

# CSCI 210 Systems Programming

Week 12

## Pipes and Named Pipes (FIFOs)

The Linux Programming Interface (Ch. 44)

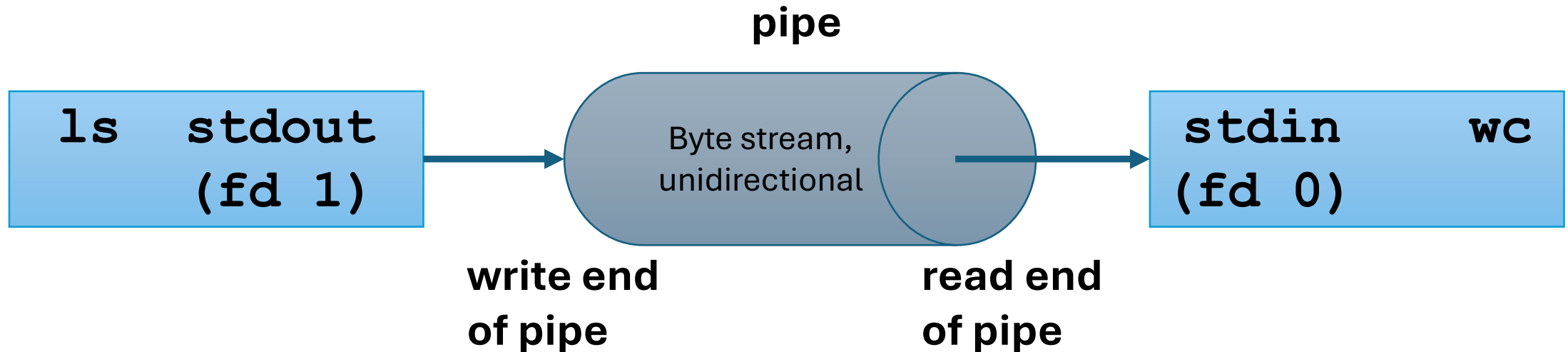
Systems Programming in Unix/Linux (3.10)

# Overview

- Pipes
  - Creating pipes with the pipe() system call
  - An example program that creates a pipe between two “related” processes
- Named Pipes (FIFOs)
  - Creating named pipes with mkfifo or mknod commands
  - Example programs that use named pipes for inter process communication
  - A more complex client-server application

# Pipes

- How does the shell execute the following:
  - `ls | wc -l`
  - By creating a “pipe” between the two processes



# Important characteristics of pipes - I

- A pipe is a byte stream
  - Meaning there are not well-defined message boundaries
  - A reader process makes a request to read any number of bytes
  - Bytes are written and read sequentially (hence the term: “stream”) - no random access.

# Important characteristics of pipes - II

- Reading from a pipe
  - If a process tries to read from an empty pipe, it is blocked until at least one byte is available to read
    - If there are no writer processes to the pipe, i.e., if the write end of the pipe is closed, then the read will receive “end-of-file”, i.e., it will not be blocked
      - End-of-file -> read() will return zero to the reader process

# Important characteristics of pipes - III

- Pipes are unidirectional
  - Although there are exceptions (such as stream pipes), we are going to assume that data can travel in only in one direction through a pipe.
- If we want bidirectional communication, we can set up two different pipes and switch the role of the writer and reader in one of the pipes

# Important characteristics of pipes - IV

- Pipes have limited capacities
  - They are managed by the Kernel in memory
  - In newer Linux systems, their size can be 64KBs (after Linux 2.6.11). In older ones, it is equal to the size of single page which is 4KBs.
- When the pipe is full, the writer process is blocked, until bytes can be freed by a reader process who reads bytes from the reading end.
  - Which means that capacity is not something we should be concerned with
- If the read end of the pipe is closed, i.e. no readers, the process receives the SIGPIPE signal, i.e., broken pipe, and terminates.

# Important characteristics of pipes - V

- There can be multiple writer process and multiple reader processes
  - When there are multiple processes, each write of up to PIPE\_BUF number of bytes is guaranteed to be atomic, i.e., the bytes will not be interleaved with the bytes written by another process
    - PIPE\_BUF is 4096 bytes in Linux



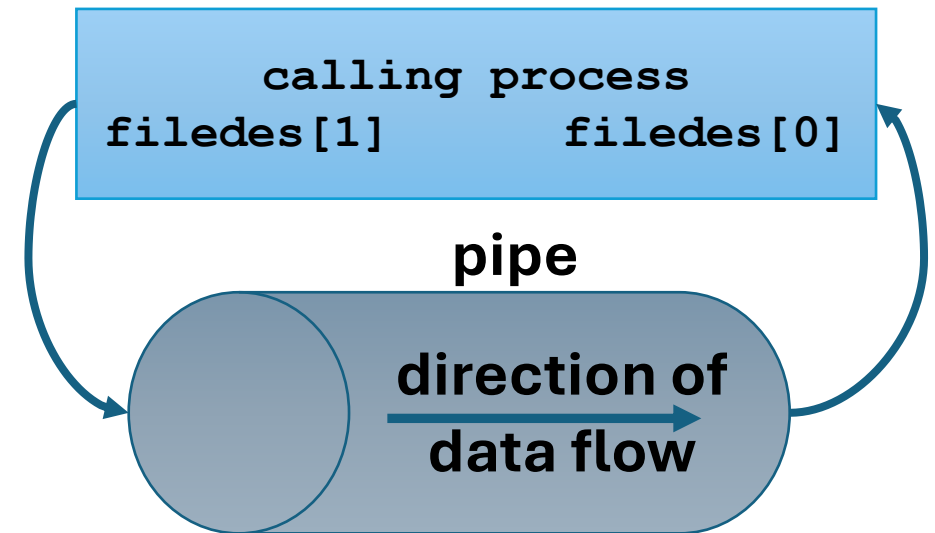
# Creating and Using Pipes - I

- The pipe() system call:

```
#include <unistd.h>
```

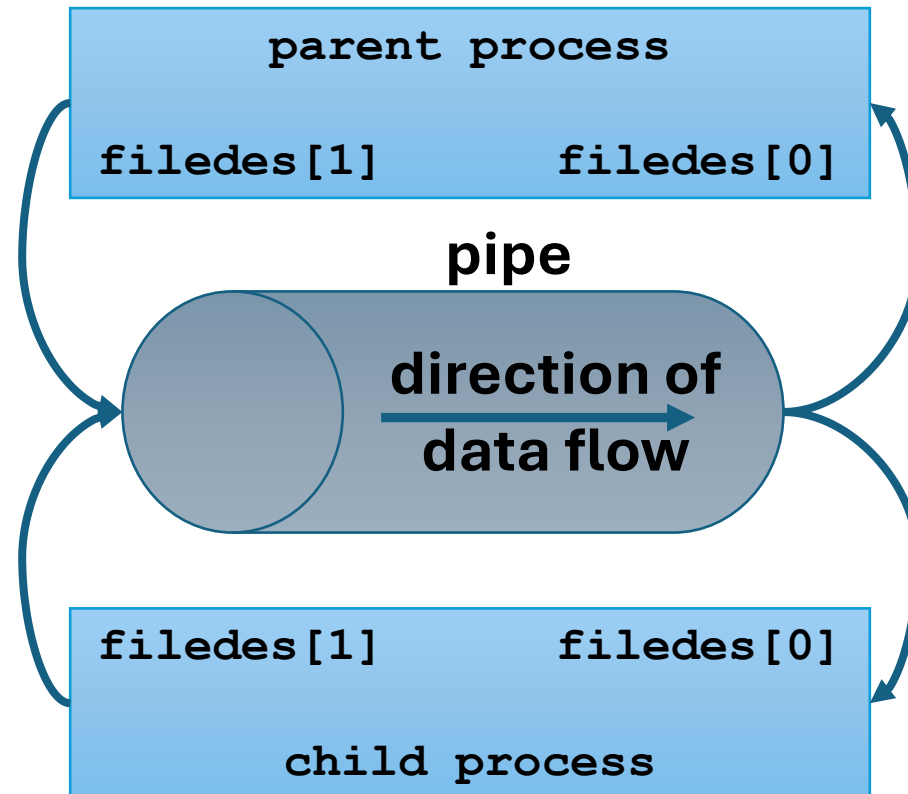
```
int pipe(int filedes[2]);
```

- Returns 0 on success, or -1 on error



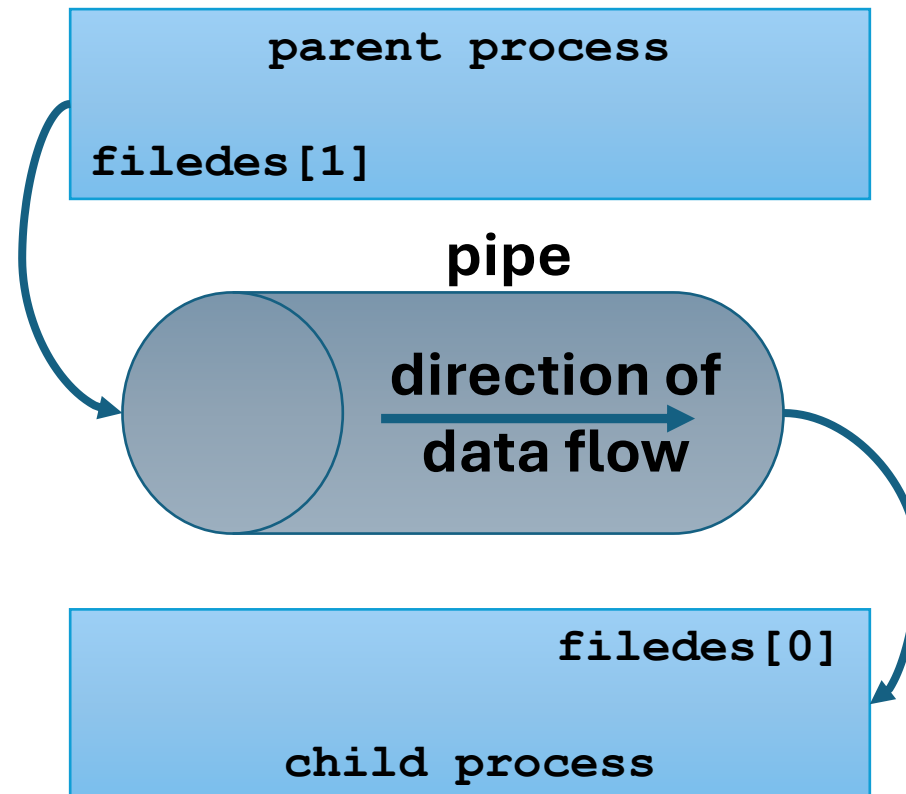
# Creating and Using Pipes - II

- The pipe is designed to be used by two different processes
- Therefore, it is usually followed by a `fork()` (and `exec()` if needed)



# Creating and Using Pipes - III

- We need to close the unused descriptors, to finish setting up the pipe between the parent and the child process



# Example: pipes.c

```
int filedес[2];
int forkReturn;
pipe(filedes);
forkReturn = fork();
if (forkReturn > 0) { /* parent process */
    close(filedes[0]);
    write(filedes[1], "Hello, from your parent!\n", 26);
    wait(NULL);
    exit(0);
}
else if (forkReturn == 0) { /* child process */
    close(filedes[1]);
    char buf[100] = {};
    read(filedes[0], buf, 24);
    printf("Message from parent: %s\n", buf);
    exit(0);
}
```

# How does the shell do it?

- In other words, how does one connect the stdout of the first process to the stdin of the second process?
  - By replacing the file descriptors 0 and 1 with the new file descriptors created by the pipe() call
  - Use the dup() system call for this purpose
    - The dup(fd) system call copies the file descriptor fd to the lowest available file descriptor. So, if we close 0, and immediately call dup(fd), the descriptor will be copied onto the standard input 0.

# Example: commandpipe.c

```
int filedес[2];
int forkReturn;
pipe(filedес);
forkReturn = fork();
if (forkReturn > 0) { /* parent process */
    close(filedес[0]);
    close(1);
    dup(filedес[1]);
    close(filedес[1]);
    execlp("ls", "ls", NULL);
}
else if (forkReturn == 0) { /* child process */
    close(filedес[1]);
    close(0);
    dup(filedес[0]);
    close(filedес[0]);
    execlp("wc", "wc", "-l", NULL);
}
```

# FIFOs – a.k.a. named pipes

- The pipe mechanism works between related processes - the file descriptors created by a parent are shared with the child (and descendant) processes.
- It is possible to establish communication between arbitrary processes by using “named pipes”
  - Named pipes are special files
  - Like pipes, they allow processes trying to read from them be “blocked” until data is written to the file by a writer process
    - In regular files, if you try to read from an empty file, you will immediately get an end of file (EOF).

# Creating named pipes

- `mkfifo [ -m mode ] pathname`
- mode is simply the access rights you want the named pipe to have as in a `chmod` command.
- Example
  - Create a named pipe and examine it using `ls -l`.
  - Do not try to edit it `vi` as a regular file – see what happens when you try to do that 😊
- `mknod` is a more generic command that can create other block special files. `mkfifo` uses `mknod` at the backend.

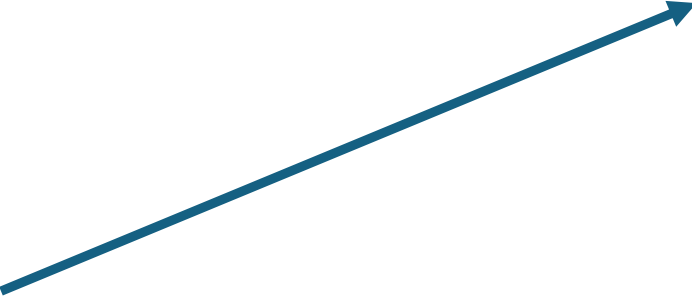


# Example: A reader process

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd;
    int n;
    char buf[500];
    fd = open("namedpipe", O_RDONLY);
    n=read(fd,buf,499);
    while (n!=0) {
        buf[n]='\0';
        printf("Got this: %s\n",buf);
        n=read(fd,buf,499);
    }
    return 0;
}
```

Try to open a regular file instead  
and see what happened when you read  
from it



# Example: A writer process

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

int main() {
    int fd;
    char buf[500]={};
    fd = open("namedpipe",O_WRONLY);
    printf("Enter something:");
    while (scanf("%s",buf)!=EOF) {
        write(fd,buf,strlen(buf));
        printf("Enter something:");
    }
    close(fd);
    return 0;
}
```

**Try to execute this one before the reader process. Does it get blocked?**

# Opening a pipe

- The pipe should be open on both ends before you can proceed to do any input or output operations on it.
- The `open()` system call will be blocked by process that tries to open it as read only, until another process opens the pipe as write only and vice versa
- From the manual page (man 3 mkfifo):
  - One you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.

# A Client-Server Application Using FIFOs

- The server: provide unique integer intervals to clients
- Clients: request integer intervals by specifying the size of the interval only
- Example run:
  - Client 1: I want 3 integers
  - Server: You can get 3 starting from 0, i.e., 0, 1, 2
  - Client 2: I want 5 integers
  - Server: You can get 5 starting from 3, i.e., 3, 4, 5, 6, 7
  - Client 1: I want 2 integers
  - Server: You can get 2 starting from 8, i.e., 8, 9

# The Architecture

- A single server FIFO to accept requests
- A separate FIFO for each client to respond to requests



# The message format

- We can use fixed sized messages to be used as responses and requests
  - Other approaches:
    - Using delimiters to identify message ends
    - Use fixed sized headers to specify info about the message (length, etc)

```
struct request {  
    char name[50]; // name of client FIFO  
    int intervalLen;  
};  
struct response {  
    int intervalStart;  
};
```

# server.c

```
server = open("serverFIFO", O_RDONLY);
int curr = 0;
while (1) {
    if (read(server, &req, sizeof(struct request)) !=
        sizeof(struct request))
        continue;
    struct response r;
    client = open(req.name, O_WRONLY);
    r.intervalStart = curr;
    curr += req.intervalLen;
    write(client, &r, sizeof(struct response));
    close(client);
}
close(server);
```

# client.c

```
server = open("serverFIFO", O_WRONLY);
printf("Make a request (enter -1 to end):");
scanf("%d", &len);
while (len != -1) {
    struct request req;
    struct response r;
    strcpy(req.name, argv[1]);
    req.intervalLen = len;
    write(server, &req, sizeof(struct request));
    client = open(argv[1], O_RDONLY);
    read(client, &r, sizeof(struct response));
    printf("Make a request (enter -1 to end):");
    scanf("%d", &len);
    close(client);
}
close(server);
```