

CS241 System Programming

IPC – Pipes (V)

Klara Nahrstedt

Lecture 38

4/24/2006



Contents

- Introduction to Pipes
- Fan Example
- Pipelines
- Pipes in Client-Server Communication

Administrative

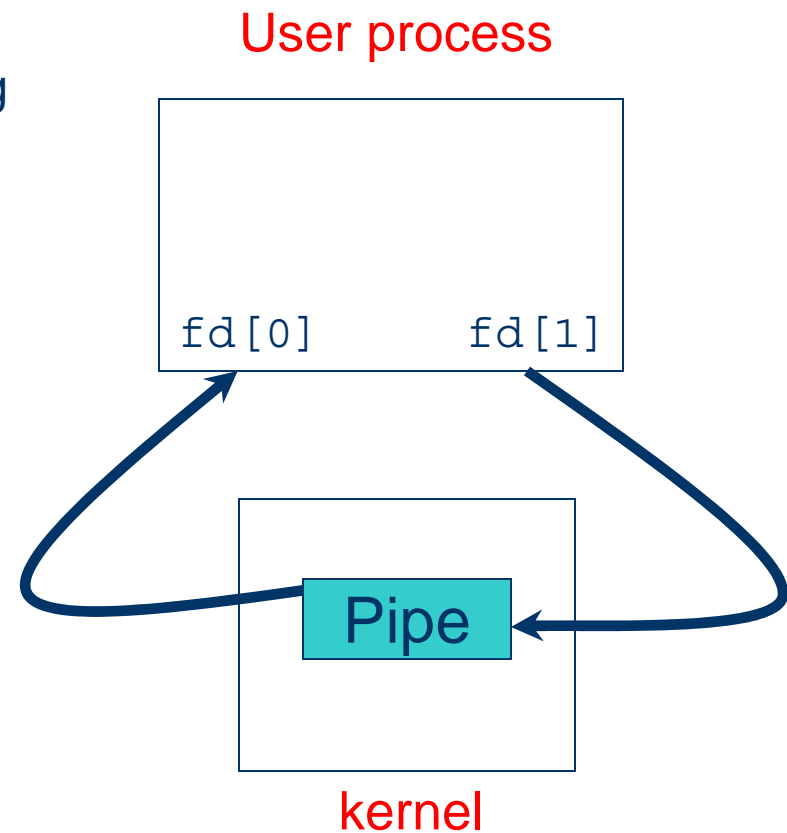
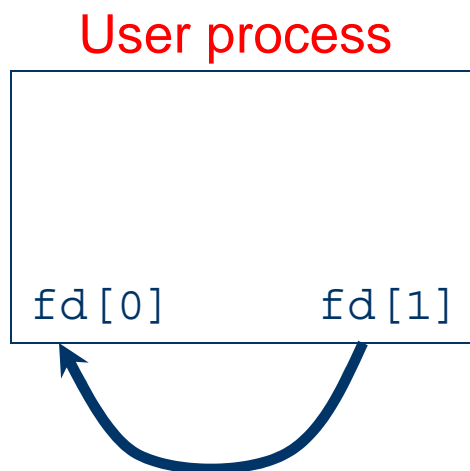
- Read R&R Chapter 6.1-6.4
- MP5 is on – deadline May 1
- Homework 2 – posted April 24 – deadline May 3, midnight
- Quiz 11 – April 28
 - Cover R&R Chapter 18.1-18.7
 - Cover R&R Chapter 20.1-20.8
- **Quiz 12 – May 3 → OPTIONAL !!!**
 - Cover Processes/Synchronization – T: Chapter 2.1-2.4
 - Cover Memory Management – T: 4.1 – 4.5.4
 - Cover File Systems – T: 5.1-5.3

Introduction to Pipes

- Pipes are the oldest form of UNIX IPC
- Pipes have two limitations:
 - They are half-duplex, i.e., data flows only one direction – one-way communication channel!!
 - They can be used only between processes that have a common ancestor
 - Normally a pipe is created by a process, that process calls fork, and the pipe is used between the parent and child
- Pipe is created by calling
 - `#include <unistd.h>`
 - `int pipe(int fildes[2]);`

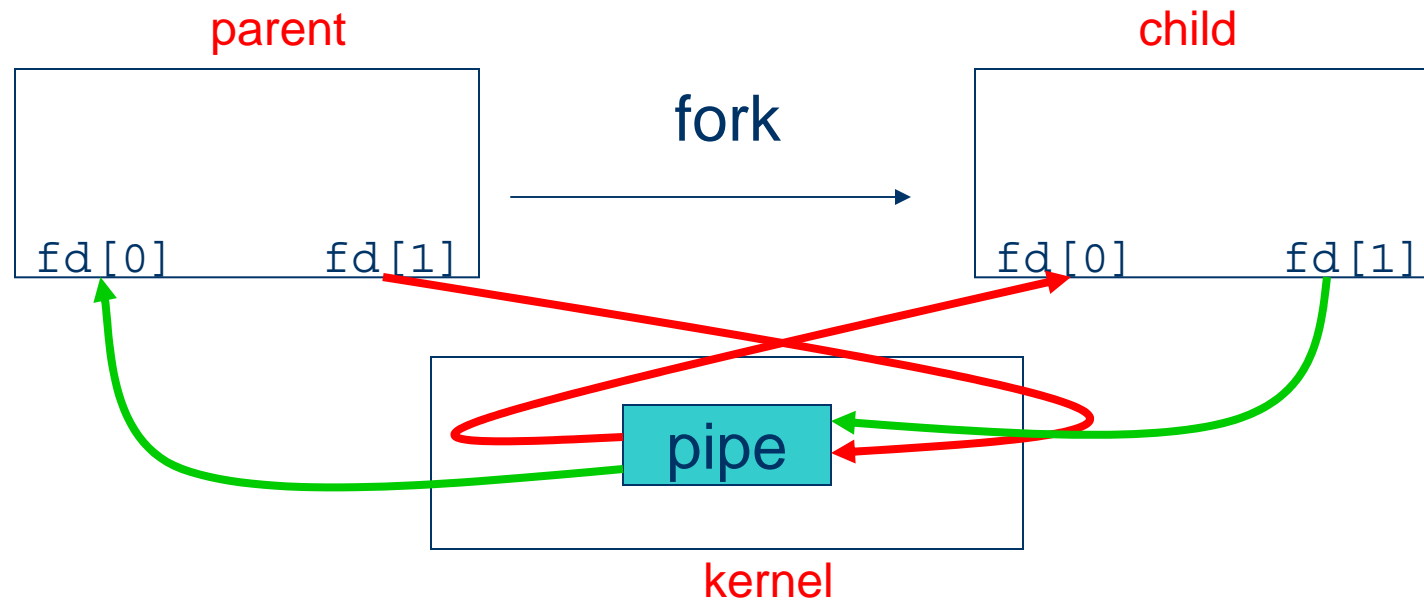
Pipe File Descriptors

- `pipe` command returns two descriptors through the `filedes` argument:
 - `filedes[0]` is open for reading
 - `filedes[1]` is open for writing
- Two ways to view a UNIX pipe:



Pipe Usage

- A pipe in a single process is next to useless.
- Process that calls `pipe` then calls `fork`, creating IPC channel from the parent to the child or vice versa



Simple Program of Using Pipe between a Parent and Child (Program 6.1)

```
.../* define h files and variables - see Program 6.1*/
int main(void) {
    char bufin[BUFSIZE] = "empty";
    char bufout[] = "hello";
    int bytesin;
    pid_t childpid;
    int fd[2];

    if (pipe(fd) == -1) { perror("failed to create the pipe"); return 1; }
    bytesin = strlen(bufin);
    childpid = fork();
    if (childpid == -1) { perror("failed to fork"); return 1; }
    if (childpid) write(fd[1], bufout, strlen(bufout)+1); /*parent*/
    else bytesin = read(fd[0], bufin, BUFSIZE); /*child*/
    .../*finish the code*/
```

Using a pipe for synchronization

- Create a **pipe** from parent to children
- Send a synchronization character to children
- Have children wait for character

Fan (Program 6.2)

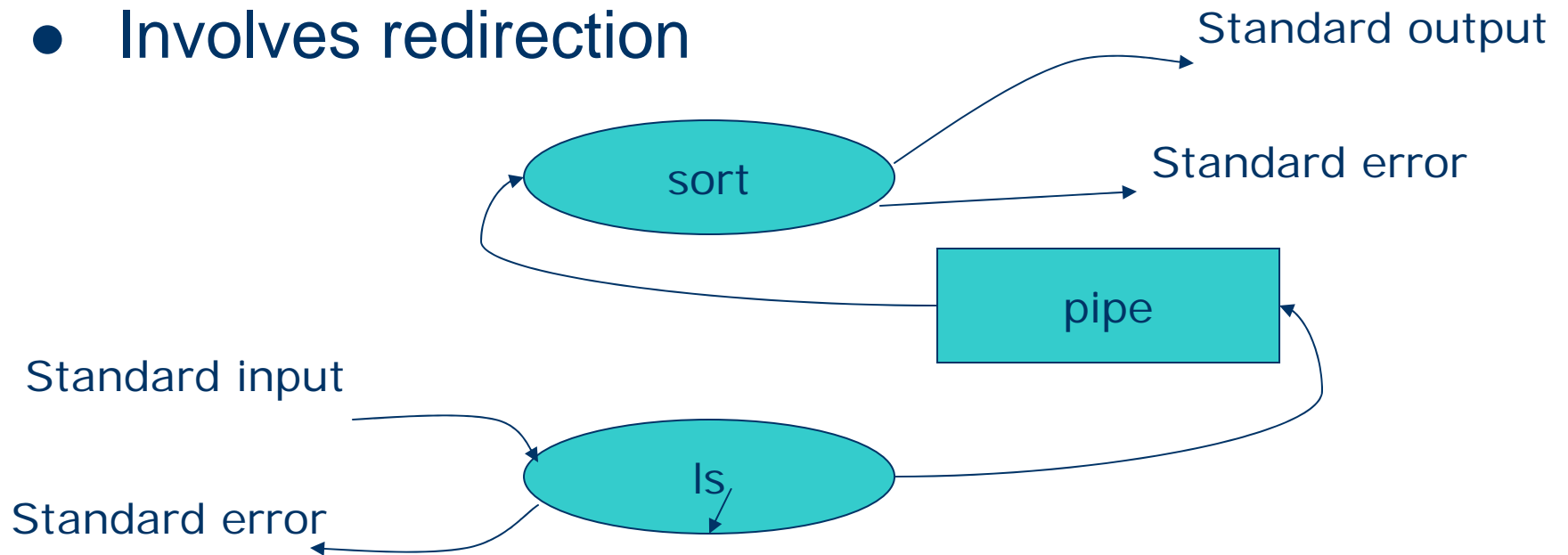
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "restart.h"
int main (int argc, char *argv[]) {
    char buf[] = "g";
    pid_t childpid = 0;
    int fd[2];
    int i, n;
    if (argc != 2){ /* check for valid number of command-line arguments */
        fprintf (stderr, "Usage: %s processes\n", argv[0]);
        return 1;}
    n = atoi(argv[1]);
    if (pipe(fd) == -1) { /* create pipe for synchronization */
        perror("Failed to create the synchronization pipe");
        return 1;
    }
}
```

Fan

```
for (i = 1; i < n; i++)          /* parent creates all children */
    if ((childpid = fork()) <= 0)
        break;
if (childpid > 0) {              /* write synchronization characters to pipe */
    for (i = 0; i < n; i++)
        if (r_write(fd[1], buf, 1) != 1)
            perror("Failed to write synchronization characters");
}
if (r_read(fd[0], buf, 1) != 1) /* synchronize here */
    perror("Failed to read synchronization characters");
fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",
        i, (long)getpid(), (long)getppid(), (long)childpid);
return (childpid == -1);
}
```

Pipelines

- `ls -l | sort -n +4`
- Sort on numeric character in 4th place
- Involves redirection



Simple Redirect (Program 6.3)

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
    pid_t childpid;
    int fd[2];

    if ((pipe(fd) == -1) || ((childpid = fork()) == -1)) {
        perror("Failed to setup pipeline");
        return 1;
    }
}
```

Simple Redirect

```
if (childpid == 0) {                                /* ls is the child */
    if (dup2(fd[1], STDOUT_FILENO) == -1)
        perror("Failed to redirect stdout of ls");
    else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
        perror("Failed to close extra pipe descriptors on ls");
    else {
        execl("/bin/ls", "ls", "-l", NULL);
        perror("Failed to exec ls");
    }
    return 1;
}
```

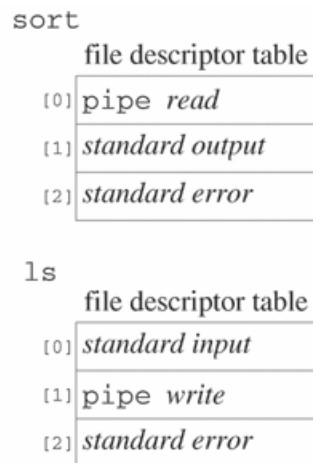
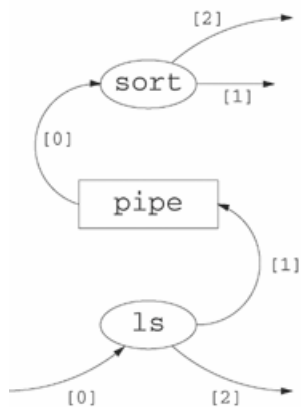
Simple Redirect

```
if (dup2(fd[0], STDIN_FILENO) == -1)          /* sort is the parent */
    perror("Failed to redirect stdin of sort");
else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
    perror("Failed to close extra pipe file descriptors on sort");
else {
    execl("/bin/sort", "sort", "-n", "+4", NULL);
    perror("Failed to exec sort");
}
return 1;
}
```

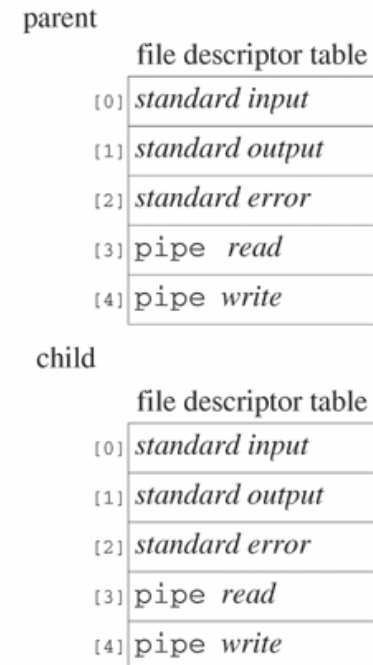
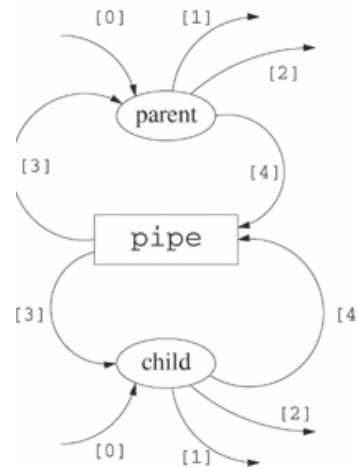
Why do we close the pipes in Prog 6.3?

Figures 6.2 and 6.3

Closed



Not Closed



If we don't, the "sort" process never detects EOF.

FIFOs

- Pipes disappear when no process has them open.
- FIFOs are named pipes that are special files that persist even after all the processes have closed them.

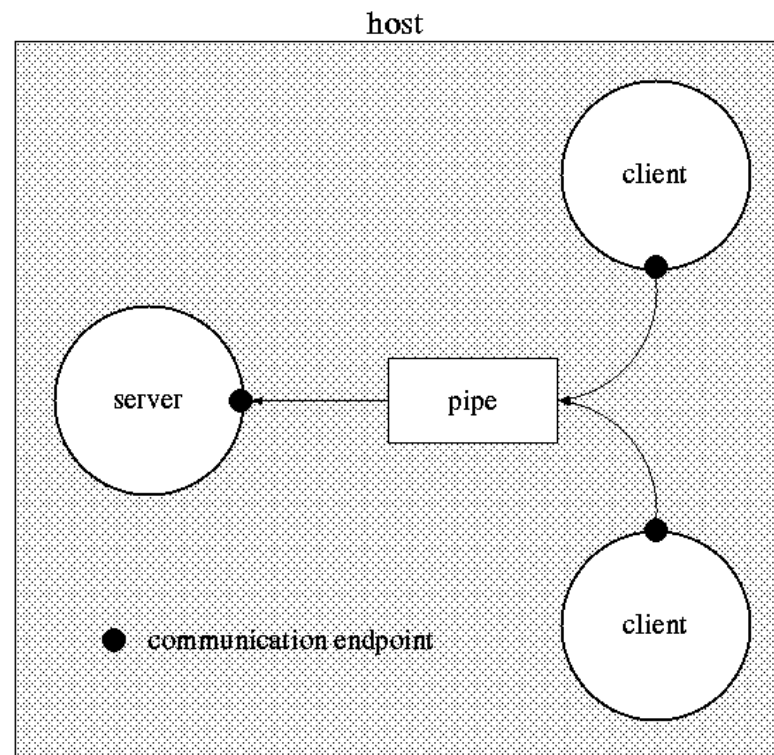
Mkfifo

```
#include <sys/stat.h>
```

```
int mkfifo(const char *path, mode_t mode);
```

Mode is for access control

Pipes and Client-Server Model



Example Client-Server

- Simple Request Model Client Server
- Client writes to fifo
- Server reads from fifo and outputs data to stdout
- Fifo ensures atomicity of write so that multiple clients can send data to file

Pipe-server (Program 6.7)

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include "restart.h"
#define FIFOARG 1
#define FIFO_PERMS (S_IRWXU | S_IWGRP | S_IWOTH)
int main (int argc, char *argv[]) {
    int requestfd;

    if (argc != 2) { /* name of server fifo is passed on the command line
        fprintf(stderr, "Usage: %s fifoname > logfile\n", argv[0]);
        return 1;
    }
```

Pipe-server (2)

```
/* create a named pipe to handle incoming requests */
if ((mkfifo(argv[FIFOARG], FIFO_PERMS) == -1)&&(errno !=
EEXIST)) {
    perror("Server failed to create a FIFO");
    return 1;
}

/* open a read/write communication endpoint to the pipe */
if ((requestfd = open(argv[FIFOARG], O_RDWR)) == -1) {
    perror("Server failed to open its FIFO");
    return 1;
}
copyfile(requestfd, STDOUT_FILENO);
return 1;
}
```

Pipe-client (Program 6.8)

```
.... /*declare h files*/
.... /*start main program */
/*name of server fifo is passed on the command line*/
if (argc !=2) {
    fprintf(stderr,"Usage: %s fifoname ", argv[0]);
    return 1; }
if ((requestfd = open(argv[FIFOARG], O_WRONLY)) == -1) {
    perror("Client failed to open log fifo for writing");
    return 1; }
curtime = time(NULL);
snprintf(requestbuf, PIPE_BUF, "%d: %s", (int)getpid(),ctime(&curtime));
...
If (r_write(requestfd, requestbuf, len) != len) {
....}
```

Summary

- Pipes
- Pipelines
- FIFOs
- Client Server