Ch8. I/O Redirection and Pipes

Prof. Seokin Hong

Kyungpook National University

Fall 2018

THE SHELL

- A shell is a program that manages processes and runs programs

- Thee main functions of shells
  (a) Shells run programs
  (b) Shells manage input and output
  (c) Shells can be programmed

THE SHELL

- How do the following commands work?

```
$ ls > my.files
$ who | sort > userlist
```

- We focus on a particular form of interprocess communication: I/O redirection and pipes

Objectives

- Ideas and Skills
  - I/O Redirection : What and why?
  - Definitions of standard input, output, and error
  - Redirecting standard I/O to files
  - Using fork to redirect I/O for other programs
  - **Pipes**
  - Using fork with pipes
- System Calls and Functions
  - dup, dup2
  - pipe

- Consider the following problem:
  You want a program that notifies you when people log in or log out of the system so you can watch for your pals.

  - You could write a C program that uses the utmp file and interval timers.
  - A simpler solution is to write a shell script : who

A SHELL APPLICATION : WATCH FOR USERS

Logic

Get list of users (call it prev)
While true
    sleep
    get list of users (call it curr)
    compare lists
        in prev, not in curr -> logout
        in curr, not in prev -> login
    make prev = curr
repeat

Shell code

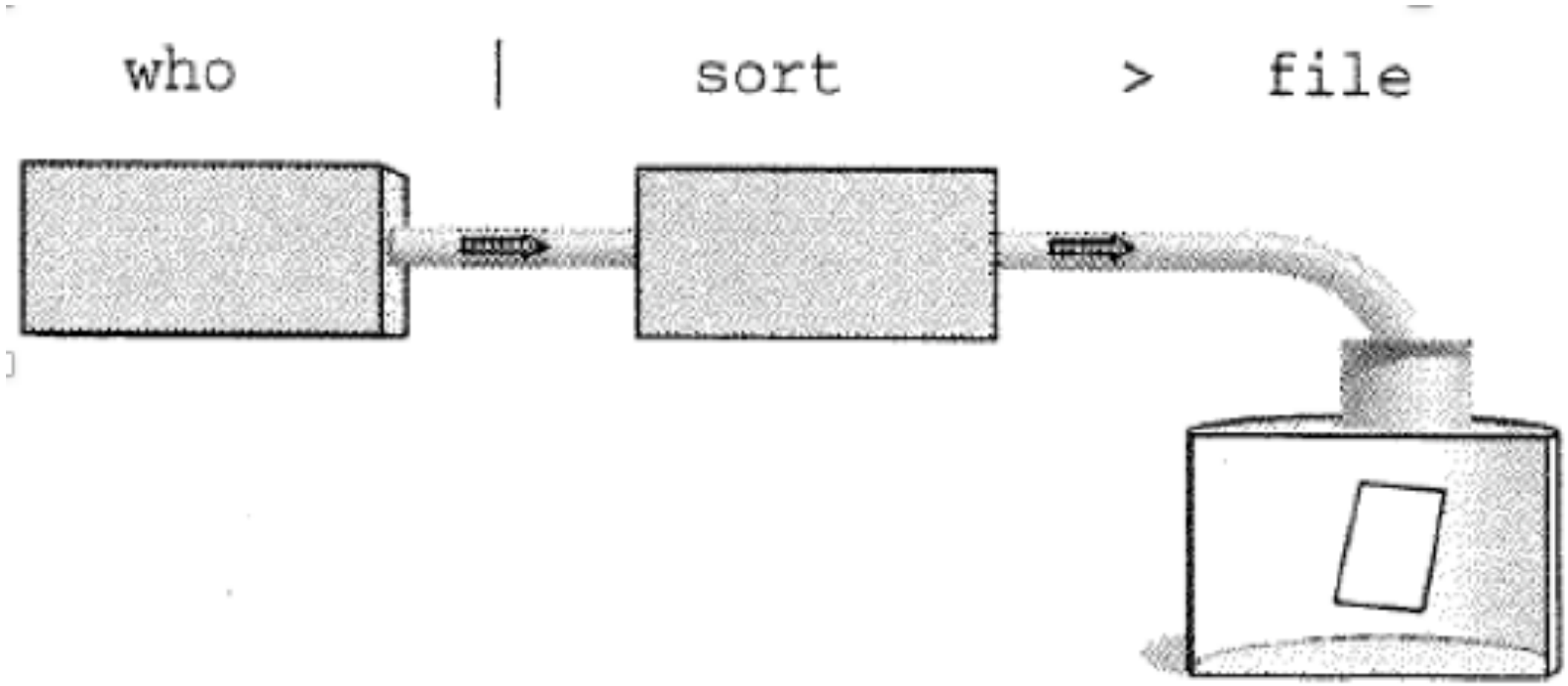
```
Who | sort > prev
While true ; do
        sleep 60
        who | sort > curr
        echo “logged out:”
        comm -23 prev curr
        echo “logged in:”
        comm -13 prev curr
        mv curr prev
done
```

A SHELL APPLICATION : WATCH FOR USERS

- who | sort > prev

shell code

```
who | sort > prev
while true ; do
   sleep 60
   who | sort > curr
   echo "logged out:"
   comm -23 prev curr
   echo "logged in:"
   comm -13 prev curr
   mv curr prev
done
```

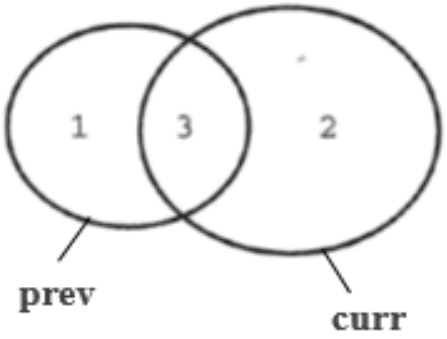


A SHELL APPLICATION : WATCH FOR USERS

- The comm command
  - compares two sorted lists
  - prints out three columns: 1, 2, and 3.

```
root@DESKTOP-K4MA2V5:~# cat file1.txt
a
b
c
d
root@DESKTOP-K4MA2V5:~# cat file2.txt
a
b
c
e
root@DESKTOP-K4MA2V5:~# comm file1.txt file2.txt
                        a
                        b
                        c
d
        e
root@DESKTOP-K4MA2V5:~#
```

```
 shell code
----------------------
who | sort > prev
while true ; do
   sleep 60
   who | sort > curr
   echo "logged out:"
   comm -23 prev curr
   echo "logged in:"
   comm -13 prev curr
   mv curr prev
done
```



A SHELL APPLICATION : WATCH FOR USERS

- comm -23 prev curr
  - drop columns 2 and 3 ➔ show lines only in prev

- comm -13 prev curr
  - drop columns 1 and 3 ➔ show lines only in curr

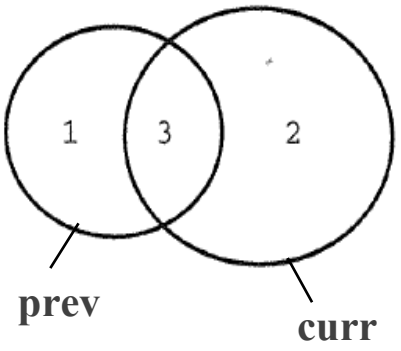


FIGURE 10.2

ccmm compares two lists and outputs three sets.

```
 shell code
----------------------
who | sort > prev
while true ; do
   sleep 60
   who | sort > curr
   echo "logged out:"
   comm -23 prev curr
   echo "logged in:"
   comm -13 prev curr
   mv curr prev
done
```

Contents

- 10.1 Shell
- 10.2 A Shell Application：Watch for Users
- 10.3 Facts about Standard I/O and Redirection
- 10.4 How to Attach stdin to a File
- 10.5 Redirecting I/O for Another Program: who>userlist
- 10.6 Programming Pipes

- Lessons:

  (a) Power of shell scripts—easier and quicker than C
  (b) Flexibility of software tools—each tool does one specific, general task
  (c) Use and value of I/O redirection and pipes

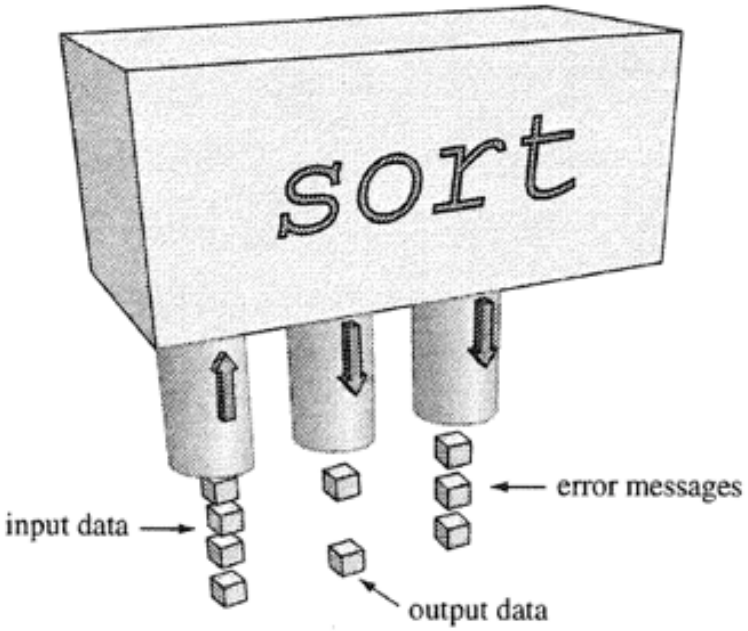
- The > operator can be used to treats *files* as *variables* of arbitrary size and structure.

```
x = func_a(func_b(y));    in C

prog_b | prog_a > x       in sh
```

- Question: How?

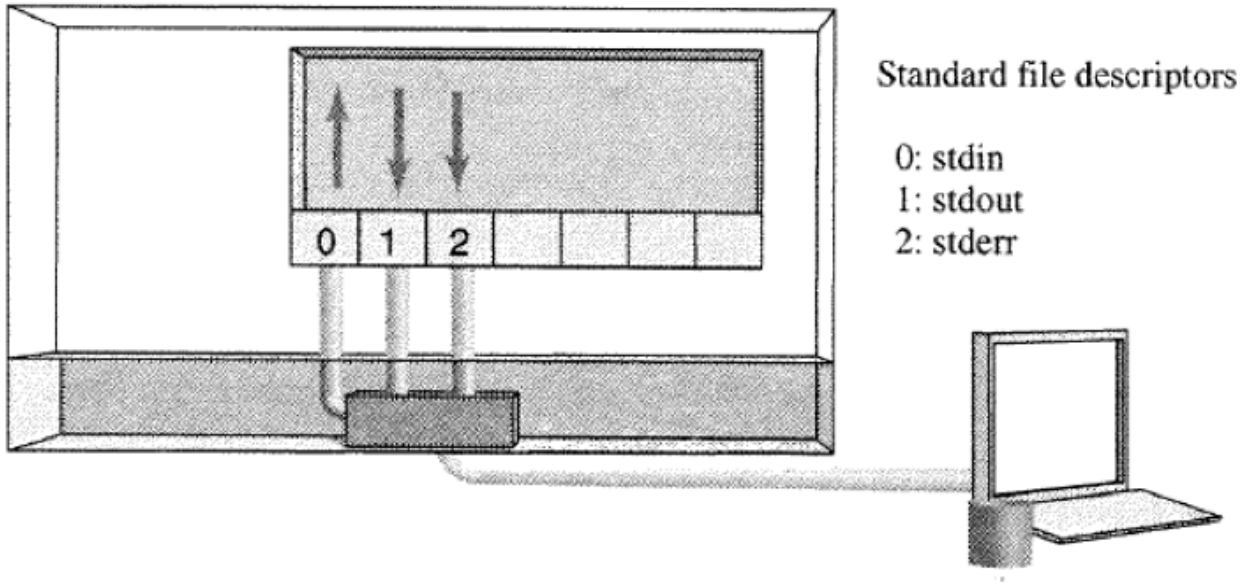
FACT ABOUT STANDARD I/O AND REDIRECTION

- All unix I/O redirection is based on the principle of standard streams of data.
- **Standard input**
  - the stream of data to process
- **Standard output**
  - the stream of result data
- **Standard error**
  - a stream of error messages



Fact One: Three Standard File Descriptors

- All Unix tools use file descriptor 0, 1, and 2.
  - 0: standard in (stdin)
  - 1: standard out (stdout)
  - 2: standard error (stderr)
- Every unix program gets three open file descriptors at startup:



Output Goes Only to stdout

- Most programs do NOT accept names for output files;
  - They always write **results** to **file descriptor 1** and **errors** to **file descriptor 2**.
  - If you want to send the output of a process to a file or to the input of another process, you need to change where the file descriptor goes.

The Shell, Not the Program, Redirects I/O

- You **tell the shell to attach file descriptor 1 to a file** by using the output redirection notation:

  $ cmd **>** filename

  - The shell **connects that file descriptor to the named file**.

  - The **program continues to write to file descriptor 1**, unaware of the new data destination.

```
/* listargs.c
 *              print the number of command line args, list the args,
 *              then print a message to stderr
 */
#include        <stdio.h>

main( int ac, char *av[] )
{
        int     i;

        printf("Number of args: %d, Args are:\n", ac);

        for(i=0;i<ac;i++)
                printf("args[%d] %s\n", i, av[i]);              // to stdout

        fprintf(stderr,"This message is sent to stderr.\n");    // to stderr
}
```

```
$ cc listargs.c -o listargs
$ ./listargs testing one two
args[0] ./listargs
args[1] testing
args[2] one
args[3] two
This message is sent to stderr.
$ ./listargs testing one two > xyz
This message is sent to stderr.
$ cat xyz
args[0] ./listargs
args[1] testing
args[2] one
args[3] two
$ ./listargs testing >xyz one two 2> oops
$ cat xyz
args[0] ./listargs
args[1] testing
args[2] one
args[3] two
$ cat oops
This message is sent to stderr.
```

1> xyz

anywhere in the command

Understanding I/O Redirection

- **Goal**:
  - Understand how I/O redirection works
  - Learn how to write programs that use it

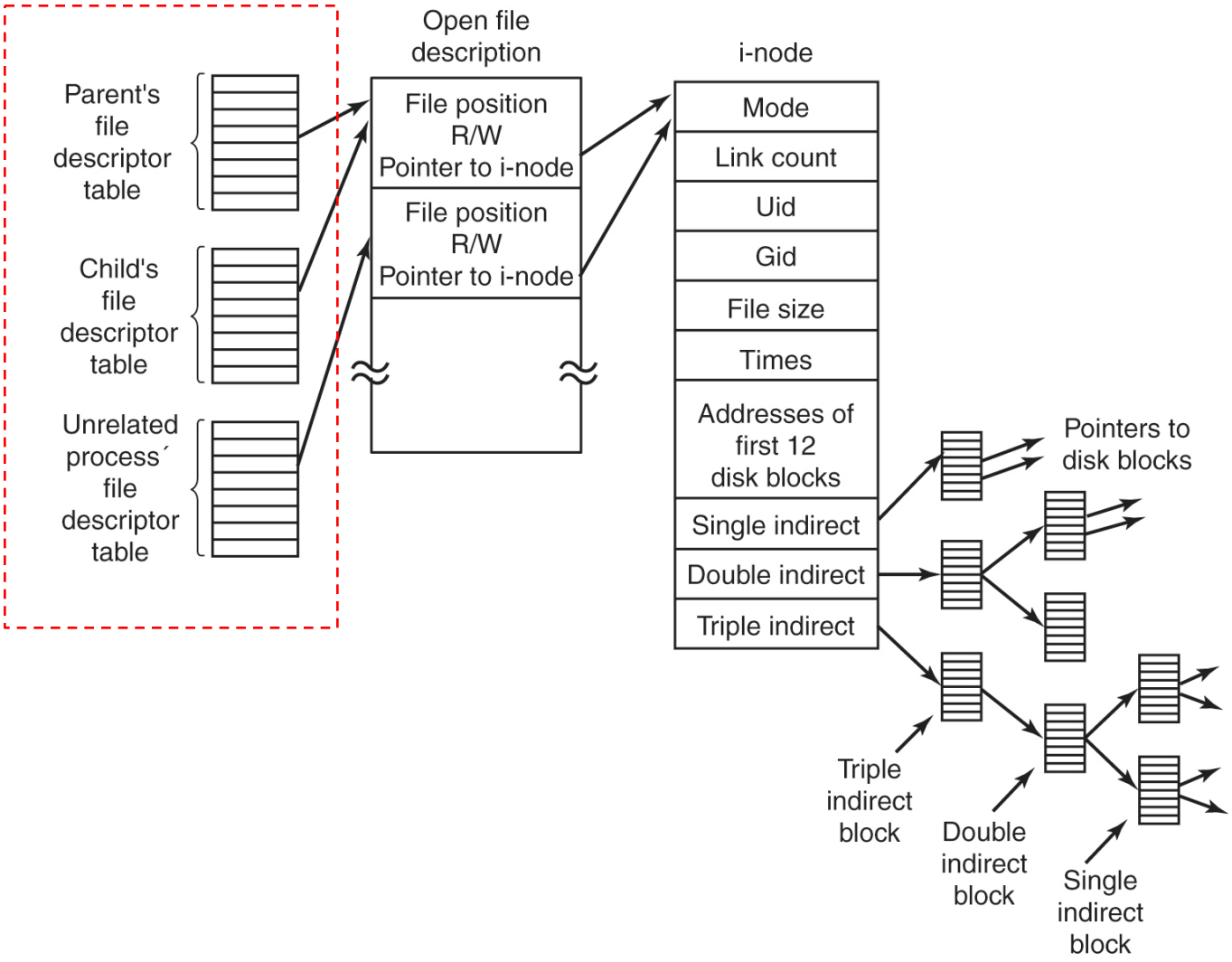
who > userlist	attach `stdout` to a file
sort < data	attach `stdin` to a file
who | sort	attach `stdout` to stdin

Fact Two: The "Lowest-Available-fd" Principle

- What is **a file descriptor**? It is an index of an array.
  - Each process has a collection of files it has open.
  - The information of those open files are kept in an array.

- **FACT:** When you open a file, you always get the lowest available spot in the array.

File descriptor



Contents

- 10.1 Shell
- 10.2 A Shell Application：Watch for Users
- 10.3 Facts about Standard I/O and Redirection
- 10.4 How to Attach stdin to a File
- 10.5 Redirecting I/O for Another Program: who>userlist
- 10.6 Programming Pipes

How to Attach stdin to a File

- How does a program redirect standard input so that data come from a file?

  ex) $ sort < data

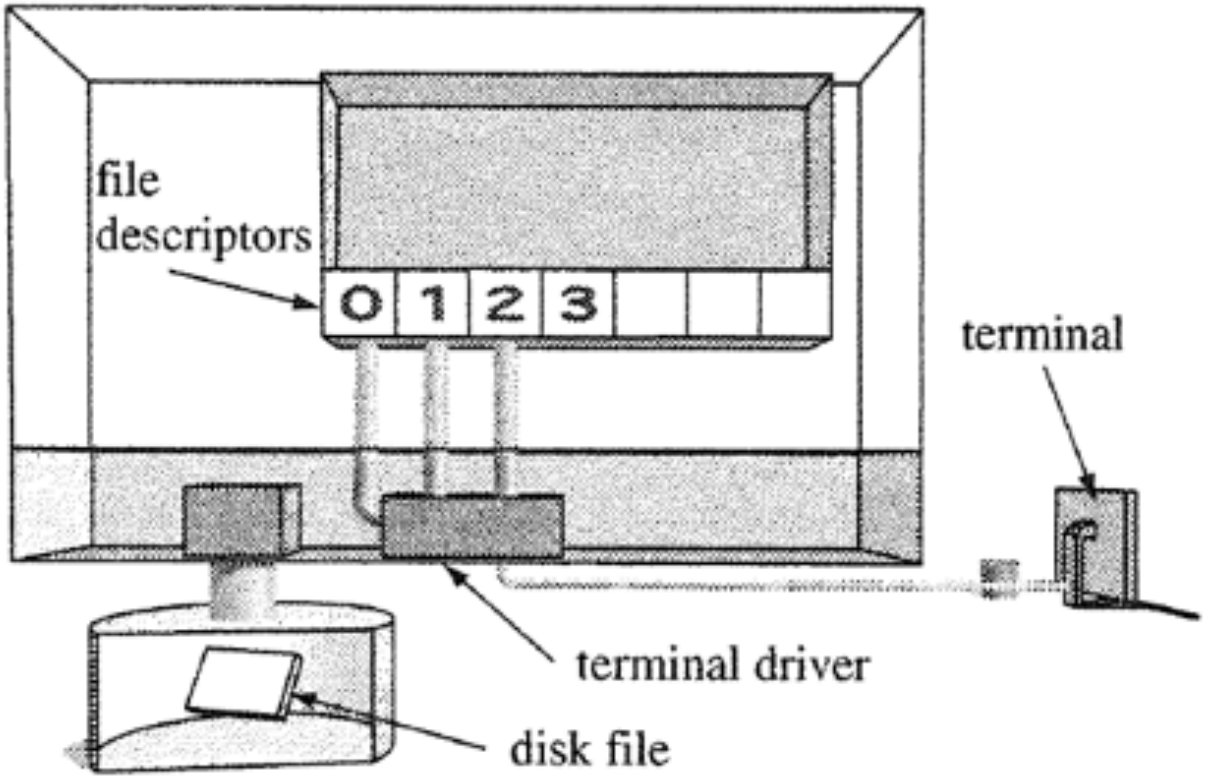
```
root@DESKTOP-K4MA2V5:~# cat data
5
4
3
2
1
root@DESKTOP-K4MA2V5:~# sort < data
1
2
3
4
5
root@DESKTOP-K4MA2V5:~#
```

- If we attach file descriptor 0 to a file, that file becomes the source for standard input.

- How? …

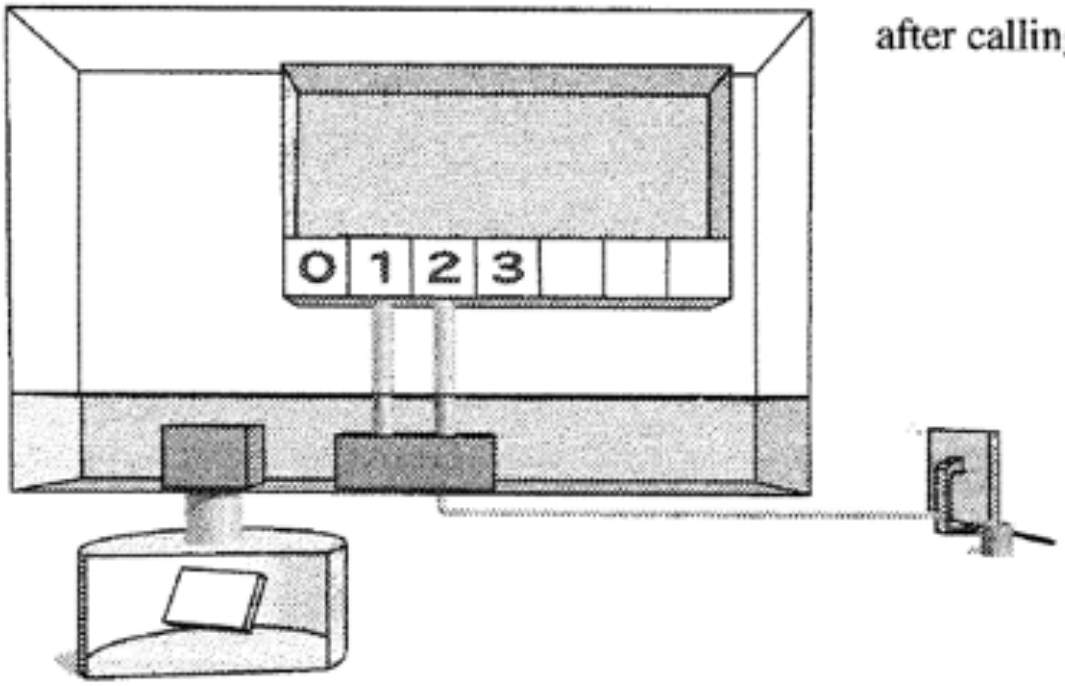
Method 1: Close Then Open

- **Starting**
  - File descriptor 0, 1, 2 attached to the terminal driver



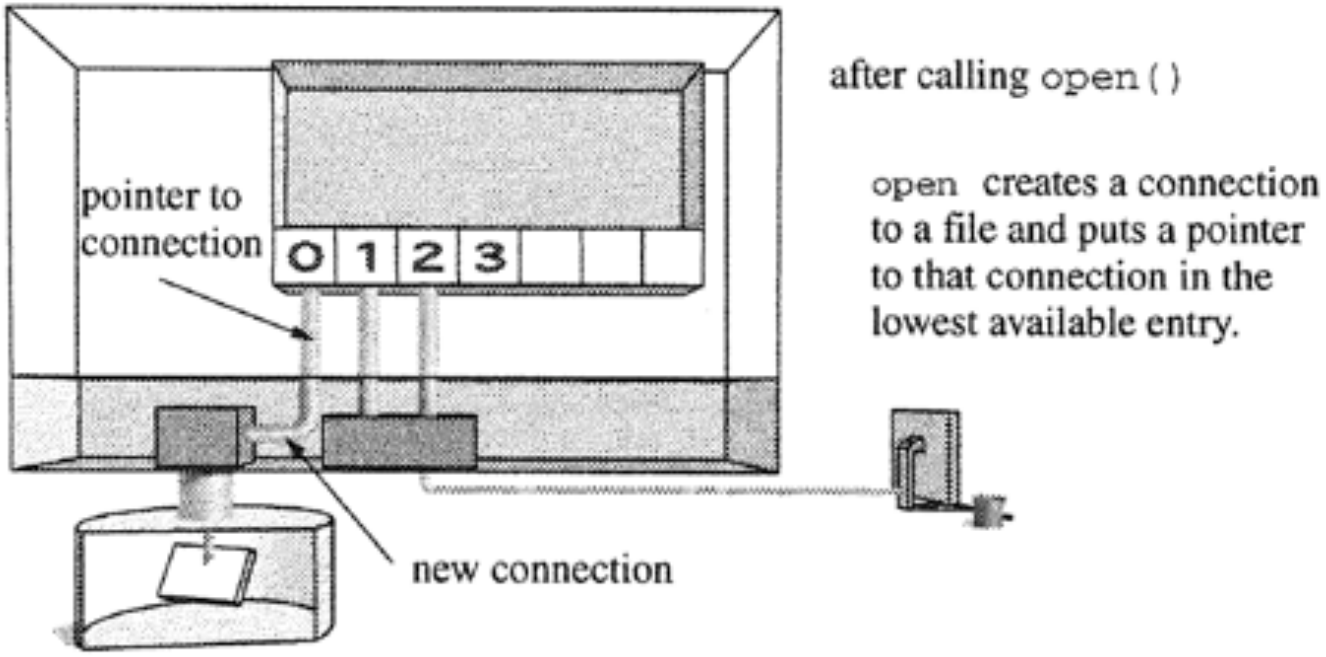
Method 1: Close Then Open

- **Then, close(0)**
  - The first element in the array of file descriptors is now unused



Method 1: Close Then Open

- **Finally, open(filename, O_RDONLY)**
  - Opens the file you want to attach to stdin.



close-then-open method

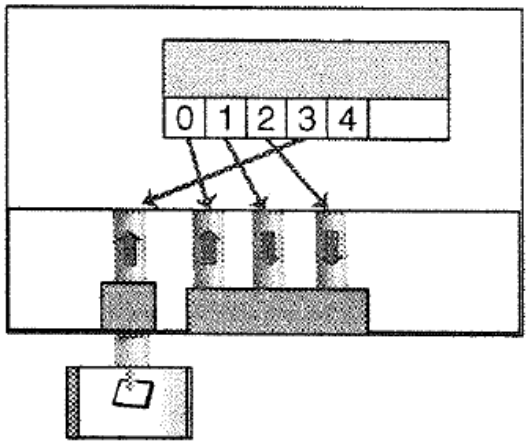
```
/* stdinredir1.c
 *      purpose: show how to redirect standard input by replacing file
 *               descriptor 0 with a connection to a file.
 *       action: reads three lines from standard input, then
 *               closes fd 0, opens a disk file, then reads in
 *               three more lines from standard input
#include      <unistd.h>
#include       <stdio.h>
#include       <fcntl.h>
#include      <stdlib.h>
main()
{
        int     fd ;
        char    line[100];

        /* read and print three lines */

        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );

        /* redirect input */

        close(0);
        fd = open("/etc/passwd", O_RDONLY);
        if ( fd != 0 ){
                fprintf(stderr,"Could not open data as fd 0\n");
                exit(1);
        }

        /* read and print three lines */

        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
}
```

Method 2: open..close..dup..close
Method 3: open..dup2..close

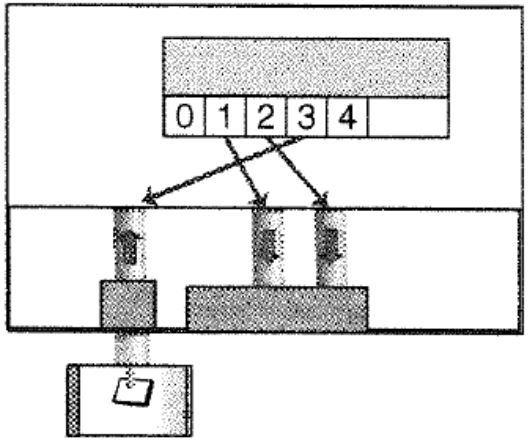
dup, dup2	
PURPOSE	Copy a file descriptor
INCLUDE	#include <unistd.h>
USAGE	newfd = dup(oldfd); newfd = dup2(oldfd, newfd);
ARGS	oldfd file descriptor to copy newfd copy of oldfd
RETURNS	-1 if error newfd new file descriptor

```
#ifdef CLOSE_DUP
        close(0);
        newfd = dup(fd);
#else
        newfd = dup2(fd,0);
#endif
```

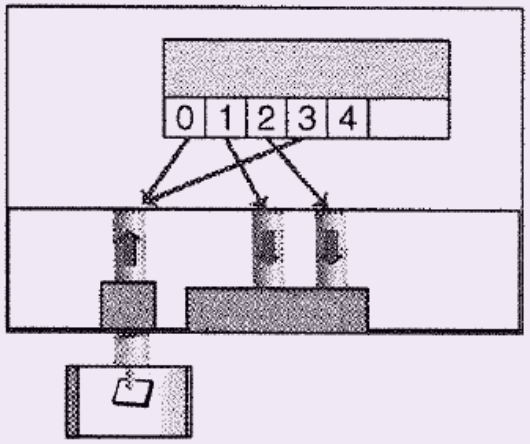
```
fd = open("f", O_RDONLY);
```




```
close(0);
```




```
dup(fd);
```




```
close(fd);
```

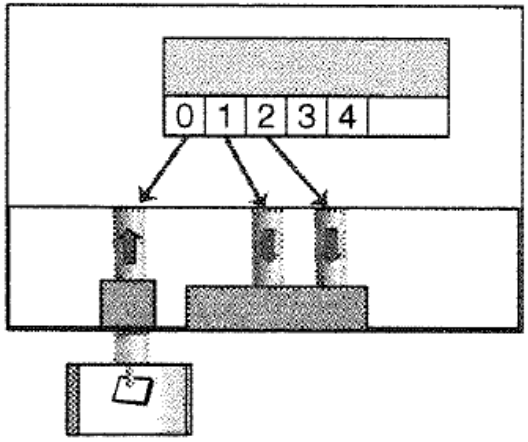


FIGURE 10.9

Using dup to redirect.

open(file) The first step is to open the file to which `stdin` should be attached. This call returns a file descriptor, which is not 0, since 0 is currently open.

close(0) The next step is to close file descriptor 0. File descriptor 0 is now unused.

dup(fd) The `dup(fd)` system call makes a duplicate of `fd`. The duplicate uses the *lowest number unused file descriptor.* Therefore, the duplicate of the connection to the file is at spot 0 in the array of open files. We have thereby attached the disk file to file descriptor 0.

close(fd) Finally, we `close(fd)`, the original connection to the file, leaving only the connection on file descriptor 0. Compare this method to the technique of moving a phone call from one extension to another.

```
/* stdinredir2.c
 *      shows two more methods for redirecting standard input
 *      use #define to set one or the other
 */ #include      <unistd.h>
#include        <stdio.h>
#include        <fcntl.h>
#include       <stdlib.h>
/* #define      CLOSE_DUP       /* open, close, dup, close */
/* #define      USE_DUP2        /* open, dup2, close */

main()
{
        int     fd ;
        int     newfd;
        char    line[100];

        /* read and print three lines */

        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
```

```
        /* redirect input */
        fd = open("data", O_RDONLY);     /* open the disk file    */
#ifdef CLOSE_DUP
        close(0);
        newfd = dup(fd);                 /* copy open fd to 0     */
#else
        newfd = dup2(fd,0);              /* close 0, dup fd to 0 */
#endif
        if ( newfd != 0 ){
                fprintf(stderr,"Could not duplicate fd to 0\n");
                exit(1);
        }
        close(fd);                       /* close original fd     */

        /* read and print three lines */

        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
}
```

※ test with "/etc/passwd"

But the Shell Redirects stdin for Other Programs

- In practice, of course, if a program wants to read a file, it can just open the file directly rather than changing standard input

- The real value of these samples is to show how one program can change standard input for another program;
  $ sort **<** data

Contents

- 10.1 Shell Programming
- 10.2 A Shell Application : Watch for Users
- 10.3 Facts about Standard I/O and Redirection
- 10.4 How to Attach stdin to a File
- 10.5 Redirecting I/O for Another Program: who > userlist
- 10.6 Programming Pipes

Redirecting I/O for other programs

- The shell redirects output for a child.
  - How “who > userlist” works?

The child inherits from the parent the pointers to open files. The child redirects standard output:

```
close(1);
creat("f");
exec();
```

```
creat(“userlist”);
exec(“who”);
```

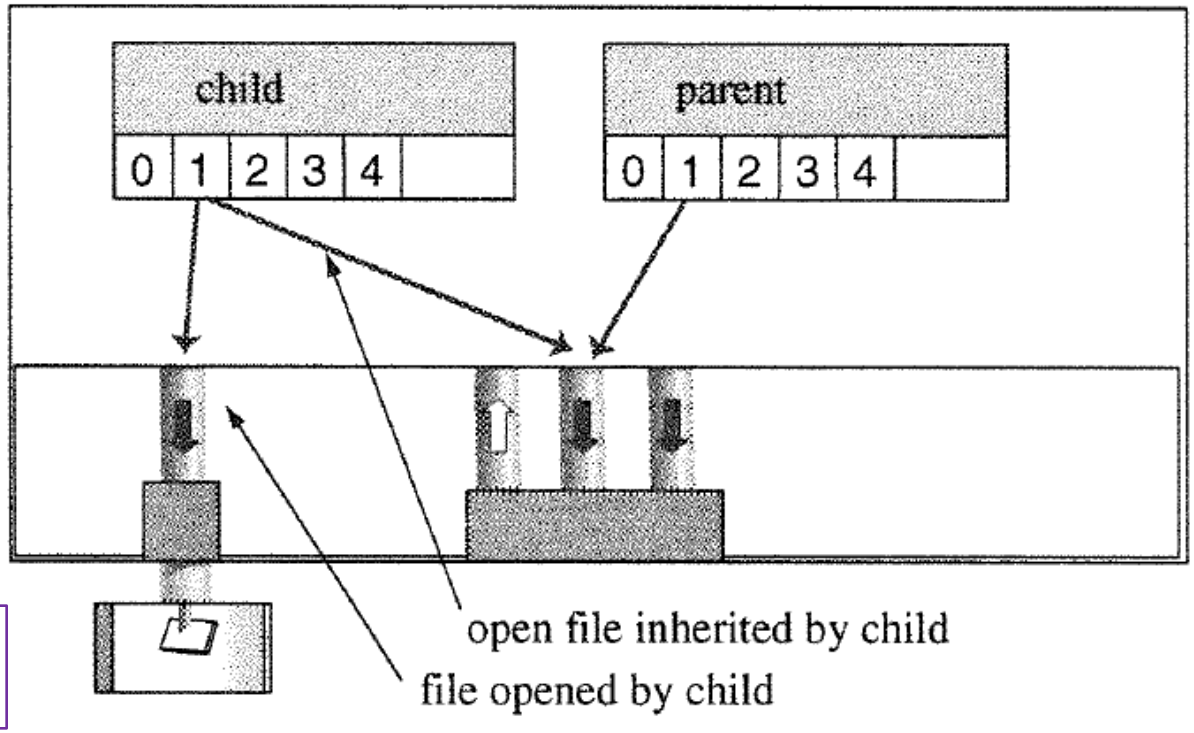
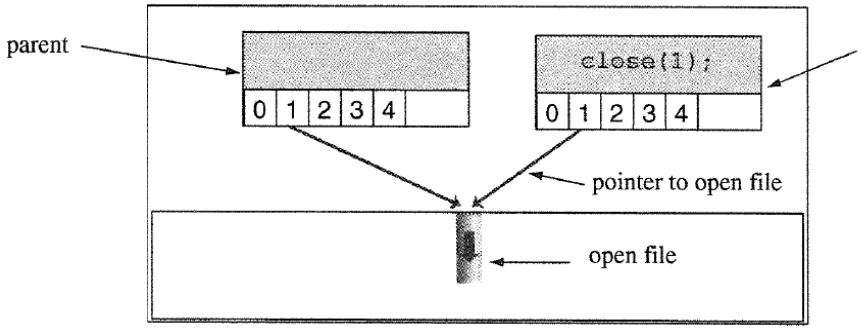


FIGURE 10.10

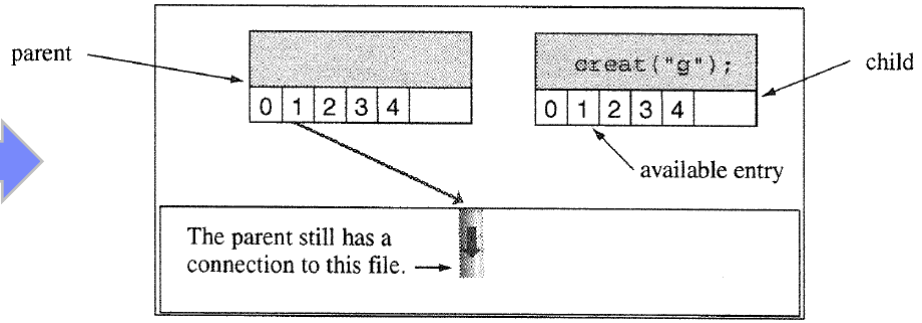
The shell redirects output for a child.

Redirecting I/O for other programs

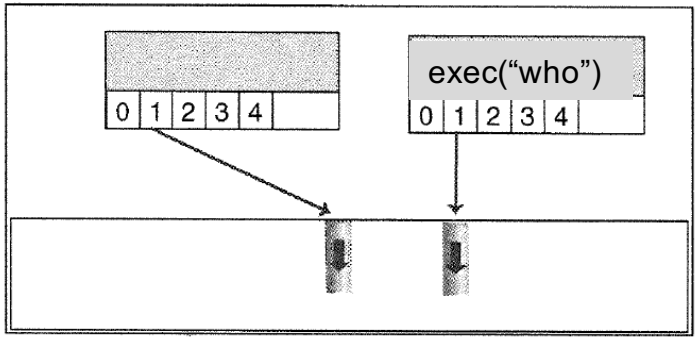
- `who > g`



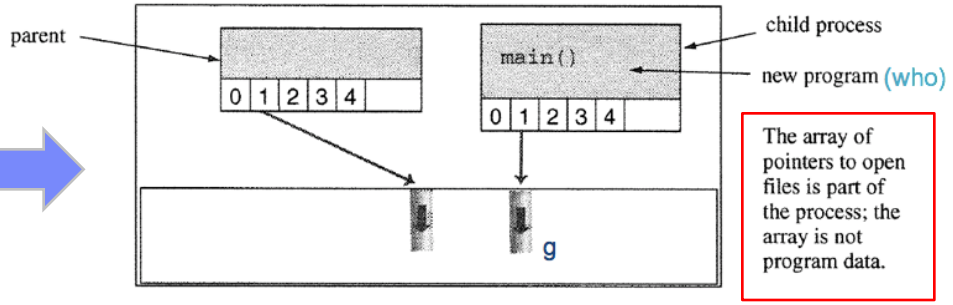
After parent calls fork().



After child calls close(1).



After child calls create("g", m).



After child execs a new program.

```c
/* whotofile.c
 *      purpose: show how to redirect output for another program
 *         idea: fork, then in the child, redirect output, then exec
 */
#include        <stdio.h>
#include        <unistd.h>    /* for execlp */
#include        <stdlib.h>    /* for exit   */
main()
{
        int     pid ;
        int     fd;

        printf("About to run who into a file\n");

        /* create a new process or quit */
        if( (pid = fork() ) == -1 ){
                perror("fork"); exit(1);
        }
        /* child does the work */
        if ( pid == 0 ){
                close(1);                               /* close, */
                fd = creat( "userlist", 0644 );         /* then open */
                execlp( "who", "who", NULL );           /* and run     */
                perror("execlp");
                exit(1);
        }
        /* parent waits then reports */
        if ( pid != 0 ){
                wait(NULL);
                printf("Done running who.  results in userlist\n");
        }
}
```

Summary of Redirection to Files

- Three basic facts:

  (a) Standard input, output, and error are file descriptors 0, 1, and 2.
  (b) The kernel always uses the lowest numbered unused file descriptor.
  (c) The set of file descriptors is passed unchanged across `exec` calls.

- The shell also supports the following forms:
  - who > userlog → write
  - who >> userlog → add
  - sort < data → read

Contents

- 10.1 Shell Programming
- 10.2 A Shell Application : Watch for Users
- 10.3 Facts about Standard I/O and Redirection
- 10.4 How to Attach stdin to a File
- 10.5 Redirecting I/O for Another Program: who > userlist
- 10.6 Programming Pipes

Pipe

Pipe

- $ who | sort
- A pipe is a one-way data channel in the kernel
  - It has a reading end and a writing end.

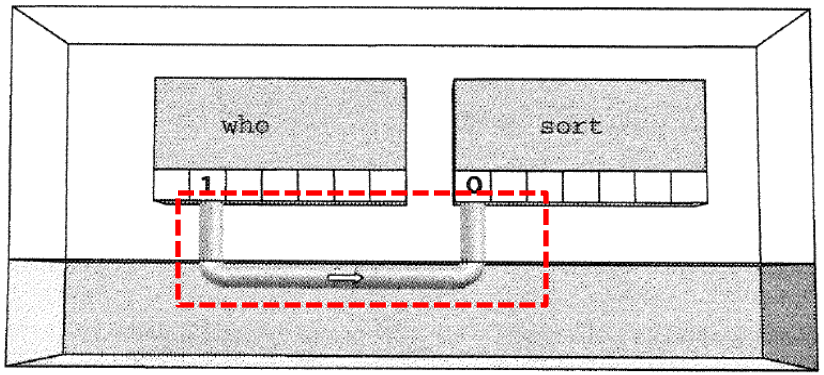


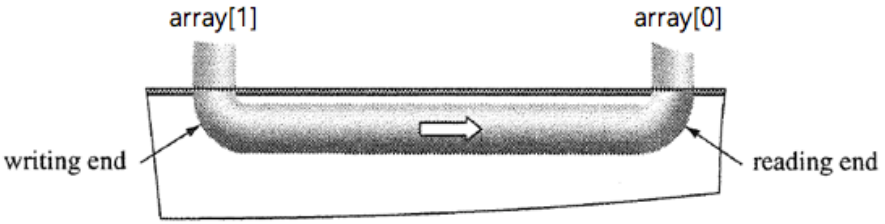
FIGURE 10.16
Two processes connected by a pipe.

- How to create a pipe and how to connect stdin and stdout to a pipe?

Creating a Pipe

- Use **`pipe()`** system call:
  - It creates the pipe and connects its two ends to two file descriptors.

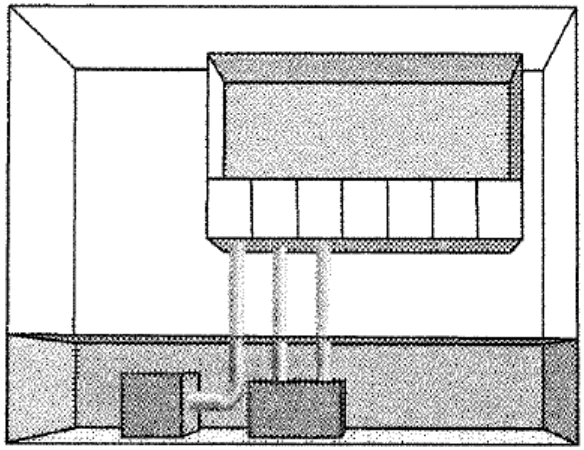
pipe		
PURPOSE	Create a pipe	
INCLUDE	#include <unistd.h>	
USAGE	result = pipe(int array[2])	
ARGS	array	an array of two ints
RETURNS	-1	if error
	0	if success



- array[0] is the file descriptor of the reading end
- array[1] is the file descriptor of the writing end

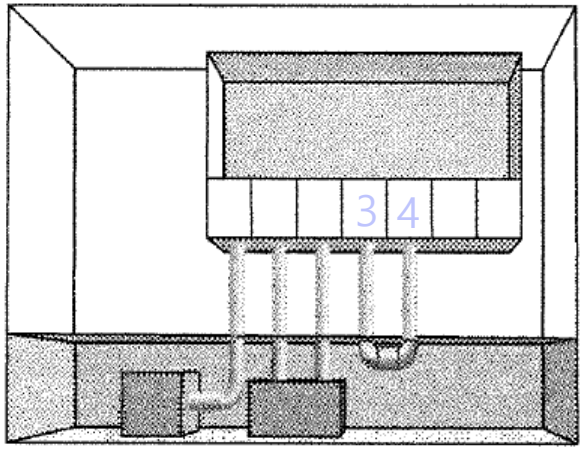
```
int     len, i, apipe[2];       /* two
char    buf[BUFSIZ];            /* for

/* get a pipe */
if ( pipe ( apipe ) == -1 ){
        perror("could not make pipe");
        exit(1);
}
```

Before `pipe`



The process has some usual files open.

After `pipe`



The kernel creates a pipe and sets file descriptors.

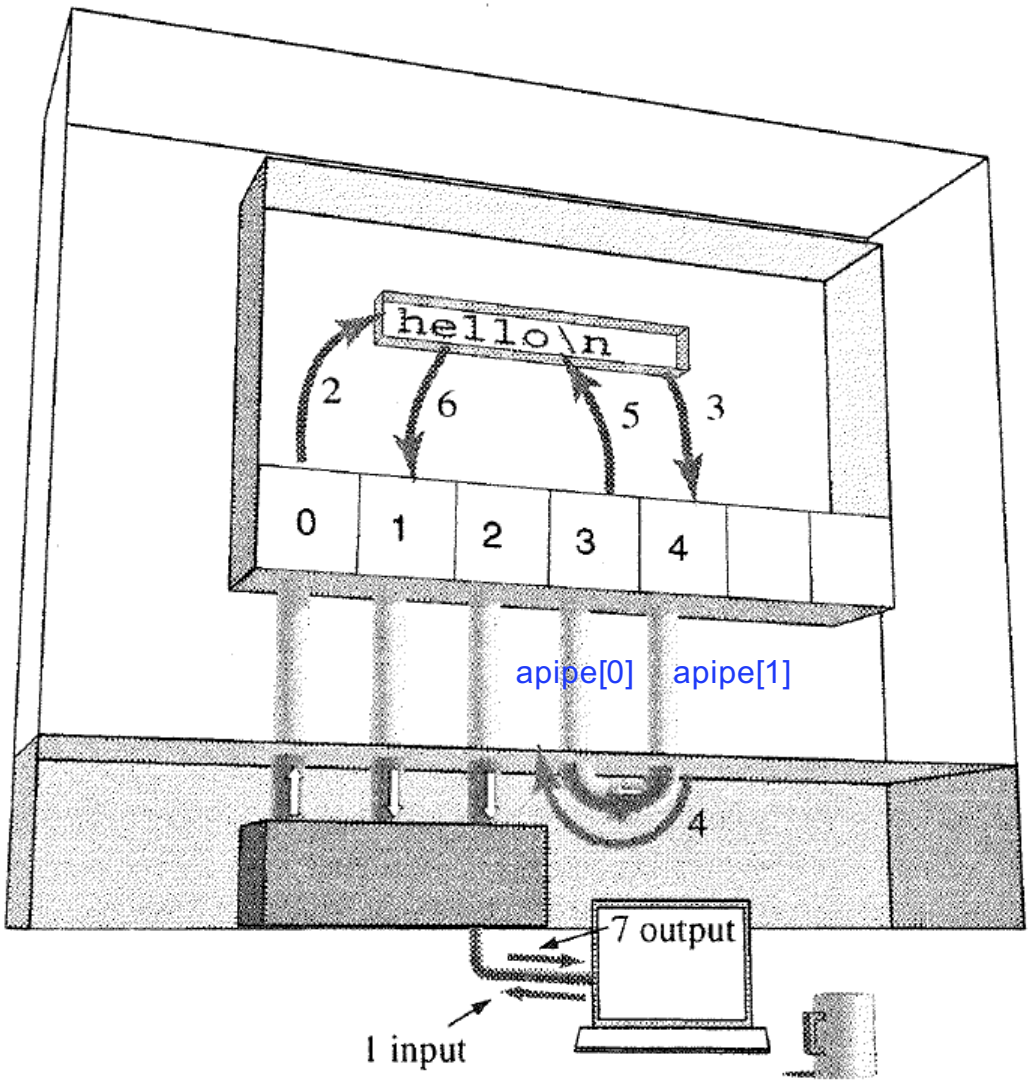
FIGURE 10.18

A process creates a pipe.

※ `pipe` uses the lowest-numbered available file descriptors.

```
/*  pipedemo.c  * Demonstrates: how to create and use a pipe
 *              * Effect: creates a pipe, writes into writing
 *                end, then runs around and reads from reading
 *                end.  A little weird, but demonstrates the idea.
 */
#include        <stdio.h>
#include        <unistd.h>

main()
{
        int     len, i, apipe[2];       /* two file descriptors */
        char    buf[BUFSIZ];            /* for reading end      */

        /* get a pipe */
        if ( pipe ( apipe ) == -1 ){
                perror("could not make pipe");
                exit(1);
        }
        printf("Got a pipe! It is file descriptors: { %d %d }\n",
                                                apipe[0], apipe[1]);

        /* read from stdin, write into pipe, read from pipe, print */
```

```
        while ( fgets(buf, BUFSIZ, stdin) ){
                len = strlen( buf );
                if ( write( apipe[1], buf, len) != len ){      /* send */
                        perror("writing to pipe");             /* down */
                        break;                                 /* pipe */
                }
                for ( i = 0 ; i<len ; i++ )                    /* wipe */
                        buf[i] = 'X' ;
                len = read( apipe[0], buf, BUFSIZ ) ;          /* read */
                if ( len == -1 ){                              /* from */
                        perror("reading from pipe");           /* pipe */
                        break;
                }
                if ( write( 1 , buf, len ) != len ){           /* send  */
                        perror("writing to stdout");           /* to    */
                        break;                                 /* stdout */
                }
        }
}
```




```
while ( fgets(buf, BUFSIZ, stdin) ){
        len = strlen( buf );
        if ( write( apipe[1], buf, len) != len ){
                perror("writing to pipe");
                break;
        }
        for ( i = 0 ; i<len ; i++ )
                buf[i] = 'X' ;
        len = read( apipe[0], buf, BUFSIZ ) ;
        if ( len == -1 ){
                perror("reading from pipe");
                break;
        }
        if ( write( 1 , buf, len ) != len ){
                perror("writing to stdout");
                break;
        }
}
```

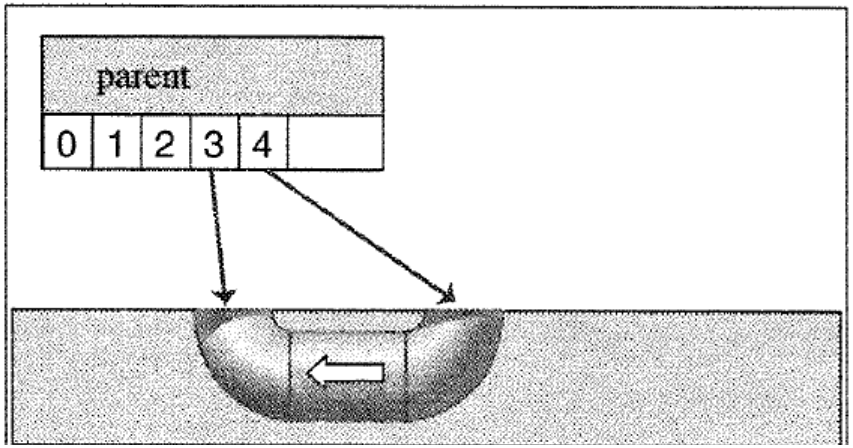
FIGURE 10.19

Data flow in `pipedemo.c`.

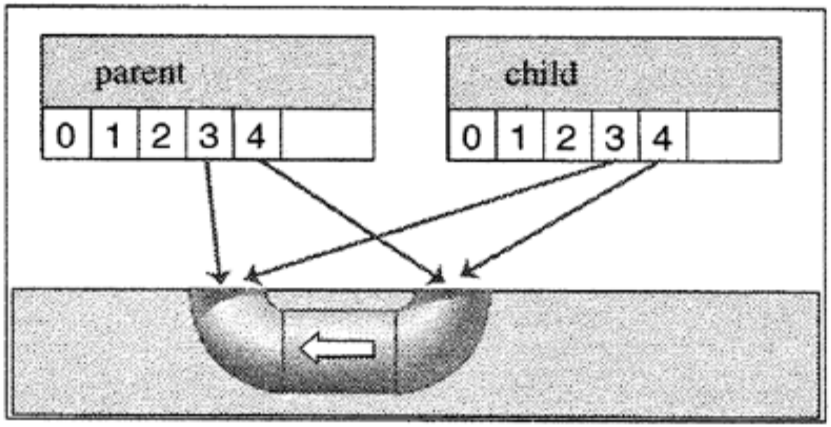
Using `fork` to Share a Pipe

Sharing a pipe:

A process calls `pipe`. The kernel creates a pipe and adds to the array of file descriptors pointers to the ends of the pipe.



The process then calls `fork`. The kernel creates a new process, and copies into that process the array of file desriptors from the parent.



Both processes have access to both ends of one pipe.

FIGURE 10.20

Sharing a pipe.

- Both processes can read and write, but a pipe is most effective when one process writes data and the other process reads data;
  - If child writes into the pipe, the parent can read those bytes

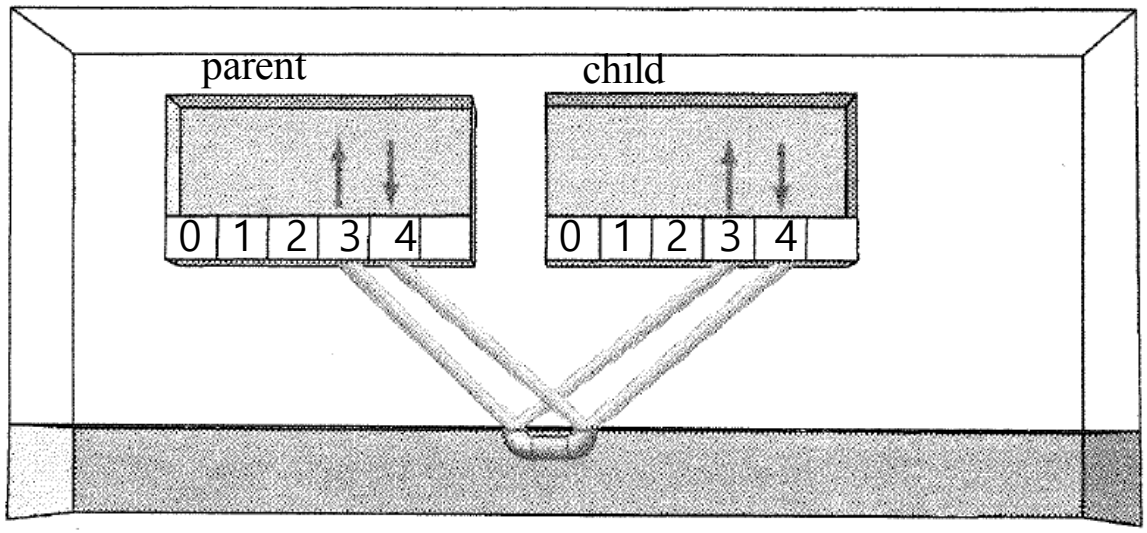


FIGURE 10.21

Interprocess data flow.

```
/* pipedemo2.c  * Demonstrates how pipe is duplicated in fork()
 *              * Parent continues to write and read pipe,
 *                but child also writes to the pipe
 */
#include        <stdio.h>

#define CHILD_MESS      "I want a cookie\n"
#define PAR_MESS        "testing..\n"
#define oops(m,x)       { perror(m); exit(x); }
```

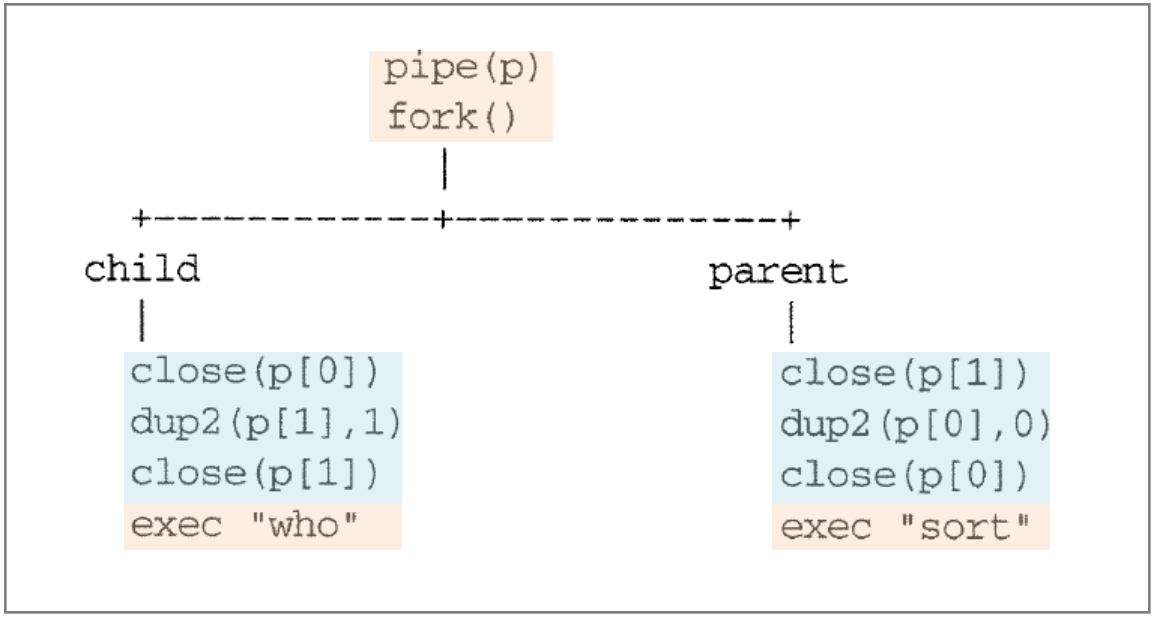
```
main()
{
        int     pipefd[2];              /* the pipe     */
        int     len;                    /* for write    */
        char    buf[BUFSIZ];            /* for read     */
        int     read_len;

        if ( pipe( pipefd ) == -1 )
                oops("cannot get a pipe", 1);

        switch( fork() ){
                case -1:
                        oops("cannot fork", 2);

                /* child writes to pipe every 5 seconds */
                case 0:
                        len = strlen(CHILD_MESS);
                        while ( 1 ){
                              if (write(pipefd[1], CHILD_MESS, len) != len )
                                        oops("write", 3);
                               sleep(5);
                        }

                /* parent reads from pipe and also writes to pipe */
                default:
                        len = strlen( PAR_MESS );
                        while ( 1 ){
                               if ( write( pipefd[1], PAR_MESS, len)!=len )
                                        oops("write", 4);
                               sleep(1);
                               read_len = read( pipefd[0], buf, BUFSIZ );
                               if ( read_len <= 0 )
                                       break;
                               write( 1 , buf, read_len );
                        }
        }
}
```

10.6.3 The Finale: Using `pipe`, `fork`, and `exec`

- Writing a general-purpose program `pipe`:

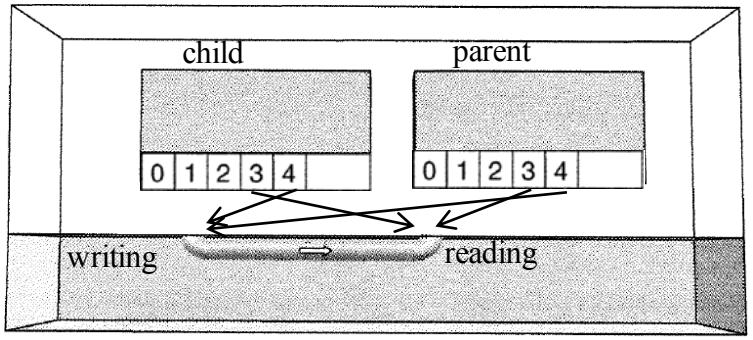
```
$ who | sort
$ ls | head
```

- Logic



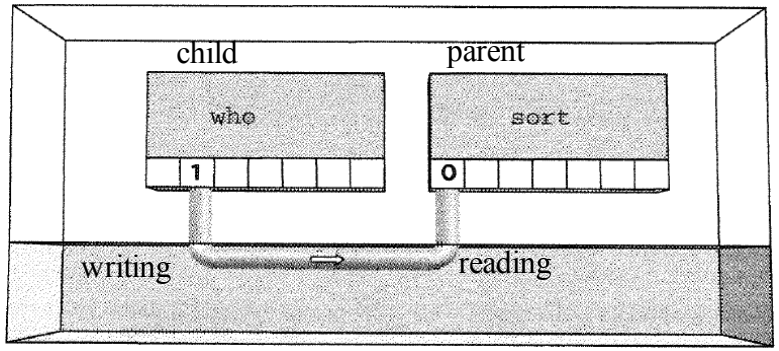
pipe.c

```
pipe(p)
fork()
```



p[0] : 3, p[1] : 4

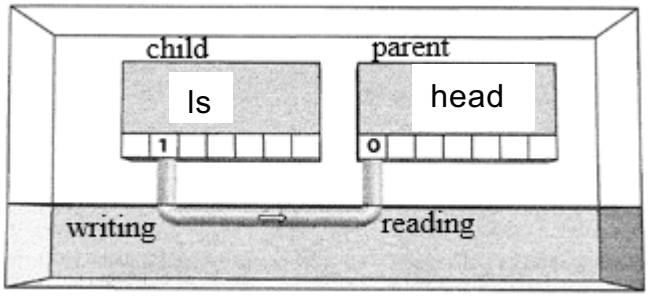
```
+-----------+-------------+
child                     parent
  |                         |
  close(p[0])               close(p[1])
  dup2(p[1],1)              dup2(p[0],0)
  close(p[1])               close(p[0])
  exec "who"                exec "sort"
```



```
# ./pipe ls head
```

```
/* pipe.c
 *      * Demonstrates how to create a pipeline from one process to another
 *      * Takes two args, each a command, and connects
 *        av[1]'s output to input of av[2]
 *      * usage: pipe command1 command2
 *        effect: command1 | command2
 *      * Limitations: commands do not take arguments
 *      * uses execlp() since known number of args
 *      * Note: exchange child and parent and watch fun
 */
#include        <stdio.h>
#include        <unistd.h>

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

main(int ac, char **av)
{
        int     thepipe[2],                     /* two file descriptors */
                newfd,                          /* useful for pipes     */
                pid;                            /* and the pid          */

        if ( ac != 3 ){
                fprintf(stderr, "usage: pipe cmd1 cmd2\n");
                exit(1);
        }
        if ( pipe( thepipe ) == -1 )            /* get a pipe           */
                oops("Cannot get a pipe", 1);
       /* ------------------------------------------------------------ */
       /*       now we have a pipe, now let's get two processes        */

        if ( (pid = fork()) == -1 )                     /* get a proc   */
                oops("Cannot fork", 2);
```

```
# ./pipe ls head
```




```
/* ----------------------------------------------------
/*      Right Here, there are two processes
/*              parent will read from pipe
if ( pid > 0 ){                     /* parent will exec av[2]       */
        close(thepipe[1]);          /* parent doesn't write to pipe */

         if ( dup2(thepipe[0], 0) == -1 )
                 oops("could not redirect stdin",3);

         close(thepipe[0]);          /* stdin is duped, close pipe   */
         execlp( av[2], av[2], NULL);
         oops(av[2], 4);
 }

/*       child execs av[1] and writes into pipe
close(thepipe[0]);                  /* child doesn't read from pipe */

if ( dup2(thepipe[1], 1) == -1 )
         oops("could not redirect stdout", 4);

close(thepipe[1]);                  /* stdout is duped, close pipe  */
execlp( av[1], av[1], NULL);
oops(av[1], 5);
}
```

```
root@DESKTOP-K4MA2V5:~# ./pipe ls head
DESKTOP-K4MA2V5
a
a.out
a.txt
data
demodir
fido.7
file1.txt
file2.txt
file3
root@DESKTOP-K4MA2V5:~#
```

Technical Details: Pipes Are Not Files

- Pipes were a special case, as they were blocks allocated but not associated to disk blocks.
  - pipe don’t have any associated disk blocks associated with it.
  - Linux has a VFS (virtual file system) module called pipefs, that gets mounted in kernel space during boot
  - pipefs cannot be directly examined by the user unlike most file systems
  - The pipe(2) syscall is used by shells and other programs to implement piping, and just creates a new file in pipefs, returning two file descriptors (one for the read end, opening using O_RDONLY, and one for the write end, opened using O_WRONLY)
  - pipefs is stored using an in-memory file system