CSCI 4061: Pipes and FIFOs

Chris Kauffman

Last Updated:
Wed Mar 24 03:35:23 PM CDT 2021

Logistics

Reading: Stevens/Rago

- Ch 15.1-5 Pipes/FIFOs
- `man pipe(7)` documentation
- Ch 15.6-12
- Wikip: Dining Philosophers

Goals

- Finish Signals
- Pipes (Unnamed)
- Pipelines
- FIFOs (Named pipe)

Assignments

- Lab09: Pipelines
- HW09: FIFOs

Project 2

Delayed because I suck

Exercise: Warm-up

Recall: Pipes

1. What's a pipe? How permanent is a pipe?
2. How does one set up a pipe in C?
3. How does one set up a pipe on the command line shell?

Answers: Warm-up

Recall: Pipes

1. What's a pipe?
   Communication buffer to allow programs to talk to one another, typically output of one program becomes input to another. OS automatically de-allocates a pipe when no processes are using it.
2. How does one set up a pipe in C?

```
int pipe_fds[2];
pipe(pipe_fds);   // 2 fds for read/write now in array
```

3. How does one set up a pipe on the command line shell?

```
$> cmd1 | cmd2
```

Pipes and Pipelines

- Have discussed pipes previously (`commando`)
- Unix **pipelines** allow simple programs to combine to solve new problems: program output becomes input for another program

History

McIlroy noticed that much of the time command shells passed the output file from one program as input to another. His ideas were implemented in 1973 when ("in one feverish night", wrote McIlroy) Ken Thompson added the pipe() system call and pipes to the shell and several utilities in Version 3 Unix. "The next day", McIlroy continued, "saw an unforgettable orgy of one-liners as everybody joined in the excitement of plumbing."
– Wikipedia: Unix Pipes

- Pipe solutions alleviate need for temporary files

Count the files in a directory

- Solution 1: write a C program using `readdir()` in a counting loop
- Solution 2: `ls`, then count by hand
- Solution 3: `ls > tmp.txt`, count lines in file
- **Pipe Solution**

  ```
  > ls | wc -l
  ```

  `wc -l file` counts lines from file / stdin

A historical note

"Programming Pearls" by Jon Bentley, CACM 1986 with special guests

- Donald Knuth, godfather of CS
- Doug McIlroy, inventor of Unix pipes

Problem statement: Top-K words

Given a text file and an integer K, print the K most common words in the file (and the number of their occurrences) in decreasing frequency.

Knuth's Solution:

- ~8 pages of text and pseudo-code / Pascal
- Demonstration of "literate programming" [1] so may be a bit more verbose than needed

McIlroy's Solution?

[1] Literate Programming is a Knuth invention involving writing code interspersed with detailed, formatted comments describing it. Humans read the combination while a program is then used to extract and compile the code.

Pipeline for Top-K Words

McIlroy's Solution (Roughly)

```
#!/bin/bash
#
# usage: topk.sh <K> <file>
K=$1                        # arg1 is K value
file=$2                     # arg2 is file to search

cat $file                 | # Feed input \
tr -sc 'A-Za-z' '\n'      | # Translate non-alpha to newline \
tr 'A-Z' 'a-z'            | # Upper to lower case \
sort                      | # Duh \
uniq -c                   | # Merge repeated, add counts \
sort -rn                  | # Sort in reverse numerical order \
head -n $K                  # Print only top 10 lines
```

- 9 lines of shell script / piped Unix commands
- Original was not a script so was only 6 lines long

Lab09: Pipelines

- Have several stock files in CSV (comma separated value) format
- Wanted top 5 dates with the biggest increase in stock price; e.g. Change = Close - Open; top 5 changes
- Construct a pipeline to calculate this
- Create a shell script for the pipeline

```
lab09-code> ls stock*
stocks-apple.csv  stocks-gamestop.csv  stocks-uber.csv

lab09-code> head stocks-gamestop.csv
Date,Open,High,Low,Close,Volume
03/19/2021,"195.73","227.00","182.66","200.27","24,677,301"
03/18/2021,"214.00","218.88","195.65","201.75","11,799,910"
03/17/2021,"217.84","231.47","204.00","209.81","16,481,590"
03/16/2021,"203.16","220.70","172.35","208.17","35,422,871"
...
```

Exercise: Tool Familiarity

- It is not possible to write complex pipelines unless you are somewhat familiar with each component
- Getting basic familiarity with available Unix tools can save you TONs of work
- Note: solutions don't necessarily involve pipelines

Diff between `DirA` and `DirB`

- Have two directories `DirA` and `DirB` with about 250 of mostly identical files
- Some files exist in only one directory, some files differ between them
- Want the *difference* between the directories

Find Phone Numbers

We have 50,000 HTML files in a Unix directory tree, under a directory called `/website`. We have 2 days to get a list of file paths to the editorial staff. You need to give me a list of the .html files in this directory tree that appear to contain phone numbers in the following two formats: (xxx) xxx-xxxx and xxx-xxx-xxxx.

From: The Five Essential Phone-Screen Questions, by Steve Yegge

Answers: Tool Familiarity

Diff between `DirA` and `DirB`

```
> find lectures/ | wc -l        # 247 files in lectures/
    247     247    9149
> find lectures-copy/ | wc -l   # 246 files in lectures-copy
    246     246   15001
> diff -rq lectures/ ~/tmp/lectures-copy
Files lectures/09-pipes-fifos.org and lectures-copy/09-pipes-fifos.org differ
Files lectures/09-pipes-fifos.pdf and lectures-copy/09-pipes-fifos.pdf differ
Files lectures/09-pipes-fifos.tex and lectures-copy/09-pipes-fifos.tex differ
Only in lectures/: new-file.txt
```

Find Phone Numbers

Here's one of many possible solutions to the problem:

```
grep -l -R \
  --perl-regexp "\b(\(\d{3}\)\s*|\d{3}-)\d{3}-\d{4}\b" * \
  > output.txt
```

But I don't even expect candidates to get that far, really. If they say, after hearing the question, "Um... grep?" then they're probably OK.

– Steve Yegge

Check out Tool Time Session 3 for more info

Exercise: Pipes have a limited size

A pipe has a limited capacity...

Since Linux 2.6.11, the pipe capacity is 16 pages (i.e., 65,536 bytes in a system with a page size of 4096 bytes).

Applications should not rely on a particular capacity: an application should be designed so that a reading process consumes data as soon as it is available, so that a writing process does not remain blocked.
– `man pipe(7)` Manual documentation

- Examine the program `fill_pipe.c`
- Observe the behavior of programs as pipes fill up
- Relate this to **a major flaw** in Project 1 `commando`
  Hint: when did `cmd_fetch_output()` get called...

Answer: Pipes have a limited size

- `commando` set up child processes to write into pipes for their standard output
- `commando` used calls to `waitpid()` to wait until a child was finished, THEN read all child output from the pipe
- Children would call `write()` to generate output going into pipes
- If the pipe filled up, the child's `write()` would block
- `commando` would be waiting on blocked child but never empty the pipe to allow it to proceed
- End result: child never finishes

This is an example of deadlock: protocol used by cooperating entities ends with both getting stuck waiting for the other

- Resolutions for `commando`?

Convenience Functions for Pipes

C standard library gives some convenience functions for use with FILE* for pipes. Demoed in `pager_demo.c` / `popen_demo.c`

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
// Does a fork() / exec() with `cmdstring` to create a child process
// which is connected to the parent via the FILE * that is returned.
// If type is "r", the parent reads child output so the file pointer,
// is connected to the standard output of `cmdstring`.
// If type is "w", the parent write to child input so the file pointer,
// is connected to the standard input of `cmdstring`.
// Returns: file pointer if OK, NULL on error

int pclose(FILE *fp);
// The pclose function closes the standard I/O stream, waits for the
// command to terminate, and returns the termination status of the
// shell.
```

Figures below from Stevens/Rago

Figure 15.9 Result of `fp = popen(cmdstring, "r")`

Figure 15.10 Result of `fp = popen(cmdstring, "w")`

Pipe I/O and Signals

Behavior of I/O operations based on state of pipe is as follows

	Has Bytes	Has Space	Read End	Write End	Effect
`read()`	Yes	-	Open	-	`read()` return #bytes
`read()`	No	-	Open	Open	process blocks
`read()`	No	-	Open	Closed	`read()` returns 0 / EOF
`write()`	-	Yes	Open	Open	`write()` returns #bytes
`write()`	-	No	Open	Open	process blocks
`write()`	-	-	Closed	Open	**`SIGPIPE` sent to proc**

Note last line: `write()` to a pipe with no readers results in `SIGPIPE`

Exercise: Signals and Pipes

```
1  // broken_pipe_signalling.c
2  void bp_handler(int sig_num) {
3    fprintf(stderr,"Received SIGPIPE!\n");
4    fflush(stderr);
5    if(getenv("EXIT_ON_BROKEN_PIPE")){
6      fprintf(stderr,"exit(1)\n");
7      exit(1);
8    }
9  }
10
11 int main () {
12   if(getenv("HANDLE_BROKEN_PIPE")){
13     struct sigaction my_sa = {};
14     my_sa.sa_handler = bp_handler;
15     my_sa.sa_flags = SA_RESTART;
16     sigaction(SIGPIPE, &my_sa, NULL);
17   }
18   for(int i=0; 1; i++){
19     printf("%d\n",i);
20   }
21   return 0;
22 }
```

What gets printed in each of the following cases and why?

```
> gcc broken_pipe_signaling.c
> ./a.out | head          # 1
...
> export HANDLE_BROKEN_PIPE=1
> ./a.out | head          # 2
...
> export EXIT_ON_BROKEN_PIPE=1
> ./a.out | head          # 3
...
```

Relate your answer to signal delivery and signal handlers

Answers: Signals and Pipes

1

```
> gcc broken_pipe_signaling.c
> ./a.out | head
0
1
2
3
4
5
6
7
8
9
>
```

Default disposition for `SIGPIPE` is `TERM`: process dies when read end closes

2

```
> export HANDLE_BROKEN_PIPE=1
> ./a.out | head
0
1
2
3
4
5
6
7
8
9
Received SIGPIPE!
Received SIGPIPE!
Received SIGPIPE!
Received SIGPIPE!
...
```

Catching `SIGPIPE` but restarting `write()` to broken pipe, receive another signal, cycles infinitely

3

```
> export EXIT_ON_BROKEN_PIPE=1
> ./a.out | head
0
1
2
3
4
5
6
7
8
9
Received SIGPIPE!
exit(1)
>
```

Catching `SIGPIPE` and exiting in handler

FIFO: Named Pipe

- Major limitation of pipes is that they must be created by a parent and shared with a child
- No way for two unrelated processes to share a pipe...
  Or is there?

First In First Out

- A Unix **FIFO** or **named pipe** is a pipe which has a place in the file system
- Can be created with either a shell command or via C calls

Command/Call	Effect
`mkfifo filename`	Create a FIFO on the command shell
`int mkfifo(char *path, mode_t perms)`	System call to create a FIFO

Working with Fifos

A FIFO looks like a normal file but it is not

```
> mkfifo my.fifo                                  # Create a FIFO
> ls -l my.fifo
  prw-rw---- 1 kauffman kauffman 0 Oct 24 12:05 my.fifo
# ^ it's a 'p' for Pipe           ^ 0 size on disk

> echo 'Hello there!' > my.fifo                   # write to pipe
# hung  C-c

> echo 'Hello there!' > my.fifo &                 # write to pipe in background job
[1] 1797
> cat my.fifo                                     # read from pipe
Hello there!                                      # got what was written in
[1]+  Done    echo 'Hello there!' > my.fifo       # writer finished

> cat my.fifo                                     # read from pipe (nothing there)
# hung  C-c

> cat my.fifo &                                   # read from pipe in background job
[1] 1933
> echo 'Hello there!' > my.fifo                   # write to pipe
Hello there!
>
[1]+  Done    cat my.fifo                         # reader finished
```

A Few Oddities for FIFOs

> *In the normal case (without `O_NONBLOCK`), an `open()` for read-only blocks until some other process opens the FIFO for writing. Similarly, an `open()` for write-only blocks until some other process opens the FIFO for reading.*
> *– Stevens/Rago pg 553 (15.5 on FIFOs)*

- Explains why following hangs

  ```
  > echo 'Hello there!' > my.fifo  # write only to pipe
  ```

- No other process is reading from the FIFO yet
- Much harder to set up non-blocking I/O in terminals and likely not worth it
- Also requires **care** to make sure processes writing to FIFOs don't hang because no reader exists
- Standard trick is to open FIFO in Read/Write mode: avoids blocking at expense of some other problems, demoed next HW

FIFOs are Pipes

- BOTH Pipes / FIFOs use the same mechanics
  - Have limited capacity, 65K by default
  - Use a RAM buffer for fast communication, so no permanent storage, 0 size on disk
  - Same behavior for `read()` / `write()` concerning blocking, end of file, signals
- EXCEPT that creating/opening them has minor differences
  - Standard/Unnamed pipes created AND opened via `pipe()`: always a read + write end
  - FIFOs created via `mkfifo()`, opened via `open()`, will block process if opening for only read OR write - *always two partners*

Differences Between Pipes and Files

- Recall: OS manages position for read/write in both Files and FIFOs but in subtly different ways
- `multiple_writes.c` forks a child, both parent and child write different messages into a File or FIFO
- Can invoke this program with command line options which dictate the order and type of where stuff is written

Study `multiple_writes.c`

1. Process opens normal file, forks, Parent / Child write.

   ```
   > multiple_writes prefork file tmp.txt 20
   ```

2. Process forks, opens file, Parent / Child write.

   ```
   > multiple_writes postfork file tmp.txt 20
   ```

3. Process opens a FIFO, forks, Parent / Child write.

   ```
   > multiple_writes prefork fifo tmp.fifo 20
   ```

4. Proccess forks, opens FIFO, Parent / Child write.

   ```
   > multiple_writes postfork fifo tmp.fifo 20
   ```

Exercise: Predict Output that Appears

```
#1 PREFORK OPEN FILE
int fd = open("tmp.file",..);
int ch = fork();
for(i=0; i<iters; i++){
  if(ch==0){
    write(fd,"child",..);
  }
  else{
    write(fd,"parent",..);
  }
}
close(fd);
```

```
#2 POSTFORK OPEN FILE
int ch = fork();
int fd = open("tmp.file",..);
for(i=0; i<iters; i++){
  if(ch==0){
    write(fd,"child",..);
  }
  else{
    write(fd,"parent",..);
  }
}
close(fd);
```

```
#3 PREFORK OPEN FIFO
int fd = open("tmp.fifo",..);
int ch = fork();
for(i=0; i<iters; i++){
  if(ch==0){
    write(fd,"child",..);
  }
  else{
    write(fd,"parent",..);
  }
}
close(fd);
```

```
#4 POSTFORK OPEN FIFO
int ch = fork();
int fd = open("tmp.fifo",..);
for(i=0; i<iters; i++){
  if(ch==0){
    write(fd,"child",..);
  }
  else{
    write(fd,"parent",..);
  }
}
close(fd);
```

Answers: Differences Between Pipes/FIFOs and Files

1. Process opens normal file, forks, Parent / Child write.

   `> multiple_writes prefork file tmp.txt 20`

   *Both parent and child output appear, OS manages a **shared write position** between parent and child*

2. Process forks, opens file, Parent / Child write. File position is NOT shared so will overwrite each other in file.

   `> multiple_writes postfork file tmp.txt 20`

   *Parent and child each have **independent write positions**, loss of data from file*

3. Process opens a FIFO, forks, Parent / Child write.

   `> multiple_writes prefork fifo tmp.fifo 20`

   *Pipes always have a **shared write position**, all data from parent and child appear*

4. Proccess forks, opens FIFO, Parent / Child write.

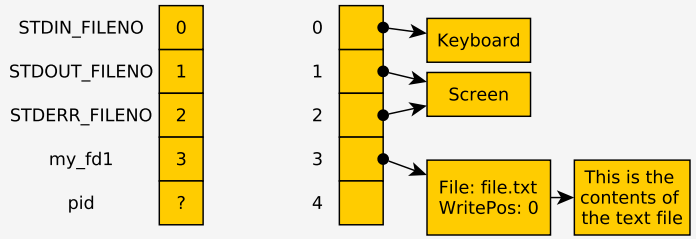
   `> multiple_writes postfork fifo tmp.fifo 20`

   *Pipes always have a **shared write position**, all data from parent and child appear*

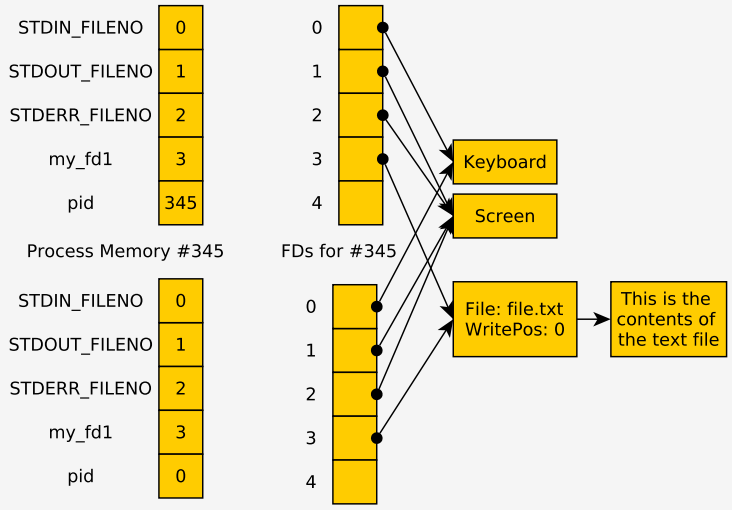
Draw some pictures of the internal FD table, Open file table, and INodes to support these.

open() normal file then call fork()

my_fd = open("file.txt"); // called by parent

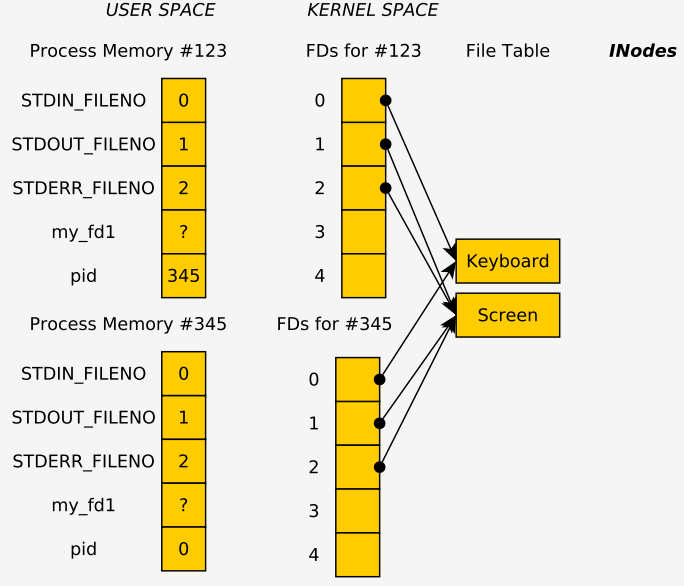


pid = fork();

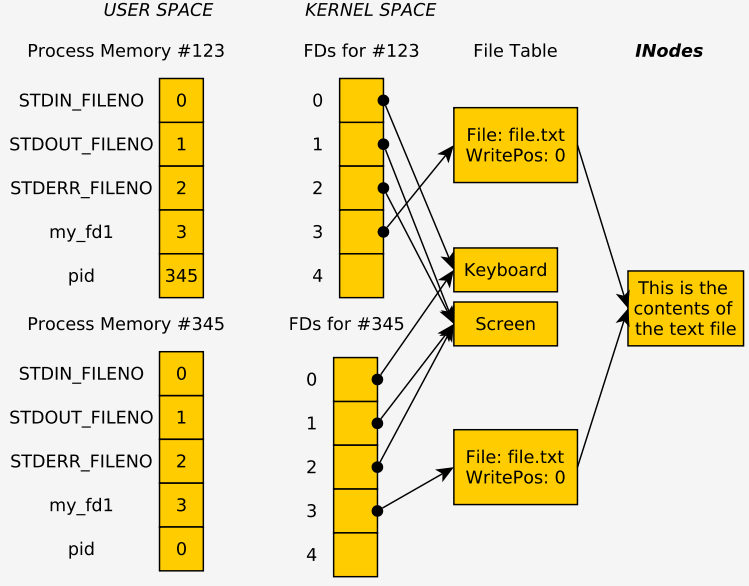


fork() then call open() normal file

pid = fork();

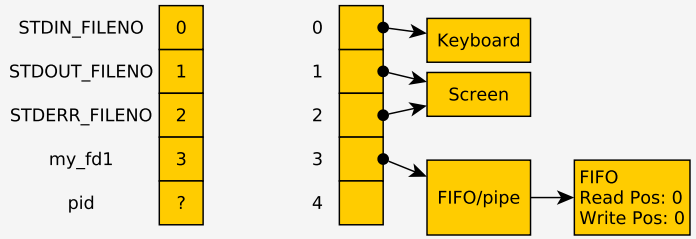


my_fd = open("file.txt"); // called by parent and child

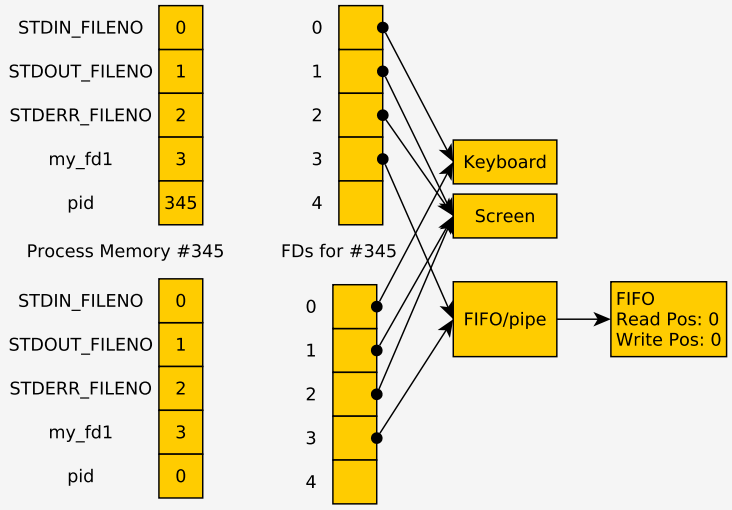


open() FIFO then call fork()

my_fd = open("my.fifo"); // called by parent

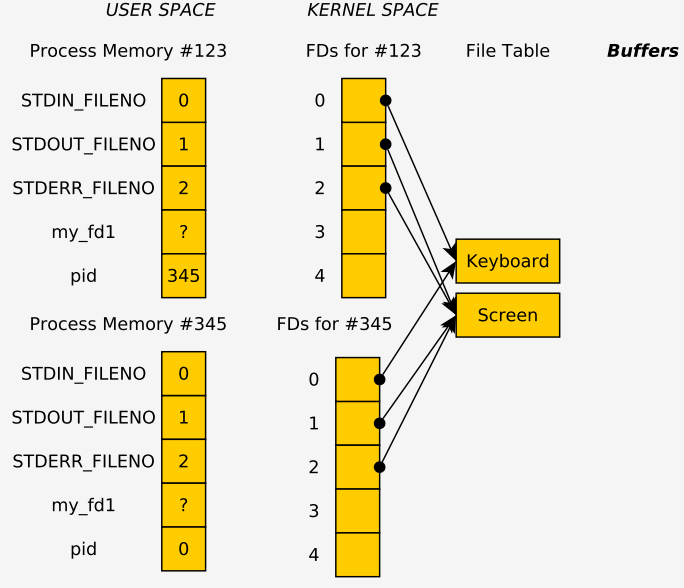


pid = fork();

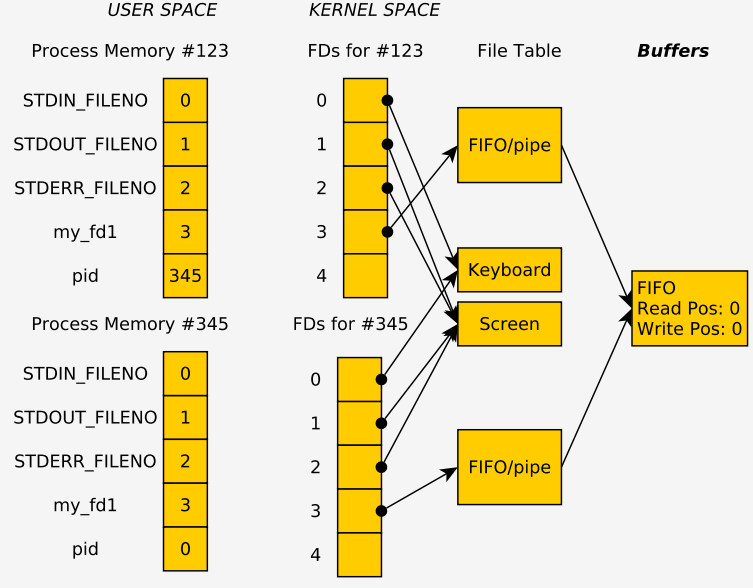


fork() then call open() FIFO

pid = fork();



my_fd = open("my.fifo"); // called by parent and child



Lessons on OS Treatment of Files/Pipes

File Descriptor Table

- One per process but stored in kernel space
- Each numbered entry refers to system wide File Table

INodes

Contains actual file and contents, corresponds to physical storage

Buffers for Pipes / Fifos

Internal kernel storage, Read/Write positions managed by kernel

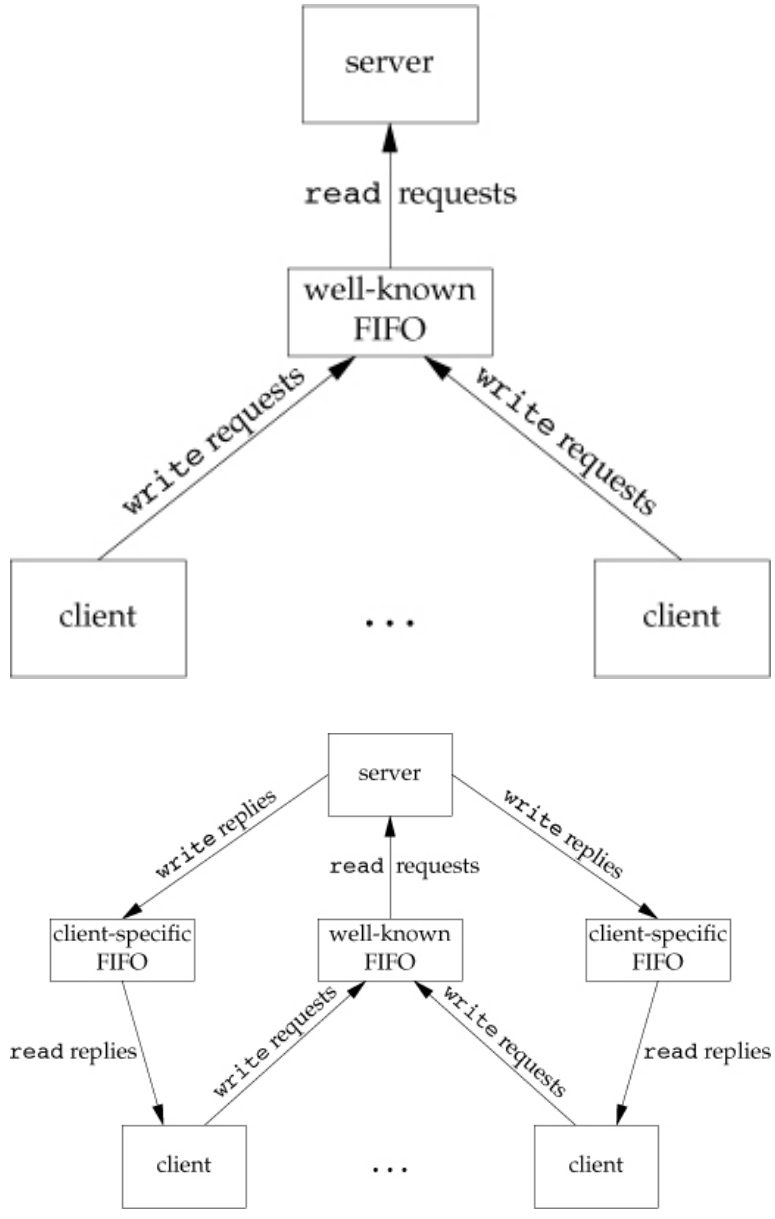
System File Table

- Shared by entire system, managed by the OS
- Each entry corresponds to open "thing" in use by a proc
- May have multiple file table entries per "real" file
- Each File Table Entry has its own Read/Write positions
- Connects File Descriptor Table to INodes, Buffers

Servers/Clients with FIFOs

- Create simple communication protocols
- Server which has names/email addresses
- Clients which have names, want email addresses
- Server are Daemon always running
- Client uses FIFOs to make requests to server and coordinate
- Basics of message passing between processes

Upcoming HW will discuss this, will be used for a project later in the semester



Source: Stevens and Rago Ch 15.5