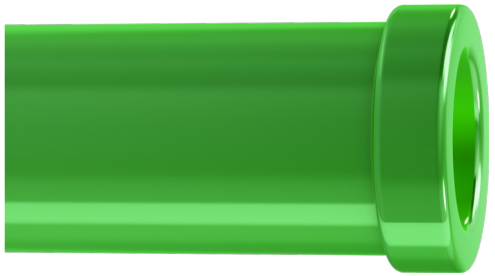
CSE 3100: Systems Programming

Part 2
Lecture 4: Pipes

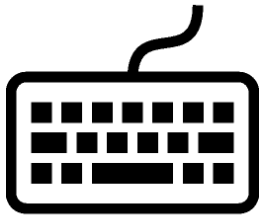


Review From Last Lecture (1)

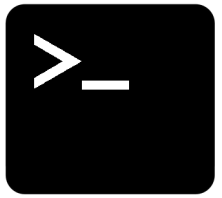
- A file descriptor is a nonnegative integer associated with a file.

FD	FILE *
0	stdin
1	stdout
2	stderr

The input, for example the process takes inputs from the keyboard.

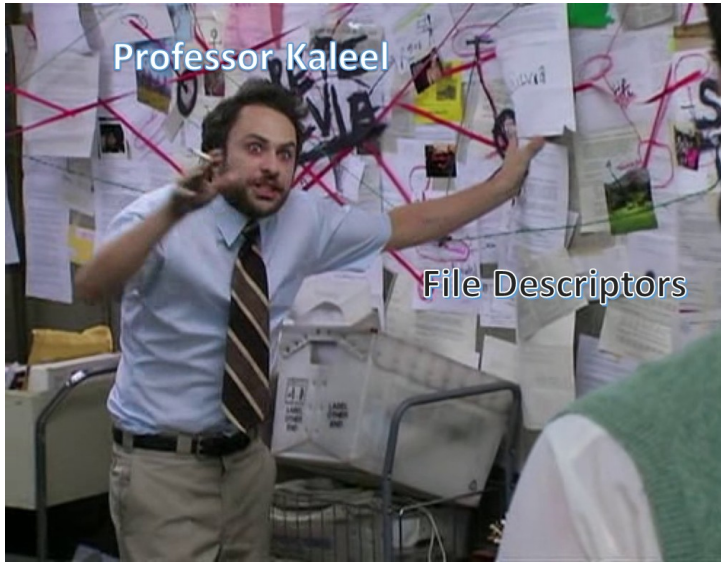


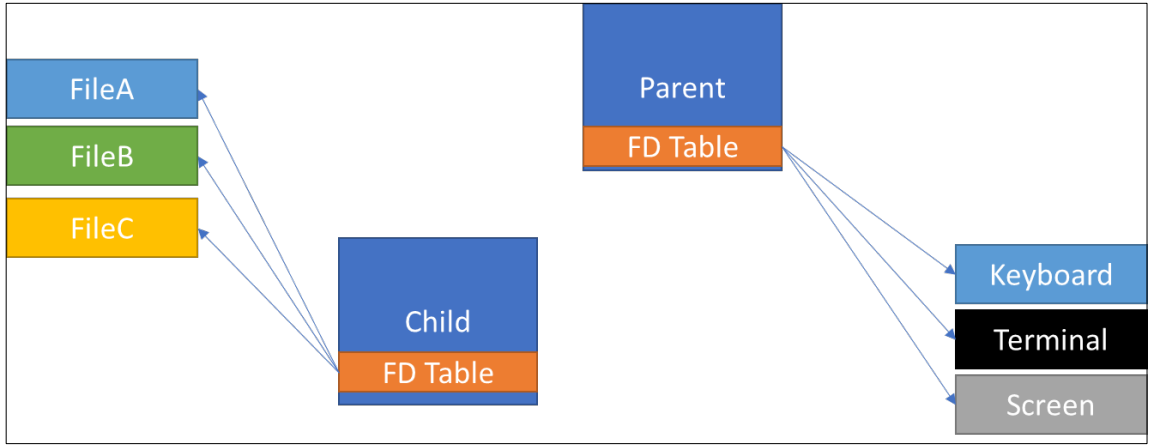
Where the process gives outputs e.g. the terminal.



Where the process reports errors, e.g. could be a log file recording the errors.

Review From Last Lecture (2)





1. We start with one process (program).
2. But we want a process to do multiple things, so we'll call fork.
3. However when we fork, both file descriptor tables point to the same things.
4. So we need to call dup() or dup2() in the child to get different functionality

Question: *What happens if we want processes to exchange data?*

- Let’s assume the child will do some computation.
- We then want to pass the value to the parent.
- Will the following code work?

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

int main(){
    int solution[2]; //variable to fill in values
    pid_t pid = fork();
```

- Create a solution variable which I want the child to fill in with values.
- Then I want the parent to be able to read those values.

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

int main(){
    int solution[2]; //variable to fill in values
    pid_t pid = fork();
    if(pid == 0)
    {
        //set value of solution
        solution[0] = 10;
        solution[1] = 15;
        printf("In child=%d\n", solution[0]);
        printf("In child=%d\n", solution[1]);
```

- Call fork
- In the child fill in the values of solution.

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

int main(){
    int solution[2]; //variable to fill in values
    pid_t pid = fork();
    if(pid == 0)
    {
        //set value of solution
        solution[0] = 10;
        solution[1] = 15;
        printf("In child=%d\n", solution[0]);
        printf("In child=%d\n", solution[1]);
    }
    else{
        sleep(2); //make the parent wait for the child
        printf("In parent=%d\n", solution[0]);
        printf("In parent=%d\n", solution[1]);
    }
    return 0;
}
```

- In the parent, wait for the child to finish first by calling sleep.
- Then check the values of the solution.

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

int main(){
    int solution[2]; //variable to fill in values
    pid_t pid = fork();
    if(pid == 0)
    {
        //set value of solution
        solution[0] = 10;
        solution[1] = 15;
        printf("In child=%d\n", solution[0]);
        printf("In child=%d\n", solution[1]);
    }
    else{
        sleep(2); //make the parent wait for the child
        printf("In parent=%d\n", solution[0]);
        printf("In parent=%d\n", solution[1]);
    }
    return 0;
}
```

Will this work correctly?

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

int main(){
    int solution[2]; //variable to fill in values
    pid_t pid = fork();
    if(pid == 0)
    {
        //set value of solution
        solution[0] = 10;
        solution[1] = 15;
        printf("In child=%d\n", solution[0]);
        printf("In child=%d\n", solution[1]);
    }
    else{
        sleep(2); //make the parent wait for the child
        printf("In parent=%d\n", solution[0]);
        printf("In parent=%d\n", solution[1]);
    }
    return 0;
}
```

Output of the code:

```
kaleel@CentralCompute:~$ ./test
In child=10
In child=15
In parent=0
In parent=0
```

Question: *What happens if we want processes to exchange data?*

- A **Pipe** is a technique used for inter process communication.
- A pipe is a mechanism by which the output of one process is directed into the input of another process.
- A pipe file is created using the pipe system call.
- A pipe has an input end and an output end.
- One can write into a pipe from input end and read from the output end

Pipe Syntax

```
#include <unistd.h>
int  pipe(int pipefd[2]);
```

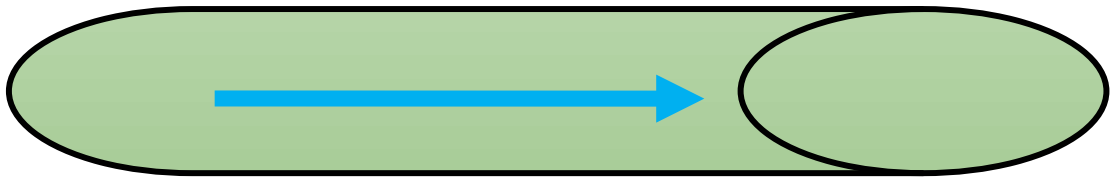
Creates a one-way pipe (a buffer to store a byte stream)

Two FDs in pipefd:

- pipefd[0] is the read end
- pipefd[1] is the write end

Returns 0 if successful

write to pipefd[1]



read from pipefd[0]

Let’s revisit this example code…can we fix it with pipes?

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

int main(){
    int solution[2]; //variable to fill in values
    pid_t pid = fork();
    if(pid == 0)
    {
        //set value of solution
        solution[0] = 10;
        solution[1] = 15;
        printf("In child=%d\n", solution[0]);
        printf("In child=%d\n", solution[1]);
    }
    else{
        sleep(2); //make the parent wait for the child
        printf("In parent=%d\n", solution[0]);
        printf("In parent=%d\n", solution[1]);
    }
    return 0;
}
```

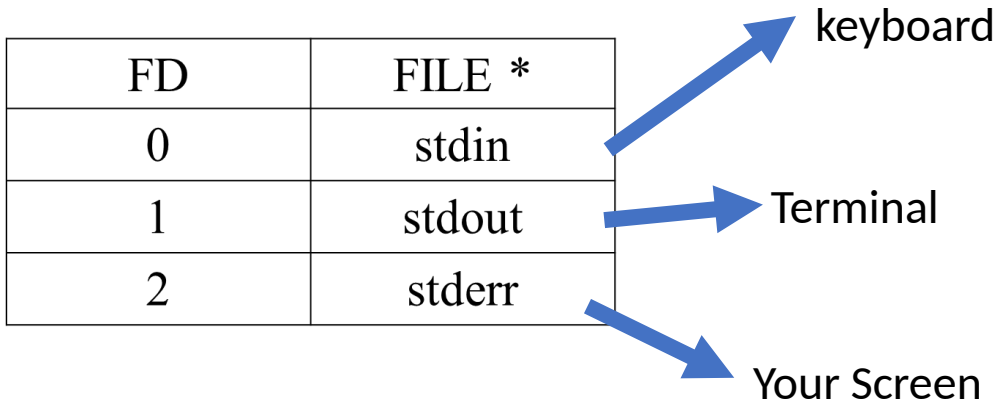
```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main(){
    //create the pipe
    int pd[2];
    pipe(pd);
```

- Create a variable (integer array of size 2) for holding the pipe.
- Call pipe to put entries in the fd table.

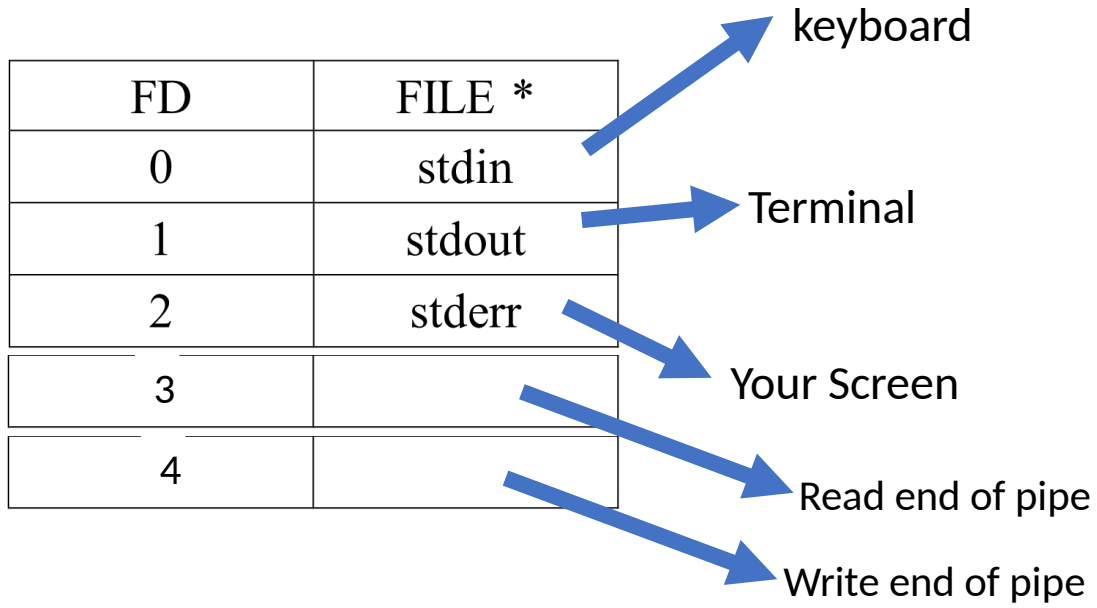
Side Note: What does the FD table look like before and after line 9?

```
9    pipe(pd);
```

BEFORE LINE 9:



AFTER LINE 9:



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

int main(){
    //create the pipe
    int pd[2];
    pipe(pd);
    pid_t pid = fork();
    int solution[2]; //variable to fill in values
```

- Call fork and create a solution variable.

```
12      if(pid == 0)
13      {
14          //set value of solution
15          solution[0] = 10;
16          solution[1] = 15;
17          close(pd[0]); //We close pd[0] since the child process doe
18          write(pd[1], &solution, sizeof(solution));
19          close(pd[1]);
20      }
```

- In the child process we set the value of solution.
- We then WRITE to the pipe using pd[1].

```
20      }
21      else{
22          printf("In the parent\n");
23          close(pd[1]); //We close pd[1] since we are in the parent
24          read(pd[0], solution , sizeof(solution));
25          printf("In parent=%d\n", solution[0]);
26          printf("In parent=%d\n", solution[1]);
27          close(pd[0]);
28      }
```

- In the parent we read from pd[0] and get solution values.
- We then print the solution values.

The Complete Code

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

int main(){
    //create the pipe
    int pd[2];
    pipe(pd);
    pid_t pid = fork();
    int solution[2]; //variable to fill i
    if(pid == 0)
    {
        //set value of solution
        solution[0] = 10;
        solution[1] = 15;
        close(pd[0]); //We close pd[0] since the child proce
        write(pd[1], &solution, sizeof(solution));
        close(pd[1]);
    }
    else{
        printf("In the parent\n");
        close(pd[1]); //We close pd[1] since we are in the parent
        read(pd[0], solution , sizeof(solution));
        printf("In parent=%d\n", solution[0]);
        printf("In parent=%d\n", solution[1]);
        close(pd[0]);
    }
    return 0;
}
```

- Create a variable (integer array of size 2) for holding the fd.
- Call pipe to set entries in the fd table.

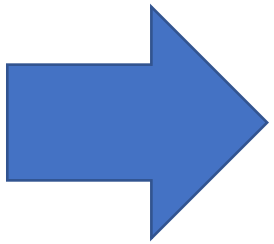
- Call fork and create a solution variable.

- In the child we fill in solution values.

- In the parent we read from pd[0] and get solution values.
- We then print the solution values.

Pipe Example Output

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

int main(){
    //create the pipe
    int pd[2];
    pipe(pd);
    pid_t pid = fork();
    int solution[2]; //variable to fill in values
    if(pid == 0)
    {
        //set value of solution
        solution[0] = 10;
        solution[1] = 15;
        close(pd[0]); //We close pd[0] since the child process does not need to read from the pipe
        write(pd[1], &solution, sizeof(solution));
        close(pd[1]);
    }
    else{
        printf("In the parent\n");
        close(pd[1]); //We close pd[1] since we are in the parent
        read(pd[0], solution , sizeof(solution));
        printf("In parent=%d\n", solution[0]);
        printf("In parent=%d\n", solution[1]);
        close(pd[0]);
    }
    return 0;
}
```



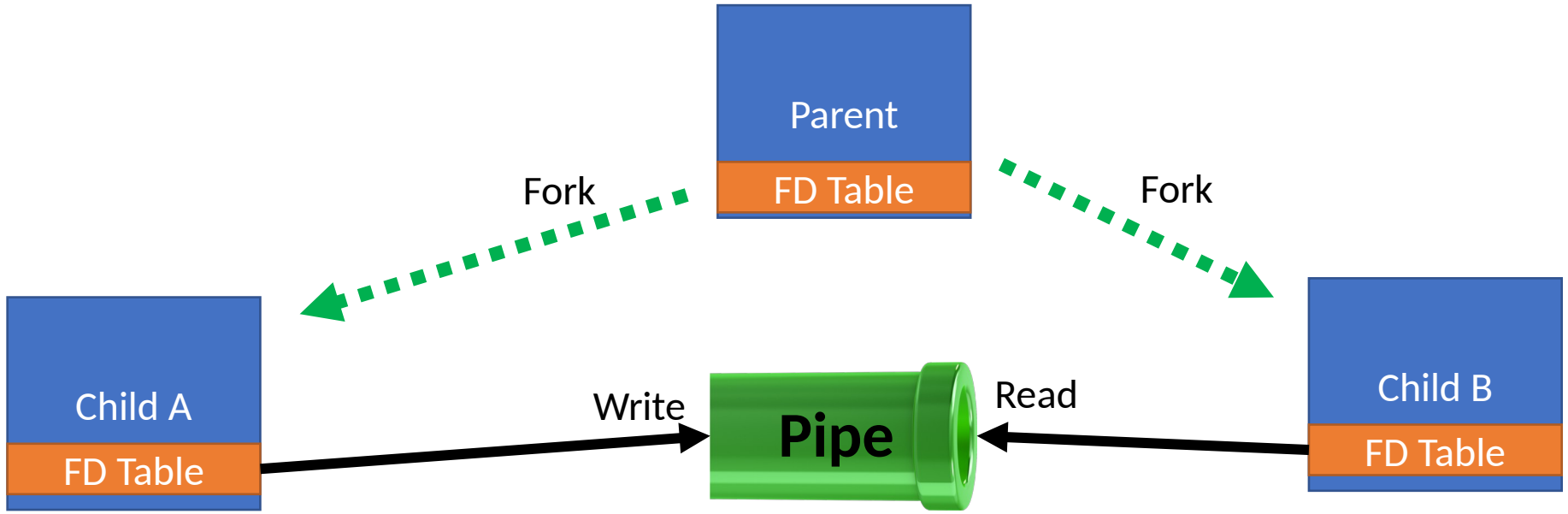
```
kaleel@CentralCompute:~$ ./test
In the parent
In parent=10
In parent=15
```

A few more notes on pipes…

- What would you do if you need two-way communications between parent and child?

- After exec, the new program gets the file descriptors for the pipe, too.
- How can the new program use the pipe?
  - ***A exe is aware of FDs 0, 1, and 2, but not 3 or 4***

Building a Pipe Setup



- Imagine we want to do the following:

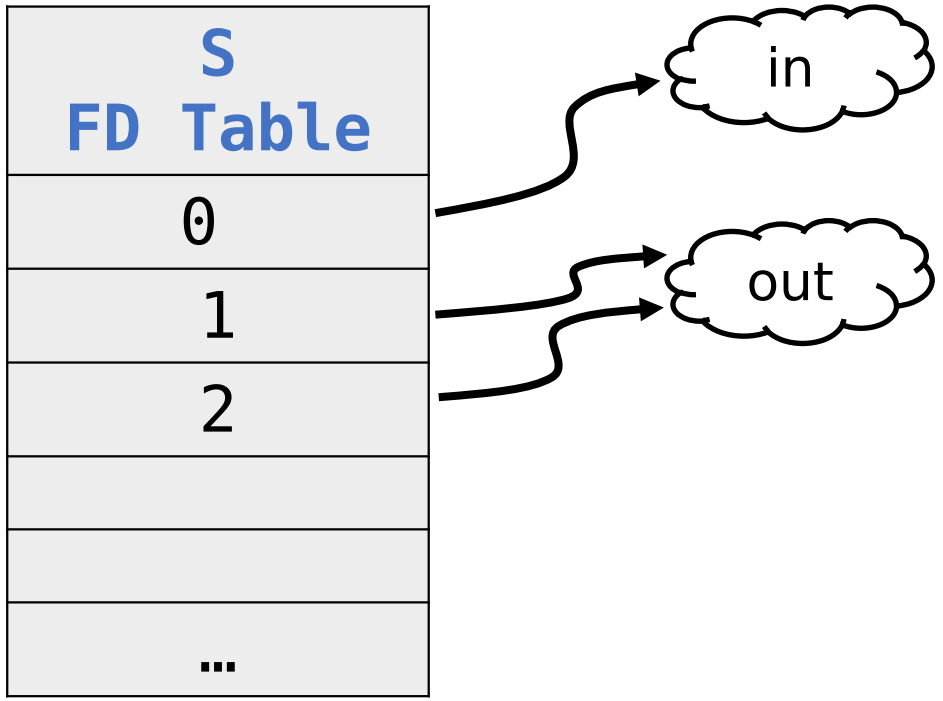
1. Have a parent process that creates two children (child A and child B).
2. Each child is going to run an executable.
3. The executable in child A is going to produce output.
4. The output of child A needs to be read as input to the executable by child B.

High Level Strategy for Pipe Setup

- High-level strategy (missing clean up !)
- Assume we want to launch two executables using two children.
  1. Create a pipe
  2. Fork #1
     - In child process
       - Redirect stdout to the write end of the pipe
       - Start A, by calling exec
  3. Fork #2
     - In child process
       - Redirect stdin to the read end of the pipe
       - Start B, by calling exec

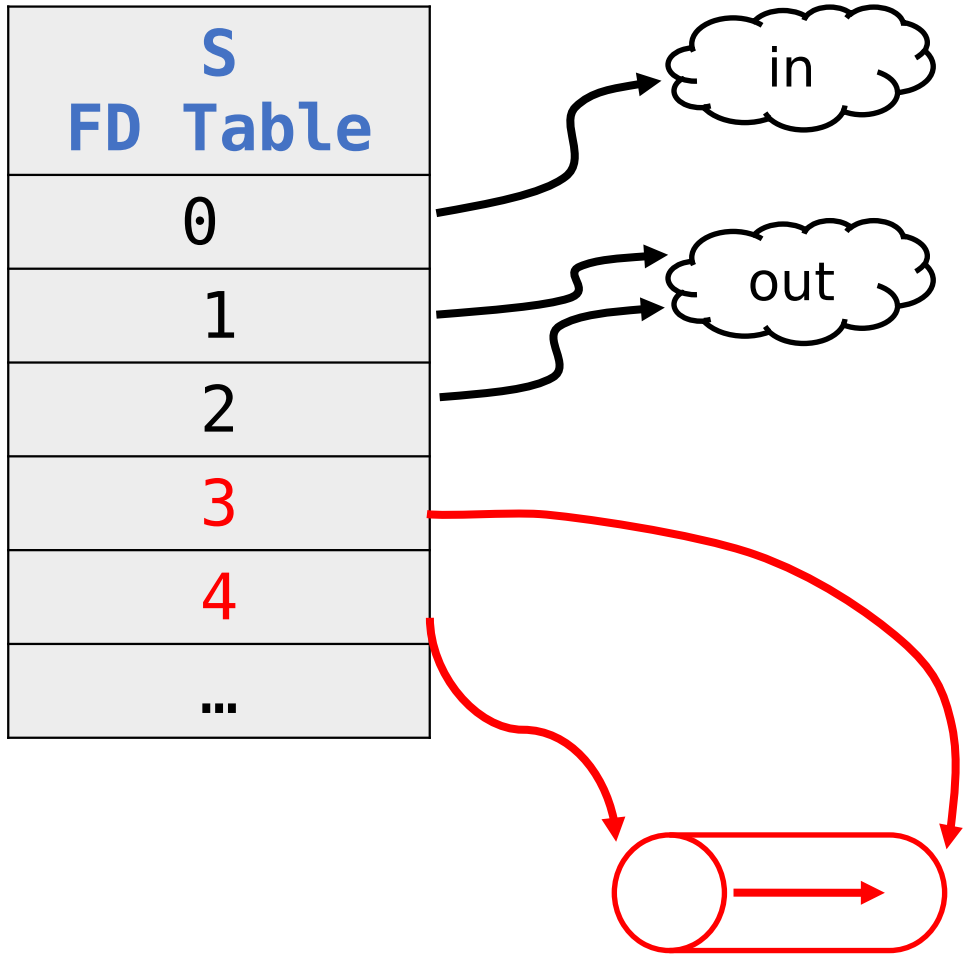
Pictorial Example

- At the beginning S has only 0, 1, and 2 open:



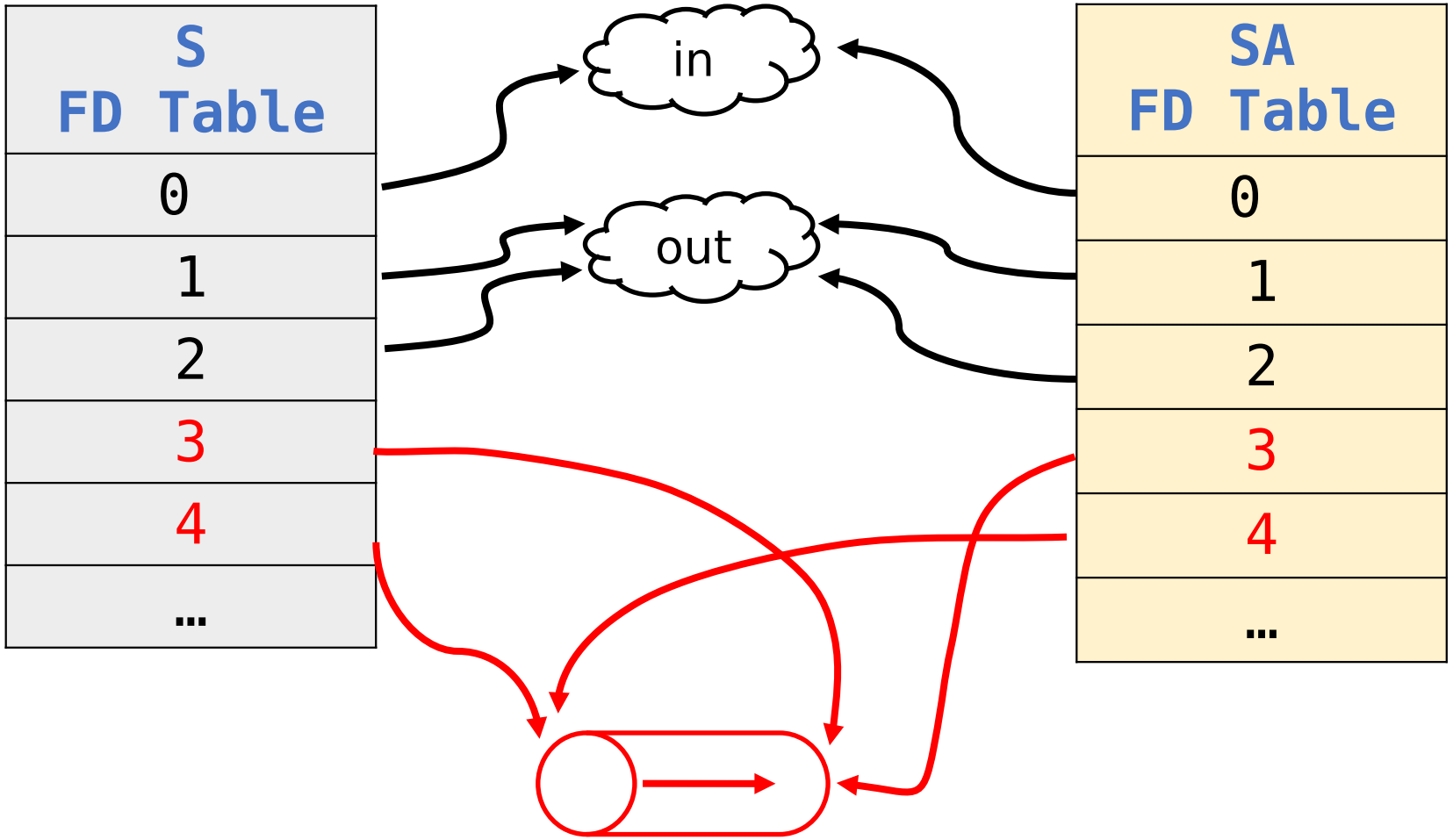
Pictorial Example: Call pipe

- S creates a pipe by calling pipe()
  - A pair of FDs is returned



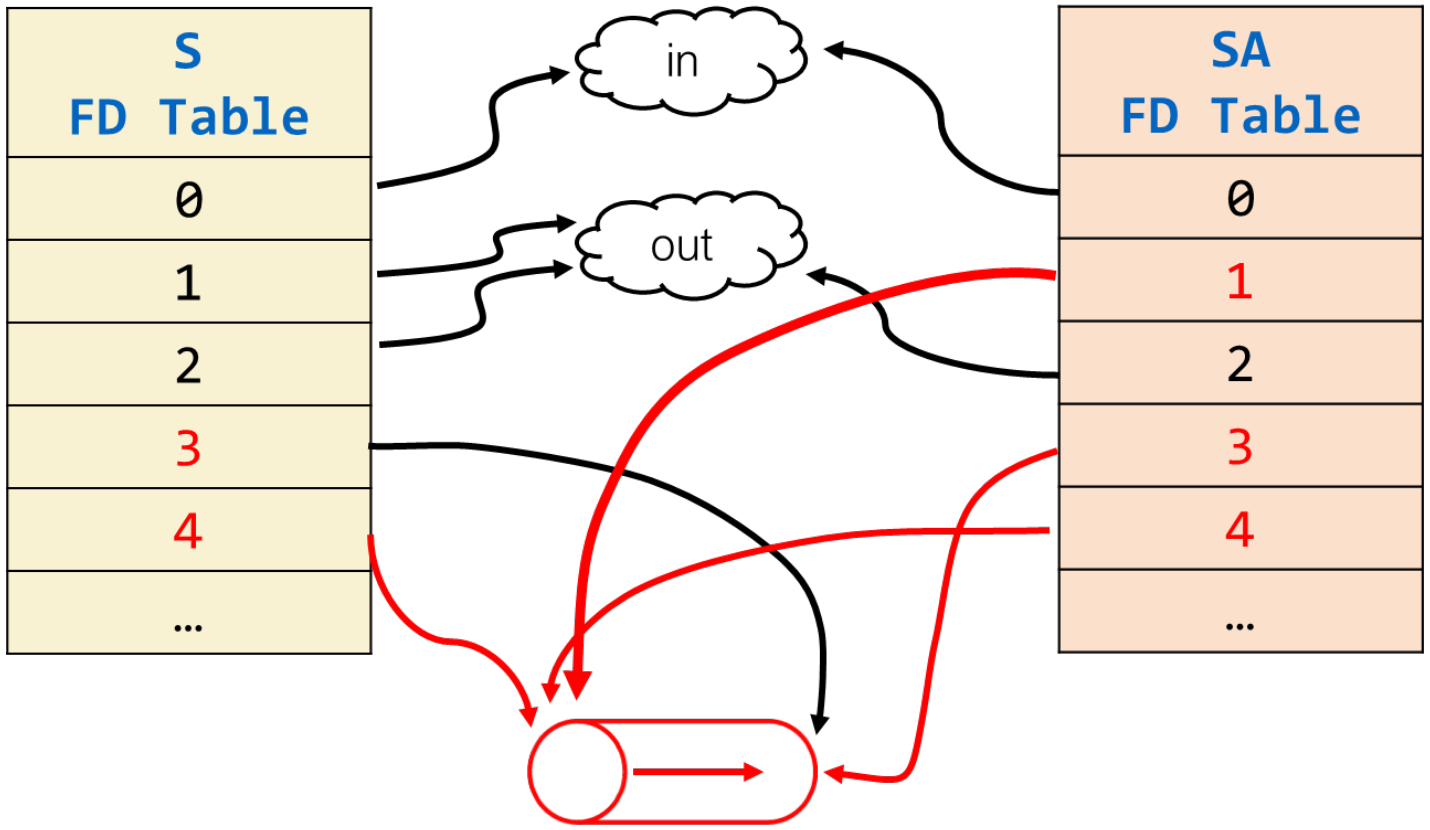
Pictorial Example: Fork

S: fork() and FD table is duplicated



Pictorial Example: Direct pipe output

SA: dup2(4, 1)
Or close(1); dup(4);

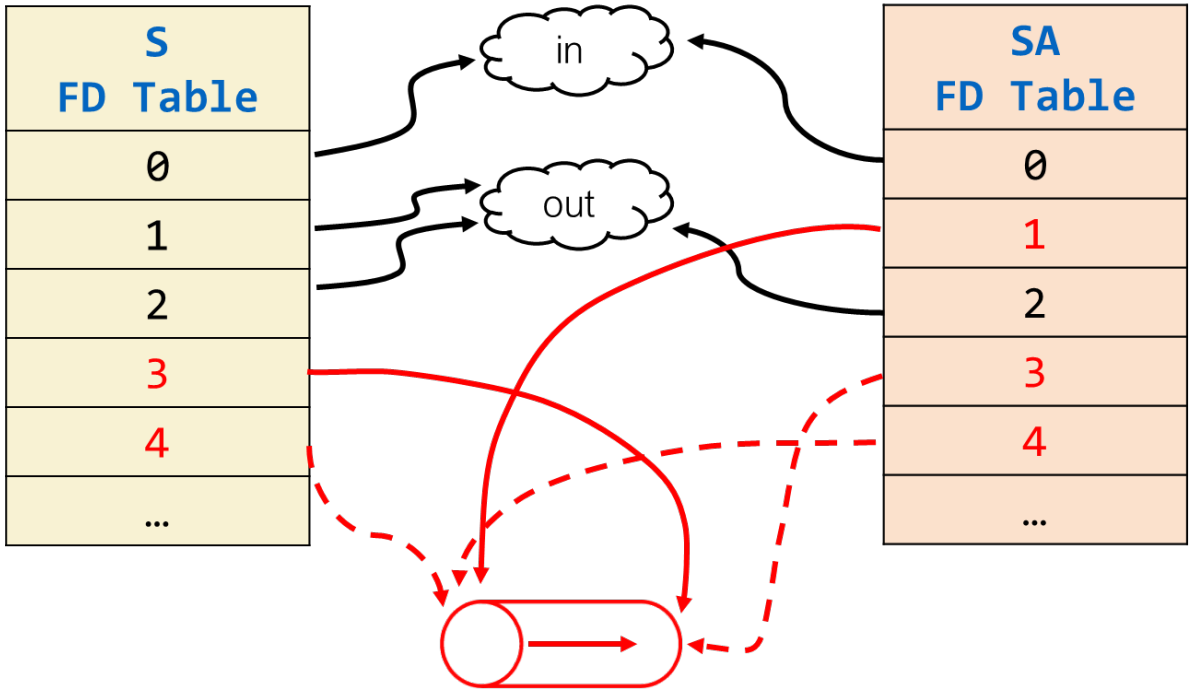


Pictorial Example: Pipe clean up #1

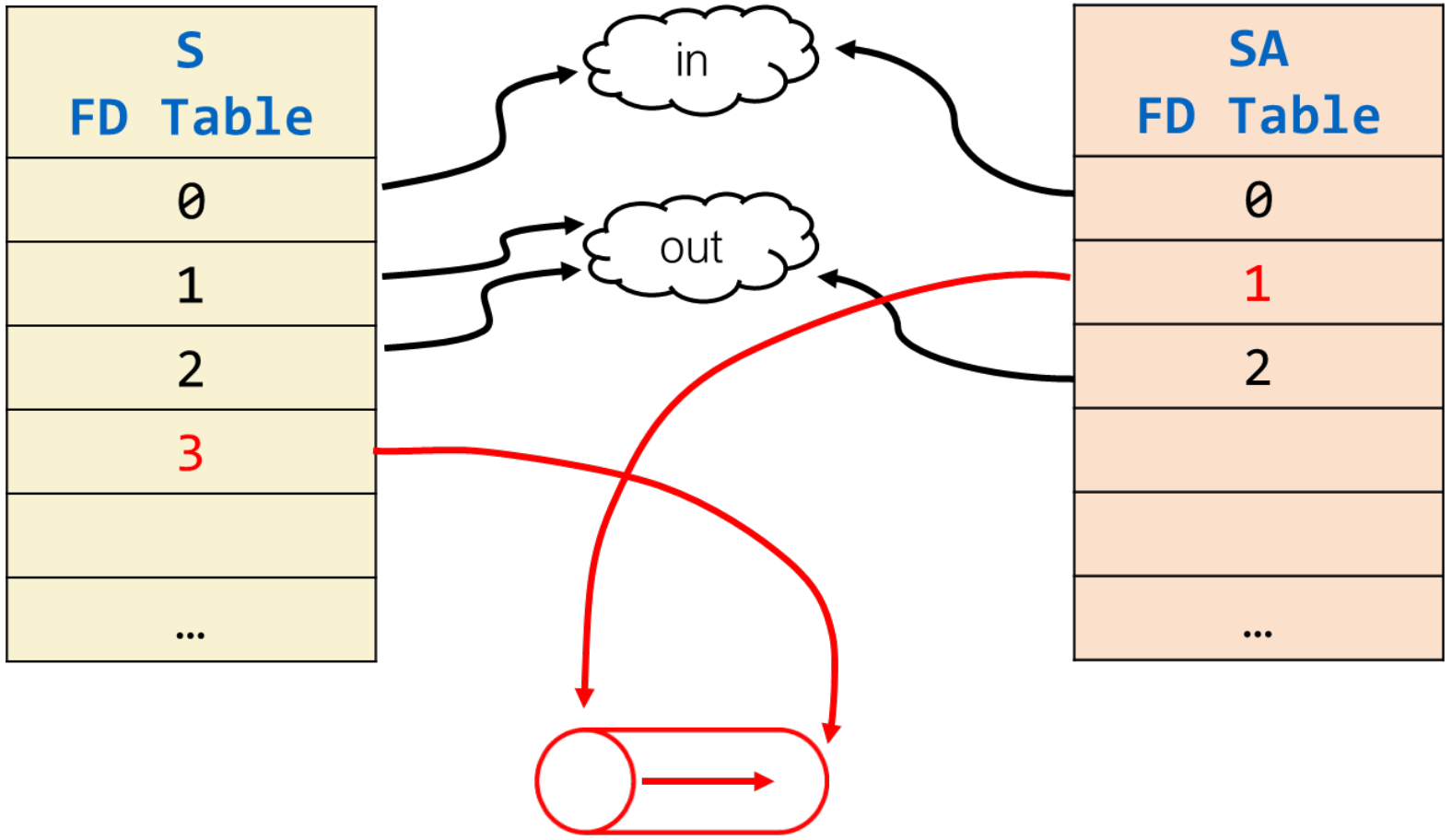
S: close(4)

SA: close(4); close(3)

SA can then exec into A

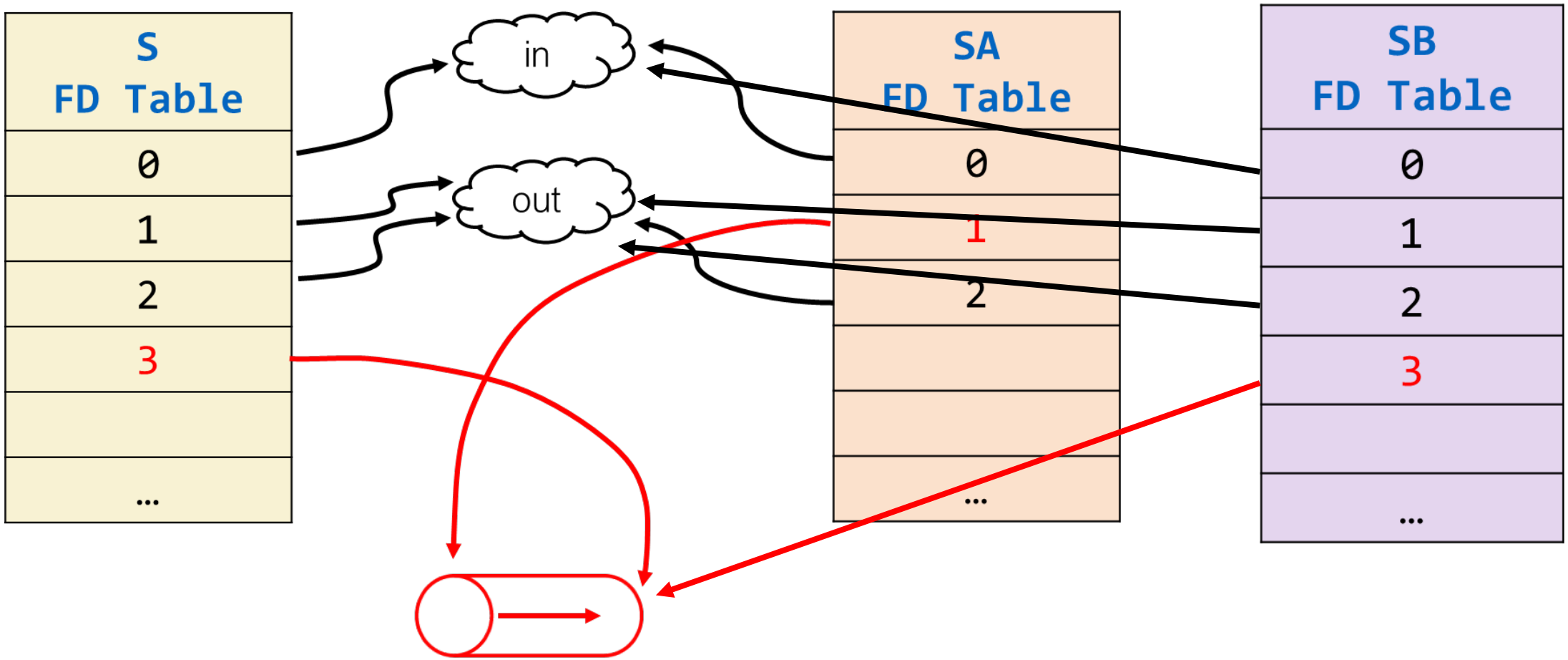


Pictorial Example: After clean up #1



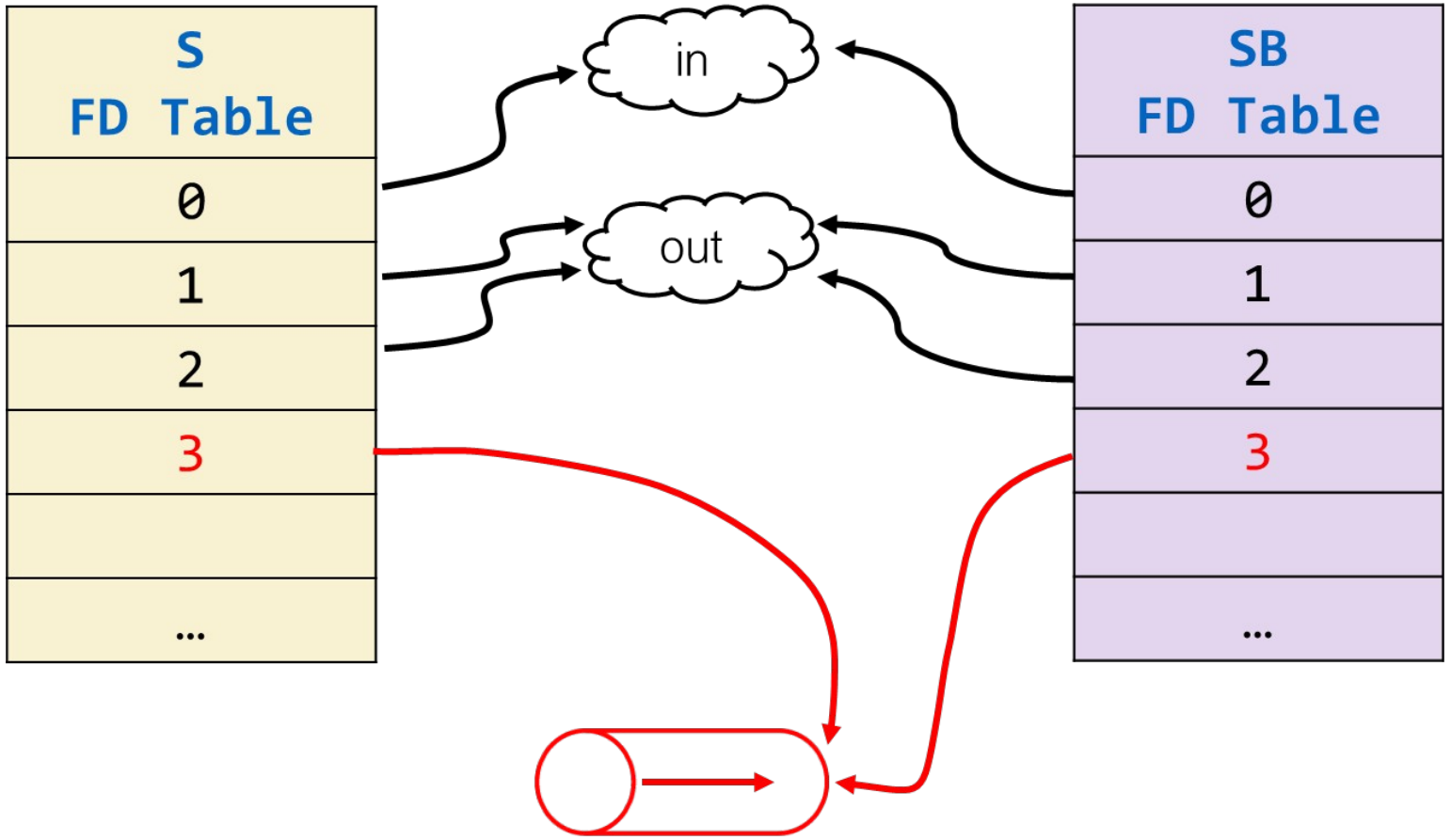
- The output of the child (SA) points to the write end of the pipe, which is what we want before launching a new executable.

Pictorial Example: Now Call Fork #2 in the parent



Let's clean up this picture a little bit by only looking at the child (SB) and the parent S…

Pictorial Example: Only looking at S and SB

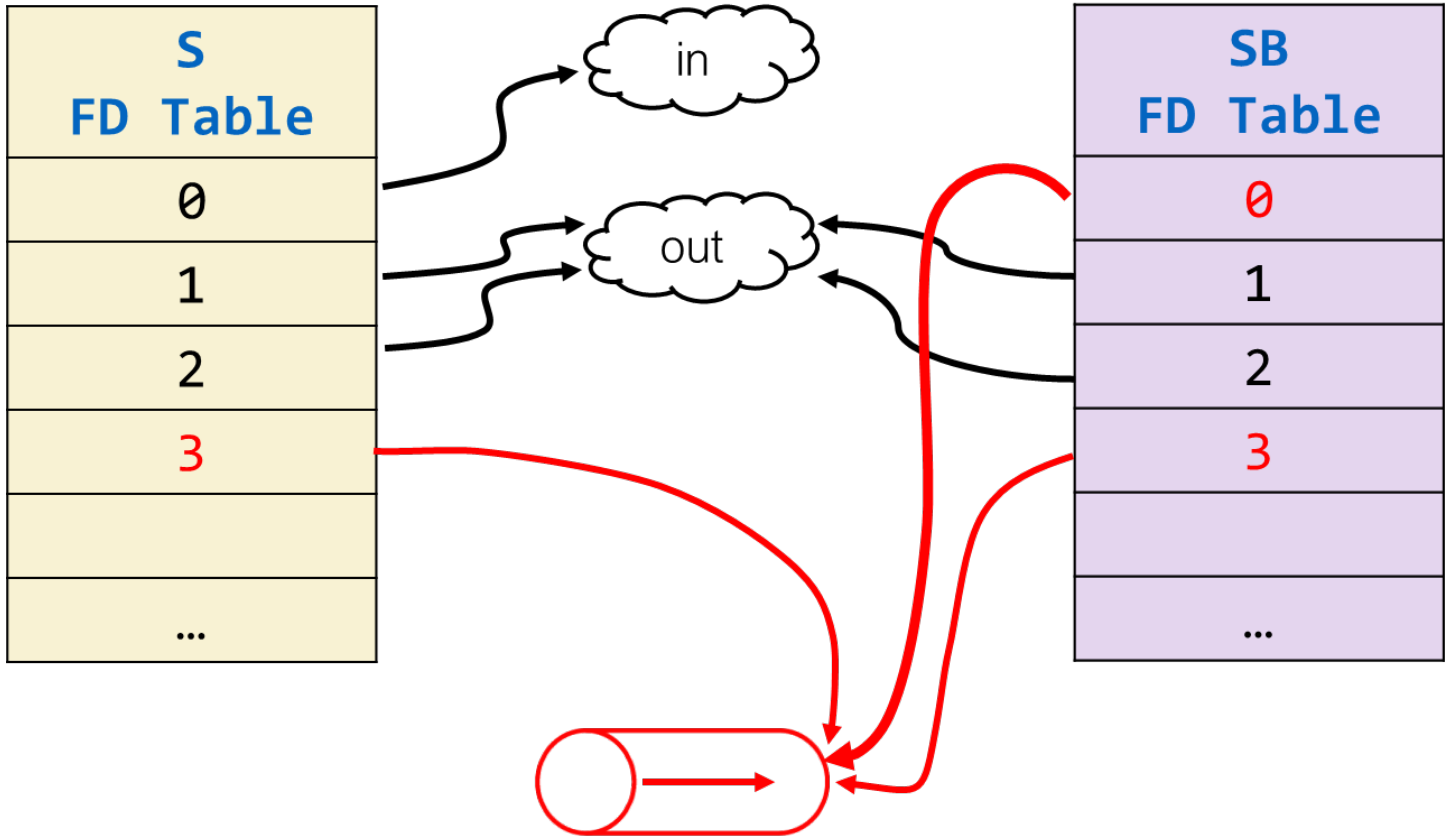


- Remember we want to run a new executable in the child which takes output from the pipe.
  - So we'll need to start changing the file descriptors in SB…

Pictorial Example: Redirect in second child process

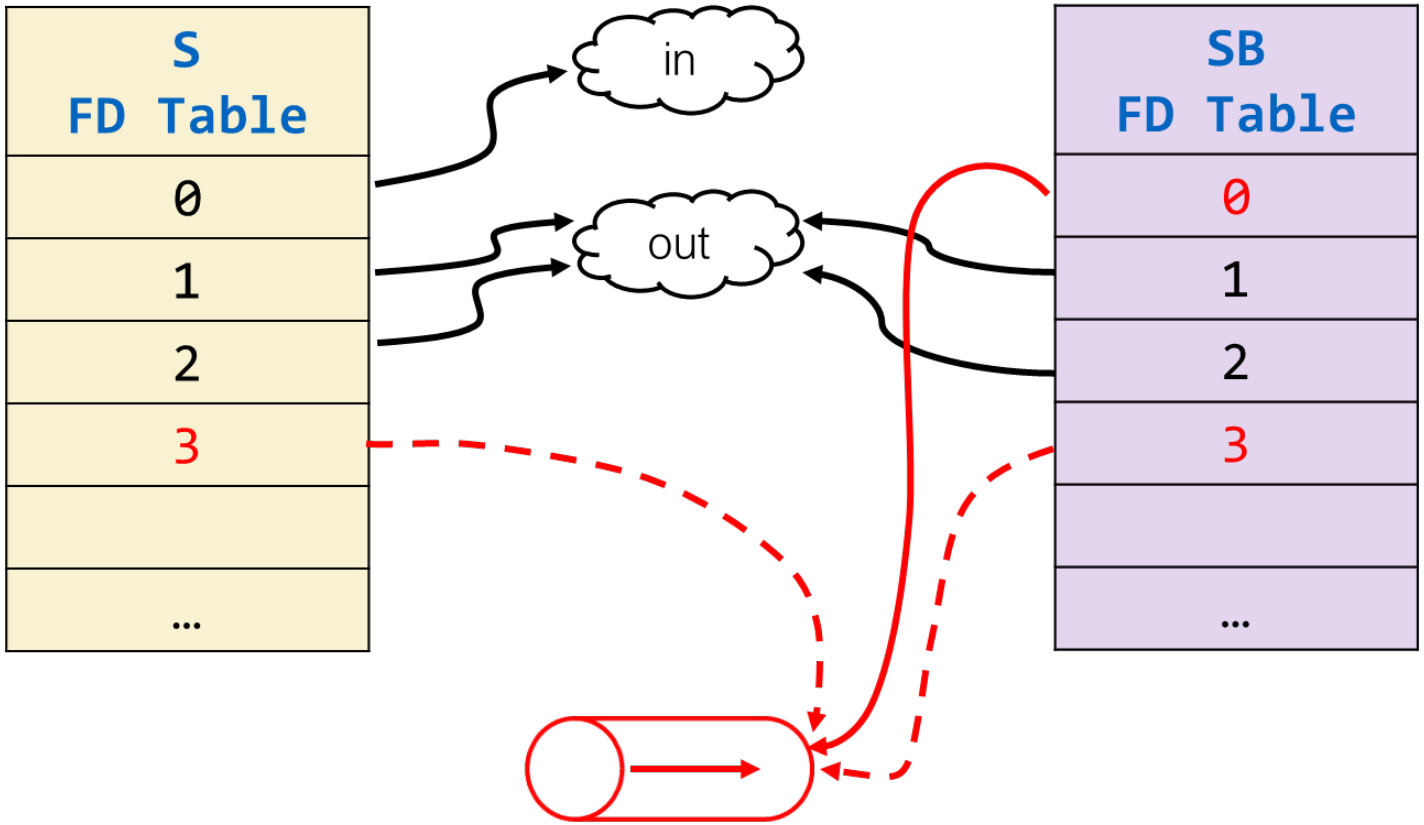
SB: dup2(3, 0)

- Or close(0); dup(3);

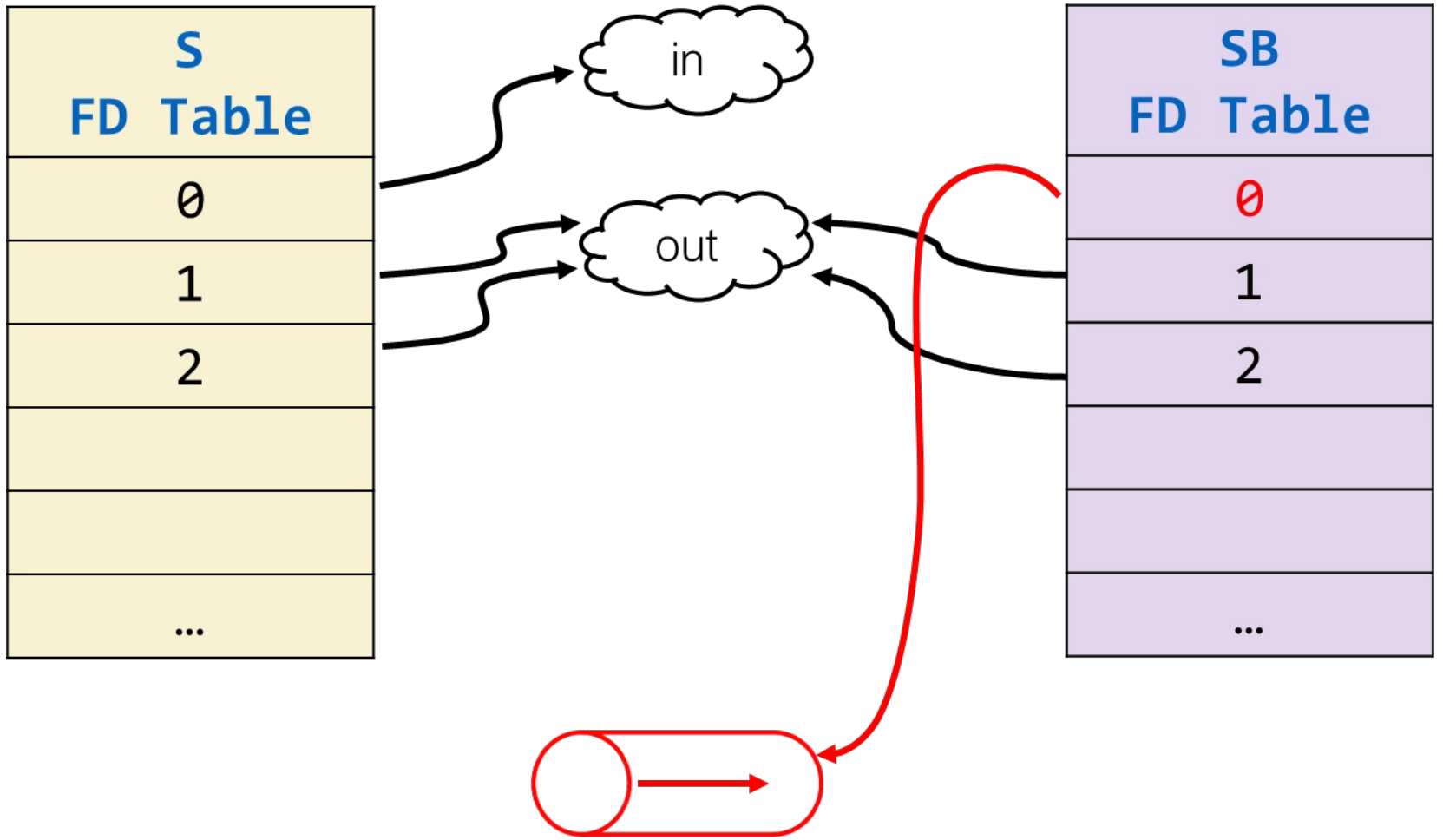


Pictorial Example: Clean up #2

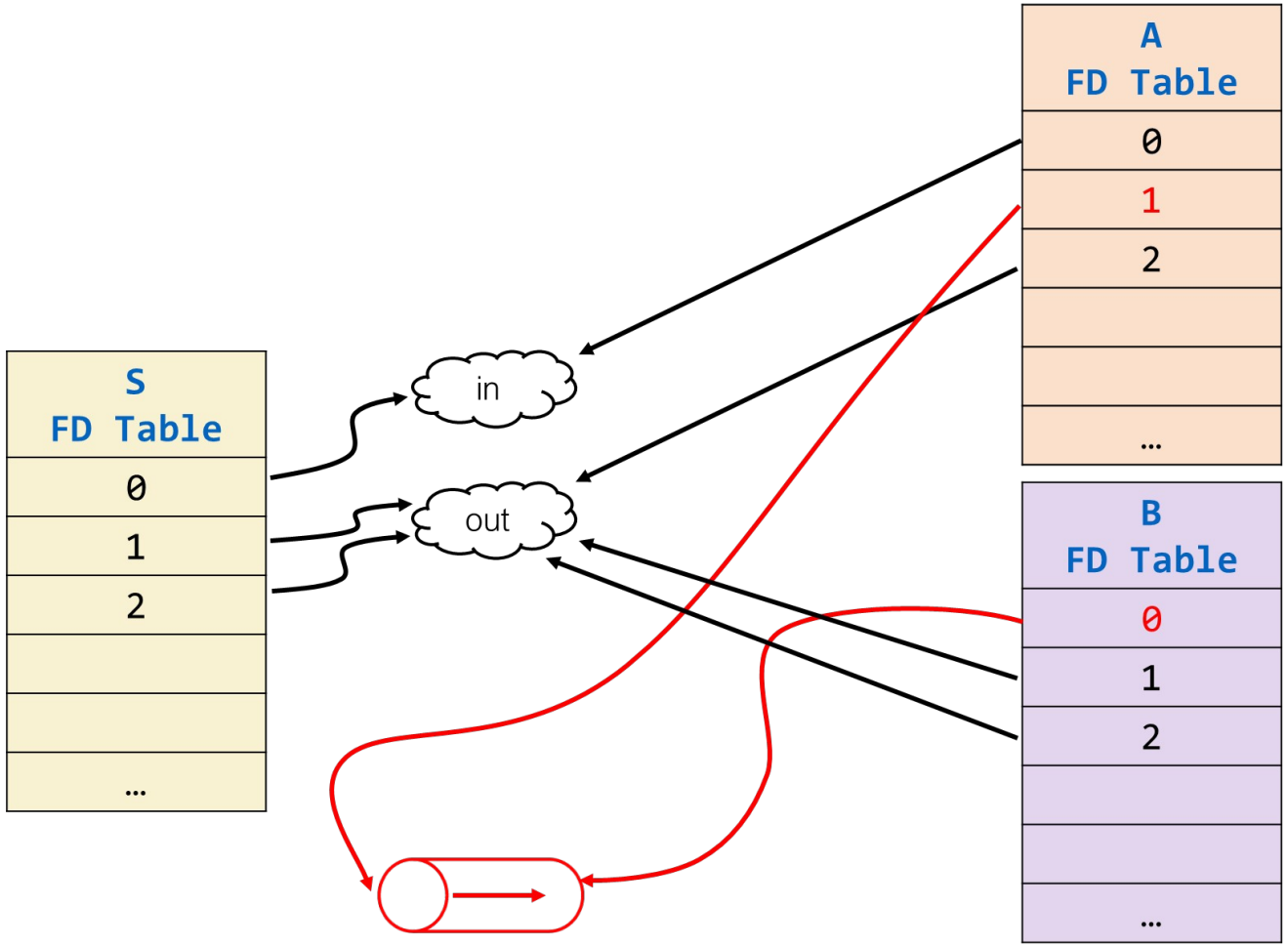
S: close(3)
SB: close(3)
SB can then exec into B



Pictorial Example: After clean up #2



Pictorial Example: Final Setup



- A writes to the pipe (using FD 1)
- B reads from the pipe (using FD 0)
- S waits

Question: *Why did we have to set fd 0 and 1 in child A and child B? Couldn't we just keep using 3 and 4?*

Executable when looking for fd 3 and fd 4

Executable when looking for fd0 and fd 1

How would we actually set this up in code?

- Before we go through the code lets take a concrete example:

The "**cat**" executable gives us text output.

The "**tr**" executable translates text.

- What we want our code to do:

1. Have a parent create two children (A and B).
2. Child A creates some text and writes it to the pipe using "cat" executable.
3. Child B reads from the pipe and uses the "tr" executable to convert the text to capital letters.

```c
int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }
```

Start in main and make sure the pipe is setup.

Parent Process FD Table

0	Std in
1	Std out
2	Std err
3	
4	

```
7   int main(int argc, char *argv[])
8   {
9       int pd[2];
10      if(pipe(pd) == -1)
11      {
12          perror("Error.");
13          return -1;
14      }
15
16      pid_t pid = fork();
17      if(pid == 0)
18      {
```

Call fork() and create a child.

Child Process FD Table

0	Std in
1	Std out
2	Std err
3	
4	

Parent Process FD Table

0	Std in
1	Std out
2	Std err
3	
4	

```c
int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
```

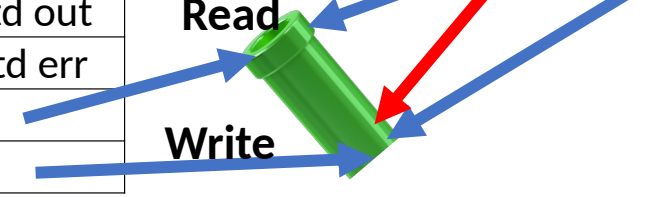
Redirect stdout of child to write end of pipe.

Child Process FD Table

0	Std in
1	Std out
2	Std err
3	
4	

Parent Process FD Table

0	Std in
1	Std out
2	Std err
3	
4	



```c
int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
        close(pd[1]);
        close(pd[0]);
```

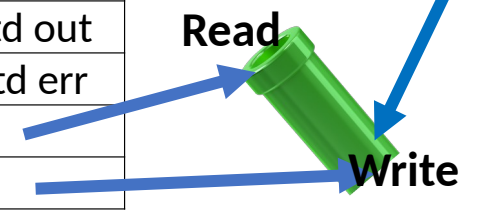
Close 3 and 4 file descriptors in the child.

Child Process FD Table

0	Std in
1	Std out
2	Std err

Parent Process FD Table

0	Std in
1	Std out
2	Std err
3	
4	



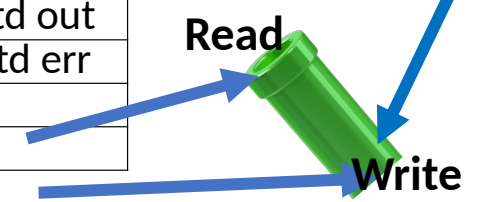
```c
int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
        close(pd[1]);
        close(pd[0]);
        char * argv_list[] = {"cat", "pipe2.c", NULL};
        execvp("cat", argv_list);
    }
```

Parent Process FD Table

0	Std in
1	Std out
2	Std err
3	
4	

Child Process FD Table

0	Std in
1	Std out
2	Std err



Child launches cat executable. This will take pipe.c and essentially turn it into text for us!
Where will cat send the text?

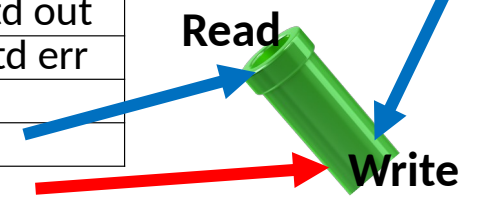
```c
int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
        close(pd[1]);
        close(pd[0]);
        char * argv_list[] = {"cat", "pipe2.c", NULL};
        execvp("cat", argv_list);
    }

    close(pd[1]);
```

Child Process FD Table

0	Std in
1	Std out
2	Std err

Parent Process FD Table

0	Std in
1	Std out
2	Std err
3	
4	



In the child we call execvp so we are not returning.
In the parent we'll close fd 4.

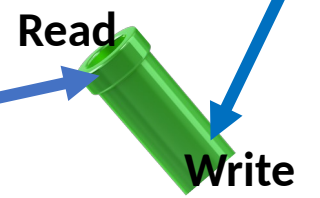
```c
int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
        close(pd[1]);
        close(pd[0]);
        char * argv_list[] = {"cat", "pipe2.c", NULL};
        execvp("cat", argv_list);
    }

    close(pd[1]);
```

Child Process FD Table

0	Std in
1	Std out
2	Std err

Parent Process FD Table

0	Std in
1	Std out
2	Std err
3	



As you can see in the picture fd 4 is gone!

```c
int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
        close(pd[1]);
        close(pd[0]);
        char * argv_list[] = {"cat", "pipe2.c", NULL};
        execvp("cat", argv_list);
    }

    close(pd[1]);


    pid_t pid1 = fork();
    if(pid1 == 0)
```

Child Process FD Table

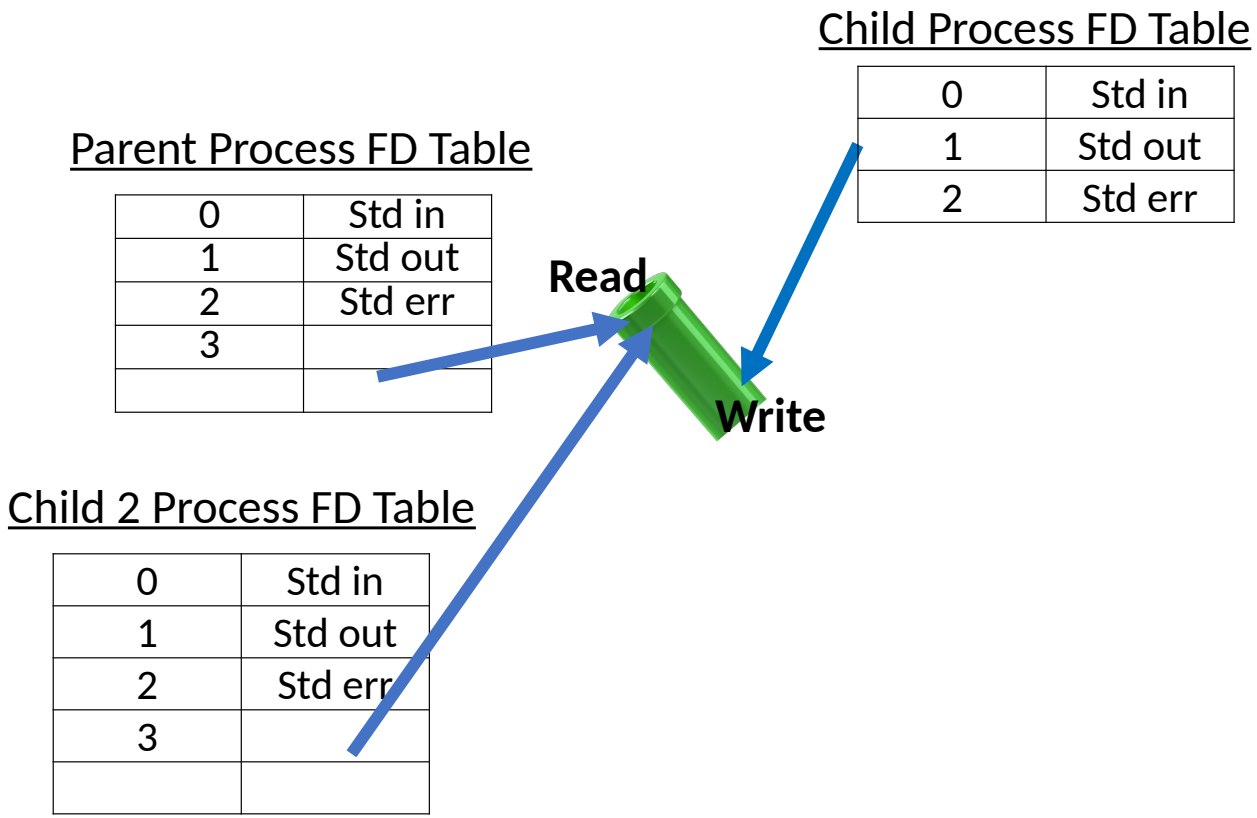
0	Std in
1	Std out
2	Std err

Parent Process FD Table

0	Std in
1	Std out
2	Std err
3	

Now time to call fork in the parent and create a new child.

```c
int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
        close(pd[1]);
        close(pd[0]);
        char * argv_list[] = {"cat", "pipe2.c", NULL};
        execvp("cat", argv_list);
    }

    close(pd[1]);


    pid_t pid1 = fork();
    if(pid1 == 0)
```



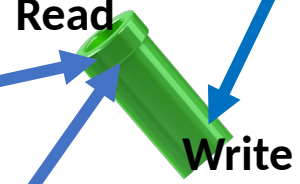
```c
int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
        close(pd[1]);
        close(pd[0]);
        char * argv_list[] = {"cat", "pipe2.c", NULL};
        execvp("cat", argv_list);
    }

    close(pd[1]);


    pid_t pid1 = fork();
    if(pid1 == 0)
    {
        dup2(pd[0], 0);
```

Child Process FD Table

0	Std in
1	Std out
2	Std err

Parent Process FD Table

0	Std in
1	Std out
2	Std err
3	

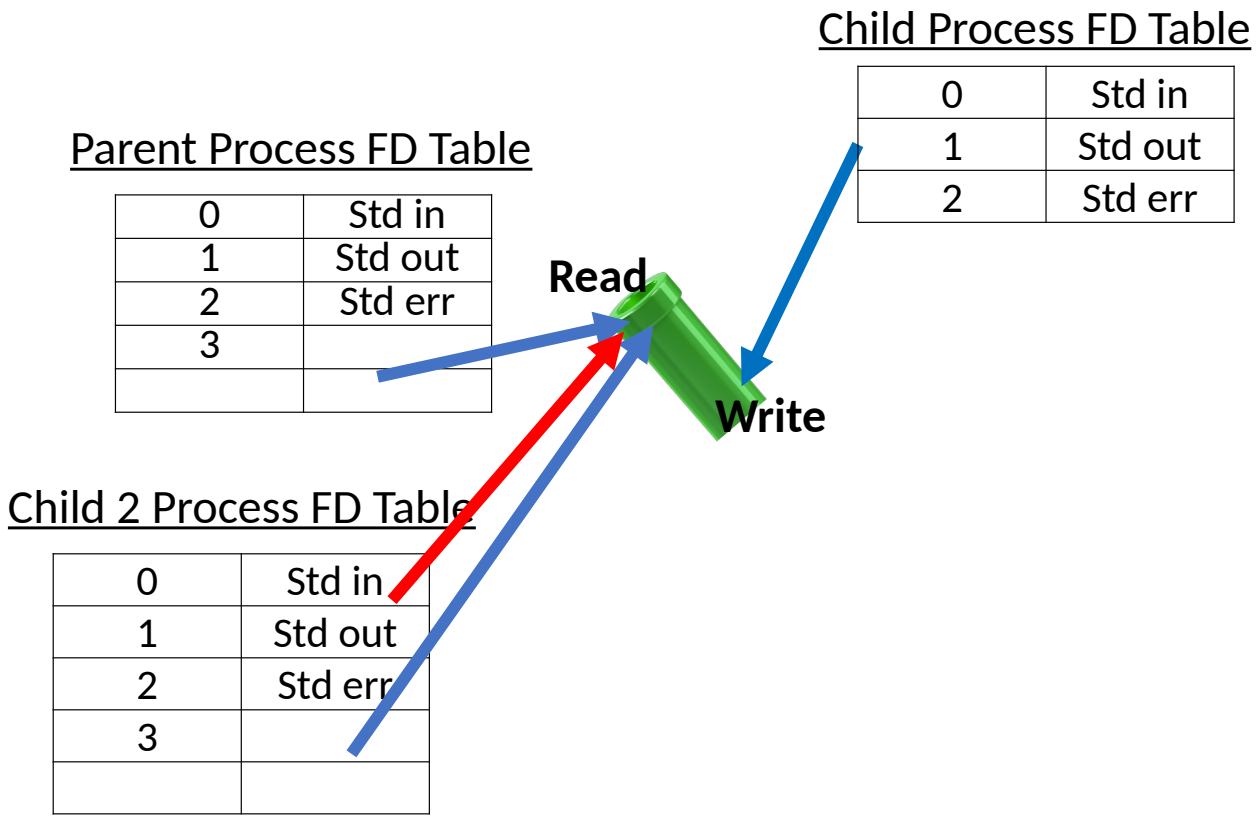


Child 2 Process FD Table

0	Std in
1	Std out
2	Std err
3	

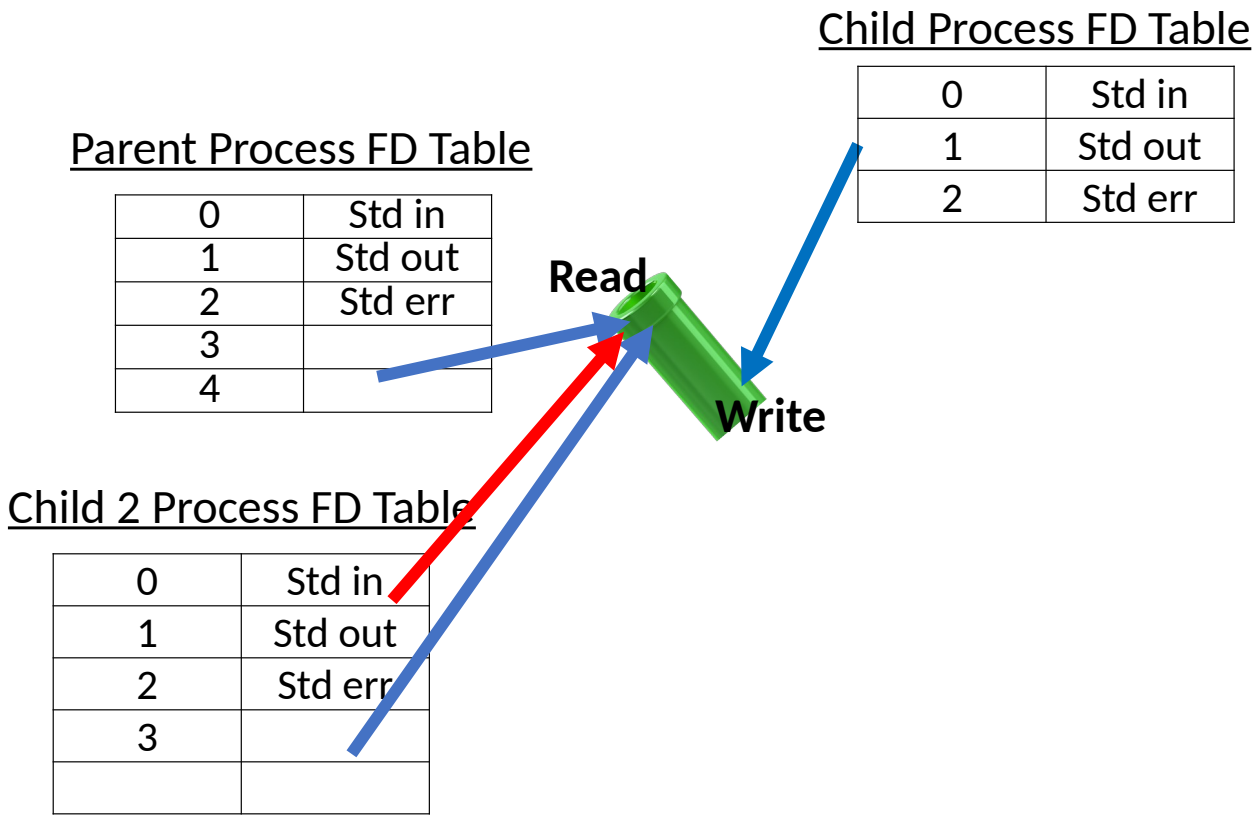
We want this process to take input from the pipe as fd 0, so call dup2() to make it happen.

```c
int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
        close(pd[1]);
        close(pd[0]);
        char * argv_list[] = {"cat", "pipe2.c", NULL};
        execvp("cat", argv_list);
    }

    close(pd[1]);


    pid_t pid1 = fork();
    if(pid1 == 0)
    {
        dup2(pd[0], 0);
```



We want this process to take input from the pipe as fd 0, so call dup2() to make it happen.

```c
int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
        close(pd[1]);
        close(pd[0]);
        char * argv_list[] = {"cat", "pipe2.c", NULL};
        execvp("cat", argv_list);
    }

    close(pd[1]);


    pid_t pid1 = fork();
    if(pid1 == 0)
    {
        dup2(pd[0], 0);
        close(pd[0]);
```



Close non-needed file descriptor to read end of the pipe.

```c
int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
        close(pd[1]);
        close(pd[0]);
        char * argv_list[] = {"cat", "pipe2.c", NULL};
        execvp("cat", argv_list);
    }

    close(pd[1]);


    pid_t pid1 = fork();
    if(pid1 == 0)
    {
        dup2(pd[0], 0);
        close(pd[0]);
```

Child Process FD Table

0	Std in
1	Std out
2	Std err

Parent Process FD Table

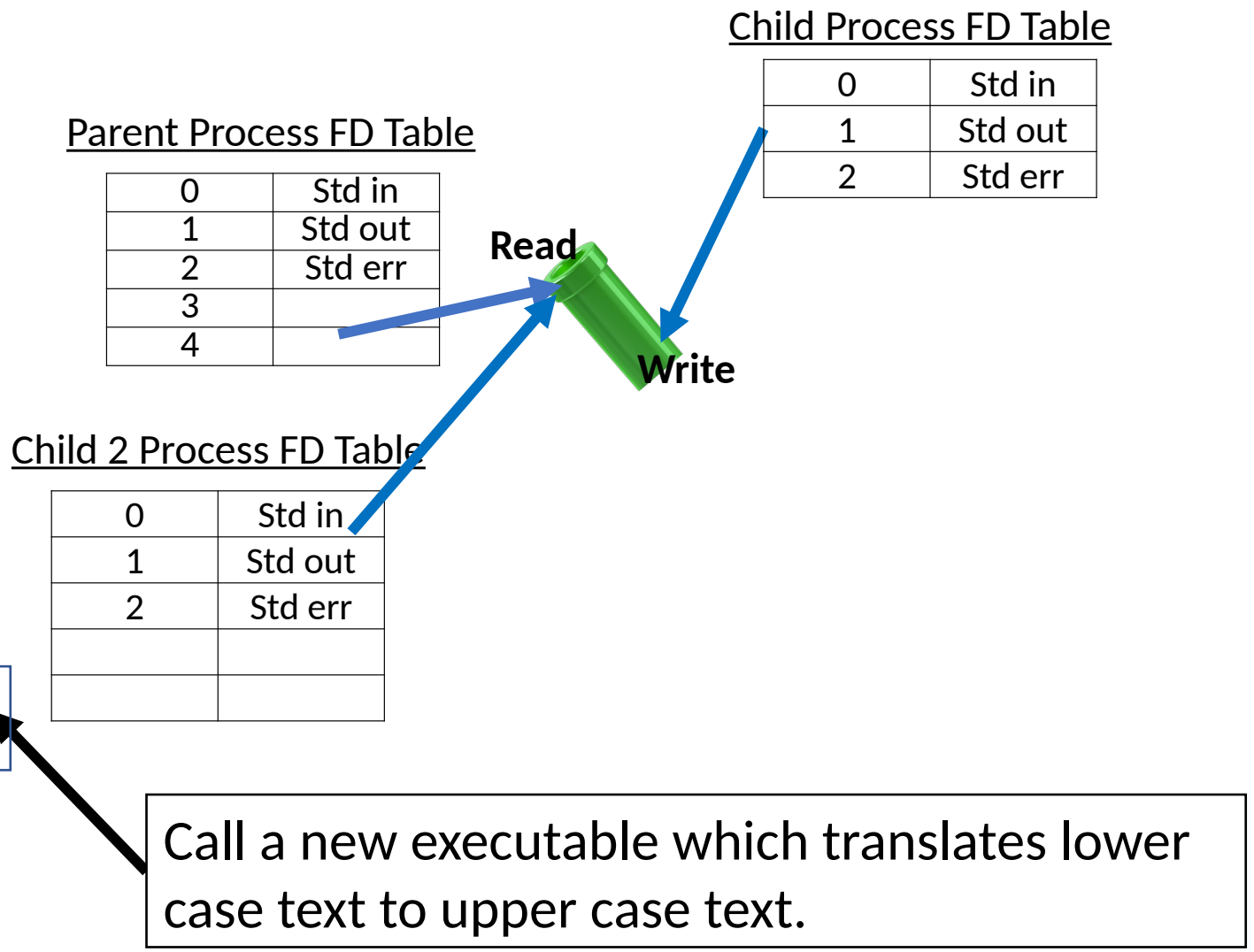
0	Std in
1	Std out
2	Std err
3	
4	

Child 2 Process FD Table

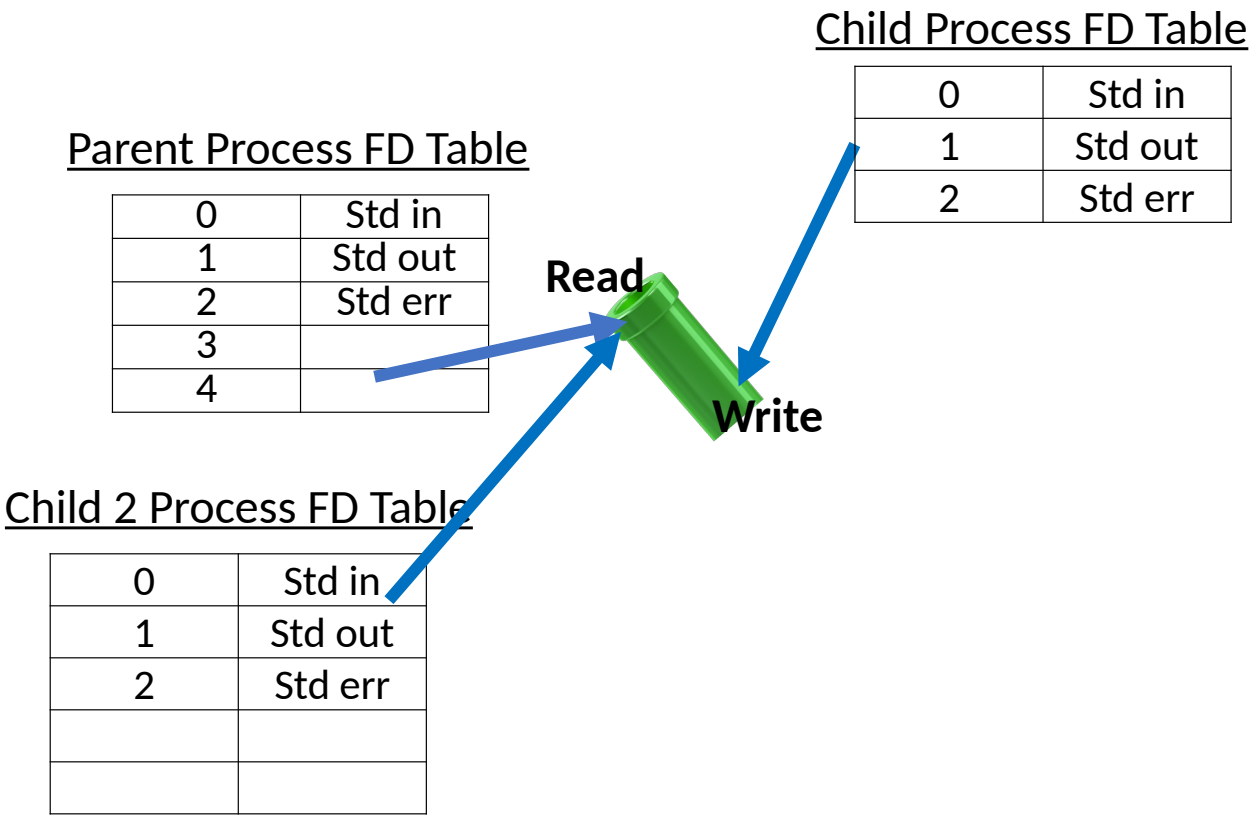
0	Std in
1	Std out
2	Std err

Close non-needed file descriptor to read end of the pipe.

```c
int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
        close(pd[1]);
        close(pd[0]);
        char * argv_list[] = {"cat", "pipe2.c", NULL};
        execvp("cat", argv_list);
    }

    close(pd[1]);


    pid_t pid1 = fork();
    if(pid1 == 0)
    {
        dup2(pd[0], 0);
        close(pd[0]);
        char * argv_list[] = {"tr", "[a-z]", "[A-Z]", NULL};
        execvp("tr", argv_list);
    }
```



```c
int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
        close(pd[1]);
        close(pd[0]);
        char * argv_list[] = {"cat", "pipe2.c", NULL};
        execvp("cat", argv_list);
    }

    close(pd[1]);


    pid_t pid1 = fork();
    if(pid1 == 0)
    {
        dup2(pd[0], 0);
        close(pd[0]);
        char * argv_list[] = {"tr", "[a-z]", "[A-Z]", NULL};
        execvp("tr", argv_list);
    }

    close(pd[0]);

    waitpid(pid, NULL, 0);
    waitpid(pid1, NULL, 0);
    return 0;
}
```



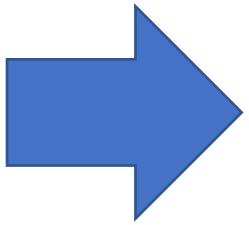
Close non-needed file descriptors in the parent.
(not pictured here)
Wait for the two processes to finish before returning.

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
        close(pd[1]);
        close(pd[0]);
        char * argv_list[] = {"cat", "pipe2.c", NULL};
        execvp("cat", argv_list);
    }

    close(pd[1]);


    pid_t pid1 = fork();
    if(pid1 == 0)
    {
        dup2(pd[0], 0);
        close(pd[0]);
        char * argv_list[] = {"tr", "[a-z]", "[A-Z]", NULL};
        execvp("tr", argv_list);
    }

    close(pd[0]);

    waitpid(pid, NULL, 0);
    waitpid(pid1, NULL, 0);
    return 0;
}
```

Full Code
(Try it on your own!)

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int pd[2];
    if(pipe(pd) == -1)
    {
        perror("Error.");
        return -1;
    }

    pid_t pid = fork();
    if(pid == 0)
    {
        dup2(pd[1], 1);
        close(pd[1]);
        close(pd[0]);
        char * argv_list[] = {"cat", "pipe2.c", NULL};
        execvp("cat", argv_list);
    }

    close(pd[1]);


    pid_t pid1 = fork();
    if(pid1 == 0)
    {
        dup2(pd[0], 0);
        close(pd[0]);
        char * argv_list[] = {"tr", "[a-z]", "[A-Z]", NULL};
        execvp("tr", argv_list);
    }

    close(pd[0]);

    waitpid(pid, NULL, 0);
    waitpid(pid1, NULL, 0);
    return 0;
}
```

Sample Output:



```
kaleel@CentralCompute:~$ gcc pipe2.c -o test
kaleel@CentralCompute:~$ ./test
#INCLUDE <STDIO.H>
#INCLUDE <UNISTD.H>
#INCLUDE <FCNTL.H>
#INCLUDE <SYS/WAIT.H>
#INCLUDE <STRING.H>

INT MAIN(INT ARGC, CHAR *ARGV[])
{
    INT PD[2];
    IF(PIPE(PD) == -1)
    {
        PERROR("ERROR.");
        RETURN -1;
    }

    PID_T PID = FORK();
    IF(PID == 0)
    {
        DUP2(PD[1], 1);
        CLOSE(PD[1]);
        CLOSE(PD[0]);
```

Going further…

- You can repeat this to create a long pipeline
  - E.g., connect B's stdout to stdin of another process C
- Draw pictures to find out how pipes are used
  - And what FDs need to be closed

Remember

- Processes are running in parallel once they are created
  - Although we showed the operations in sequence
- All processes in the pipeline are running concurrently on Linux
  - As soon as data are sent in the pipe…
  - The next process can pick them up and start to work

Slides To Review Yourself

Atomicity of read() and write()

```
nr = read(fd, buf, N);
nw = write(fd, buf, N);
```

write() and read() returns the number of bytes actually read/written

The returned values may be less than the requested

- Aotmicity of write () is guaranteed if the number of bytes is less than PIPE_BUF
  - The bytes will be consecutive
  - The default value of PIPE_BUF is 4096 on Linux
- For read(), it is fine if all writes and reads are of the same size
  - Otherwise, need special handling

Starting a 2-stage pipeline - 1

```
// A | B

pipe(pipefd)       // pipefd is an array of 2 int's
pid_a = fork()        // for A
if (pid_a == 0) {     // child process for A
    dup2();           // setup stdout for A
    close both FDs in pipefd
    exec to start A // remember to exit from child on error
}
close(pipefd[WR_END]); // No need to keep it open in parent
```

Starting a 2-stage pipeline - 2

```
pid_b = fork()          // for B
if (pid_b == 0) {       // child process for B
    dup2();             // setup stdin for B
    close(pipefd[RD_END]);
    exec to start B // remember to exit from child on error
}
close(pipefd[RD_END]); // No need to keep it open in parent

// Add code to check return value for errors!
```

Using Pipes to Sum Matrix Rows Concurrently

See the complete code in the demo repo.

```
int main(void)
{
   int i, row_sum, sum = 0, pd[2], a[N][N] = {{1, 1, 1}, {2, 2, 2}, {3, 3, 3}};

   if (pipe(pd) == -1) error_exit("pipe() failed");  /* create pipe */

   for (i = 0; i < N; ++i)
      if (fork() == 0) {            /* create a child process for each row */
         row_sum = add_vector(a[i]); /* compute the sum of a row */
         if (write(pd[1], &row_sum, sizeof(int)) == -1) /* write to pipe */
            error_exit("write() failed");
         return 0;                                  /* exit from child */
      }
   /* better to close the write end in the parent */
   for (i = 0; i < N; ++i) {
      if (read(pd[0], &row_sum, sizeof(int)) == -1) /* read from pipe */
         error_exit("read() failed");
      sum += row_sum;                                  /* calculate the
total */
   }
   printf("Sum of the array = %d\n", sum);
   /* wait for child processes*/
```

Figure Sources

1. https://i.kym-cdn.com/photos/images/newsfeed/001/273/780/f05.png
2. http://s3.amazonaws.com/pix.iemoji.com/images/emoji/apple/ios-12/256/face-screaming-in-fear.png
3. https://ychef.files.bbci.co.uk/976x549/p03lcphh.jpg
4. https://i.kym-cdn.com/entries/icons/original/000/023/987/overcome.jpg
5. https://mario.wiki.gallery/images/thumb/f/f0/Warp_Pipe_Artwork_-_Super_Mario_3D_World.png/800px-Warp_Pipe_Artwork_-_Super_Mario_3D_World.png
6. https://i.kym-cdn.com/entries/icons/original/000/022/524/tumblr_o16n2kBlpX1ta3qyvo1_1280.jpg
7. https://www.indiewire.com/wp-content/uploads/2018/12/Screen-Shot-2018-12-27-at-10.28.15-AM.png?w=780
8. https://i.pinimg.com/originals/32/c5/23/32c52334d4bdabde17a4743e086c1ae5.jpg
9. https://pbs.twimg.com/media/D4C4-rwWsAEgv-Ling