System Programming
(ELEC462)

I/O Redirection and Pipes

Dukyun Nam

HPC Lab@KNU

Contents

Introduction

- 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`

Shell Programming

- How do the following commands work?

```
ls > myfiles
who | sort > userlist
```

- Questions
  - How does the shell tell a program to send its output to a file instead of the screen?
  - How does the shell connect the output stream of one process to the input stream of another process?
  - What's the real meaning of 'standard input'?
- In this chapter, we'll focus on a particular form of IPC (Interprocess communication)
  - Input/output (I/O) redirection and pipes

A Shell Application: Watch for Users (`watch.sh`)

- Consider the following program.
    - You have a list of buddies accessing the same Linux machine
    - A program notifies you when people “log in” or “log out” of the system so you can watch for your peers
        - Let’s write a watch shell script!

```
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
```

```sh
#!/bin/sh
#
#  watch.sh  - a simple version of the watch utility, written in sh
#
        who | sort > prev                 # get initial user list
        while true                        # true is a program: exit(0);
        do
                sleep 10                  # wait a while
                who | sort > current      # get current user list
                echo "Logged out:"        # print header
                comm -23 prev current     # and results
                echo "Logged in:"         # header
                comm -13 prev current     # and results
                mv current prev           # make now past
        done
```

A Shell Application: Watch for Users (cont.)

- Execution

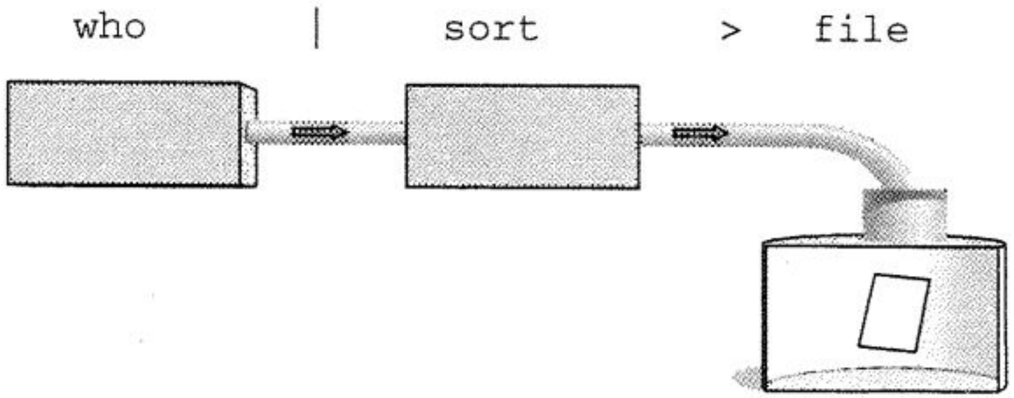
```
dynam@DESKTOP-Q4IJBP7:~/lab11$ ./watch.sh
Logged out:
Logged in:
Logged out:
dynam    pts/1        2022-11-15 14:59
Logged in:
Logged out:
Logged in:
Logged out:
Logged in:
dynam    pts/1        2022-11-15 15:00
Logged out:
```

  - In WSL, if you have no utmp file, run the following commands

```
$ sudo bash -c "echo '[1] [00049] [~~  ] [runlevel] [~           ] [4.4.0-17115-Micoroso] [0.0.0.0        ] [Wed Feb 28 13:27:14 2018 STD]' | utmpdump -r > /var/run/utmp"

$ sudo login -f username
```

A Shell Application: Watch for Users (cont.)

- `who | sort > prev`
  - Tells the three things to the shell:
    - 1) Run the commands who and sort at the same time
    - 2) Send the output of who directly to the input to sort
      - Not necessary to finish analyzing the `utmp` file before sort begins its task
      - The two processes are scheduled to run in small time slices, sharing CPU time with other processes on the system
    - 3) Send the output of sort into a file, called `prev`: what if it already exists?

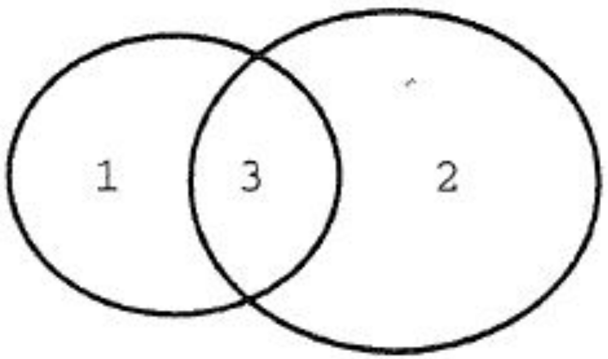


< Connecting output of `who` to input of `sort` >

A Shell Application: Watch for Users (cont.)

- comm: a command to find lines common to two sorted files
  - Compares two sorted lists and prints out the three columns
  - In the example, it produces exactly the two sets we want
    - Logouts: who did leave? (`comm -23 prev current`)
    - Logins: who are new? (`comm -13 prev current`)

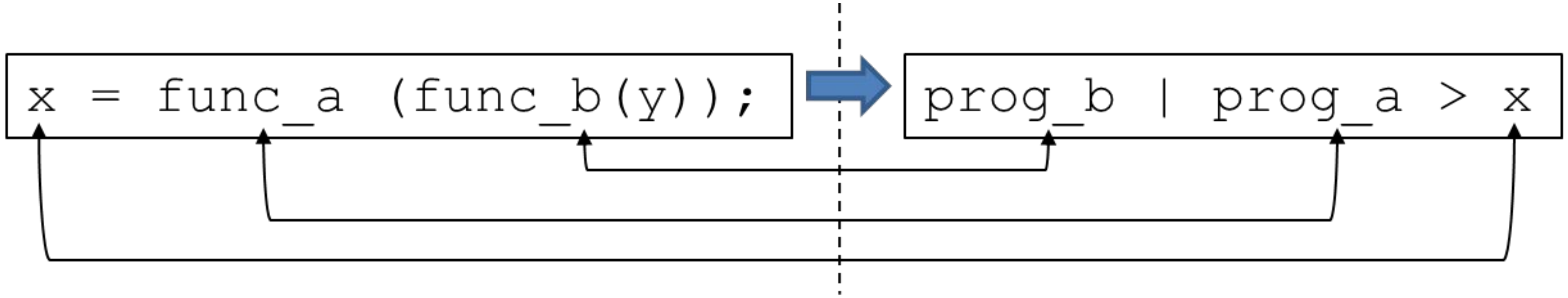
```
COMM(1)                      User Commands                      COMM(1)

NAME
       comm - compare two sorted files line by line

SYNOPSIS
       comm [OPTION]... FILE1 FILE2

DESCRIPTION
       Compare sorted files FILE1 and FILE2 line by line.

       When FILE1 or FILE2 (not both) is -, read standard input.

       With  no  options,  produce  three-column  output.   Column one contains lines
       unique to FILE1, column two contains lines unique to FILE2, and  column  three
       contains lines common to both files.

       -1     suppress column 1 (lines unique to FILE1)

       -2     suppress column 2 (lines unique to FILE2)

       -3     suppress column 3 (lines that appear in both files)
```



< `comm` compares two lists and outputs three sets >

A Shell Application: Watch for Users (cont.)

- Three important ideas behind `watch.sh`
  - (1) Power of shell scripts
    - Easier and quicker than C (or other programming languages requiring compiling)
  - (2) Flexibility of software tools
    - Each tool (or command or program) does one specific, general task
  - (3) Use and value of I/O redirection and pipes
- And one more …
  - The script shows how to use the ‘>’ operator to treat files as variables of arbitrary size and structure

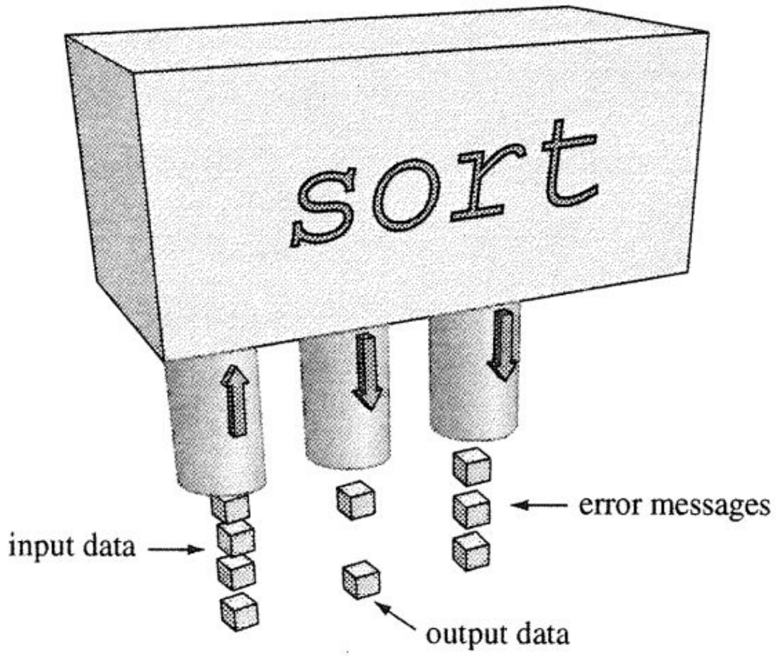


A Shell Application: Watch for Users (cont.)

- Questions
  - How does all of *these connected programs* jointly work?
  - What role does the *shell* play in connecting processes?
  - What role does the *kernel* play to get the processes to work?
  - What role do the *individual programs* play?

Facts about Standard I/O & Redirection

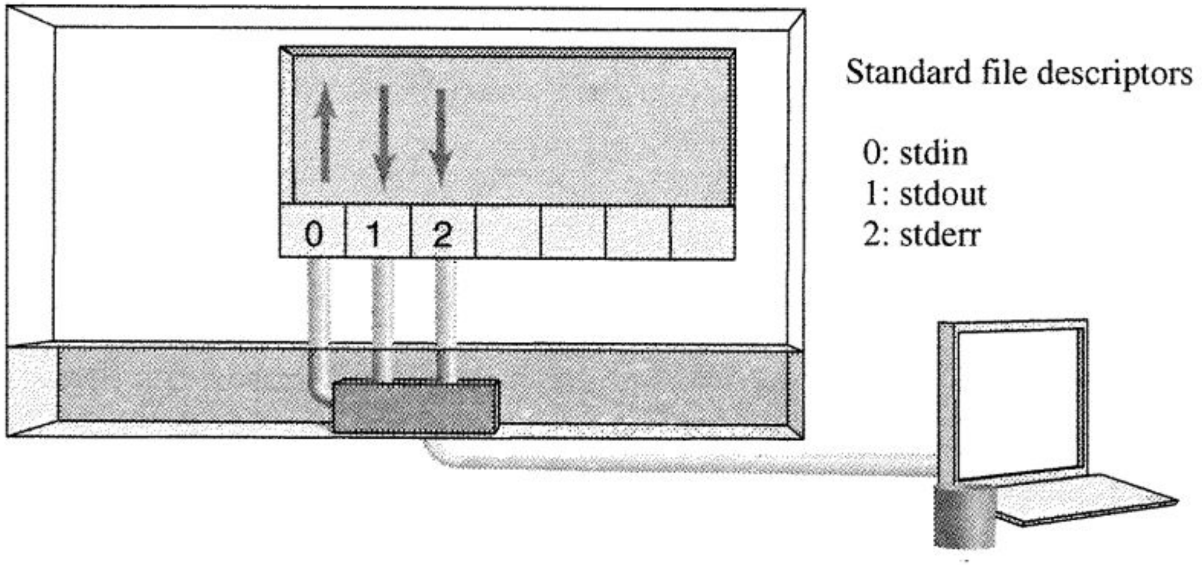
- All Linux/Unix I/O redirection based on
  - Principle of ***standard streams of data***
  - e.g., The task of `sort`
    - Reads bytes from one stream of data
      - Then it performs the sorting task on the read byte stream
    - Writes the sorted results to another stream
    - Reports any errors to a third stream
  - The three channels for data flow are as follows:
    - **Standard input**: The stream of data to process
    - **Standard output**: The stream of result data
    - **Standard error**: A stream of error messages



< A software tool reads input and writes output and errors >

Fact 1: 3 Standard File Descriptors

- All Linux/Unix tools make use of the three-stream model
  - Each of the stream is a specific *file descriptor* (`fd`)
  - Linux/Unix tools find file descriptors 0, 1, and 2 already open for reading, writing, and writing, respectively



< Three special file descriptors >

Default Connections: the `tty`

- When you run a Linux tool (`who`, `sort`, `comm`, ...) on your shell
  - The three (`stdin`, `stdout`, and `stderr`) streams are usually “connected” to your terminal
- So the tool `sort`:
  - Reads from the keyboard (`stdin`)
  - Writes output (`stdout`)
  - Writes error messages to the screen (`stderr`)
    - If Ctrl-D is pressed, then sort starts to begin the input and writes the result to stdout.

- Most tools process data from files or `stdin`
  - Given file names => they read input from those files
  - No files given => they read from standard input

The Shell, Not the Program, Redirects I/O

- *`cmd > filename`*
  - Tells the shell to attach `fd 1` to a file
    - By using the output redirection notation as specified above

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

int 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]);

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

        return 0;
}
```

The Shell, Not the Program, Redirects I/O (cont.)

- `listargs` prints to standard output the list of command-line arguments
    - Does not print the redirection symbol and filename

```
$ 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.
```

Some Important Facts …

- The shell *doesn't* pass the redirection symbol and filename to the command
- The redirection request can appear *anywhere* in the command.
  - Doesn't require spaces around the redirection symbol (>)
    - Even a command like '`> listing ls`' is acceptable
  - Doesn't terminate the command and arguments: just an added request
- Many shells provide notation for redirecting other fds
  - e.g., `2>filename`
    - Redirects fd 2, that is, standard error, to the named file

Understanding I/O Redirection

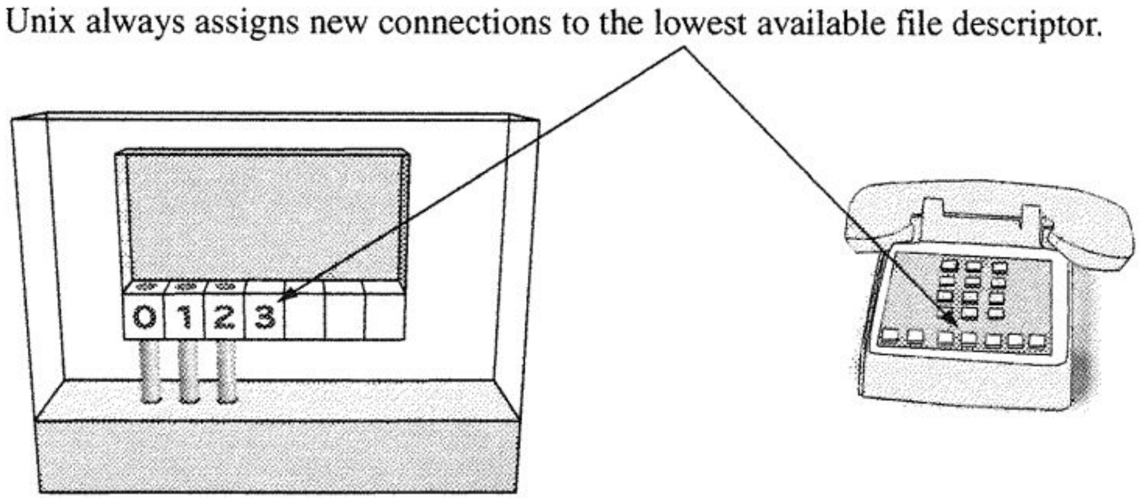
- Goal
  - Understand how I/O redirection works

    AND learn how to write programs that use it
- Method: write programs that do
  - `sort < data` attach `stdin` to a file
  - `who > userlist` attach `stdout` to a file
  - `who | sort` attach `stdout` to `stdin`

Fact 2: the “Lowest-Available-`fd`” Principle

- The meaning of a file descriptor? **An array index!**
  - Each process has a collection of files it has open
    - Those “open” files are kept in an array
  - So a file descriptor: simply an index for an item in that array
  - Making a new connection with file descriptors is like receiving a connection on a multiline phone

    the next incoming call => mapped to the lowest available line



< The “lowest-available-file-descriptor” rule >

How to Attach `stdin` to a File

- Standard I/O?
  - Standard input (`stdin`), output (`stdout`), and error (`stderr`) are indicated by file descriptors 0,1 and 2, respectively
  - Three predefined streams:

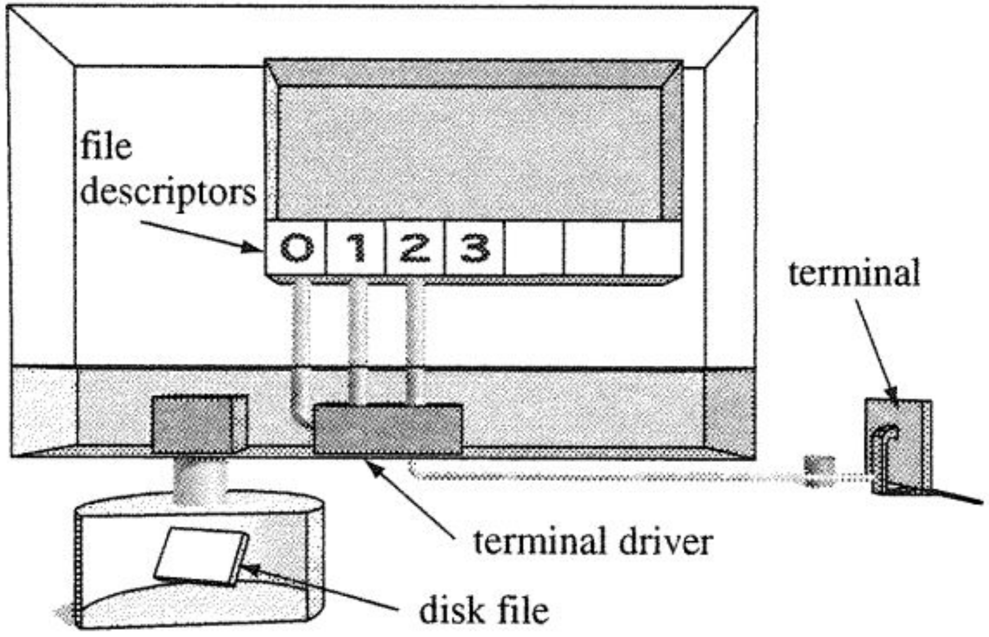
Integer value	Name	file stream in `<stdio.h>`
0	standard input	`stdin`
1	standard output	`stdout`
2	standard error	`stderr`

How to Attach `stdin` to a File (cont.)

- How does a Linux program redirect `stdin` in order for data to come from a file?
  - Linux processes don't read from *files*, but actually from ***file descriptors***
  - If `fd 0` is attached to a file, then the attached file becomes the source for standard input
- There are three methods for attaching `stdin` to a file
  - Method 1: *Close-then-open*
  - Method 2: *Open-close-dup-close*
  - Method 3: *Open-dup2-close*

Method 1: Close-then-Open

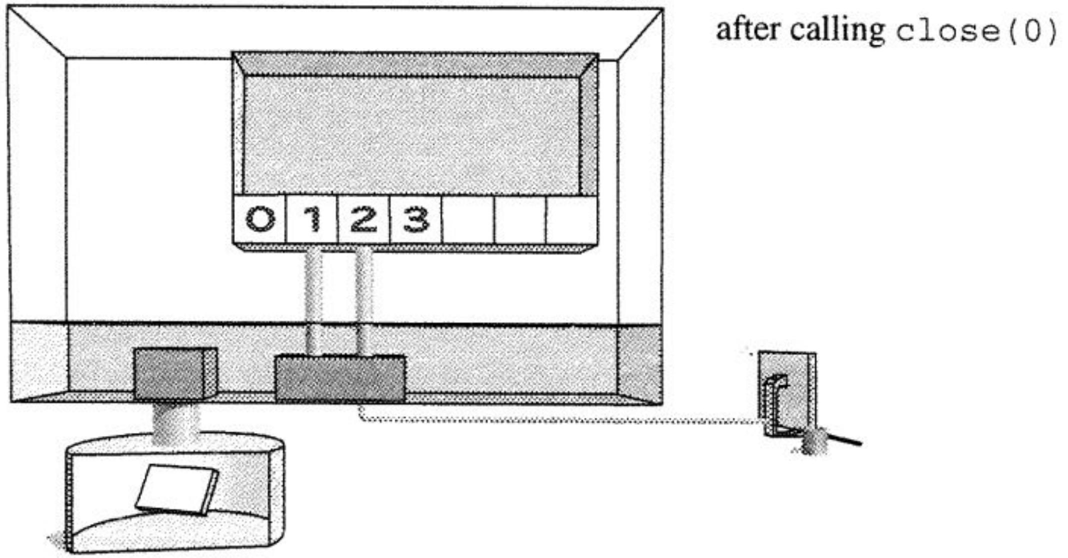
- Step 1) *Starting* with the three standard streams connected to the terminal driver
  - File descriptors 0, 1, 2 attached to `/dev/tty`
    - 0 for reading
    - 1 for writing
    - 2 for writing



< Typical starting configuration >

Method 1: Close-then-Open (cont.)

- Step 2) *Then, close(0)*: hang up the connection to stdin
  - Breaks the connection from standard input to the driver
    - See the "unused" element in the array below
  - If the process closes file descriptor 0,
    - that entry in its array of I/O channels is free



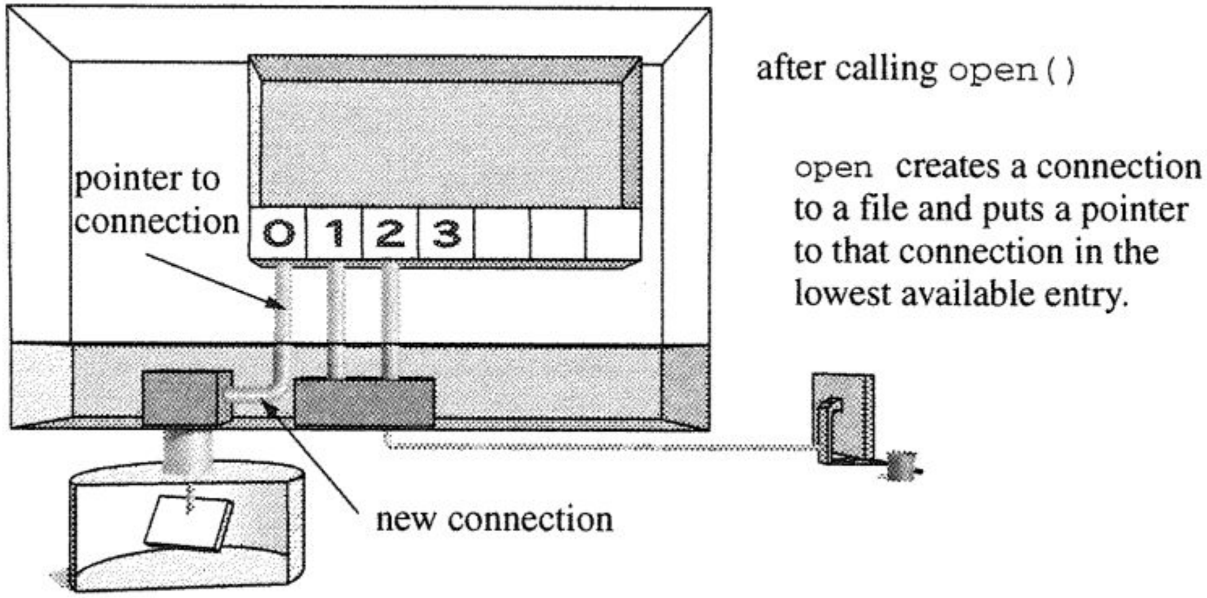
< `stdin` is now closed >

Method 1: Close-then-Open (cont.)

- Step 3) *Finally,* *open(filename, O_RDONLY)*
  - If the process opens another file,
    - that connection is attached to

      the FIRST FREE entry

      in the array of I/O channels



< `stdin` now attached to file >

Method 1: Close-then-Open (cont.)

- `stdinredir1.c`

What's returned?

```c
/* 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        <stdio.h>
#include        <stdlib.h>
#include        <unistd.h>
#include        <fcntl.h>

int 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 );

        return 0;
}
```

Method 1: Close-then-Open (cont.)

- Execution

```
dynam@DESKTOP-Q4IJBP7:~/lab11$ ./stdinredir1
line1
line1
testing line2
testing line2
line3
line3
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

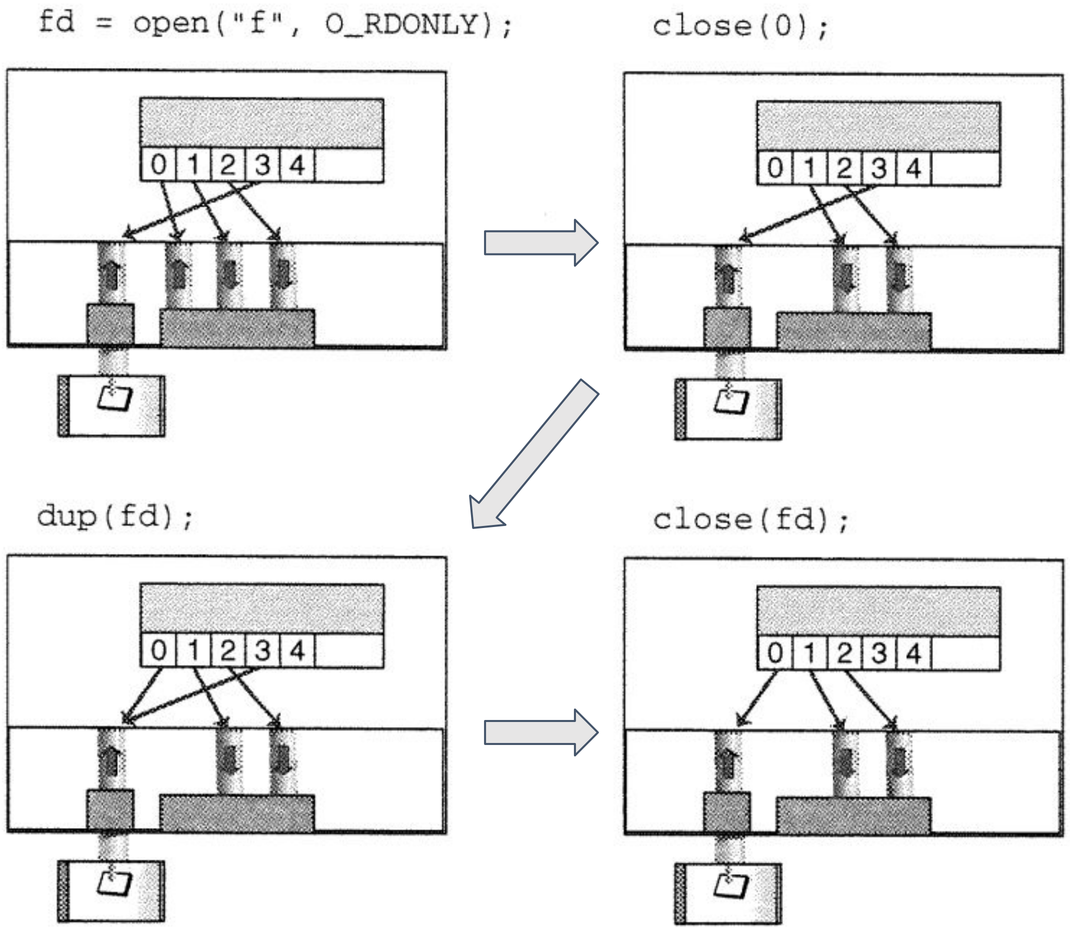
```
dynam@DESKTOP-Q4IJBP7:~/lab11$ head -3 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
dynam@DESKTOP-Q4IJBP7:~/lab11$
```

Method 2: `Open..close..dup..close`

- The system call `dup` makes a second connection to an existing file descriptor
  - `open(file)`: open a file to which `stdin` should be attached
    - Will return a file descriptor with a non-zero, as 0 is still in use
  - `close(0)`: close fd 0, which becomes now "unused"
  - `dup(fd)`: makes a "duplicate" of `fd`
    - Uses the lowest but not yet used number for fd
    - So what would be the number?
      - The duplicate of the connection to the file: located at spot 0 in the array open files
  - `close(fd)`: invokes `close(fd)`, the original connection to the file
    - Leaving only the connection to file descriptor 0

Method 2: `Open..close..dup..close` (cont.)

```
        /* redirect input */
        fd = open("/etc/passwd", 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    */
```



< Using `dup` to redirect >

Method 2: `Open..close..dup..close` (cont.)

- `stdinredir2.c`

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

#define CLOSE_DUP               /* open, close, dup, close */
/* #define      USE_DUP2        /* open, dup2, close */

int 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 );
```

```c
        /* redirect input */
        fd = open("/etc/passwd", 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 );

        return 0;
}
```

Method 2: `Open..close..dup..close` (cont.)

- Execution

```
dynam@DESKTOP-Q4IJBP7:~/lab11$ ./stdinredir2
line1
line1
line2
line2
line3
line3
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

Method 3: Open..dup2..close

- The code for stdinredir2.c includes #ifdef-ed code
  - to replace the close(0) and dup(fd) system calls with dup2 (fd, 0)

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

/* #define      CLOSE_DUP               /* open, close, dup, close */
#define USE_DUP2        /* open, dup2, close */
```

System Call Summary: `dup`

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

- The system call `dup` creates a copy of *oldfd* as *newfd*
- The system call `dup2` gets *newfd* associated with the copy of *oldfd*

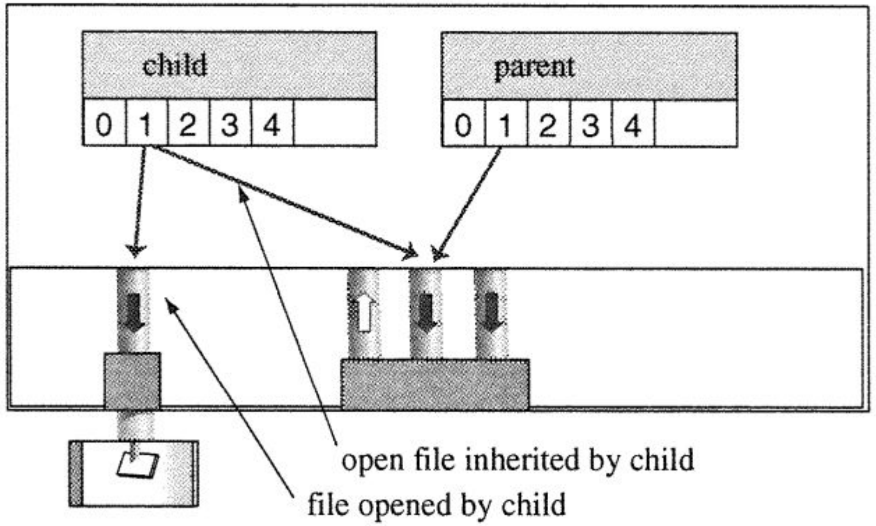
⇒ The two newfds actually refer to the same open file pointed by oldfd!

Redirecting I/O for Another Program

- `who > userlist`
  - The shell runs the command `who`
    - with the standard output of `who` attached to the file called `userlist`
- Key: the *split second* between `fork` and `exec`
  - After `fork`, the child is running the shell code, but is about to execute `exec`
  - `exec` will replace the program running in the process with no change of attributes or connections of the process
  - The process will have the same file descriptors it had before the `exec`

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

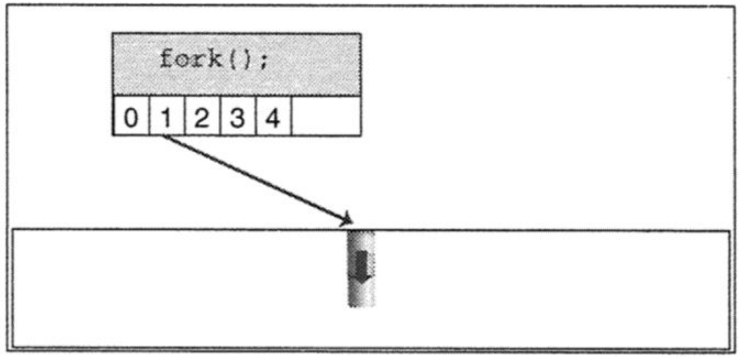
```
close(1);
creat("f");
exec();
```



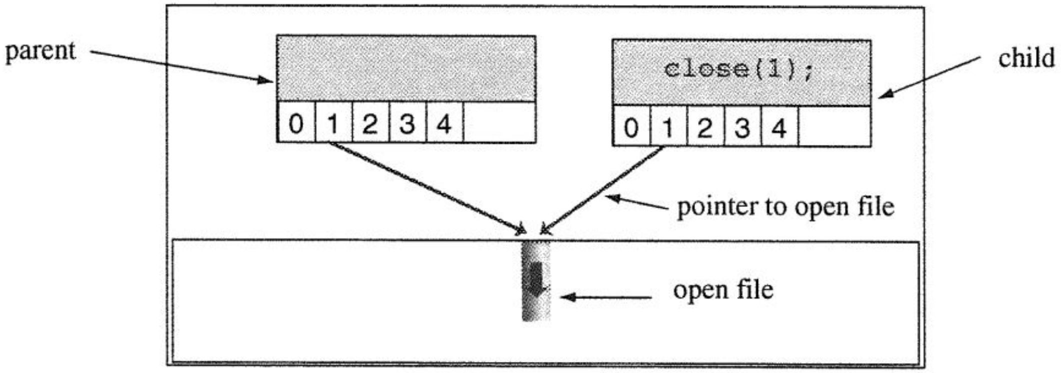
< The shell redirects output for a child >

Redirecting I/O for Another Program: `who > userlist` (cont.)

- Step 1. Start here: a process that is about to `fork` and its `stdout`
- Step 2. After parent calls `fork()`: `stdout` of a child copied from parent



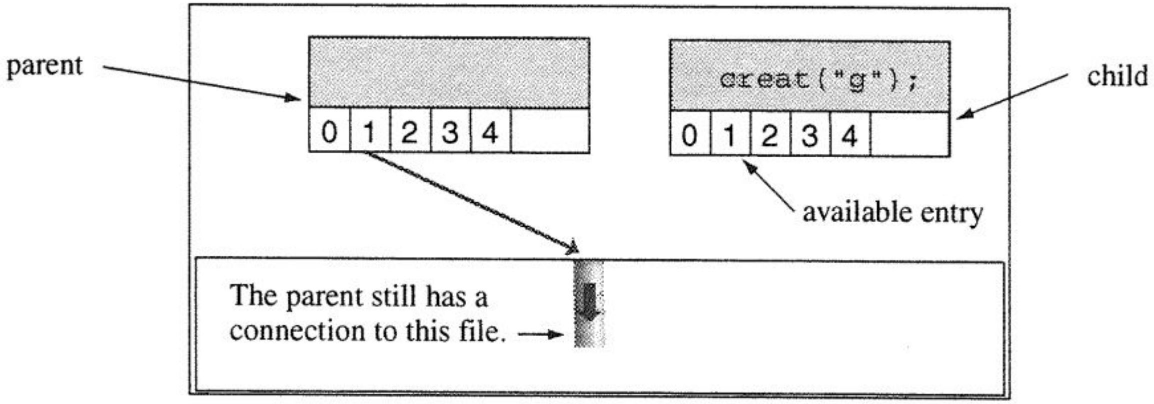
< A process about to fork and its standard output >



< Standard output of child is copied from parent >

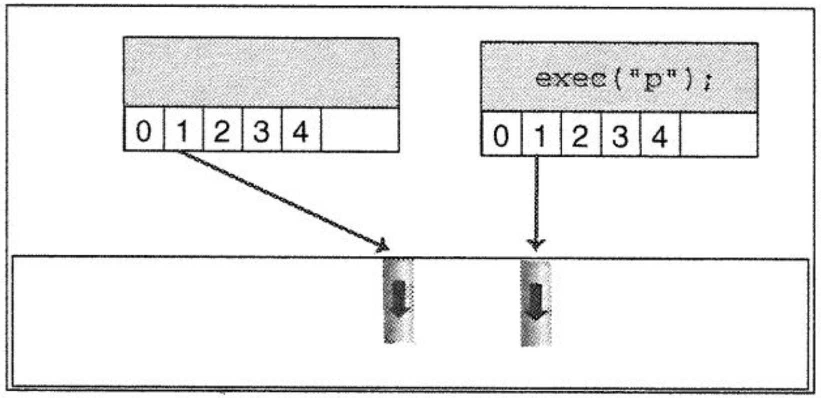
Redirecting I/O for Another Program: `who > userlist` (cont.)

- Step 3. After child calls `close(1)`: child can close its `stdout`



< The cold can close its standard output >

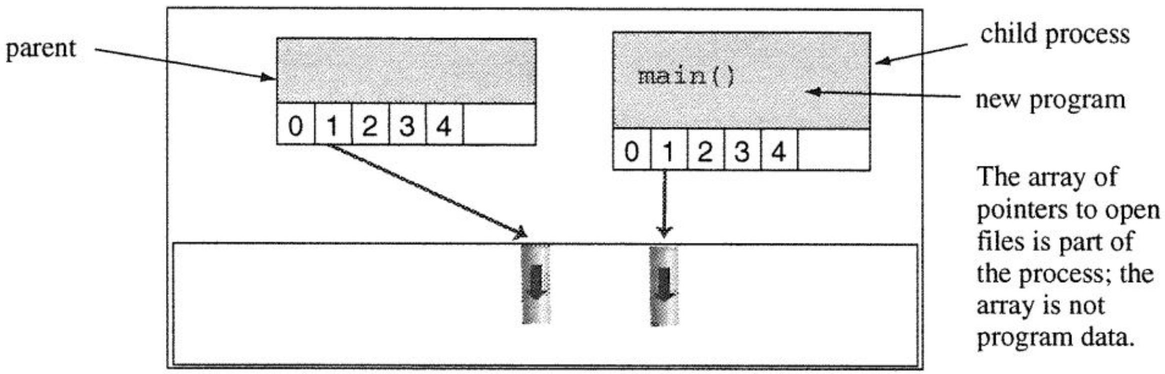
- Step 4. After child calls `creat("g",m)`: child opens a new file, taking `fd` = 1



< Child opens a new file, getting `fd = 1` >

Redirecting I/O for Another Program: `who > userlist` (cont.)

- Step 5. After child execs a new program (e.g., `who`)
  - child runs a program with the new standard output (e.g., `userlist`)



< Child runs a program with new standard output >

- The code and data for the shell are:
  - Removed from the child process
  - Replaced by the code and data for `who`
- But the file descriptors are retained across the `exec`
- Note that open files are not part of the code nor data of a program (here, `who`); they are attributes of a process

Redirecting I/O for Another Program: `who > userlist` (cont.)

- `whotofile.c`

```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>
#include        <stdlib.h>
#include        <fcntl.h>
#include        <sys/wait.h>

int 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");
        }

        return 0;
}
```

Redirecting I/O for Another Program: `who > userlist` (cont.)

- Execution

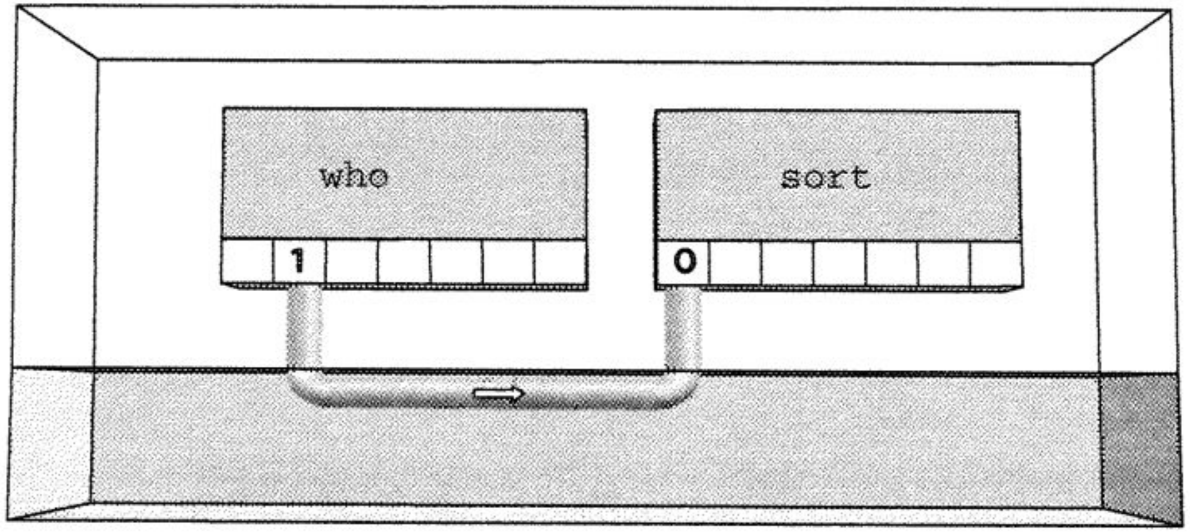
```
dynam@DESKTOP-Q4IJBP7:~/lab11$ make
cc -o whotofile whotofile.c
dynam@DESKTOP-Q4IJBP7:~/lab11$ ./whotofile
About to run who into a file
Done running who.  results in userlist
dynam@DESKTOP-Q4IJBP7:~/lab11$ cat userlist
dynam    pts/0        2022-11-15 14:56
dynam    pts/1        2022-11-15 15:11
dynam@DESKTOP-Q4IJBP7:~/lab11$ who
dynam    pts/0        2022-11-15 14:56
dynam    pts/1        2022-11-15 15:11
```

Summary of Redirection to Files

- (1) File descriptors 0, 1, and 2 represent standard input, output, and error, respectively
- (2) The kernel always uses the lowest numbered unused file descriptor
- (3) The set of file descriptors is passed unchanged across `exec` calls
  - To make I/O redirection to another program, the shell uses the interval in the child process between `fork` and `exec`
    - Reason: for the purpose of attaching the standard data streams to (external) files

What is a Pipe? How It Works?

- Pipe
  - "**One-way**" data channel in the kernel
  - Has a "writing" end and "reading" end
    - e.g., `who | sort`

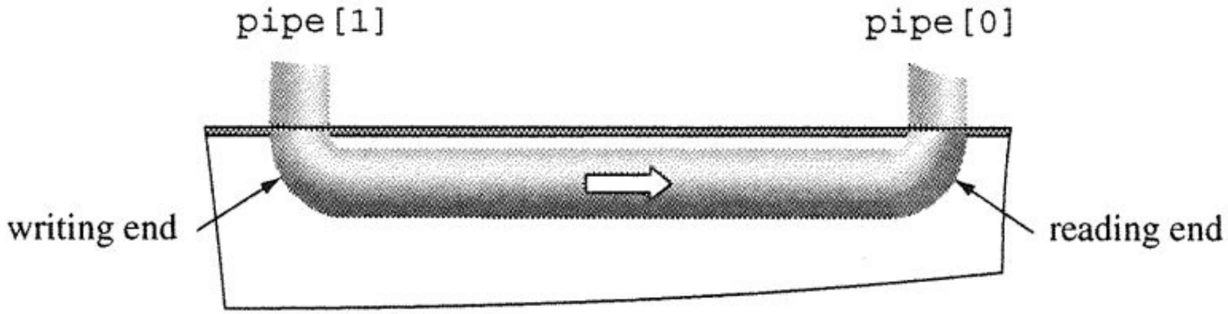


< Two processes connected by a pipe >

How to Create a Pipe? Use the System call, `pipe()`

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

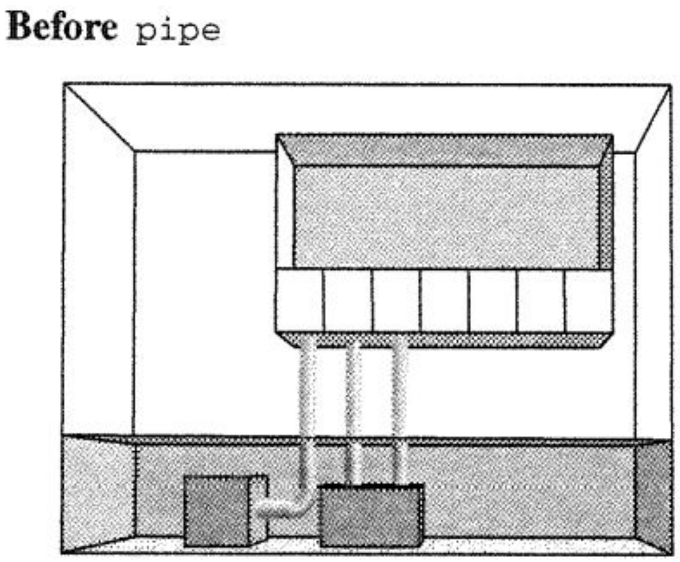
- `array[1]`: fd of the writing end
- `array[0]`: fd of the reading end



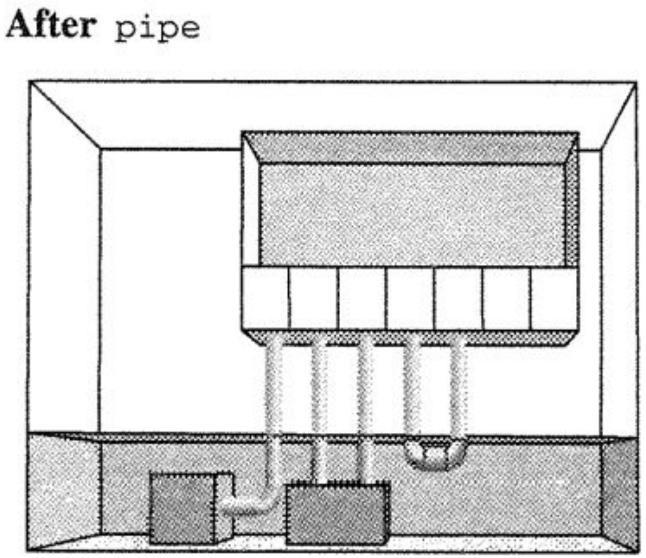
< A pipe >

Creating a Pipe

- Pipe creation by a process
    - Pipe uses the *lowest-numbered* available file descriptors: like `open()`



The process has some usual files open.



The kernel creates a pipe and sets file descriptors.

< A process creates a pipe >

Creating a Pipe (cont.)

- `pipedemo.c`

```c
/*  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>
#include        <stdlib.h>
#include        <fcntl.h>
#include        <string.h>
#include        <sys/wait.h>

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

```c
        /* 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 */
                }
        }

        return 0;
}
```

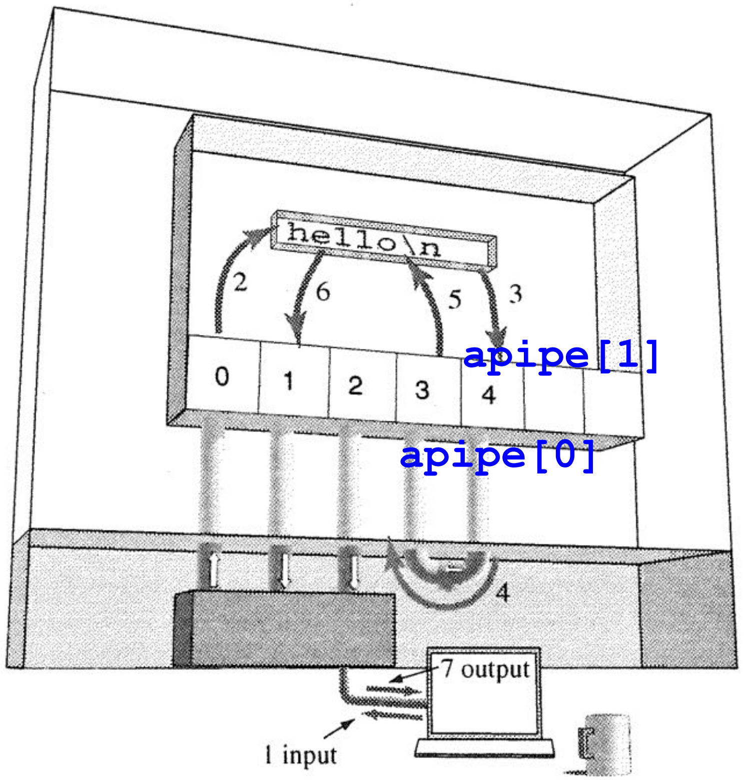
Creating a Pipe (cont.)

- Execution
    - Creates a pipe and then
    - Uses the pipe to send the data itself

```
dynam@DESKTOP-Q4IJBP7:~/lab11$ ./pipedemo
Got a pipe! It is file descriptors: { 3 4 }
hello
hello
^C
```

Creating a Pipe (cont.)

- Depicts the flow of bytes:
  - From keyboard to process: 1 ⟶ 2
  - From process to pipe: 3 ⟶ 4
  - From pipe to process, and: 5 ⟶ 6
  - From process back to terminal: 6 ⟶ 7

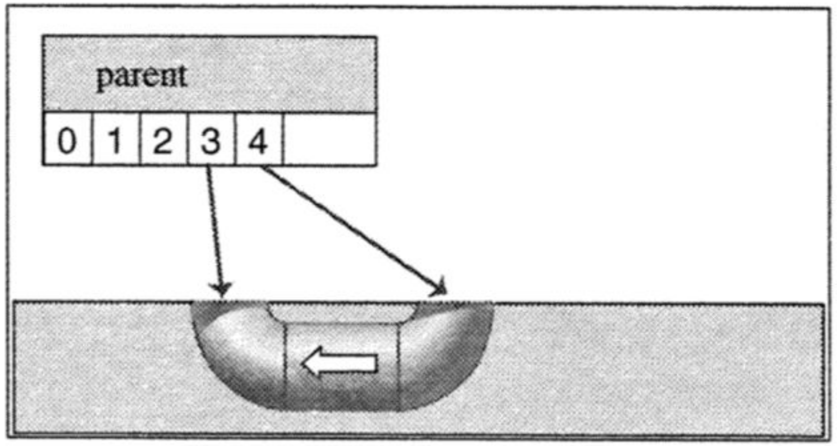


< Data flow in `pipedemo.c` >

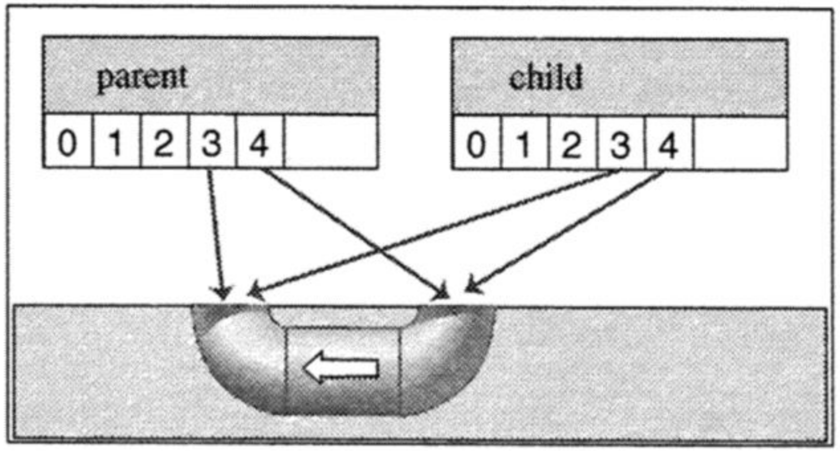
- Indeed, `pipe` and `fork` can be "combined" to connect two processes
- So the pipe can be *shared* between them

Using fork to Share a Pipe

- Sharing a pipe
  - A process calls `pipe`
    - The process has connections to both ends of the pipe
  - The kernel creates a pipe and adds to the array of file descriptors pointers to the ends of the pipe



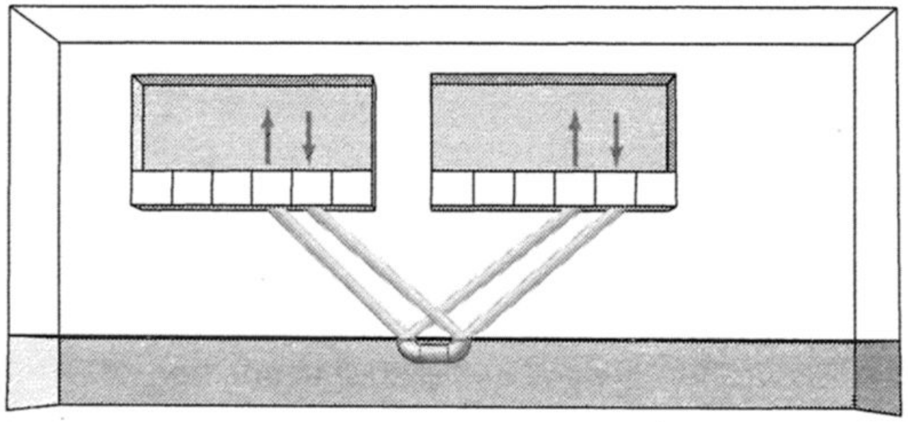
  - The process then calls `fork`
    - The child process also has connections to the pipe
  - The kernel creates a new process, and copies into that process the array of file descriptors from the parent



- Both processes now have access to both ends of one pipe!

Using fork to Share a Pipe (cont.)

- 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

Using fork to Share a Pipe (cont.)

- Shows how to combine pipe and fork
  - To create a pair of processes via pipe communication
- `pipedemo2.c`

```
/* 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>
#include        <string.h>
#include        <stdlib.h>
#include        <unistd.h>
#include        <fcntl.h>
#include        <sys/wait.h>

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

Using fork to Share a Pipe (cont.)

- `pipedemo2.c` (cont.)

```c
int 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 );
                        }
        }

        return 0;
}
```

Using fork to Share a Pipe (cont.)

- Execution

```
dynam@DESKTOP-Q4IJBP7:~/lab11$ ./pipedemo2
testing..
I want a cookie
testing..
testing..
testing..
testing..
I want a cookie
testing..
testing..
testing..
testing..
testing..
I want a cookie
testing..
testing..
testing..
^C
dynam@DESKTOP-Q4IJBP7:~/lab11$ 
```

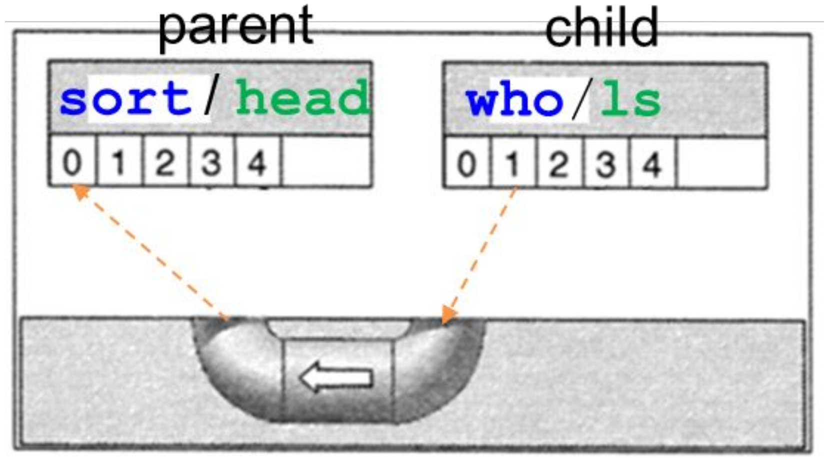
The Finale: Combining All Skills

- Let’s write a general-purpose program, called pipe.
  - It takes the names of two programs as arguments in the following:

```
pipe who sort

pipe ls head
```

  - The logic of the program as follows:

```
            pipe(p)
            fork()
              |
  +-----------+-------------+
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"
```



The Finale: Combining All Skills (cont.)

- pipe.c

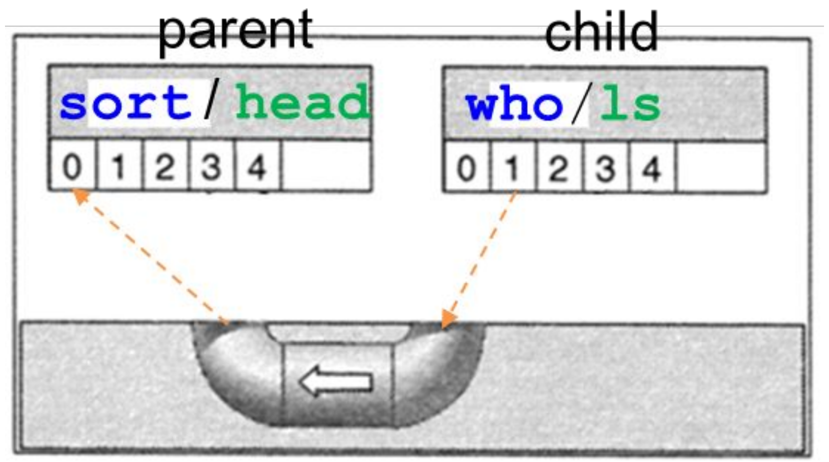
```
                pipe(p)
                fork()
                  |
   +------------+--------------+
child                          parent
  |                               |
  close(p[0])                     close(p[1])
  dup2(p[1],1)                    dup2(p[0],0)
  close(p[1])                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

```c
/* 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        <string.h>
#include        <stdlib.h>
#include        <fcntl.h>
#include        <unistd.h>
#include        <sys/wait.h>

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

int 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);
```

The Finale: Combining All Skills (cont.)

- `pipe.c` (cont.)

```
             pipe(p)
             fork()
               |
  +------------+-------------+
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"
```




```c
	/* ------------------------------------------------------------ */
	/*	now we have a pipe, now let's get two processes		*/

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

	/* ------------------------------------------------------------ */
	/*	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);

	return 0;
}
```

The Finale: Combining All Skills (cont.)

- Execution

```
dynam@DESKTOP-Q4IJBP7:~/lab11$ ./pipe ls sort
Makefile
listargs
listargs.c
pipe
pipe.c
pipedemo
pipedemo.c
pipedemo2
pipedemo2.c
sample.txt
sortfromfile
sortfromfile.c
stdinredir1
stdinredir1.c
stdinredir2
stdinredir2.c
stdinredir3.c
userlist
watch.sh
watch2.sh
whotofile
whotofile.c
whotofile2
whotofile2.c
```

Similarities between Pipes and Files

- Pipes look like regular files
  - Use `write()` to put data into a pipe
  - Use `read()` to get the data from a pipe
  - Appears as a sequence of bytes without any particular block or record

Differences between Pipes and Files

- Reading from Pipes
  - 1. `read` on a pipe blocks
    - When a process tries to `read` from a pipe, the call blocks until some bytes are written into the pipe
  - 2. Reading EOF on a pipe
    - When all writers close the writing end of the pipe, attempts to `read` from the pipe return 0, which means the end of file
  - 3. Multiple readers can cause trouble
    - A pipe is queue: first-in-first-out structure
    - When a process reads bytes from a pipe, those bytes (after reading) will be gone in the pipe
    - If two processes try to read from the same pipe, one process will get some of the bytes from the pipe, and the other process get the other bytes

Differences between Pipes and Files (cont.)

- Writing to Pipes
  - 4. `write` to a pipe blocks until there is space
    - Pipes have a **finite capacity**, far lower than the file-size limit on disk files
    - The `write` call to a pipe will get blocked until enough space is prepared
  - 5. `write` guarantees a minimum chunk size
    - The kernel will not split up chunks of data into blocks ***no smaller than 512 bytes***
    - Linux guarantees an ***unbroken buffer size of 4K bytes*** for pipes
  - 6. `write` fails if no readers
    - If all readers have closed the reading ends of pipe, then an attempt to `write` to the pipe can lead to trouble
    - Kernel’s two methods of notifying a process that `write` is no long valid:
      - 1) Sends `SIGPIPE` to that process, which will terminate
      - 2) If the kernel doesn’t kill the process, then `write` returns -1 and sets `errno` to `EPIPE`

Summary

- Input/output redirection allows separate programs to work as a team, each program a specialist
- Linux assumes that programs read input from fd 0 (`stdin`), write results to fd 1 (`stdout`), and report errors fd 2 (`stderr`)
- The log-in procedure sets up fds 0, 1, and 2
    - These connections and all open file descriptors are passed from parent to child and across the `exec` system call

Summary (cont.)

- System calls creating fds always use the lowest-number free fd
- Redirecting std input/output/error means changing where fds 0, 1, or 2 connect
- Pipe is a data queue in the kernel with each end attached to a fd
    - `pipe` system call can create a pipe
- Both ends of a pipe are copied to a child process when the parent calls `fork`
- Pipes can only connect processes sharing a common parent