# CS143A Principles on Operating Systems Discussion 08:

Instructor: Prof. Anton Burtsev

TA: Saehanseul Yi (Hans)

Nov 22, 2019 **Noon** 

#### Agenda

- pipe() and fork(): visualization
- How to debug a user-program in xv6
- sh.c call structure

```
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
  panic("pipe");
-----Point A-----
if(fork1() == 0){
  close(1);
 dup(p[1]);
 close(p[0]);
  close(p[1]);
-----Point B------
 runcmd(pcmd>left);
if(fork1() == 0){
  close(0);
  dup(p[0]);
 close(p[0]);
  close(p[1]);
  runcmd(pcmd>right);
```

```
close(p[0]);
close(p[1]);
------Point C------wait();
wait();
break;
```

```
$ | s | grep asdf asdfasdf $
```

```
close(p[0]);
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
  panic("pipe");
                                     wait();
-----Point A-----
                                     wait();
if(fork1() == 0){
                                     break;
  close(1);
 dup(p[1]);
 close(p[0]);
  close(p[1]);
-----Point B-----
 runcmd(pcmd>left);
if(fork1() == 0){
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  runcmd(pcmd>right);
```

```
close(p[1]);
------Point C-----
wait();
wait();
break;

parent process

$ ls | grep asdf
asdfasdf
$
```

```
close(p[0]);
case PIPE:
pcmd = (struct pipecmd*)cmd;
                                    close(p[1]);
if(pipe(p) < 0)
                                    -----Point C-----
  panic("pipe");
                                    wait();
-----Point A-----
                                    wait();
if(fork1() == 0){
                                    break;
  close(1);
                                                                            parent process
  dup(p[1]);
  close(p[0]);
  close(p[1]);
-----Point B-----
                                                                 asdfasdf
  runcmd(pcmd>left);
                                                fork1 child
                                                (left)
if(fork1() == 0){
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  runcmd(pcmd>right);
```

saehansy@uci.edu

fork1 child

(right)

```
close(p[0]);
case PIPE:
pcmd = (struct pipecmd*)cmd;
                                     close(p[1]);
if(pipe(p) < 0)
                                     -----Point C-----
  panic("pipe");
                                     wait();
-----Point A-----
                                     wait();
if(fork1() == 0){
                                     break;
  close(1);
                                                                             parent process
  dup(p[1]);
  close(p[0]);
                                                                                           fork1 child
  close(p[1]);
                                                                                           (right)
-----Point B-----
                                                                  asdfasdf
  runcmd(pcmd>left);
                                                fork1 child
                                                (left)
if(fork1() == 0){
                                                                             parent process
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  runcmd(pcmd>right);
                                              saehansy@uci.edu
```

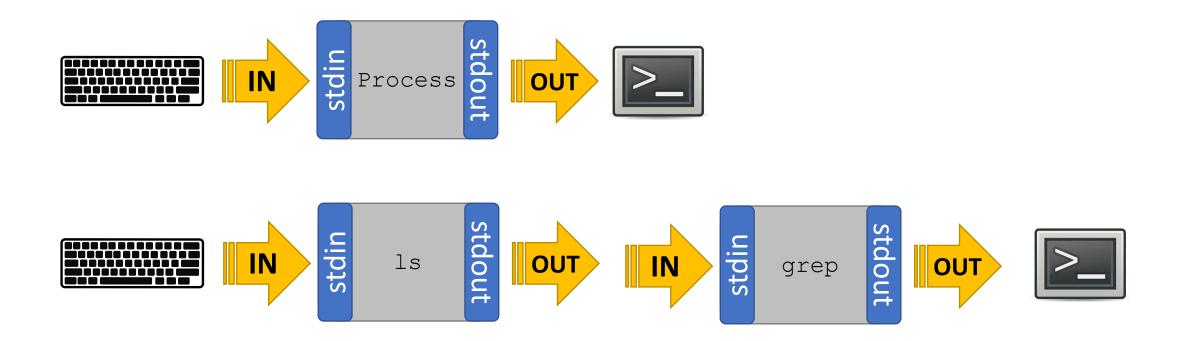
(standard input, standard output)

```
$
$ Is | grep asdf
asdfasdf
$
```



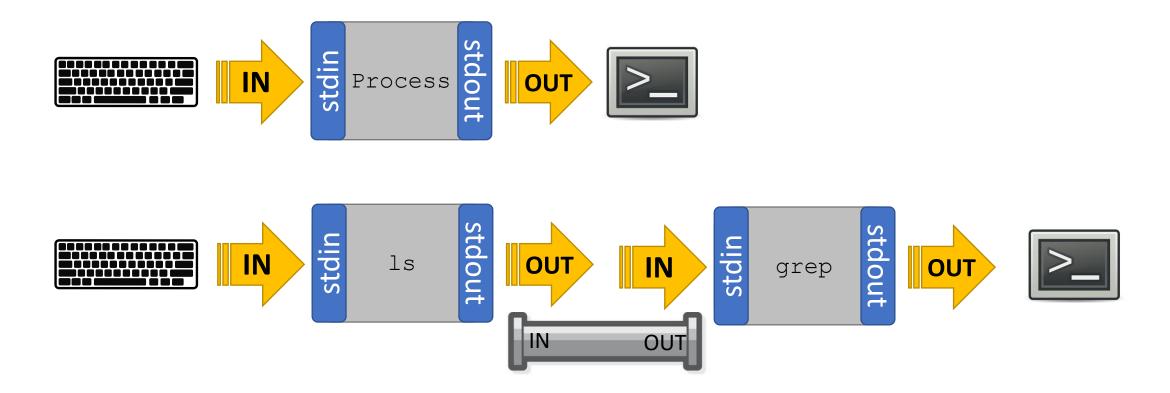
(standard input, standard output)

```
$ | grep asdf asdfasdf $
```



(standard input, standard output)

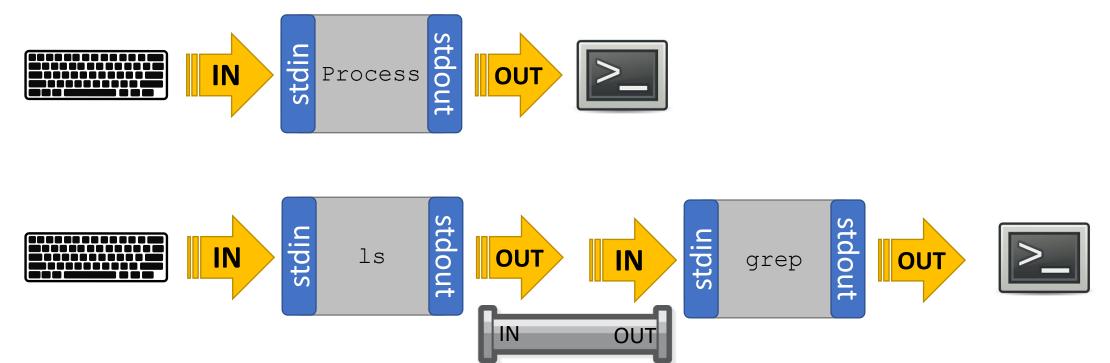
```
$
$ Is | grep asdf
asdfasdf
$
```



(standard input, standard output)



 stdin(0), stdout(1), and stderr(2) are file descriptors(i.e. just an integer for user-program)

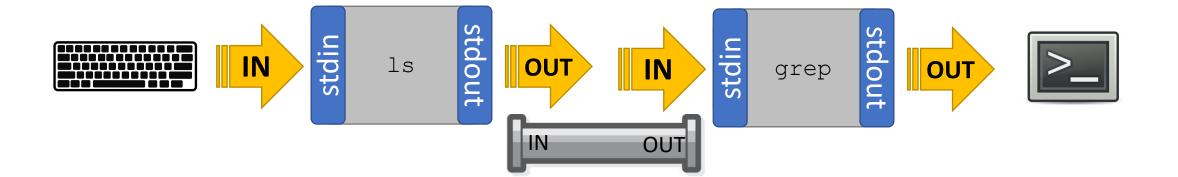


(standard input, standard output)



- stdin(0), stdout(1), and stderr(2) are file descriptors(i.e. just an integer for user-program)
- Each program has its own descriptor array(?)
   (e.g. A's stdin is 0 and B's stdin is 0 as well)





(standard input, standard output)

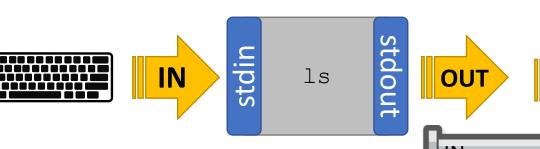


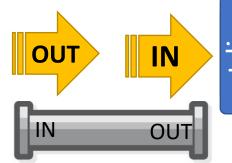
- stdin(0), stdout(1), and stderr(2) are file descriptors(i.e. just an integer for user-program)
- Each program has its own descriptor array(?) (e.g. A's stdin is 0 and B's stdin is 0 as well)
- How to modify process' file descriptors?

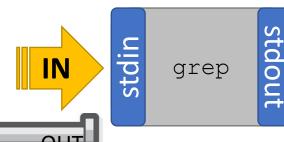
















(standard input, standard output)

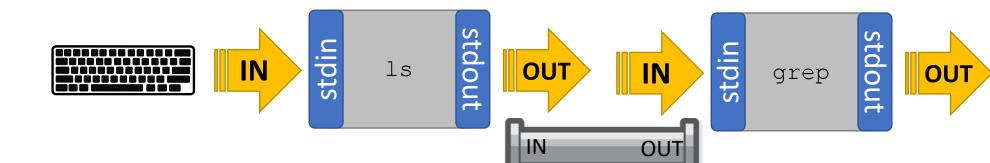


- stdin(0), stdout(1), and stderr(2) are file descriptors(i.e. just an integer for user-program)
- Each program has its own descriptor array(?) (e.g. A's stdin is 0 and B's stdin is 0 as well)
  - How to modify process' file descriptors?
    - close, dup(or open)







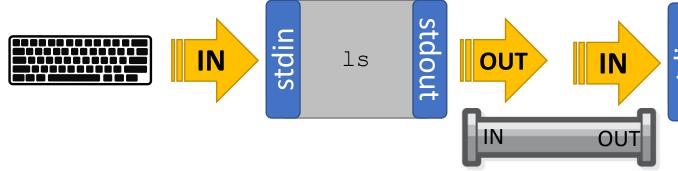


(standard input, standard output)





- stdin(0), stdout(1), and stderr(2) are file
   descriptors(i.e. just an integer for user-program)
- Each program has its own descriptor array(?)
   (e.g. A's stdin is 0 and B's stdin is 0 as well)
  - How to modify process' file descriptors?
    - close, dup(or open)
  - What we need to do: close appropriate descriptors for each process and set the appropriate descriptor by copying

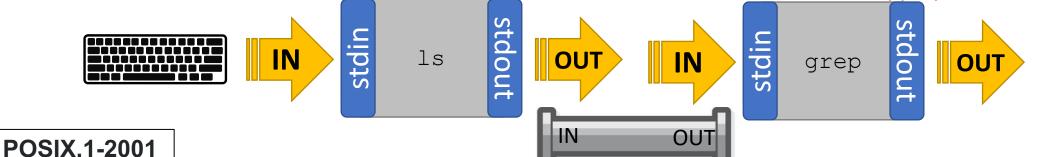




(standard input, standard output)



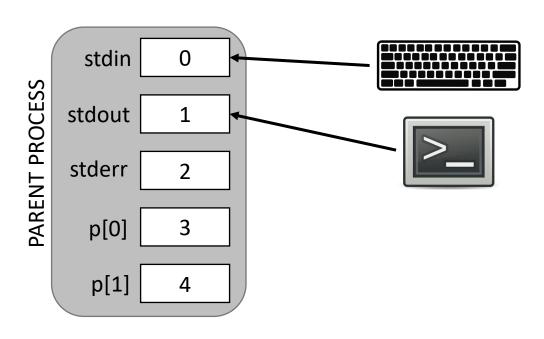
- IN Process OUT OUT
- stdin(0), stdout(1), and stderr(2) are file
   descriptors(i.e. just an integer for user-program)
- Each program has its own descriptor array(?)
   (e.g. A's stdin is 0 and B's stdin is 0 as well)
  - How to modify process' file descriptors?
    - close, dup(or open)
  - What we need to do: close appropriate descriptors for each process and set the appropriate descriptor by copying



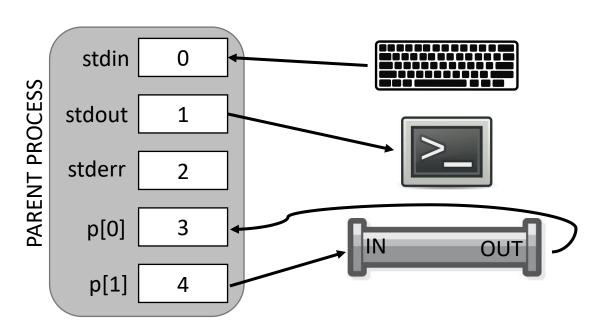
pipe() creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. filedes[0] is for reading, filedes[1] is for writing plansy@uci.edu pipe is uni-directional

```
-----Point 0-----
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
 panic("pipe");
-----Point A-----
if(fork1() == 0){
 close(1);
 dup(p[1]);
 close(p[0]);
 close(p[1]);
  -----Point B------
 runcmd(pcmd>left);
```

※ Throughout the example, stderr is always connected to the screen. Omitted for simplicity as well as p[0] and p[1] to the parent process

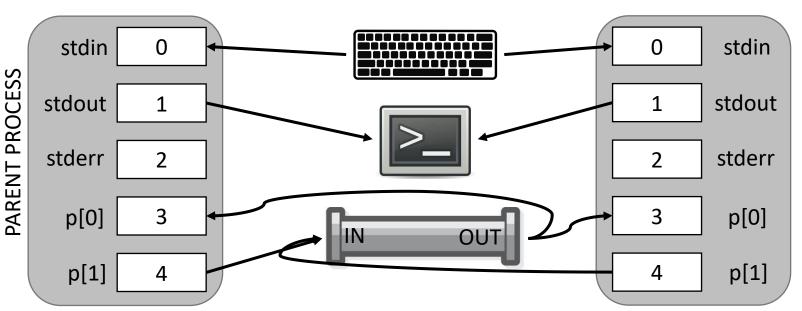


```
-----Point 0-----
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
 panic("pipe");
-----Point A-----
if(fork1() == 0){
 close(1);
 dup(p[1]);
 close(p[0]);
 close(p[1]);
            --Point B-----
 runcmd(pcmd>left);
```



```
-Point 0-
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
  panic("pipe");
 -----Point A-----
if(fork1() == 0){
  close(1);
  dup(p[1]);
  close(p[0]);
  close(p[1]);
              -Point B-----
  runcmd(pcmd>left);
```

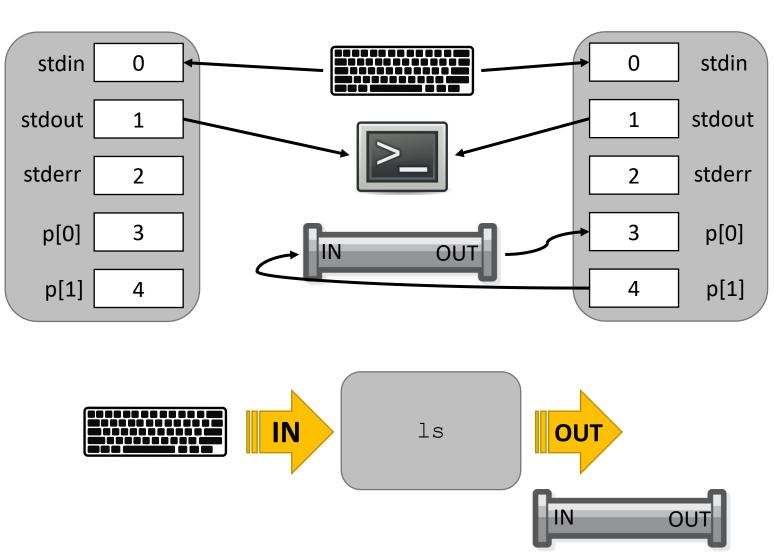
#### fork() copies the descriptors too!



PARENT PROCESS

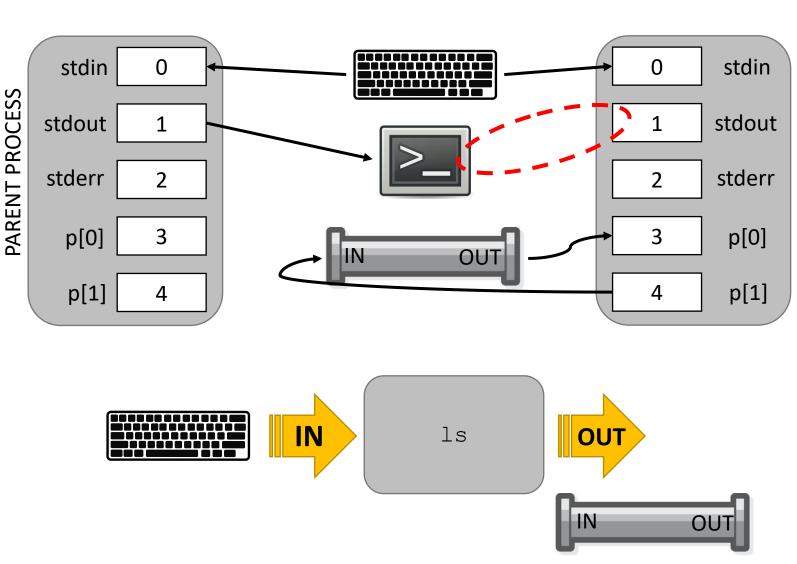
```
-Point 0-
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
  panic("pipe");
 -----Point A-----
if(fork1() == 0){
  close(1);
  dup(p[1]);
  close(p[0]);
  close(p[1]);
              -Point B--
  runcmd(pcmd>left);
```

#### fork() copies the descriptors too!



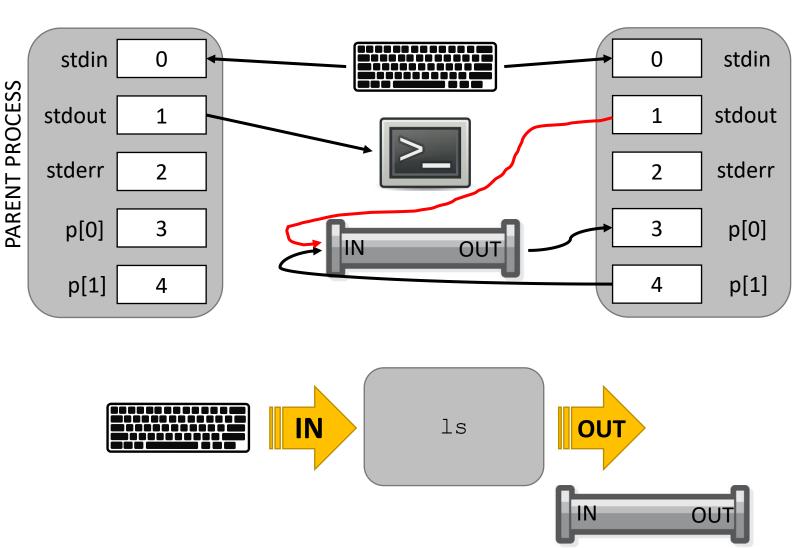
```
-Point 0-
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
  panic("pipe");
 -----Point A-----
if(fork1() == 0){
  close(1);
  dup(p[1]);
  close(p[0]);
  close(p[1]);
               -Point B--
  runcmd(pcmd>left);
```

#### fork() copies the descriptors too!



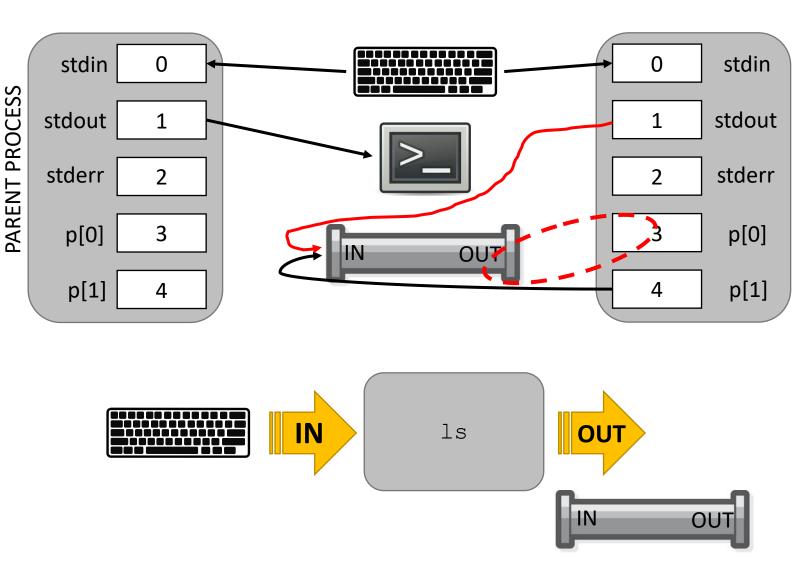
```
-Point 0-
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
  panic("pipe");
 -----Point A-----
if(fork1() == 0){
  close(1);
  dup(p[1]);
  close(p[0]);
  close(p[1]);
              -Point B--
  runcmd(pcmd>left);
```

fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!



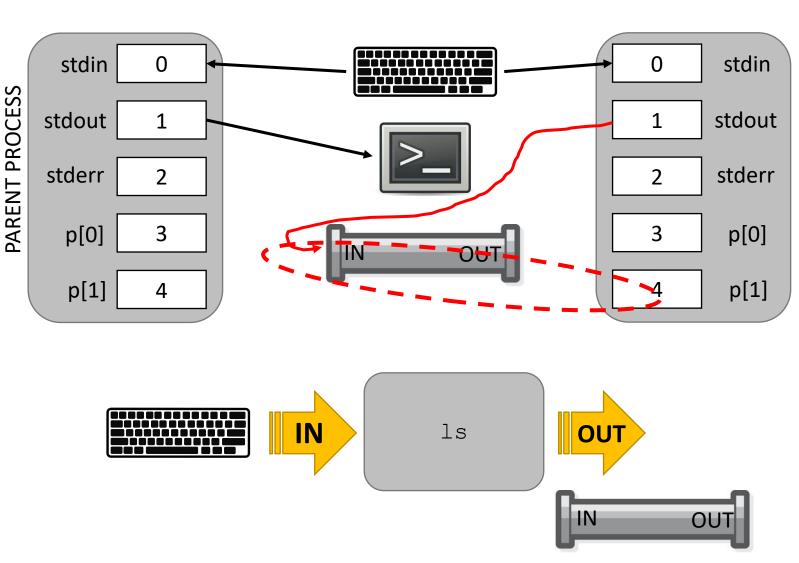
```
-Point 0-
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
  panic("pipe");
 -----Point A-----
if(fork1() == 0){
  close(1);
  dup(p[1]);
  close(p[0]);
  close(p[1]);
              -Point B--
  runcmd(pcmd>left);
```

fork() copies the descriptors too! dup()'s destination is the lowest & unused file descriptor!



```
--Point 0-
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
  panic("pipe");
 -----Point A-----
if(fork1() == 0){
  close(1);
  dup(p[1]);
  close(p[0]);
  close(p[1]);
               -Point B--
  runcmd(pcmd>left);
```

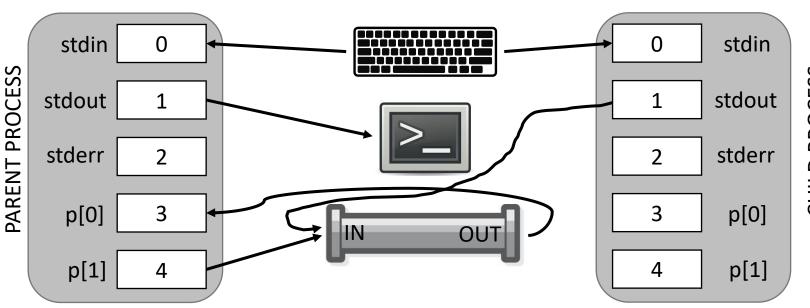
fork() copies the descriptors too! dup()'s destination is the lowest & unused file descriptor!



```
-Point B-
  runcmd(pcmd>left);
if(fork1() == 0){
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  runcmd(pcmd>right);
close(p[0]);
close(p[1]);
                -Point C--
wait();
wait();
```

break;

fork() copies the descriptors too! dup()'s destination is the lowest & unused file descriptor!

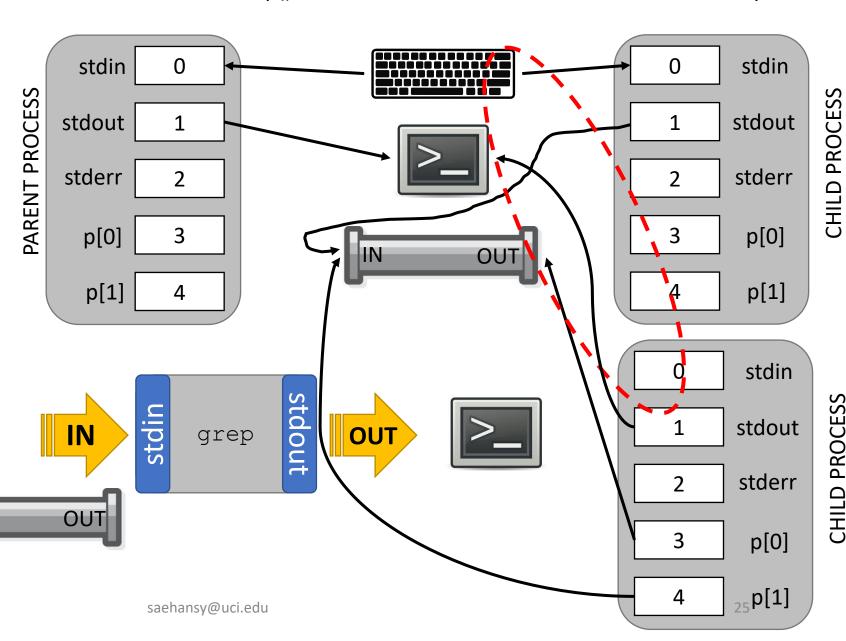


fork() copies the descriptors too! dup()'s destination is the lowest & unused file descriptor!

CHILD PROCESS

pipe() and fork() stdin stdin 0 PARENT PROCESS CHILD PROCESS -Point B stdout stdout runcmd(pcmd>left); stderr stderr if(fork1() == 0){ p[0] p[0] close(0); OUT dup(p[0]); p[1] p[1] 4 close(p[0]); close(p[1]); stdin runcmd(pcmd>right); stdin OUT stdout IN grep close(p[0]); close(p[1]); stderr -Point C--IN OUT wait(); p[0] wait(); <sub>24</sub>p[1] break; saehansy@uci.edu

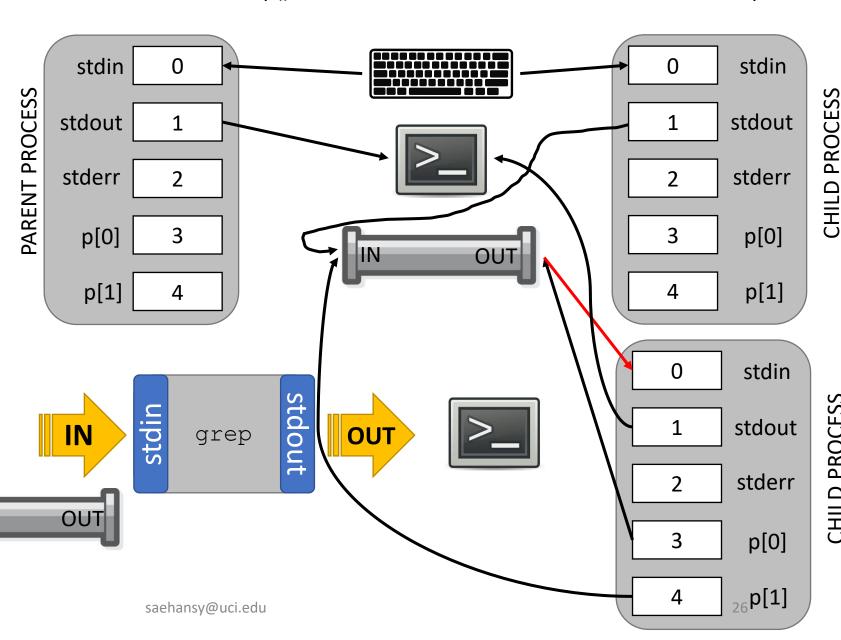
```
-Point B
  runcmd(pcmd>left);
if(fork1() == 0){
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  runcmd(pcmd>right);
close(p[0]);
close(p[1]);
                -Point C--
                                  IN
wait();
wait();
```



CHILD PROCESS

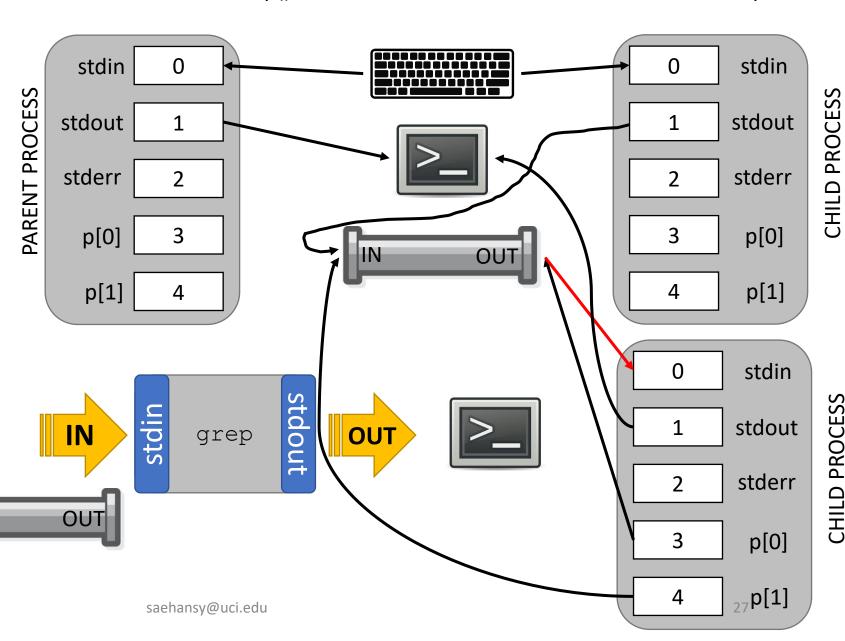
pipe() and fork()

```
-Point B
  runcmd(pcmd>left);
if(fork1() == 0){
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  runcmd(pcmd>right);
close(p[0]);
close(p[1]);
                -Point C--
                                  IN
wait();
wait();
```

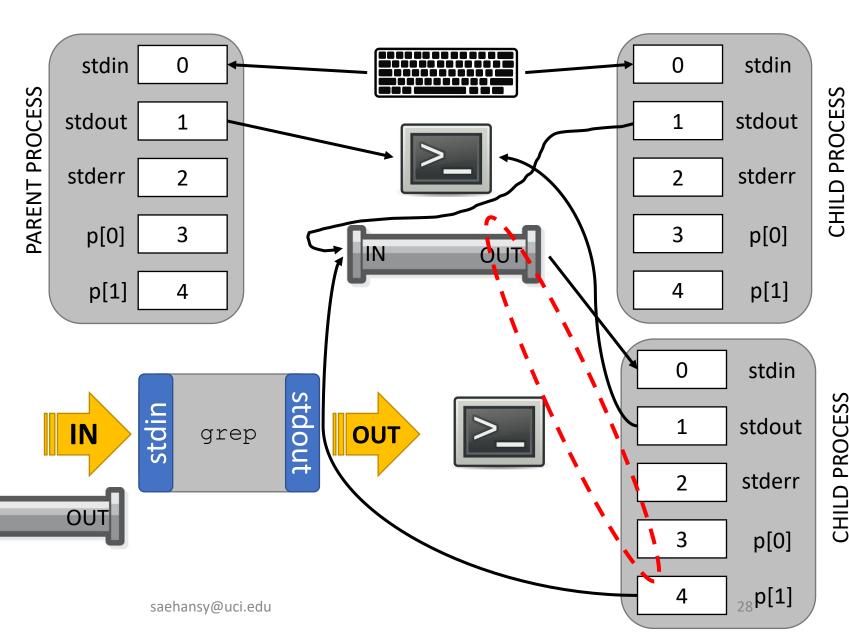


```
runcmd(pcmd>left);
if(fork1() == 0){
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  runcmd(pcmd>right);
close(p[0]);
close(p[1]);
                -Point C--
                                  IN
wait();
```

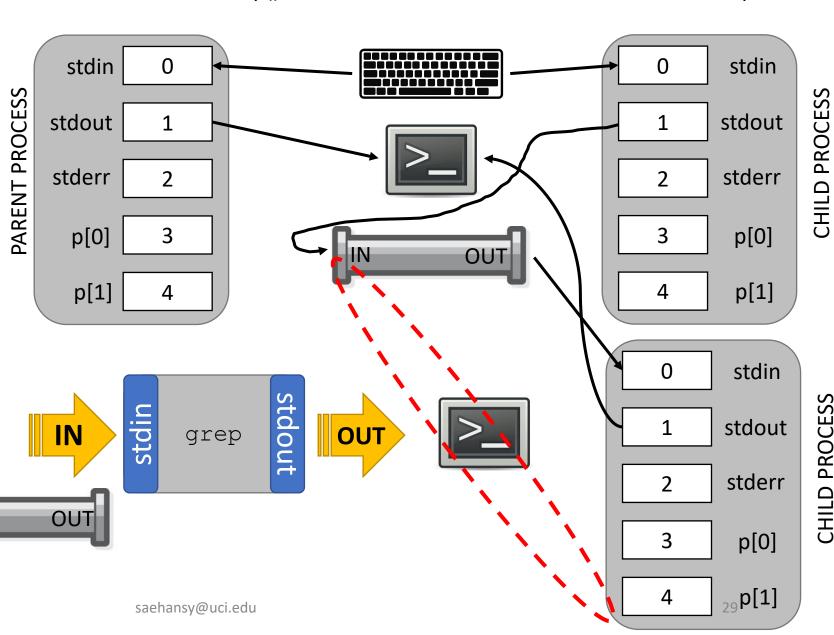
wait();



```
Point B
  runcmd(pcmd>left);
if(fork1() == 0){
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  runcmd(pcmd>right);
close(p[0]);
close(p[1]);
                -Point C--
                                  IN
wait();
wait();
```



```
Point B
  runcmd(pcmd>left);
if(fork1() == 0){
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  runcmd(pcmd>right);
close(p[0]);
close(p[1]);
                -Point C-
                                  IN
wait();
wait();
```



stdin

stdout

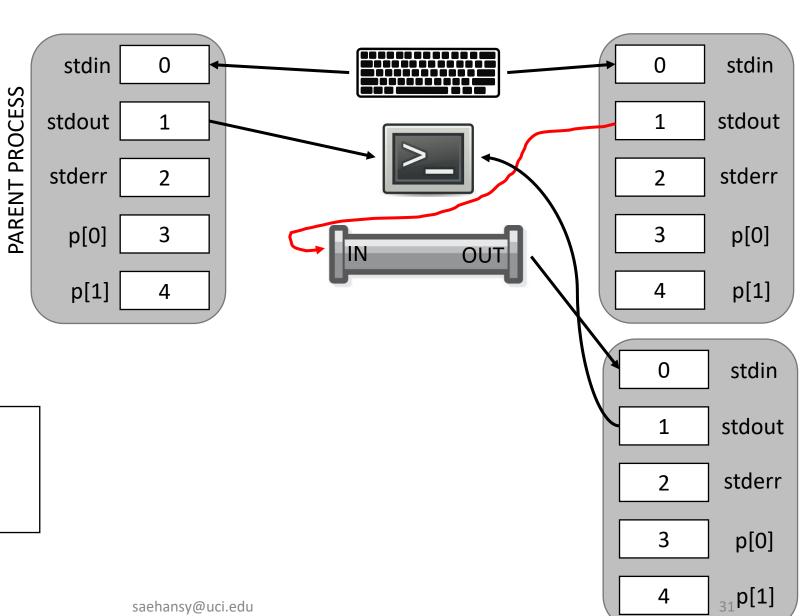
stderr

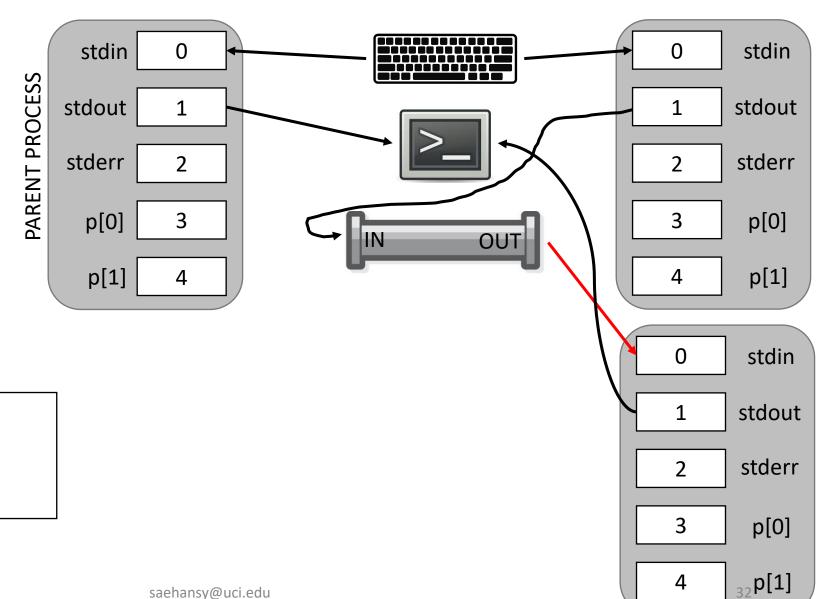
p[0]

p[1] 4 stdin stdout stderr as d f as d fp[0] <sub>30</sub>p[1] saehansy@uci.edu

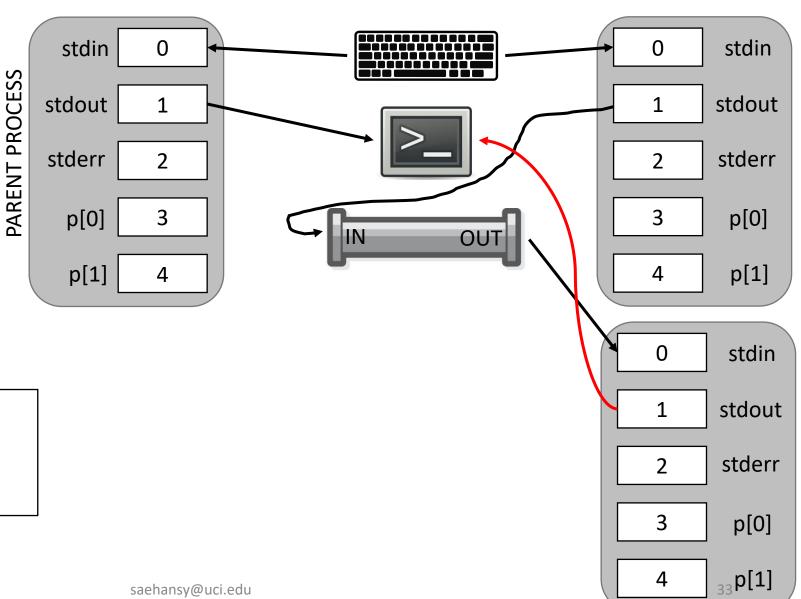
\$ Is | grep asdf

as d f as d f





\$ Is | grep asdf as d f as d f



\$ | grep asdf asdfasdf \$

#### Debugging xv6 user-programs

- If you start gdb with make 'qemu-nox-gdb' only kernel symbols are loaded
- The symbols of user programs(UPROGS in Makefile)—including sh, grep, ls—must be loaded for debugging
- file <binary> followed by break main
- UPROGS binary names start with \_ (e.g. \_sh)

```
(gdb) file _nsh
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Load new symbol table from "/home/saehansy/Workspace/ics143a/FQ19/qemu/xv6-public/_nsh"? (y or n)y
Reading symbols from /home/saehansy/Workspace/ics143a/FQ19/qemu/xv6-public/_nsh...done.
```

#### Debugging xv6 user-programs

- We are dealing with shell which has fork() and exec()
- Tell GDB what to follow (parent? children? or new process? old one?)
  - set follow-fork-mode (parent|children)
  - set follow-exec-mode (**new**|old)
  - make sure set the breakpoint inside child's code!
- if you having trouble booting xv6 after setting breakpoints, set them just before sh is executed
  - break exec
  - continue
  - 1<sup>st</sup> break
  - continue
  - 2<sup>nd</sup> break
  - if you type continue here, it will execute the shell. Type necessary things before typing continue including del br 1

it's a little buggy.. gdb is not always correct

#### Understanding sh.c

- Try out various commands, and use gdb to follow the call stack(graph)
- Make a note on each function
- Drawing a call graph for each scenario helps understanding the structure