P4: Inter-Process Communication with Pipes (ABC 12.3)

Ion Mandoiu
Laurent Michel
Revised by M. Khan and J. Shi

Inter-process communication (IPC)

- Files
- **Pipes**
- Named pipes
- Sockets
- Message queues
- Shared memory

Synchronization primitives

- Semaphores, Signals, etc.

pipe()

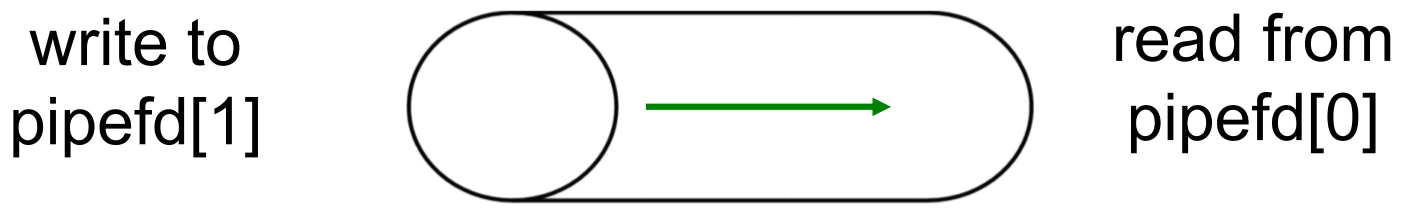
```
#include <unistd.h>
int  pipe(int pipefd[2]);
```

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

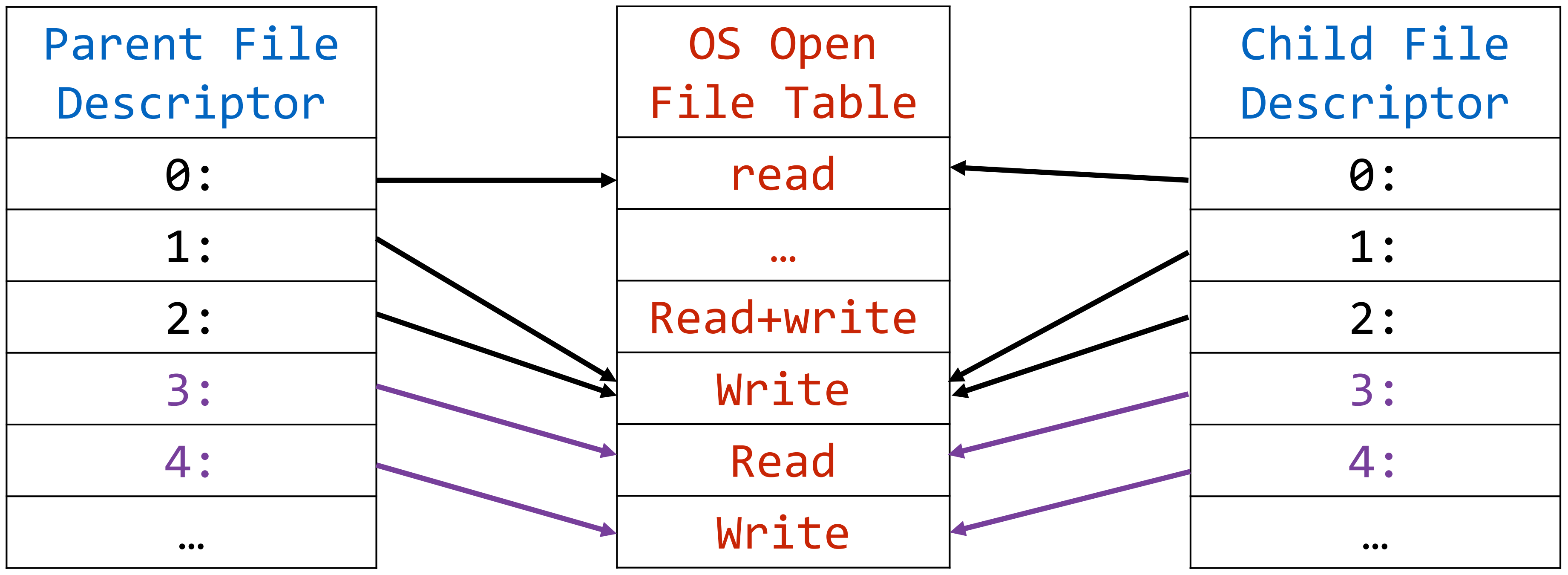
Returns 0 if successful

Pipes allow IPC. One process writes and the other one reads



Connecting two processes

- Parent creates a pipe and gets two FDs (e.g., 3 and 4)
- After fork(), the child has 3 and 4, too
- One process can write to FD 3, and the other can read from FD 4
  - Close unused FD!



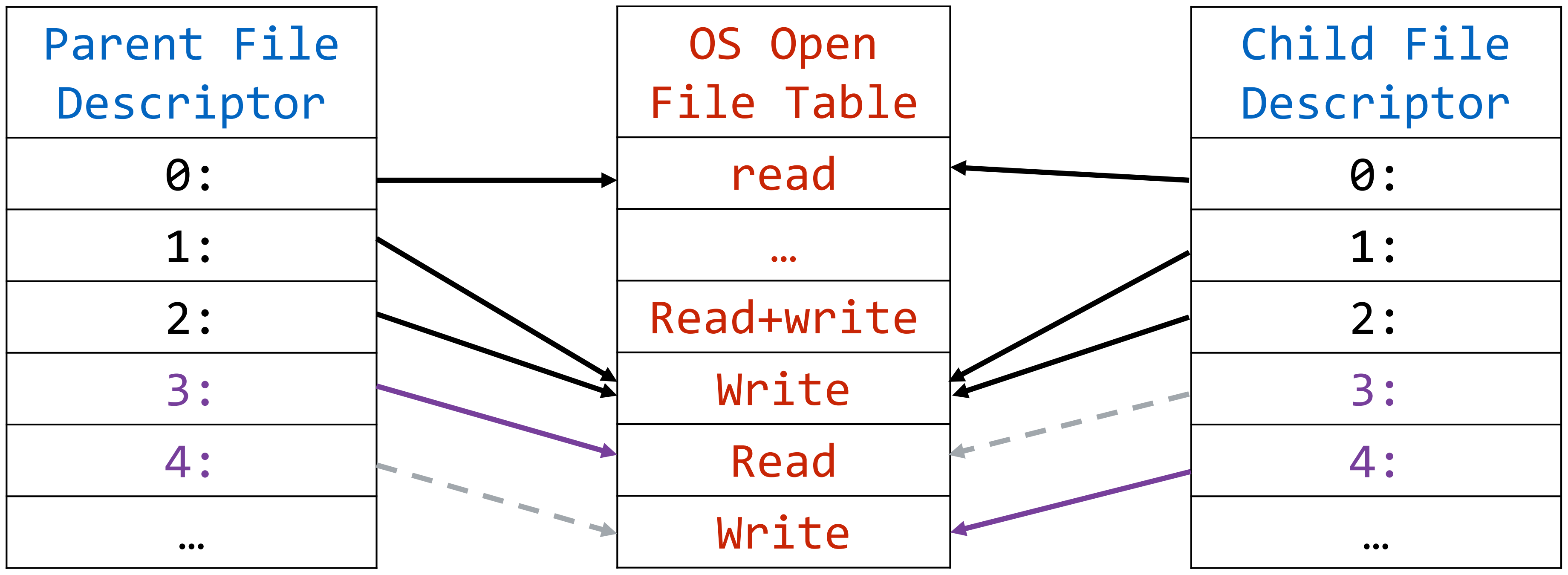
Closing FDs not in use

If the pipe is for parent to read and for child to write,

Parent: close(4);

Child: close(3);

Then child can write to and parent can read from the pipe. See demo code!



Questions

- 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 program is aware of FDs 0, 1, and 2, but not 3 or 4

Pipeline in shell

- Shell supports pipelines

```
cmd1  | cmd2 arg21 arg22 | cmd3 arg31 …
```

- stdout of a command is connected to stdin of the next command
  - Done with pipes on Linux/Unix
  - cmd1 writes to a pipe and cmd2 reads from it

Example:

```
ls | tr a-z A-Z | wc
```

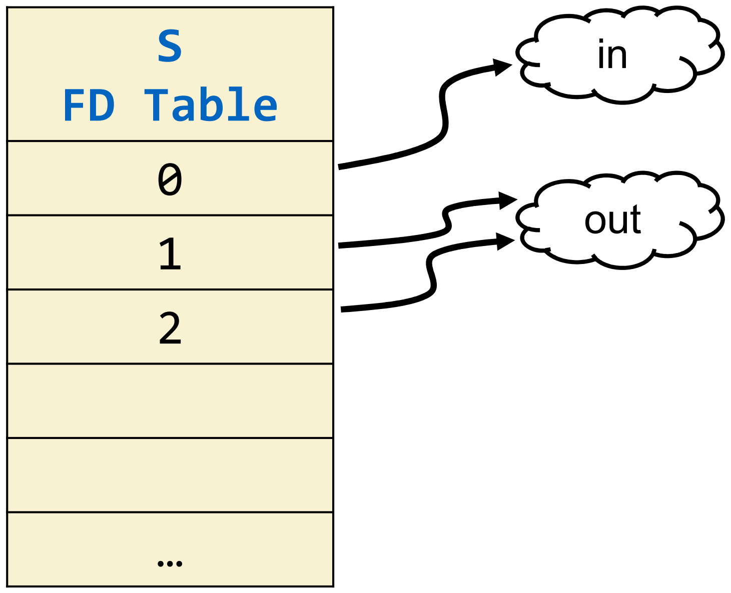
Example: connect two programs with a pipe

Start a pipeline in program S (aka, the shell): A | B

- High-level strategy (missing clean up !)
  - Create a pipe
  - Fork #1
    - In child process
      - Redirect stdout to the write end of the pipe
      - Start A, by calling exec
  - Fork #2
    - In child process
      - Redirect stdin to the read end of the pipe
      - Start B, by calling exec

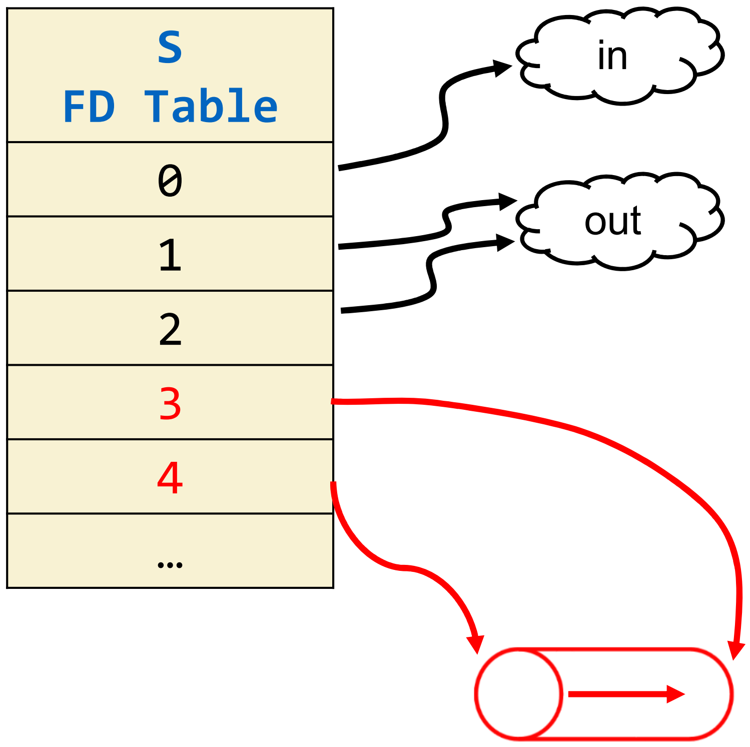
At the beginning

- S has only 0, 1, and 2 open



Pipe creation

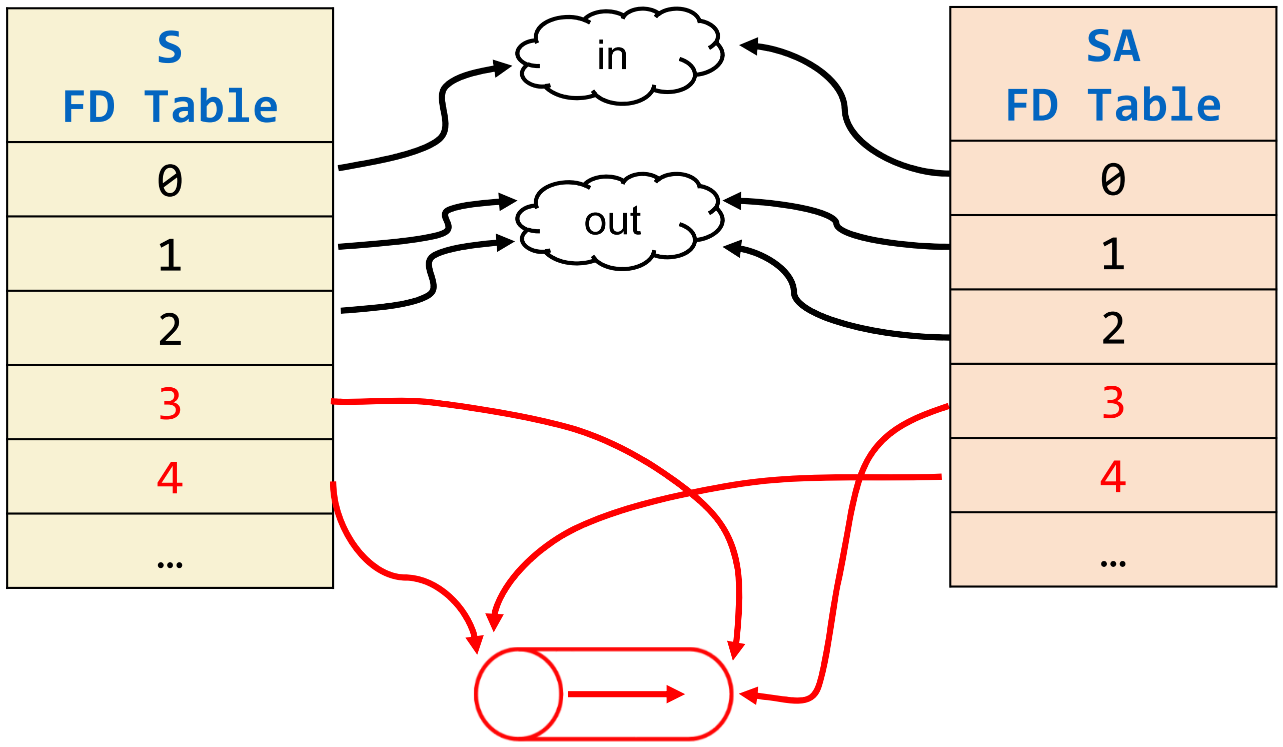
- S creates a pipe by calling pipe()
  - A pair of FDs is returned



Fork #1

S: fork()

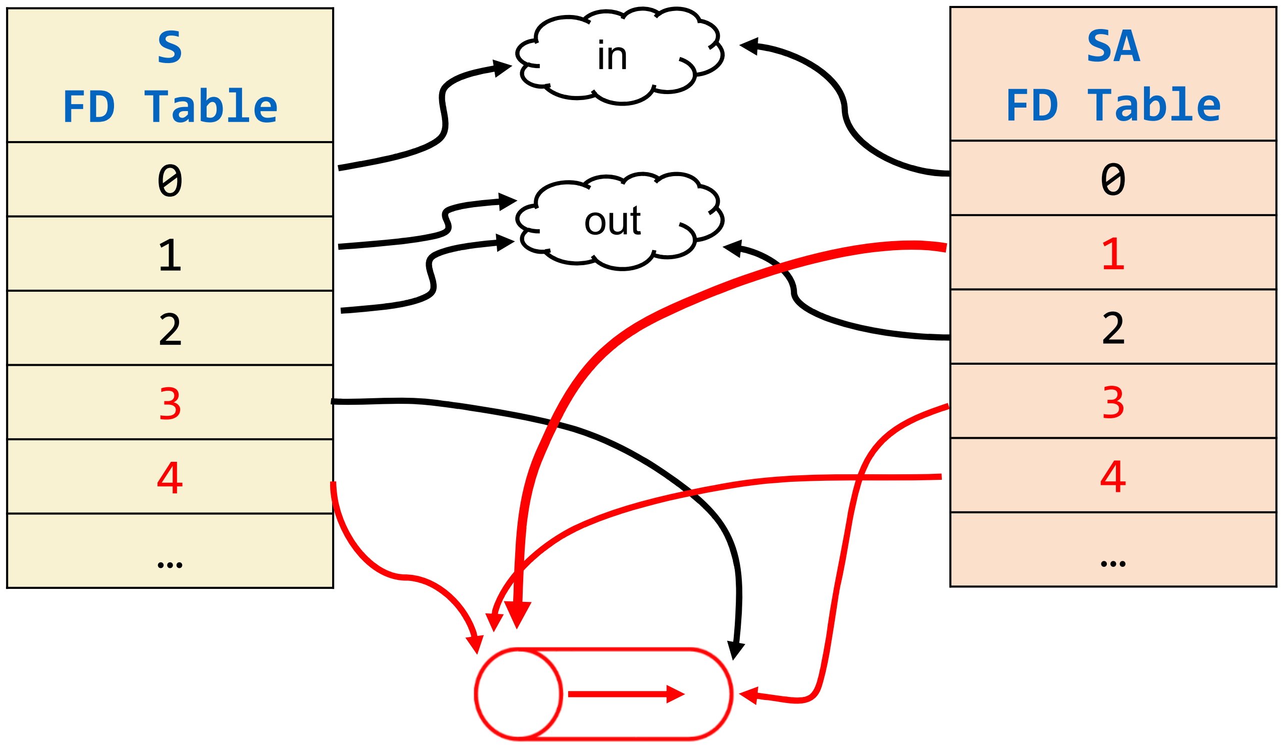
- FD table is duplicated



Redirect in first child process

SA: dup2(4, 1)

- Or close(1); dup(4);

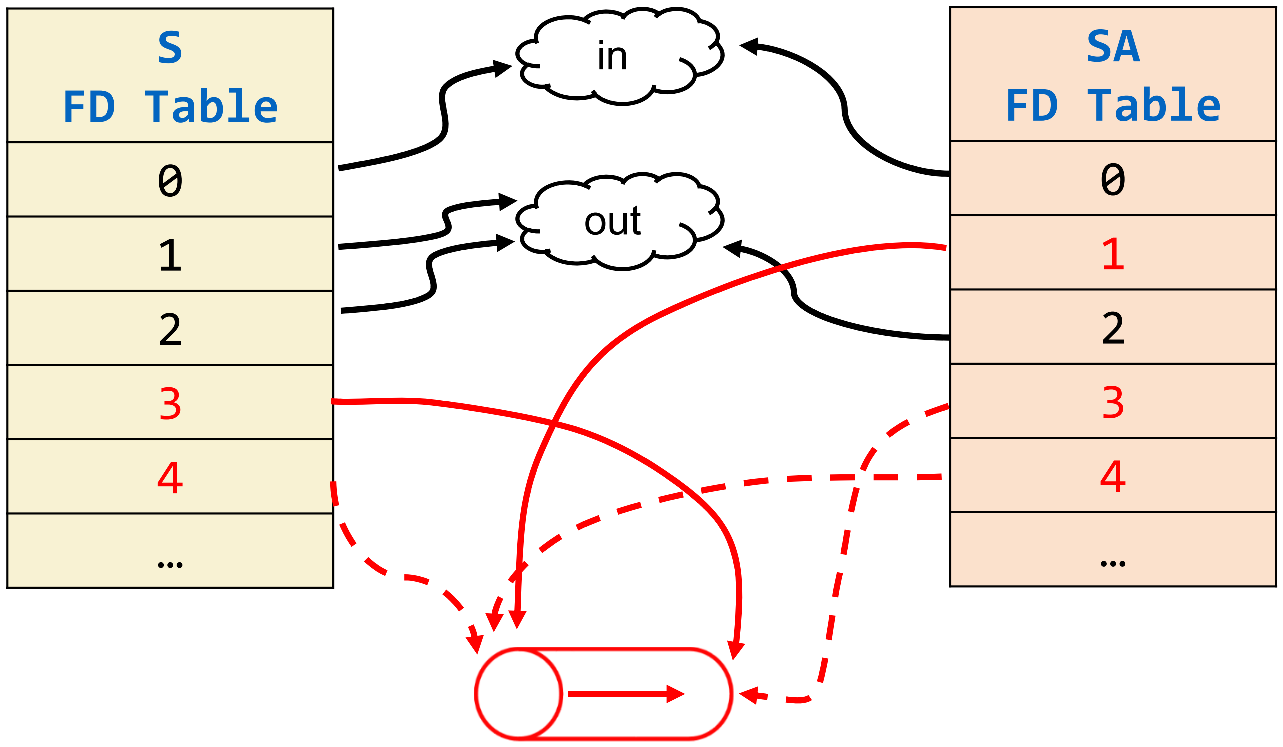


Clean up #1

S: close(4)

SA: close(4); close(3)

SA can then exec into A

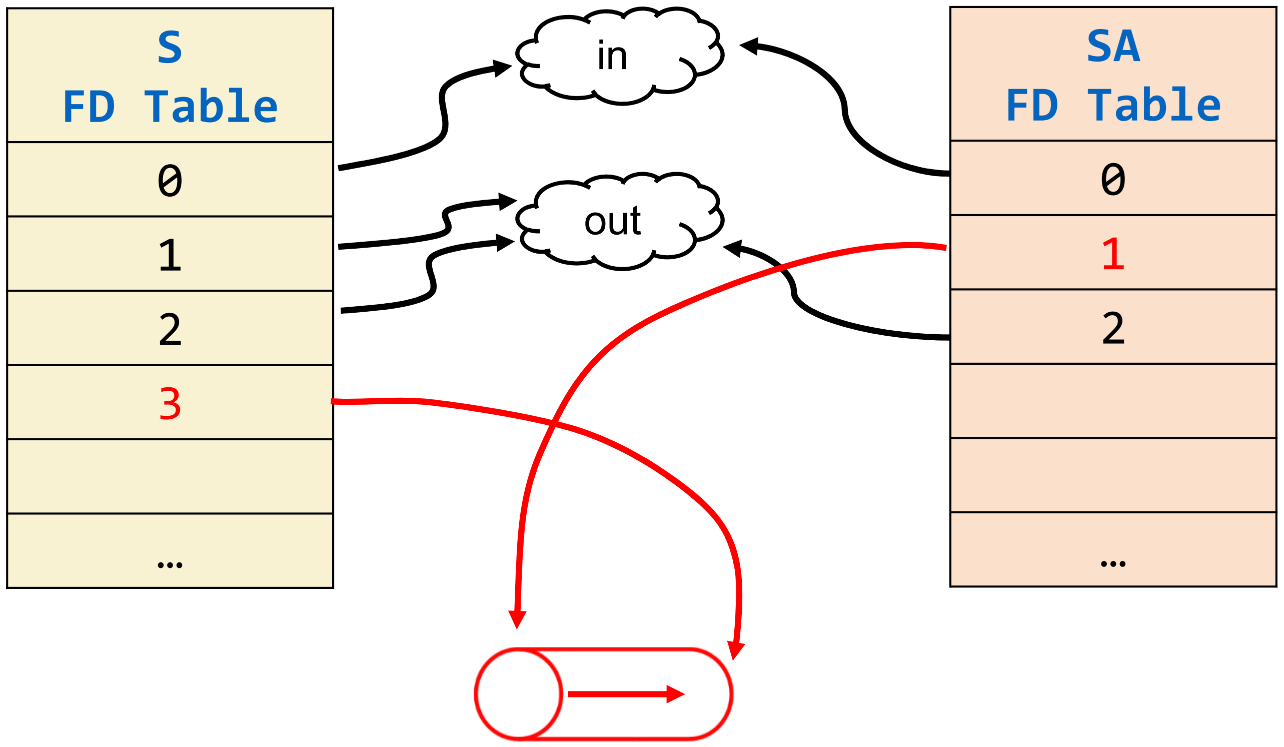


After clean up #1

S: close(4)

SA: close(4); close(3)

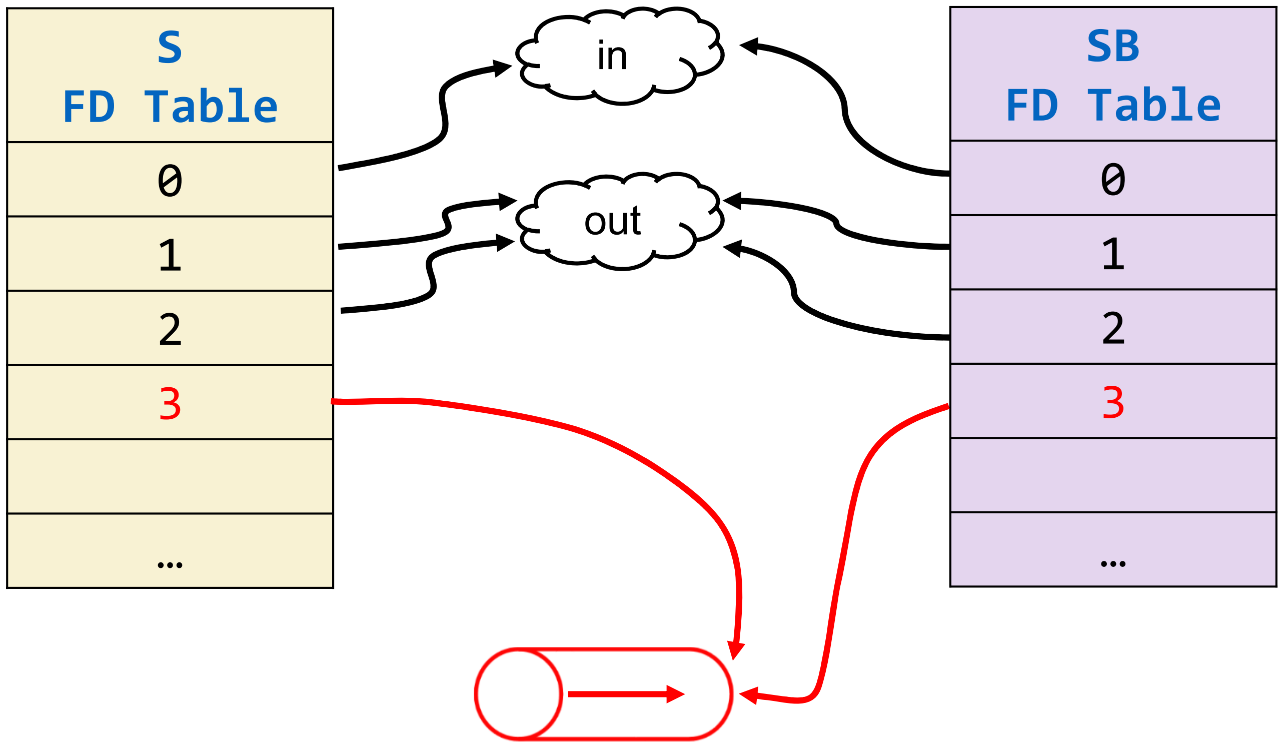
SA can then exec into A



Fork #2

S: fork()

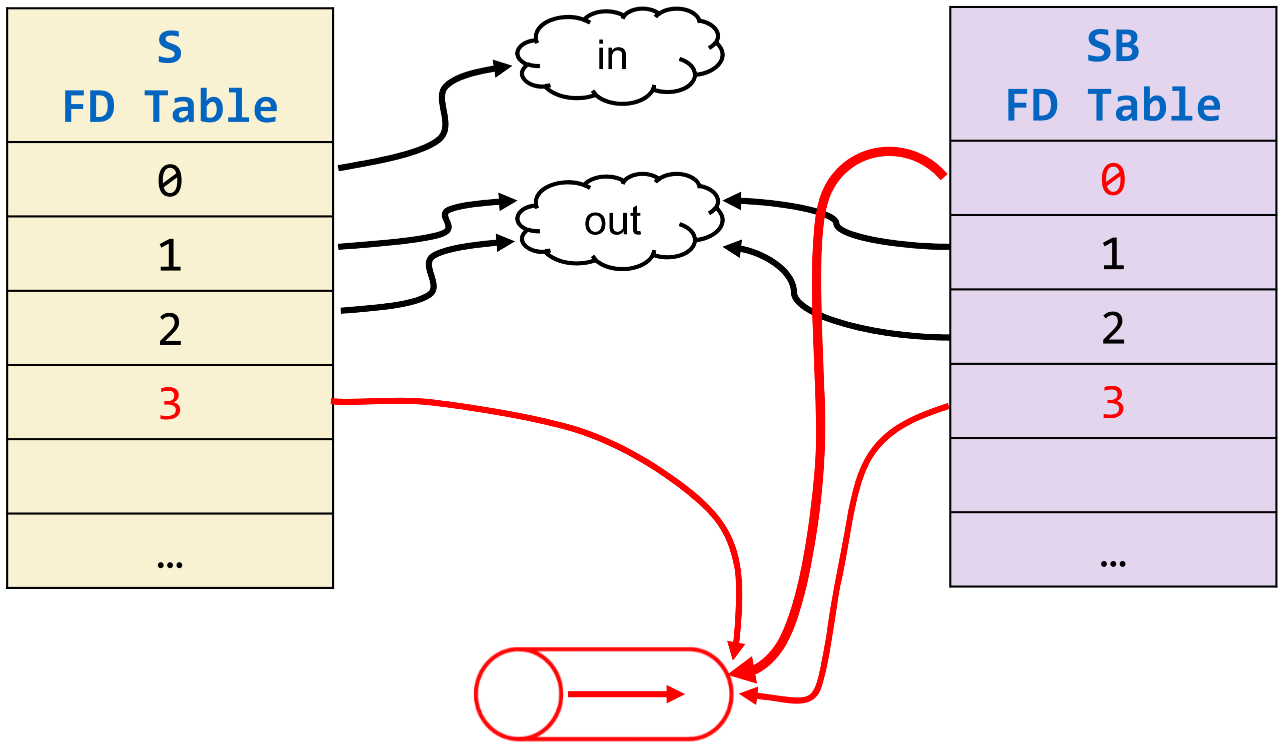
- Note that 4 has been closed in S



Redirect in second child process

SB: dup2(3, 0)

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

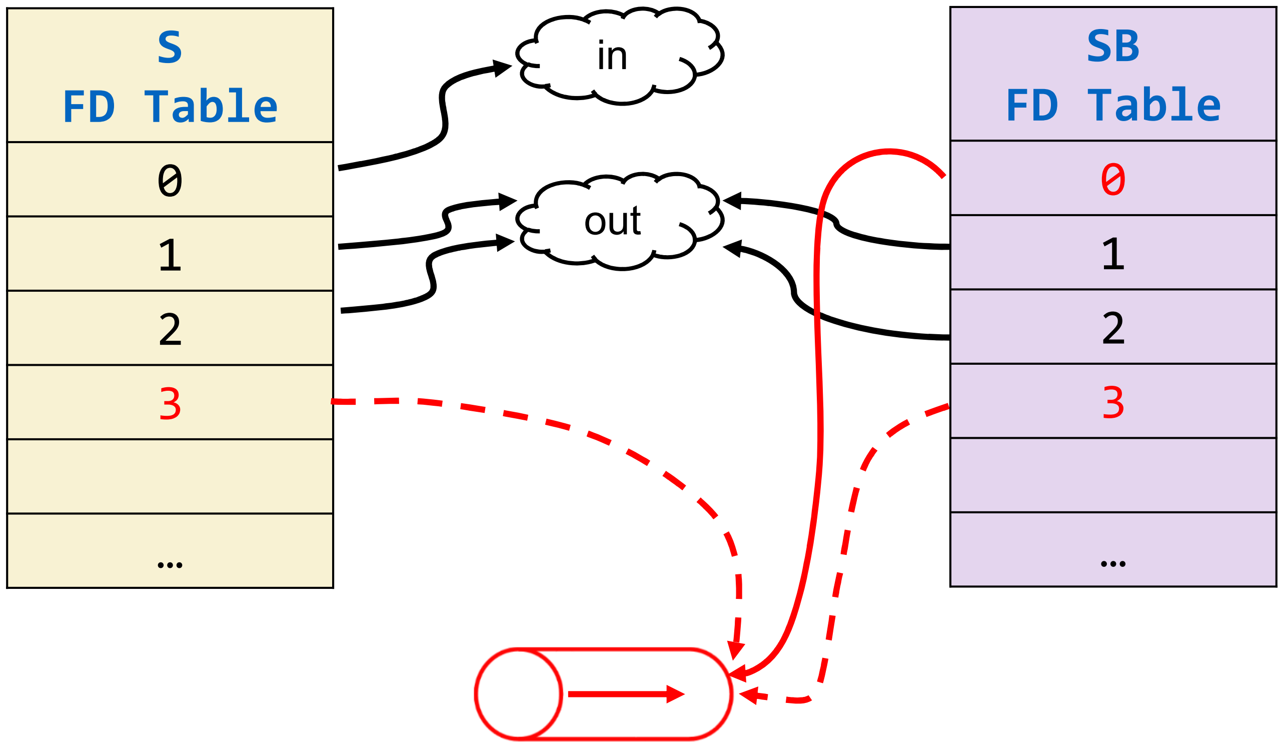


Clean up #2

S: close(3)

SB: close(3)

- SB can then exec into B

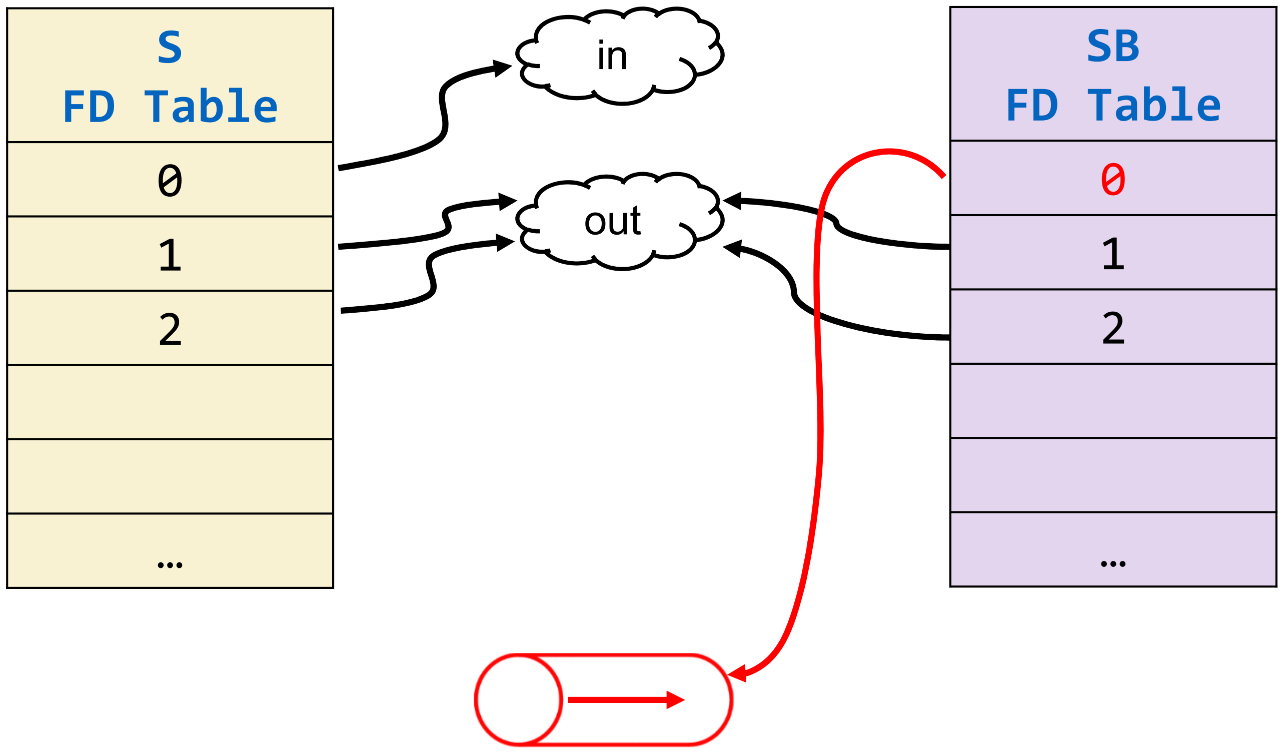


After clean up #2

S: close(3)

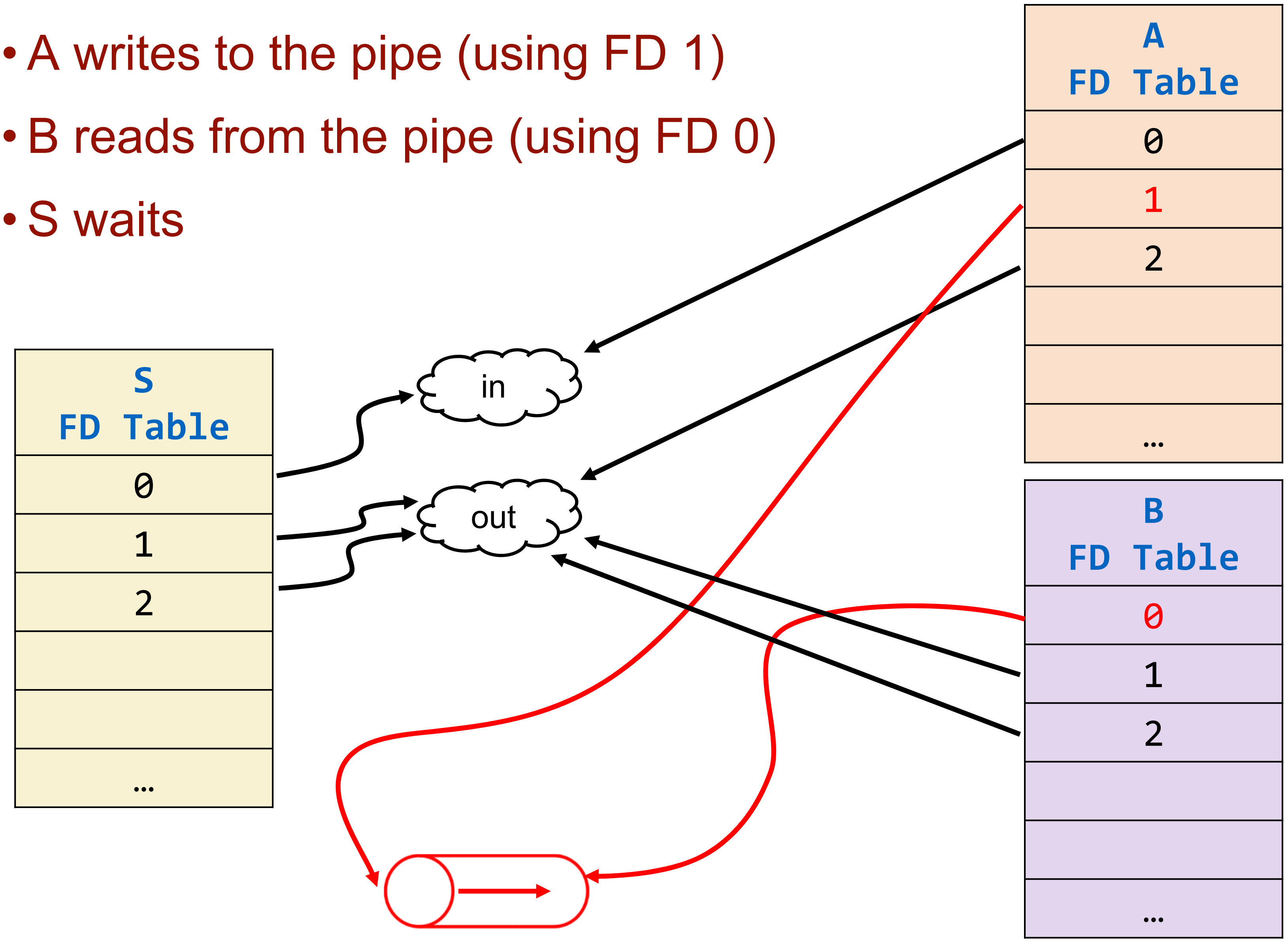
SB: close(3)

SB can then exec into B



Final set up

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



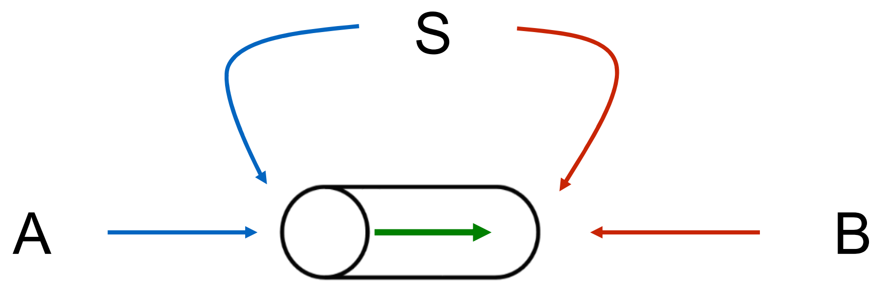
FDs of a dying process

- When a process ends, all its open FDs are automatically closed
- What happens to the processes on the other end of the pipe?

Example:

Assume S does not read or write, but have FDs of the pipe

- If both A and S die, B gets EOF when all buffered data are consumed
- If A dies, B will wait for more data (assuming S may write)
- If both B and S die, A gets an error (SIGPIPE) when writing
- If B dies, A will wait if the pipe is full (assuming S will read)



Close file descriptors
a process does not use!

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

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

Study the remaining slides yourself

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

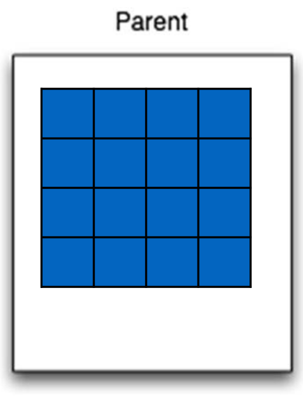
```
pid_b = fork()          // for B
if (pid_b == 0) {       // child process for B
    dup2();             // setup stdin for B
    close(pipefd[RD_END));
    exec to start B // remember to exit from child on error
}
close(pipefd[RD_END]); // No need to keep it open in parent

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

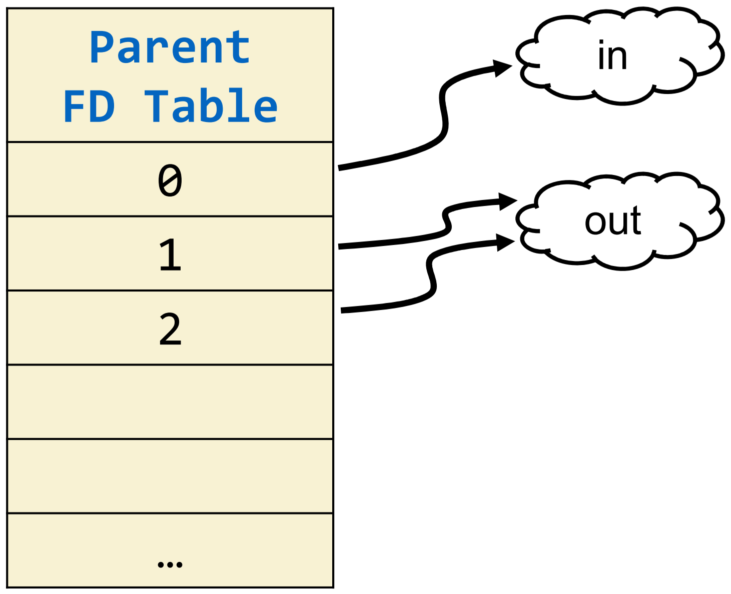
Using Pipes to Sum Matrix Rows Concurrently

See the complete code in the demo repo.

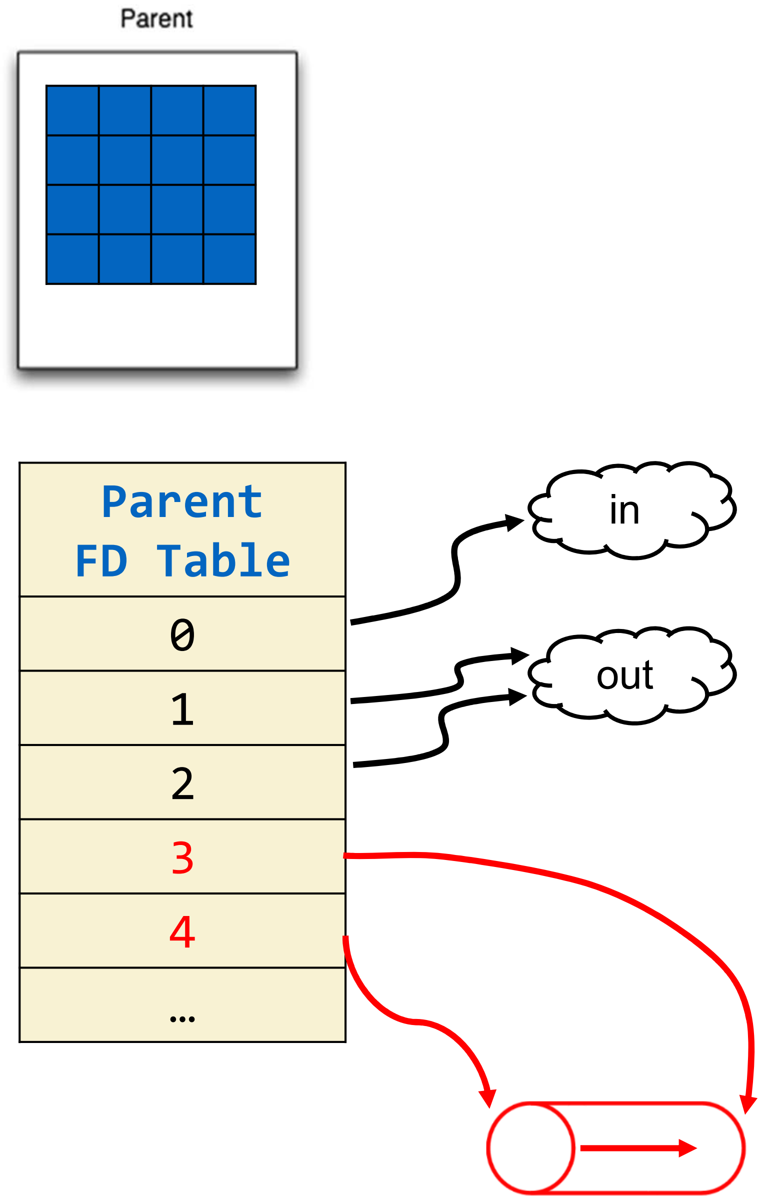
```
int main(void)
{
   int i, row_sum, sum = 0, pd[2], a[N][N] = {{1, 1, 1}, {2, 2, 2}, {3, 3, 3}};

   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*/
```

Parent process

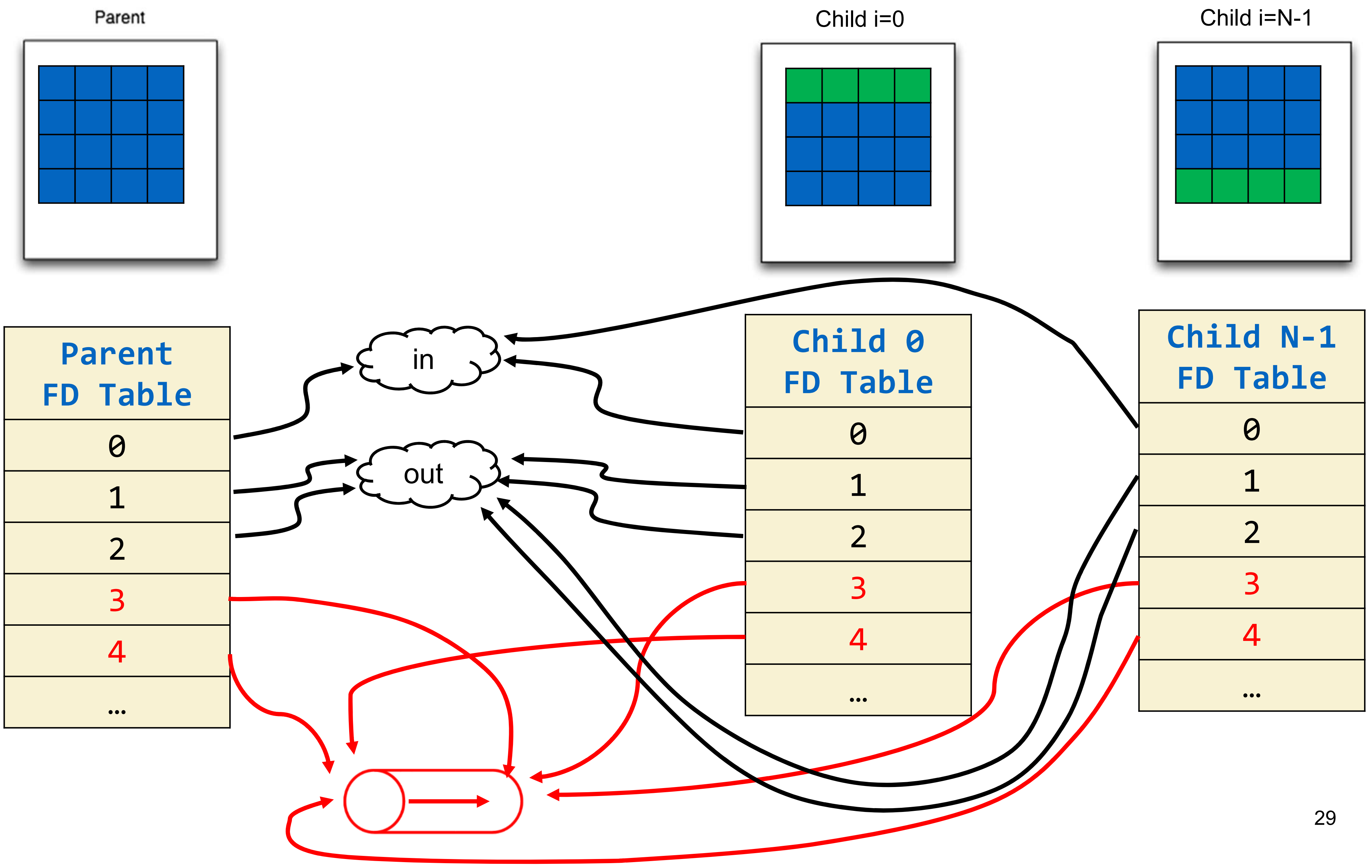




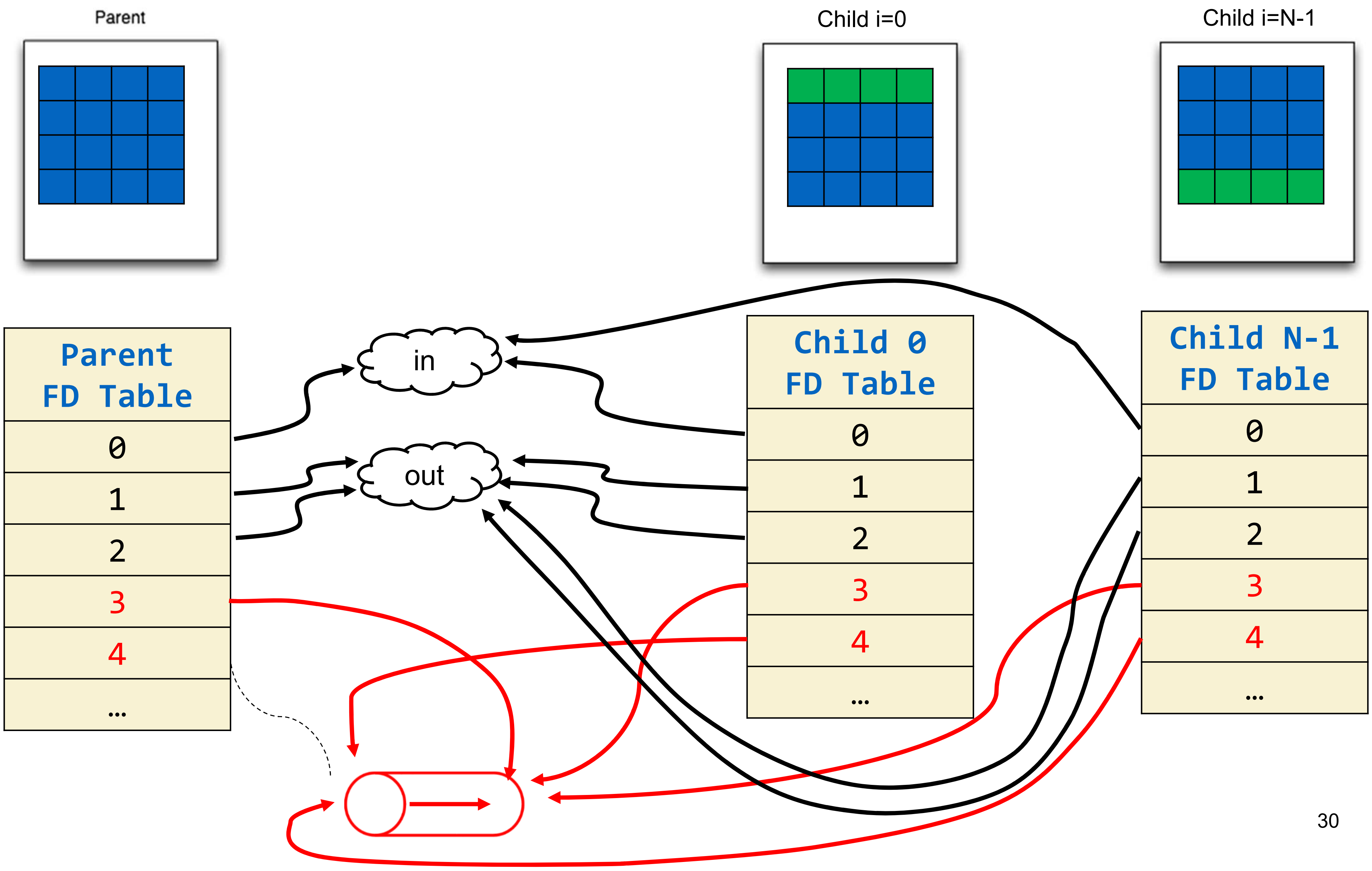
Pipe creation



After forking (without cleanup)



After forking (with cleanup in parent)


Parent
Child i=0
Child i=N-1
Parent FD Table
0
1
2
3
4
…
in
out
Child 0 FD Table
0
1
2
3
4
…
Child N-1 FD Table
0
1
2
3
4
…