        }
```

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

Connecting to Processes Near and Far

11.2 Introductory Metaphor : A Beverage Interface

11.3 **bc**: A Unix Calculator

11.4 **popen**: Making Process Look like Files