# Pipes and FIFOs

**COMP 321**

**Dave Johnson**



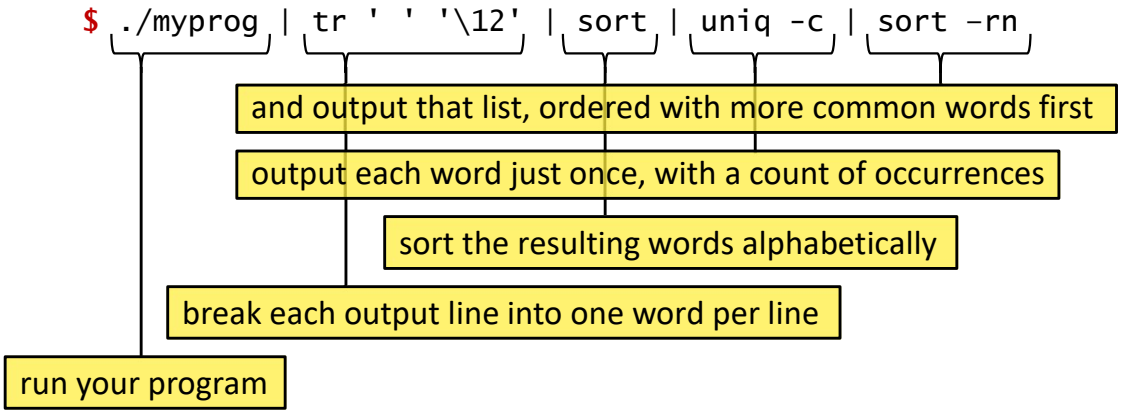



1

## Pipes are Widely Used in Unix and Similar Systems

***You can do a lot of interesting things connecting programs with pipes***

- An example – what does this shell command do?








2

# Pipes are Widely Used in Unix and Similar Systems

***You can do a lot of interesting things connecting programs with pipes***

- An example of what you can do with a pipe in a program

```
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- popen() is like fopen() but opens a shell command as a "FILE *" stream
  - type must be "r" or "w"
  - "r": Can read from the "FILE *" to get the command's standard output
  - "w": Can write to the *FILE *" to provide the command's standard input
- ***The calling program is connected to the command process using a pipe***
- Important: this "FILE *" must be closed with pclose(), not fclose()





# Creating a Process Pipeline

```
int fds[2];          /* declare an array to pass to pipe() kernel call */
int pipe(fds);
```

***A pipe provides a <u>unidirectional</u> byte stream channel***

- The pipe() kernel call creates a new "pipe" and returns 2 file descriptors
  - fds[***1***] is a file descriptor number open O_WRONLY to the new pipe (the "***write end***" of the pipe)
  - fds[***0***] is a file descriptor number open O_RDONLY to the same pipe (the "***read end***" of the pipe)

## Example – Running "`ls | wc -l`"

```
int main(void)
{
    int fds[2];
    int pid1, pid2;

    pipe(fds);
    if ((pid1 = fork()) == 0) {
        dup2(fds[1], 1);
        . . .
        execl("/usr/bin/ls", "ls", NULL);
    }

    If ((pid2 = fork()) == 0) {
        dup2(fds[0], 0);
        . . .
        execl("/usr/bin/wc", "wc", "-l", NULL);
    }
    . . .
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
    exit(0);
}
```

- The first child will exec "ls" to write into the pipe as its standard output
- The second child will exec "wc -l" to read from the pipe as its standard input





## Kernel Buffering Inside a Pipe

***Bytes written and not yet read are buffered "inside" the pipe in the kernel***

- (Remember the Bounded-Buffer Problem when we talked about threads)
- The amount of buffering inside the pipe is limited
- Doing a write() into the pipe adds bytes into that buffering
  - The default buffering capacity is, e.g., 16 pages = 16 × 4 kB = 64 kB, but it may be different
  - If the pipe buffering is full, any new write() is blocked until space in the pipe is available
- Doing read() from the pipe removes bytes from that buffering
  - If the pipe is empty, any new read() is blocked until data is available to complete the read

## End of File on Reading From a Pipe

***When does end of file occur on a read() from a pipe?***

- Suppose the first process has written 100 bytes into the pipe . . .
- . . . and the second process has read 100 bytes from the pipe
- The pipe is now empty (no buffered data), but another read from the pipe now ***should not*** be treated as end of file on the pipe
  - The other process might (sometime) still write more data into the pipe
  - The two processes run asynchronously, so sometimes the ordering of reads vs. writes may be different
- Reading from an empty pipe is ***only*** treated as end of file when
  - You read and the pipe is empty, ***and***
  - No file descriptor is open ***anywhere*** that can write into the pipe
- The read then returns 0 = the number of bytes read





## Special Rules for Writing Into a Pipe

***What about writing to a pipe when no one remains to possibly read it?***

- If no file descriptor is open ***anywhere*** that can read from the pipe
  - Any further writes to the pipe are useless
  - Creating a pipe returns one read and one write file descriptor
  - But once all read file descriptors (and dups of them) are closed, there is no way to make any new read fds
- So any future attempted write to the pipe is then an error
  - Causes a SIGPIPE signal in the writing process (default action is to terminate the process)
  - But if SIGPIPE is ignored, or if it is caught and the signal handler returns, the write to the pipe returns -1 with errno = EPIPE

# Complete Example – Running "ls | wc -l"

```
int main(void)
{
    int fds[2];
    int pid1, pid2;

    pipe(fds);
    if ((pid1 = fork()) == 0) {
        dup2(fds[1], 1);
        close(fds[0]);
        close(fds[1]);
        execl("/usr/bin/ls", "ls", NULL);
    }
    If ((pid2 = fork()) == 0) {
        dup2(fds[0], 0);
        close(fds[0]);
        close(fds[1]);
        execl("/usr/bin/wc", "wc", "-l", NULL);
    }
    close(fds[0]);
    close(fds[1]);
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
    exit(0);
}
```

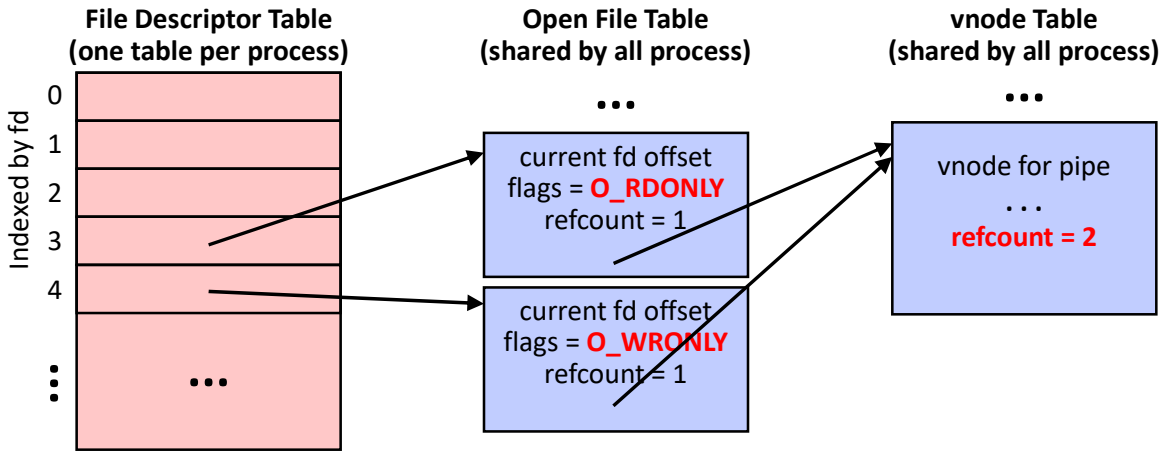
***It is important to close all unneeded/unused fds for a pipe so end of file and SIGPIPE conditions can be recognized correctly***





# Pipe File Descriptor Kernel Data Structures

***Immediately after call to pipe() returns***



## Pipe File Descriptor Kernel Data Structures

***The vnode refcount can never be greater than 2***

- The pipe() call creates 2 open file table entries (this vnode refcount = 2)
- No way to ever create additional open file table entries for the pipe
  - dup/dup2 and fork only create new file descriptors, but those only make references to the 2 existing open file table entries, never to new ones

***Recognizing end of file on a read()***

- The pipe buffering is empty and the vnode refcount == 1 (that 1 must be the read end, so must be no write end open!)

***Recognizing need to SIGPIPE on a write()***

- The vnode refcount == 1 (that 1 must be the write end, so no read end!)





## Interleaving Pipe Writes From Multiple Processes

***What if multiple processes write to the same pipe at the "same" time?***

- If pipe write() length **<** PIPE_BUF constant size
  - Each such write() is ***atomic***
  - One write() of this size ***won't*** be mixed with another write() of this size
  - The first such write() will be added to the pipe buffering entirely before the second, or the second entirely before the first
- For ***larger*** pipe write() lengths
  - Two such write() calls may arbitrarily interleave with each other in the pipe's buffering
- For Linux, the constant PIPE_BUF = 4096 (POSIX only requires ≥ 512)

## A Pipe Can Be Used Only Within Same Family Tree

***No way to have a pipe open other than to create it or inherit it***

- An open pipe file descriptor is inherited by your children when you fork
- An open pipe file descriptor is passed from one program to the next across an execve
- There's no other way to get a file descriptor to some pipe other than to create the pipe yourself (and you can then pass it on to your descendants)
- Example: Processes are all descendants of the original shell

```
$ ./myprog | tr ' ' '\12' | sort | uniq -c | sort –rn
```

- Example: With popen(), the command is a child of the calling process





## FIFOs – A "Named Pipe"

***A FIFO has a name in the filesystem, can be opened like any other file***

- The FIFO data isn't stored in the filesystem, but the FIFO name is
  - A directory entry, just like any other name
  - The inode's type marks this as a FIFO (rather than a regular file or a symbolic link, etc.)
  - The inode's protection and uid and gid owner work like any file

***Making a new FIFO – similar to creat() for a regular file***

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

- "pathname" is the FIFO's name, and "mode" is its protection
- Unlike open() or creat(), returns 0 on success, not an open file descriptor

## Opening and Using a FIFO

***Any* process must open() the FIFO before it can read or write it**

- fd = open(pathname, ***O_RDONLY***)
  - Blocks until at least one process has it open for ***writing***
- fd = open(pathname, ***O_WRONLY***)
  - Blocks until at least one process has it open for ***reading***

***End of file and SIGPIPE – work the same as for a regular (i.e., unnamed) pipe***

- ***Reading*** from an empty FIFO (no buffered data in it) when no process has it open for ***writing*** returns end of file from the read()
- ***Writing*** to a FIFO when no process has it open for ***reading*** causes SIGPIPE signal (if the process has SIGPIPE ignored or if the handler returns, then write() returns -1, with errno = EPIPE)





## Pipes vs. FIFOs

| | ***Pipe*** | ***FIFO*** |
|---|---|---|
| ***Created with*** | int = pipe(fds) | int mkfifo(pathname, mode) |
| ***Accessed with*** | Inherited file descriptor | fd = open(pathname, flags)<br>O_RDONLY or O_WRONLY |
| ***Accessible by*** | Only processes in the same process family tree | Any process (subject to standard file protection) |
| ***Name in filesystem*** | No (no name) | Yes (ls -l shows as “p”) |
| ***State persistence*** | No, everything is volatile | Name and inode persist on disk, data is volatile |

# An Example of Using FIFOs

***A server that is accessed locally using FIFOs rather than over the network***

- Client processes each write their requests into a FIFO created by the server
- The server replies to each client process in a FIFO created by that client