Ch10-2. Pipes

Prof. Seokin Hong

Kyungpook National University

Fall 2019

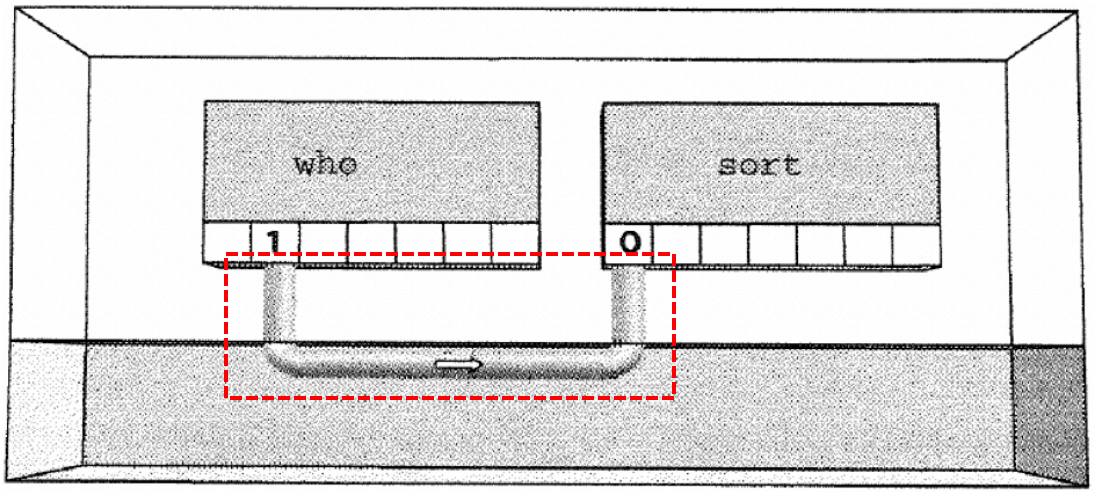
Contents

- Shell Programming
- A Shell Application : Watch for Users
- Facts about Standard I/O and Redirection
- How to Attach stdin to a File
- Redirecting I/O for Another Program: who > userlist
- 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.

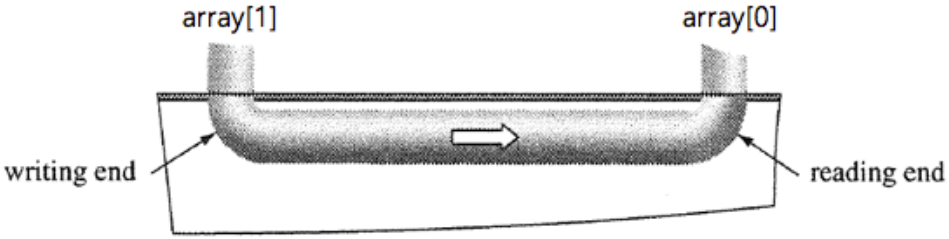


- How to create a pipe and how to connect stdin and stdout to a pipe?

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

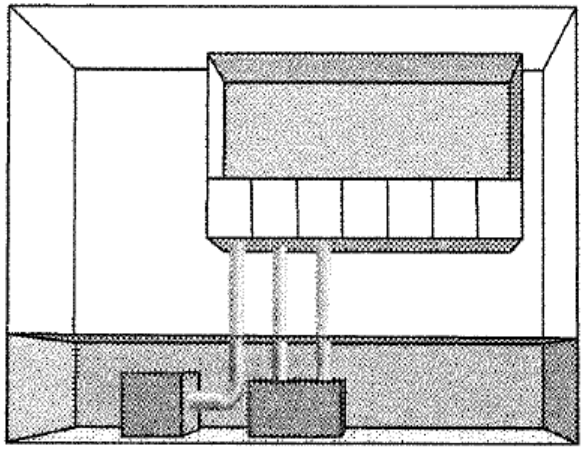


- array[0] is the file descriptor of the reading end
- array[1] is the file descriptor of the writing end

Creating a Pipe

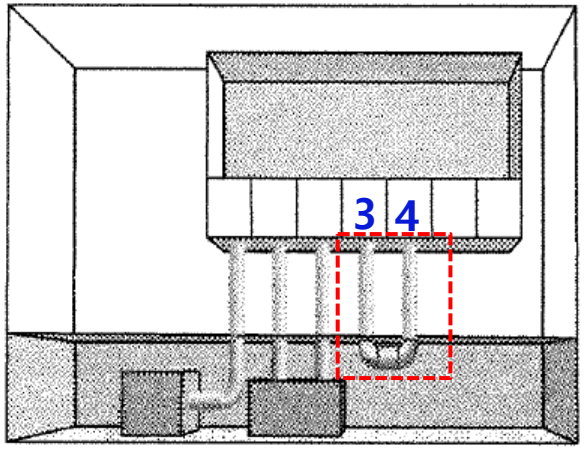
```
int len, I, apipe[2];
char buf[BUFSIZ];

pipe(apipe);
```

Before `pipe`



The process has some usual files open.

After `pipe`

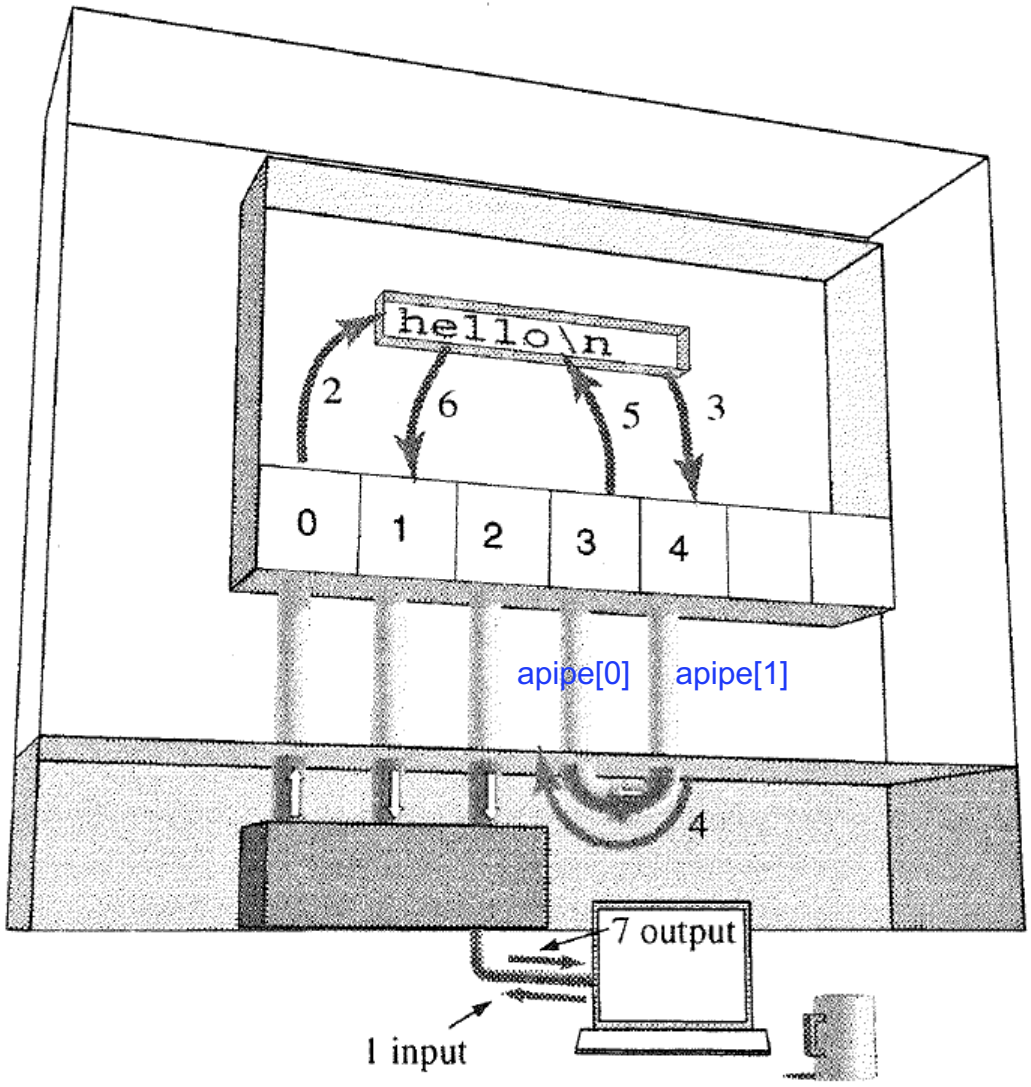


The kernel creates a pipe and sets file descriptors.

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

Ex1. pipedemo.c

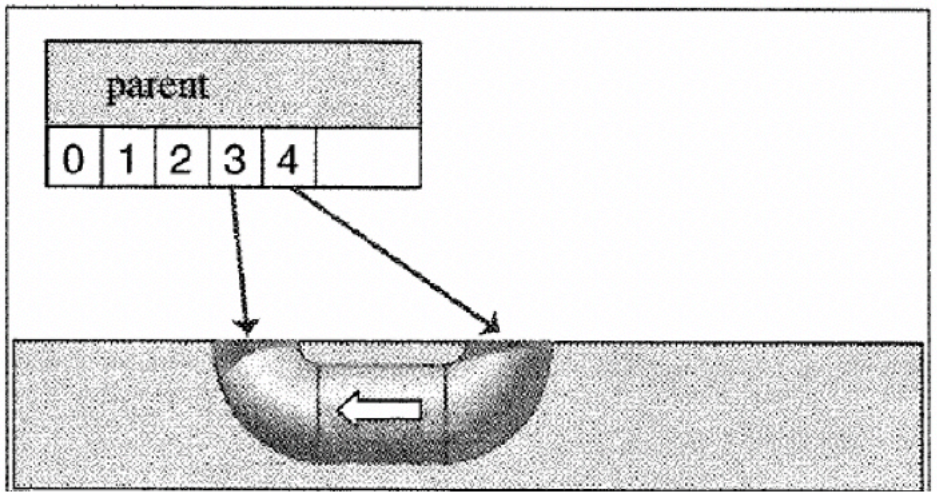
```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

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, printf */
    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;
        }
}
```

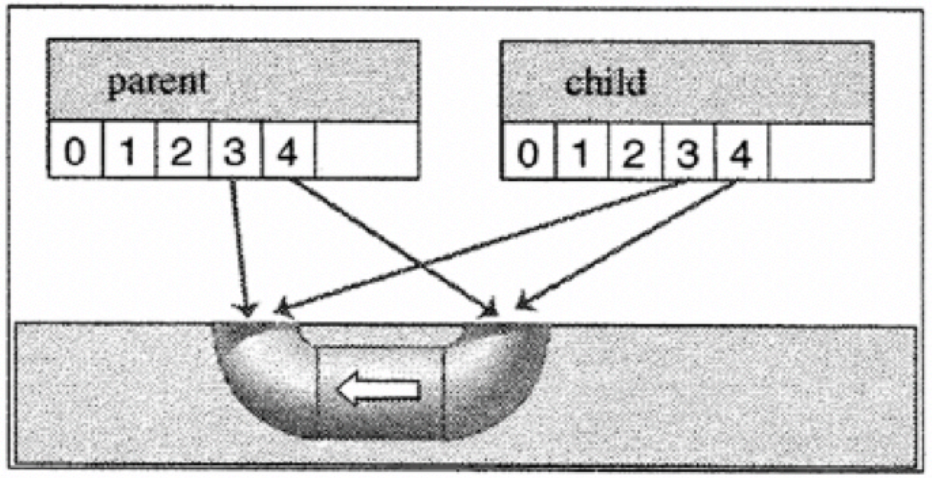
Using `fork` to Share a Pipe

A process calls pipe().
The kernel creates a pipe and stores the allocated file descriptors in the array.



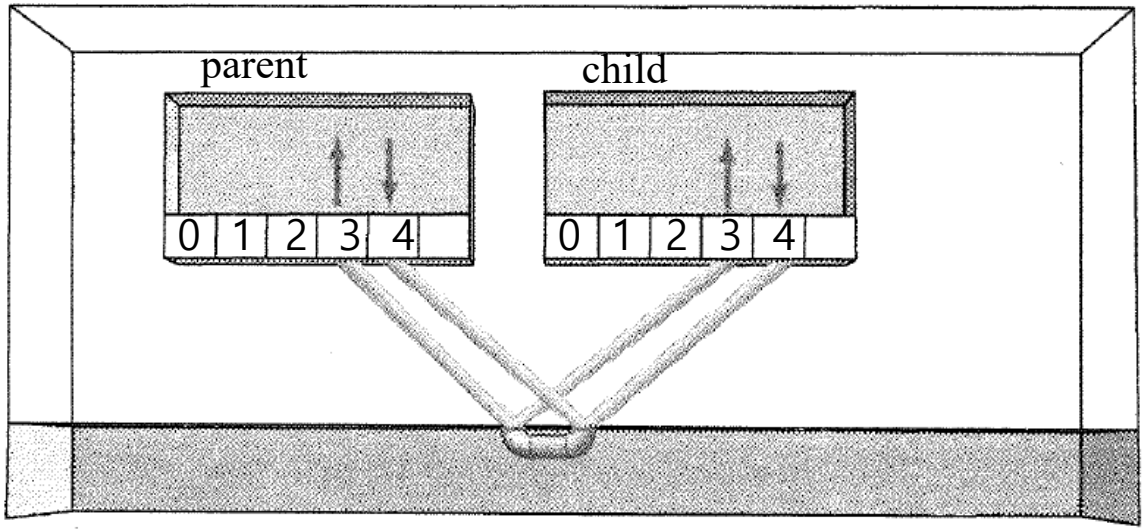
The process calls fork().
The kernel creates a new process and copies the array of file descriptors to the memory space of the new process.

Both processes have access to both ends of one pipe.



Using `fork` to Share 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 data into the pipe, the parent can read the data



Ex2. pipedemo2.c

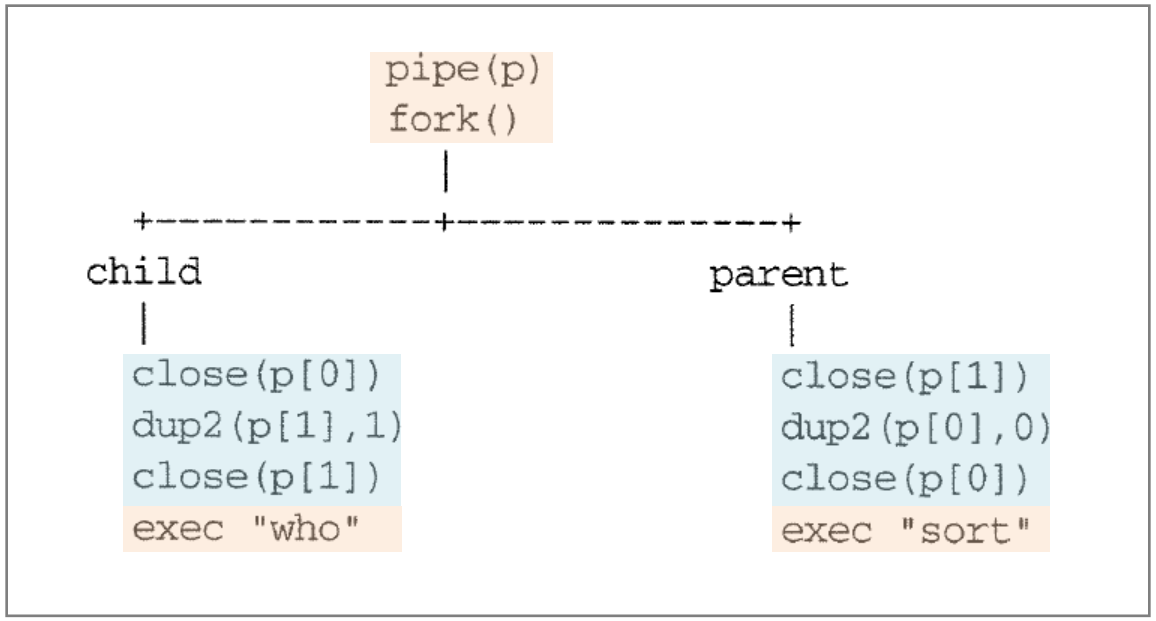
```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#define CHILD_MESS  "I want a cooke\n"
#define PAR_MESS    "tesing..\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);
            }
    }
}
```

Using `pipe`, `fork`, and `exec`

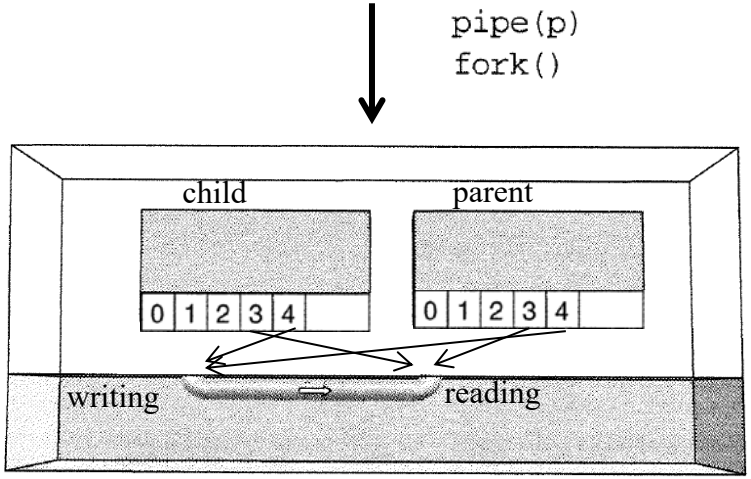
- Writing a general-purpose program `pipe`:

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

- Logic

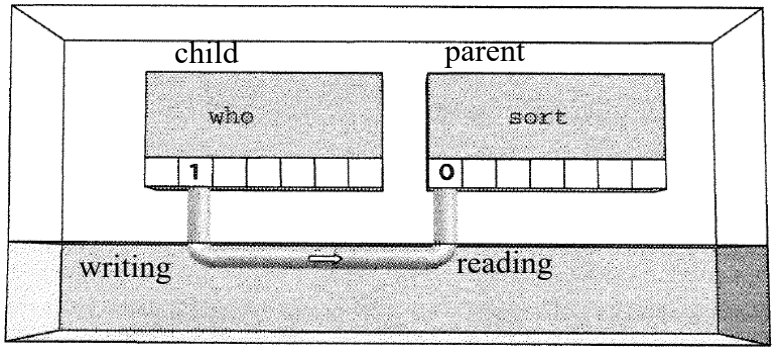


Using `pipe`, `fork`, and `exec`



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"
```



Ex3. pipe.c

```c
#include    <stdio.h>
#include    <unistd.h>
#include    <stdlib.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);

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

Ex3. pipe.c

```
[seokin@compasslab2 ch10]$ ./pipe ls head
ch10.zip
listargs
listargs.c
oops
pipe
pipe.c
pipedemo
pipedemo2
pipedemo2.c
pipedemo.c
```