

# System Programming: Communication through pipes

Raymond Namyst  
Dept. of Computer Science  
University of Bordeaux, France

<https://gforgeron.gitlab.io/progsys/>

# The concept of pipe

- Major mechanism used by the shell
  - `ls | grep pattern`
  - `./prog | cat -n | less`
  - Etc.
- Some operating systems (MS-DOS) implement pipes using files
  - Example: `ls | grep pattern`
    - The output of “ls” is redirected to a temporary file
    - The system waits for the termination of “ls”
    - “grep pattern” is executed, with its input redirected from the file
  - No parallelism
  - Max file size limit can be reached

# The concept of pipe

- In Unix systems, pipes are special objects allocated in the kernel
  - FIFO ordering
  - Fixed capacity



# The concept of pipe

- In Unix systems, pipes are special objects allocated in the kernel
  - FIFO ordering
  - Fixed capacity



- The “pipe” syscall

```
int pipe(int fildes[2]);
```

- Creates a pipe and returns two file descriptors
  - One for reading (fildes[0]) and one for writing (fildes[1])

# The concept of pipe

- In Unix systems, pipes are special objects allocated in the kernel
  - FIFO ordering
  - Fixed capacity

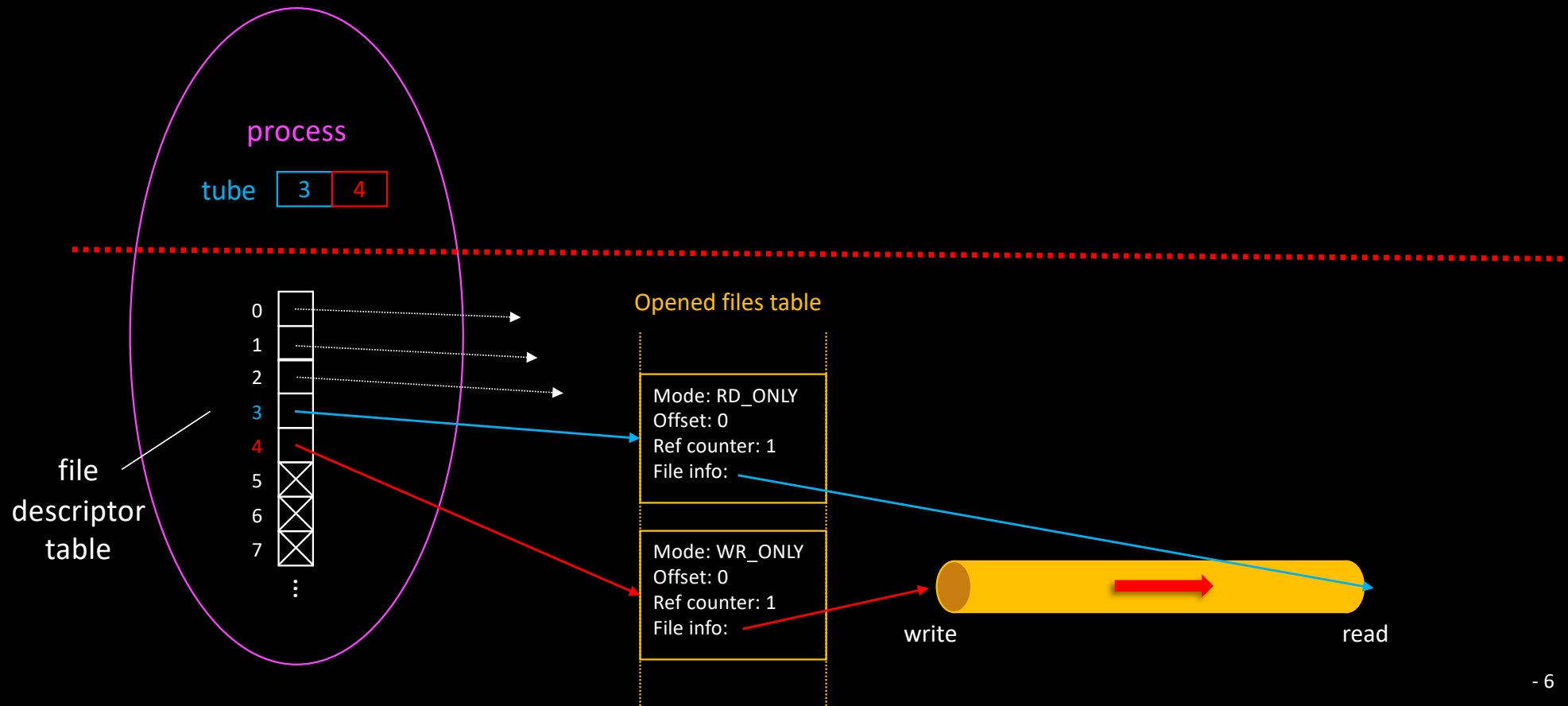


- The “pipe” syscall

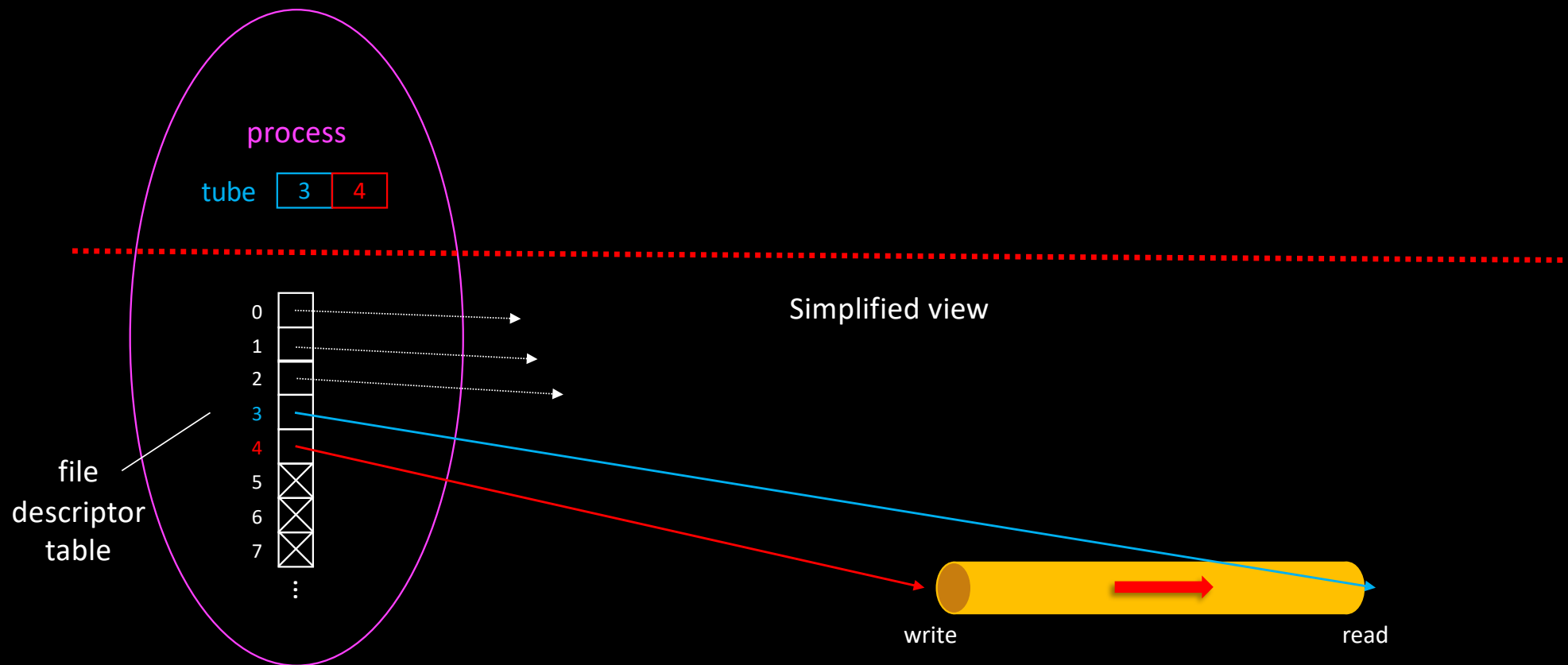
```
int pipe(int fildes[2]);
```

- Creates a pipe and returns two file descriptors
  - One for reading (fildes[0]) and one for writing (fildes[1])
- Reading and writing are done using usual read/write
  - No lseek

```
int tube[2];  
pipe (tube);
```



```
int tube[2];  
pipe (tube);
```



# Trying out our first pipe

- pipe.c

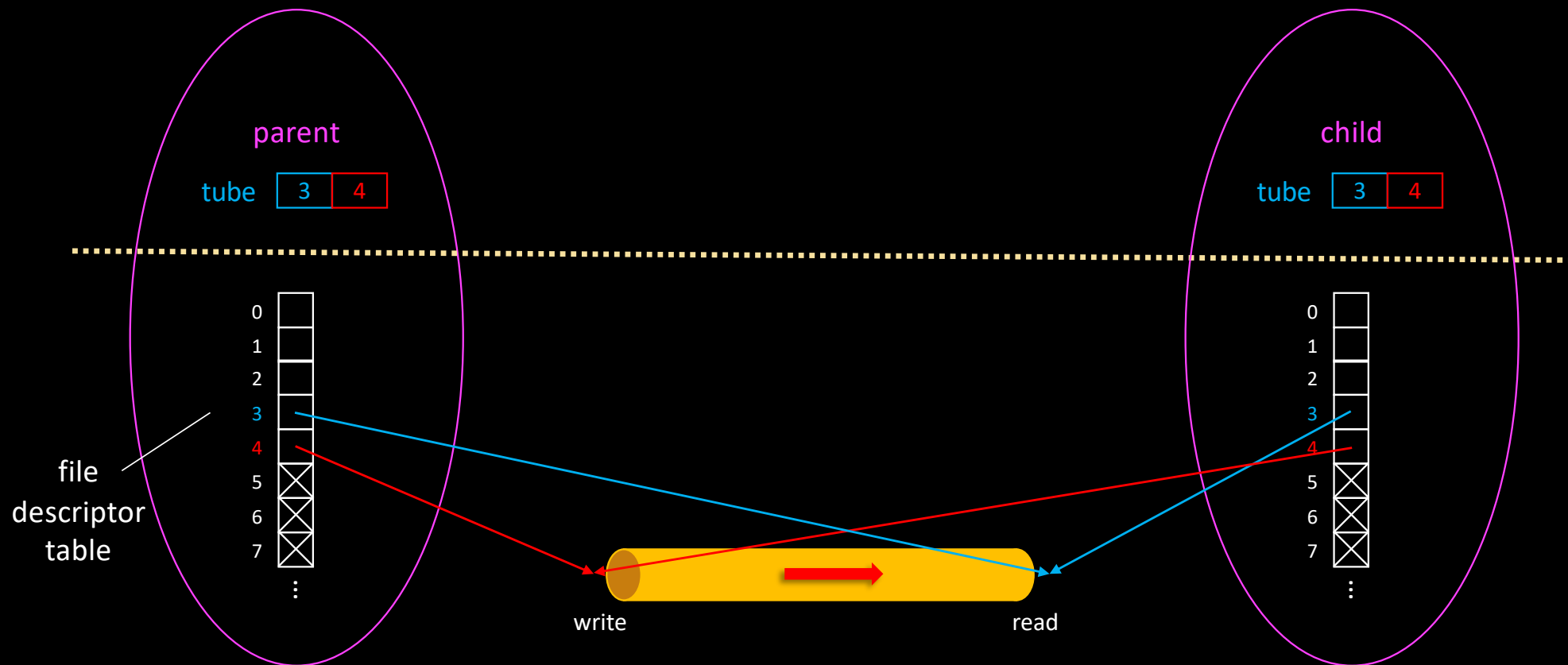




# Pipe are intended to allow communication between processes

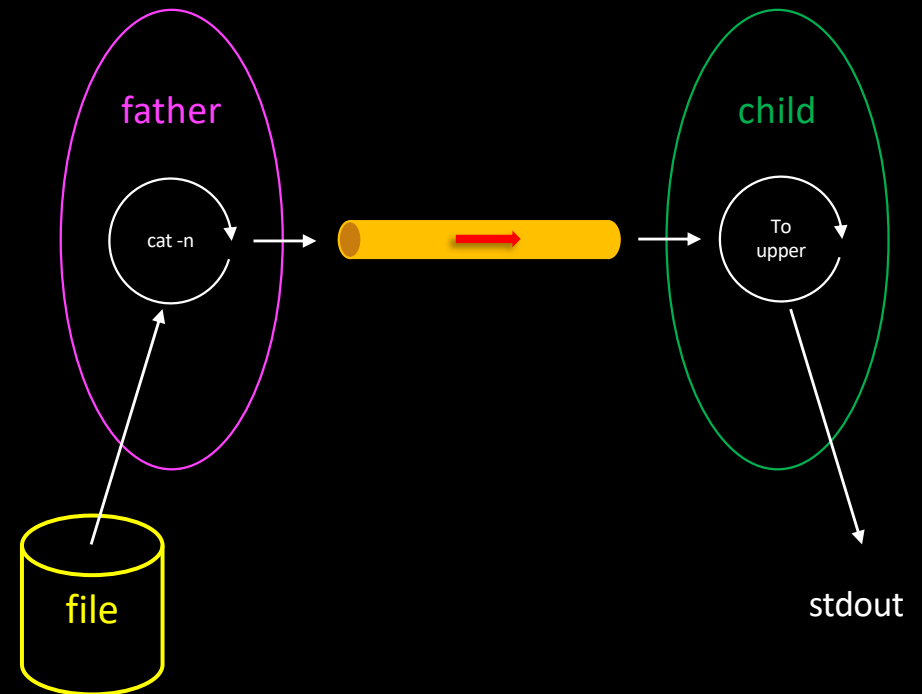
- But pipes created with “pipe” are anonymous
  - No way to share their name
    - Look at `mkfifo` if named pipes are really what you need
- Fortunately, pipes are inherited when forking new processes...

```
int tube[2];  
pipe (tube); fork();
```



# Sending characters through a pipe

- Hand-made implementation of
  - Line numbering + to upper case
- pipe-n-fork.c



# The concept of pipe

- **When both sides of a pipe are opened**
  - `read` is blocking if the pipe is empty
  - `write` is blocking if the pipe is full

# The concept of pipe

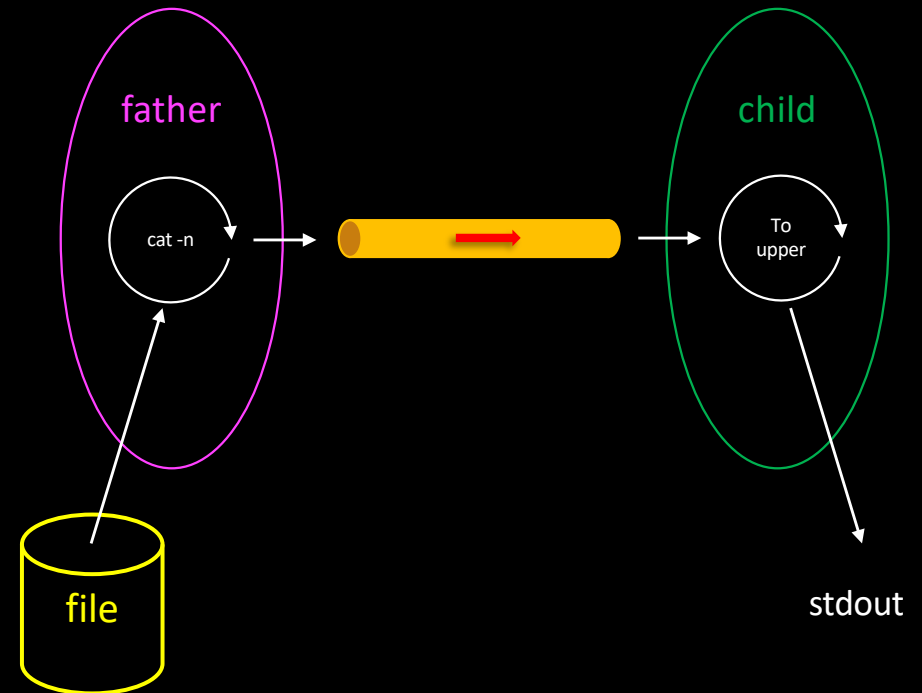
- **When both sides of a pipe are opened**
  - `read` is blocking if the pipe is empty
  - `write` is blocking if the pipe is full
- **Pipe closed on the write side**
  - `read` returns 0 (end of file)

# The concept of pipe

- **When both sides of a pipe are opened**
  - `read` is blocking if the pipe is empty
  - `write` is blocking if the pipe is full
- **Pipe closed on the write side**
  - `read` returns 0 (end of file)
- **Pipe closed on the read side**
  - `write` raises an exception (“Broken pipe”)

# Sending characters through a pipe

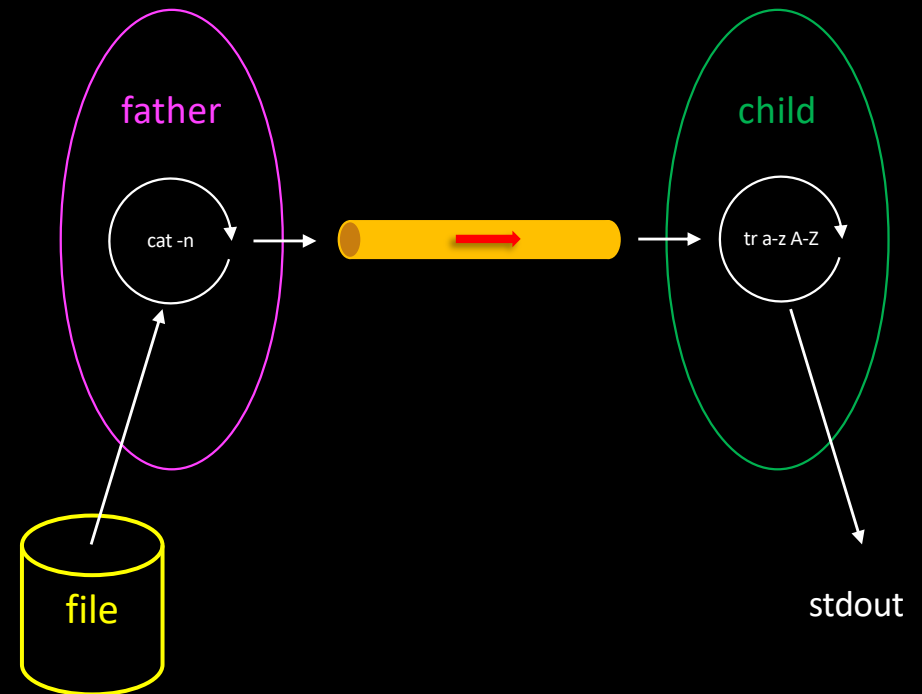
- Redirections to only use STDIN/STDOUT
- pipe-n-redir.c





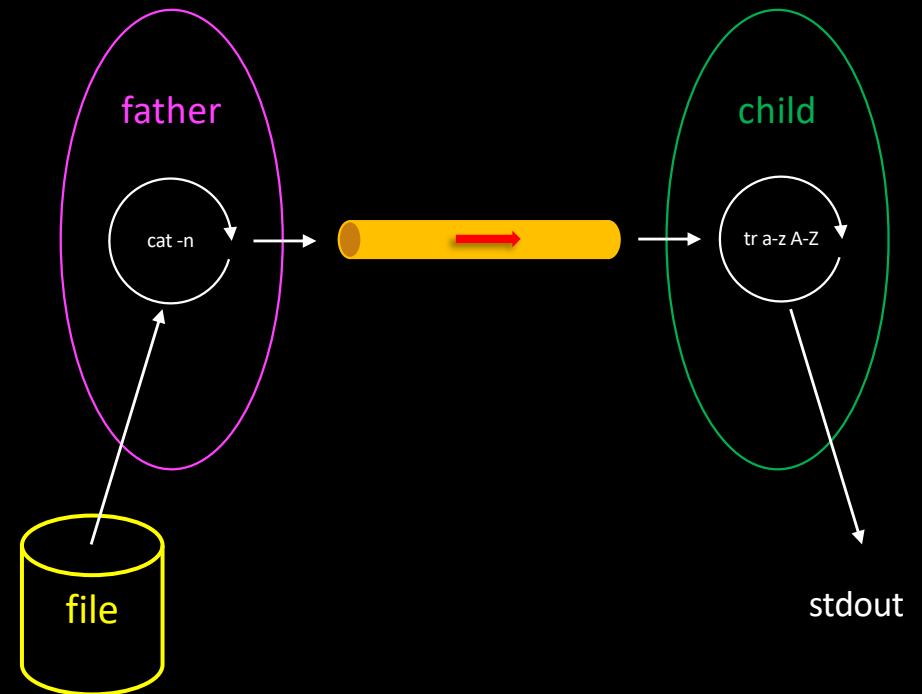
# Sending characters through a pipe

- Redirections to only use STDIN/STDOUT
- Exec to use legacy
  - `cat -n`
  - `tr a-z A-Z`
- `pipe-n-exec.c`



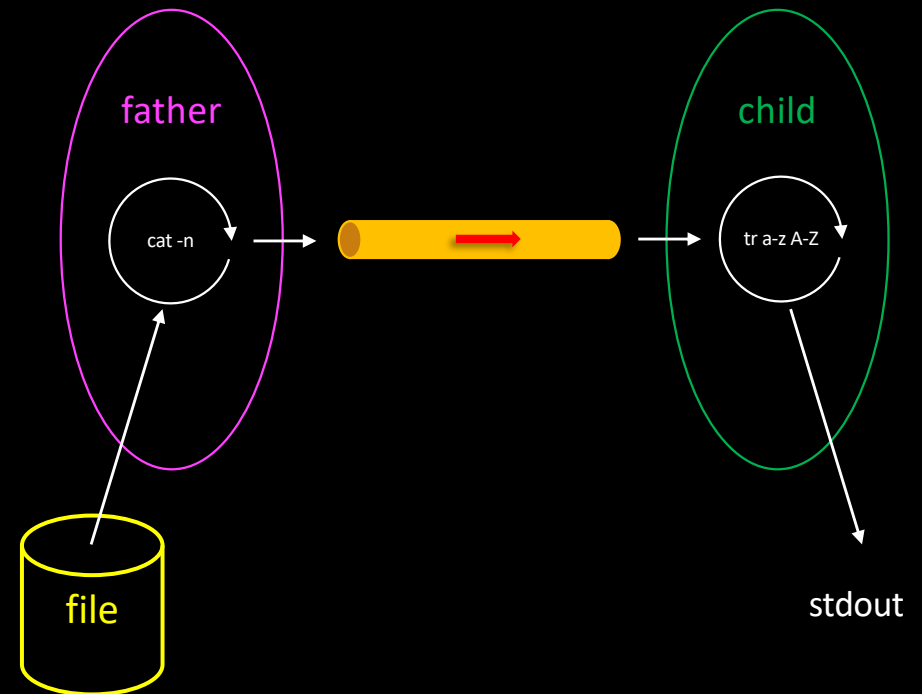
# Sending characters through a pipe

- **Problem:**
  - Child still output characters after shell prompt



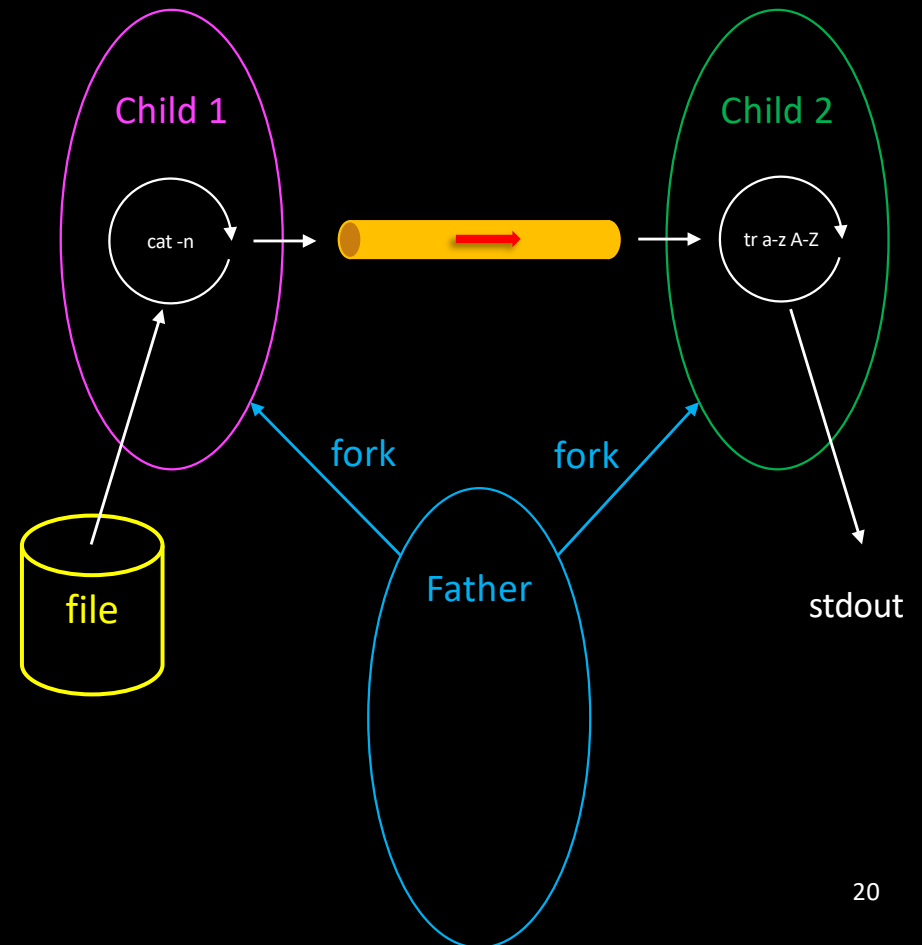
# Sending characters through a pipe

- **Problem:**
  - Child still output characters after shell prompt
- **One solution would be to swap roles of father & child...**



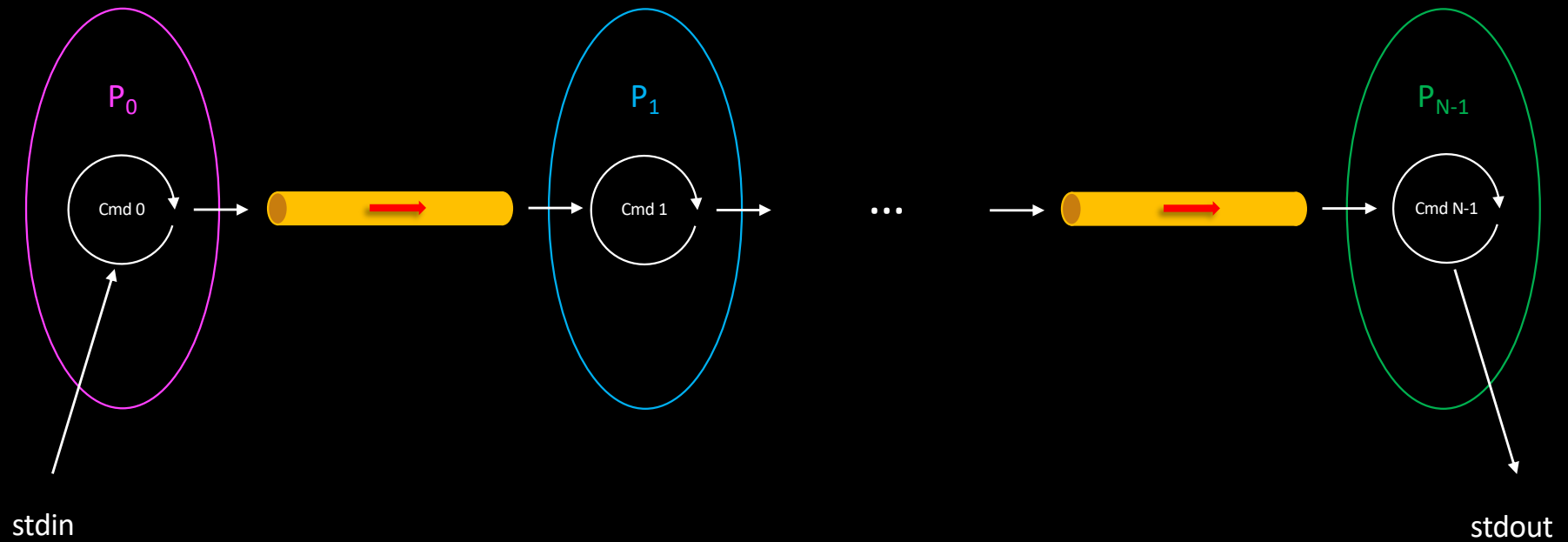
# Sending characters through a pipe

- **Problem:**
  - Child still output characters after shell prompt
- **But the shell acts differently: it creates two children!**
- **ultimate-pipe.c**



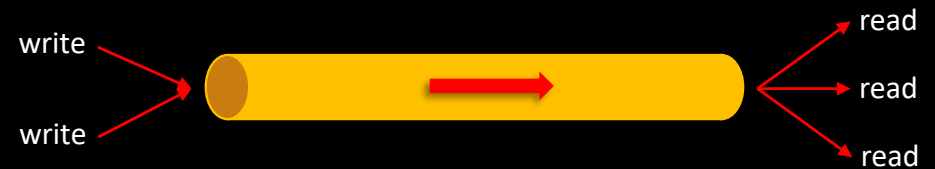
# Setting up a chain of processes

- chain.c



# Atomicity of read/write

- What happens if multiple processes simultaneously write into (or read from) the same pipe?
  - Atomic if size < PIPE\_BUF
    - (typically 512 bytes)



Additional resources  
available on

<http://gforgeron.gitlab.io/progsys/>