

# Lesson 3B

## Signals, exceptions and pipes

Operating Systems  
Computer Science and Engineering



# To remember...

Before classes

Class

After class

Prepare the prerequisites.

Study the material associated with the **bibliography**:  
slides alone are not enough.  
Please ask questions (especially after study).

Exercising skills:

- ▶ Perform all **exercises**.
- ▶ Carrying out the **practice notebooks** and **the practical exercises** progressively.



# Recommended reading

## Base



1. **Carretero 2020:**
  1. Cap. 5
2. **Carretero 2007:**
  1. Cap. 3.6 and 3.7
  2. Cap. 3.9 and 3.13

## Suggested



1. **Tanenbaum 2006:**
  1. (es) Cap. 2.2
  2. (en) Cap.2.1.7
2. **Stallings 2005:**
  1. 4.1, 4.4, 4.5 and 4.6
3. **Silberschatz 2006:**
  1. 4

# Contents

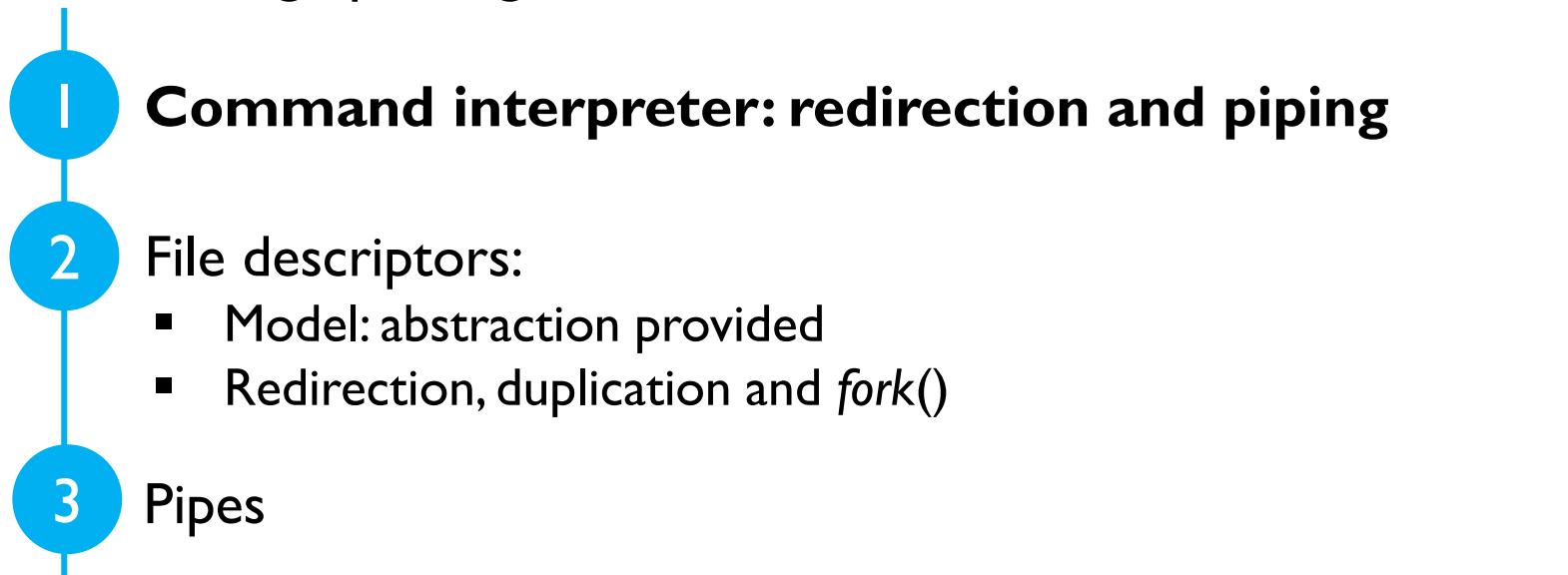
---

1. Signals and exceptions.
2. Timers.
3. Process environment.
4. **Process communication with pipes,  
local message passing**

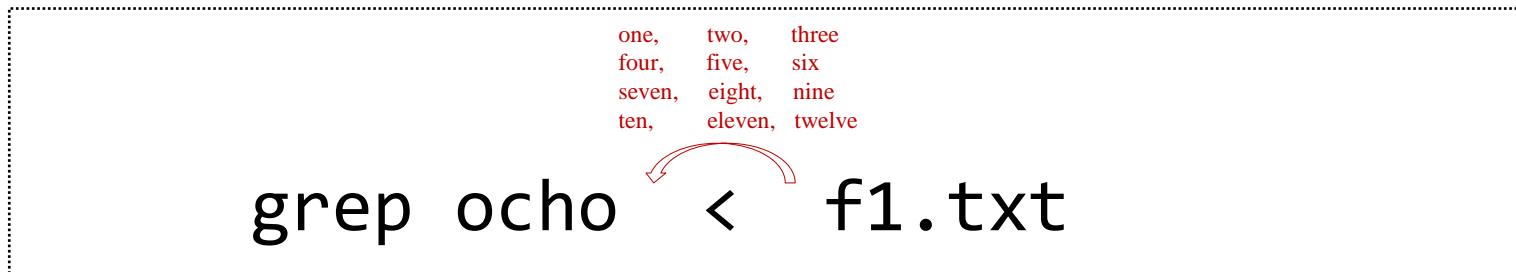
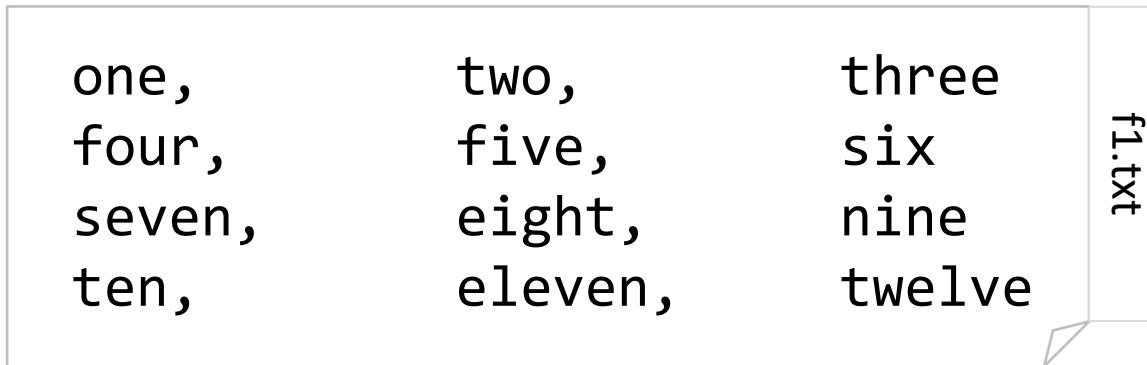
# Contents

---

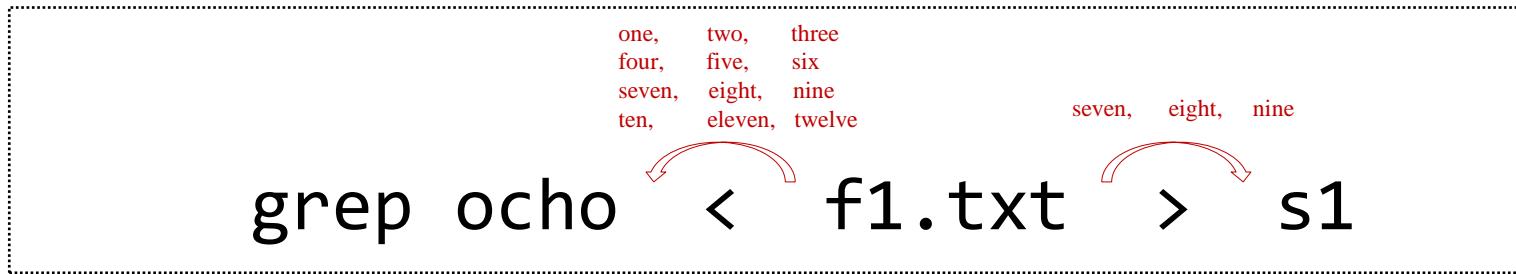
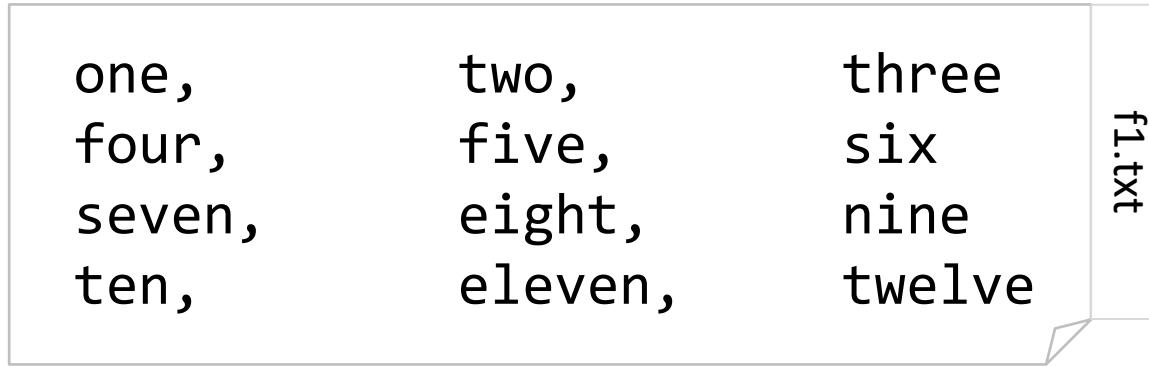
1. Signals and exceptions.
2. Timers.
3. Process environment.
4. Process communication with pipes,  
local message passing



# Example of input redirection



# Example of output redirection



# Example of output redirection

```
one,          two,          three  
four,         five,         six  
seven,        eight,        nine  
ten,          eleven,       twelve
```

f1.txt

Dependent on  
the command  
interpreter used

```
grep ocho f1.txt 1> s1
```

seven, eight, nine

# Example of **error** redirection

```
one,          two,          three  
four,         five,         six  
seven,        eight,        nine  
ten,          eleven,       twelve
```

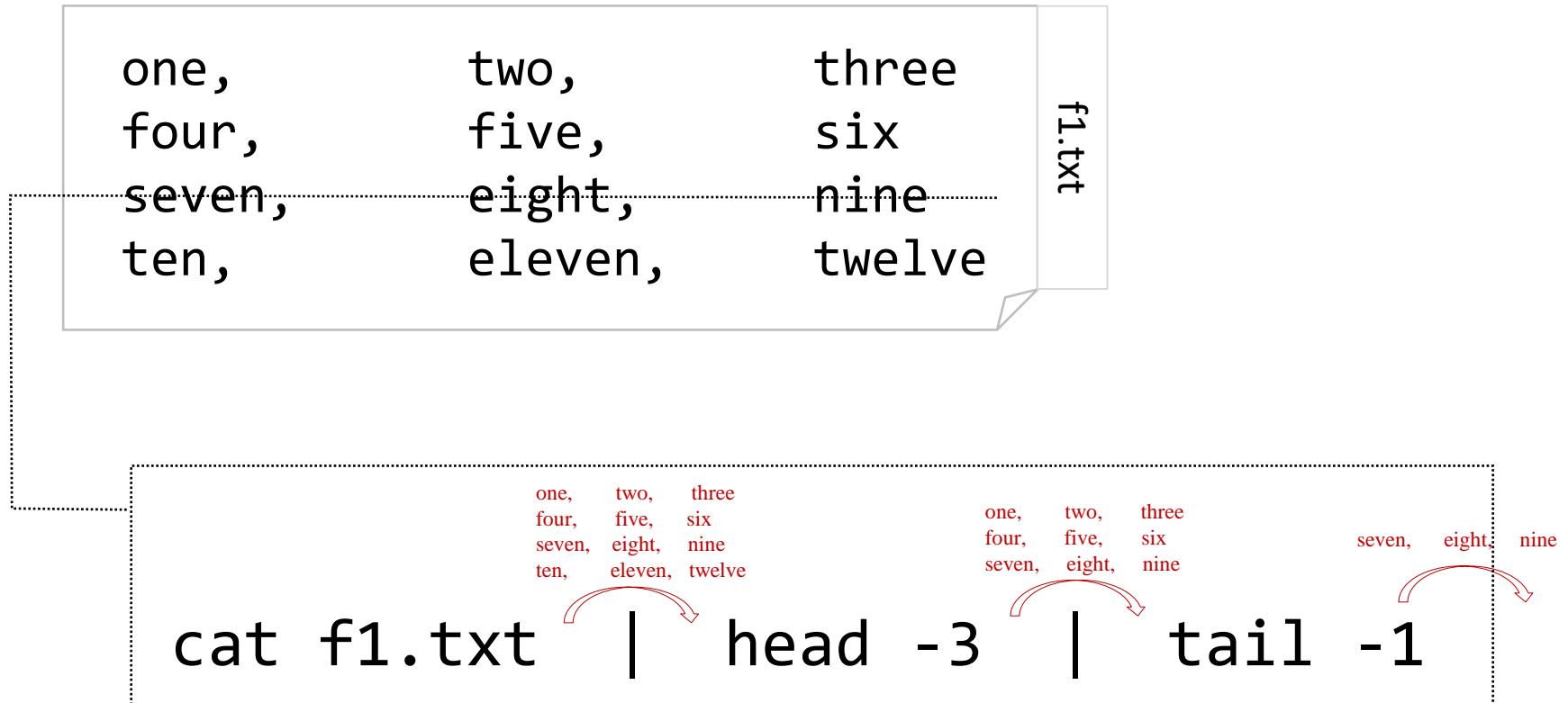
f1.txt

Dependent on  
the command  
interpreter used

```
grep ocho xx 2> s1
```

grep: xx: No such file or directory

# Example of the use of pipes



# Contents

---

1. Signals and exceptions.
2. Timers.
3. Process environment.
4. Process communication with pipes,  
local message passing



Command interpreter: redirection and piping



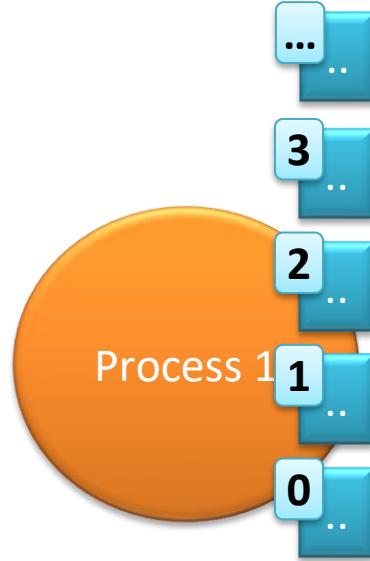
- File descriptors:**
- **Model: abstraction provided**
  - **Redirection, duplication and fork()**



Pipes

# File descriptors

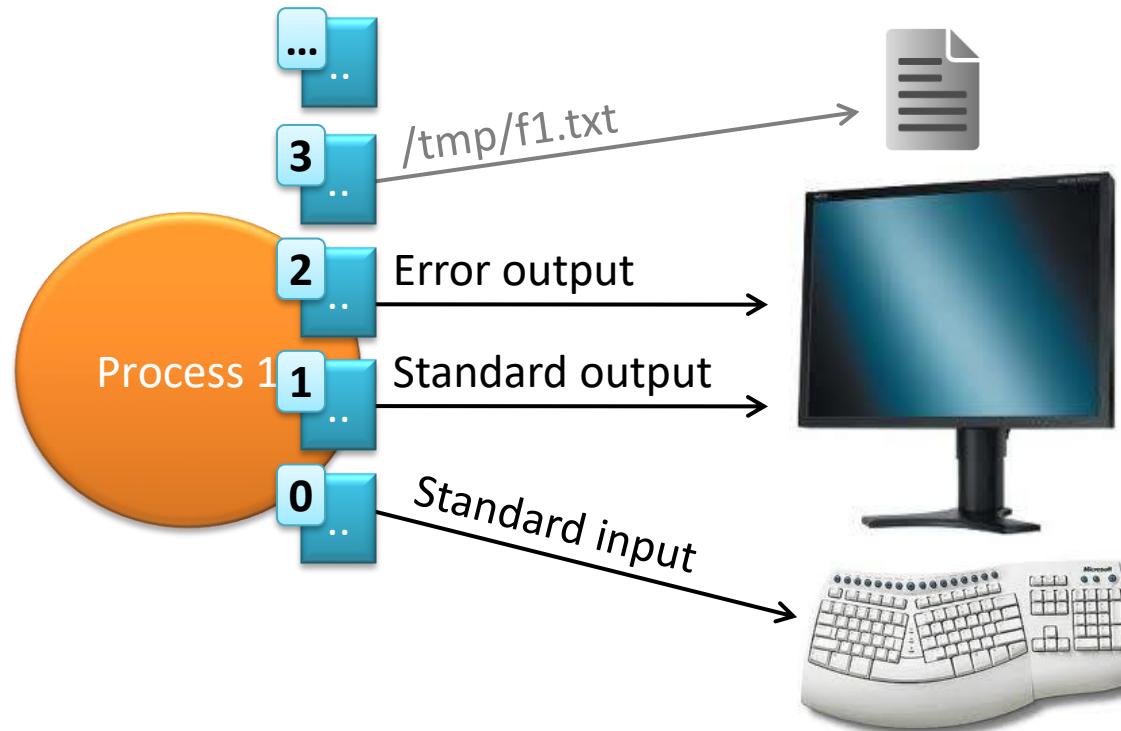
model | redirection | duplication | *fork()*



File descriptors are the index of a table per process (in the BCP) that allows you to identify the possible files (or devices) with which to communicate.

# File descriptors

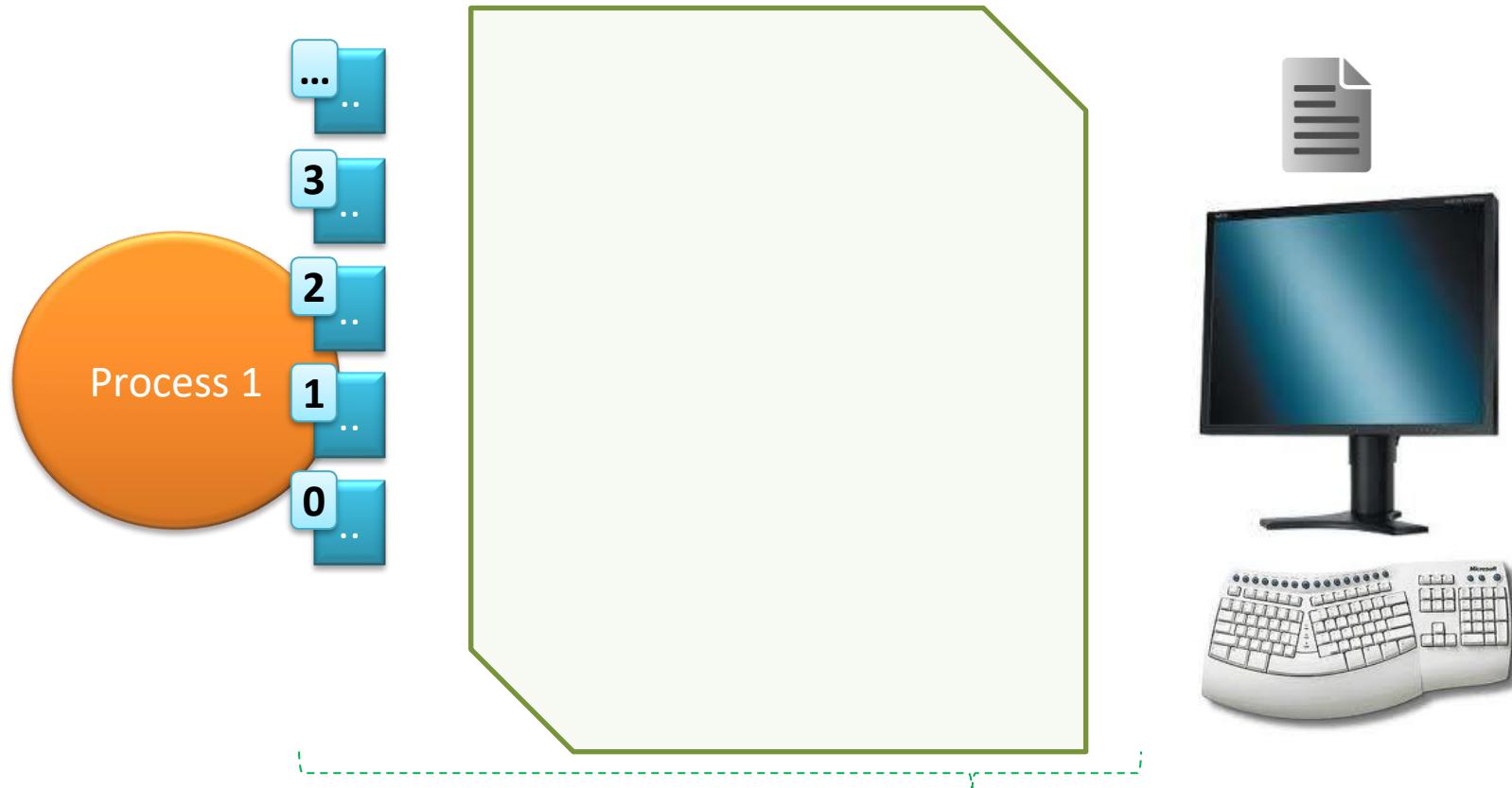
model | redirection | duplication | *fork()*



By default, the first three indexes (file descriptors) are used for standard input, standard output and error output respectively.

# File descriptors

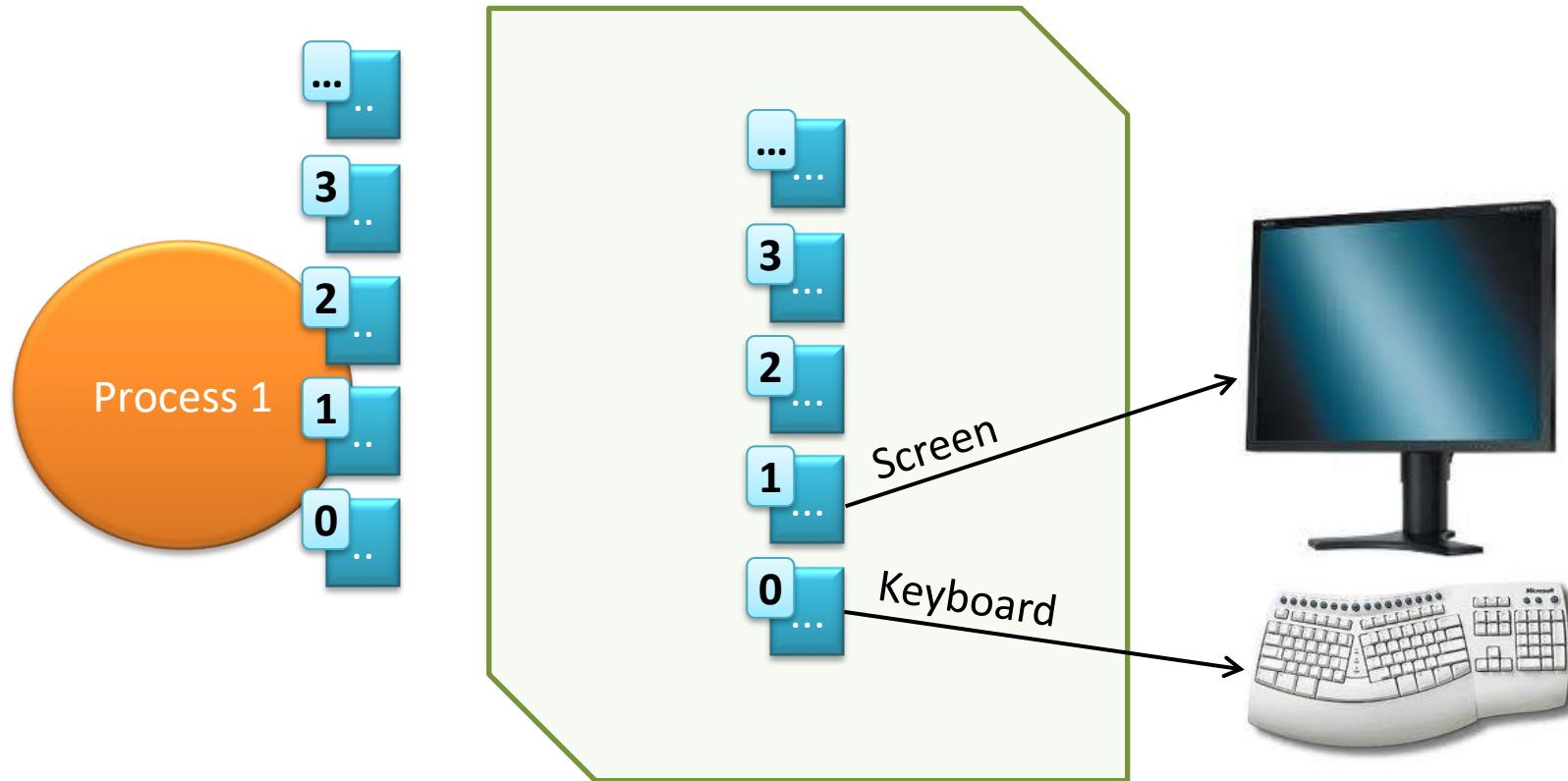
**model | redirection | duplication | fork()**



File descriptors: abstraction provided by the operating system to reference the actual devices and files. Same as a numbered key for a locker.

# File descriptors

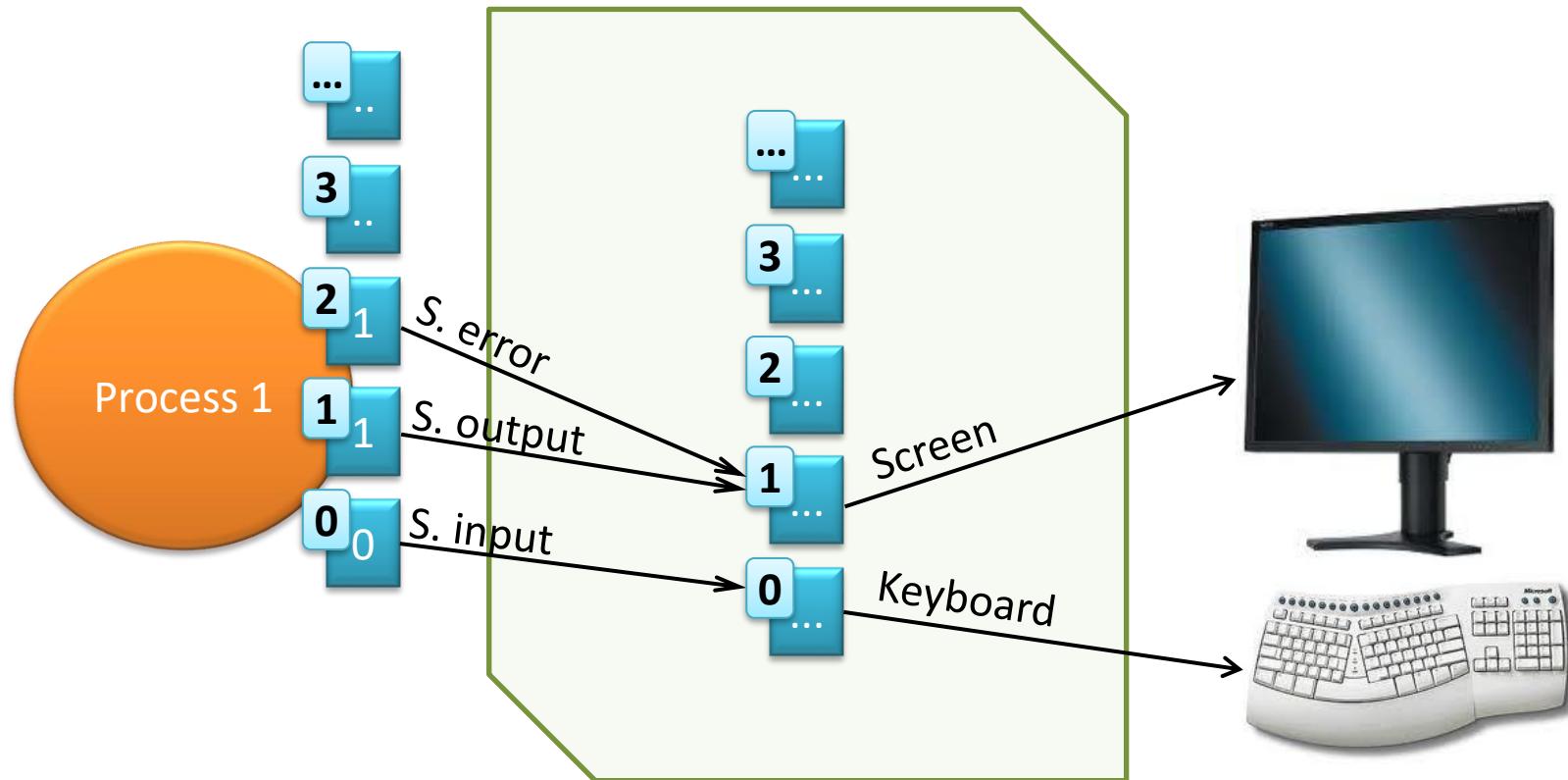
**model | redirection | duplication | fork()**



The operating system maintains an internal table with the actual contact information with the devices and files with which the processes request to communicate...

# File descriptors

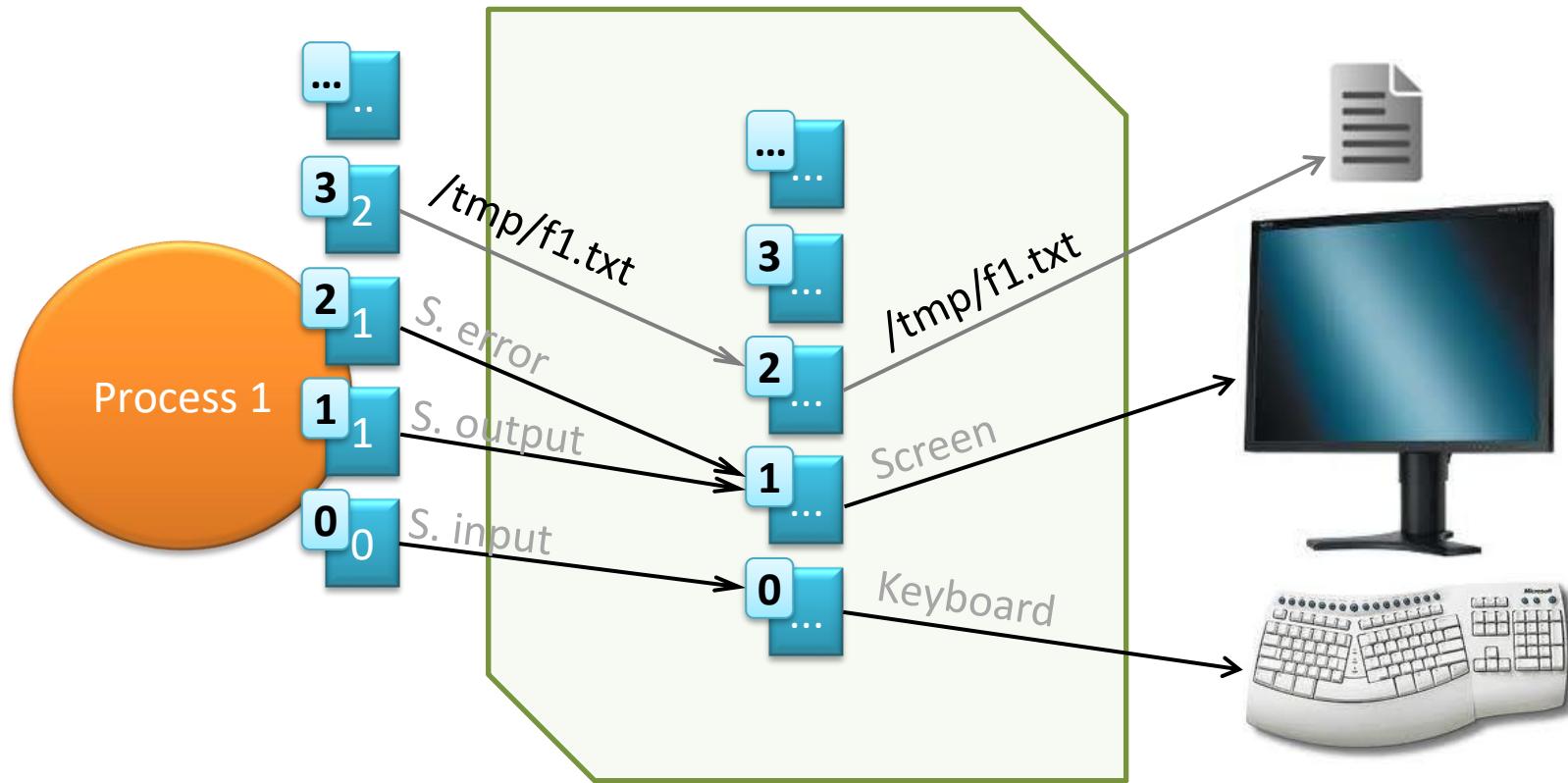
**model | redirection | duplication | fork()**



...And File descriptors are the index of the per-process table, whose content is in turn the index of the internal operating system table.

# File descriptors

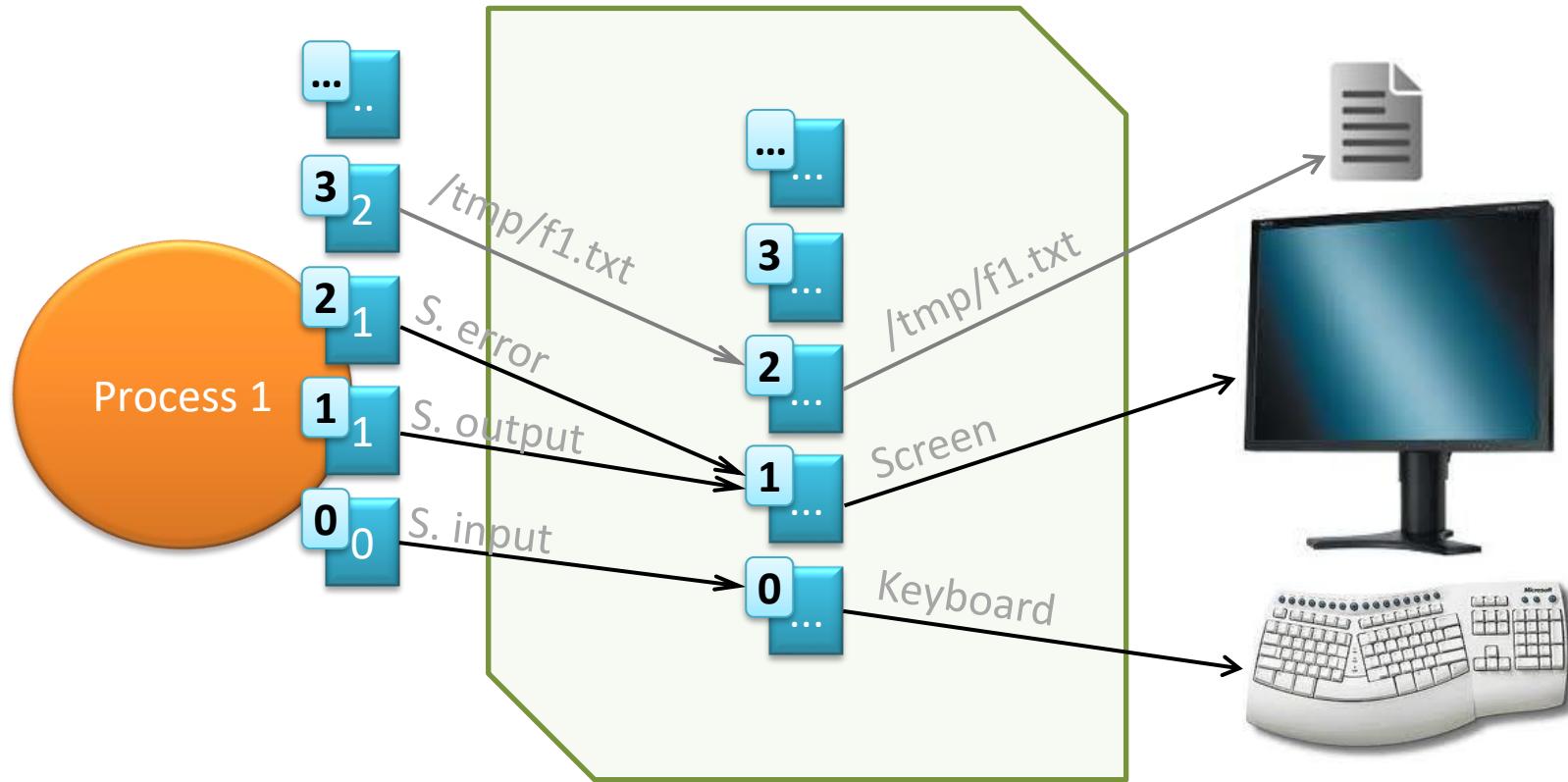
**model | redirection | duplication | fork()**



When a new file descriptor is requested (when opening a file) the first free slot in the table is searched and the index of that position is the assigned descriptor.

# File descriptors

model | redirection | duplication | *fork()*

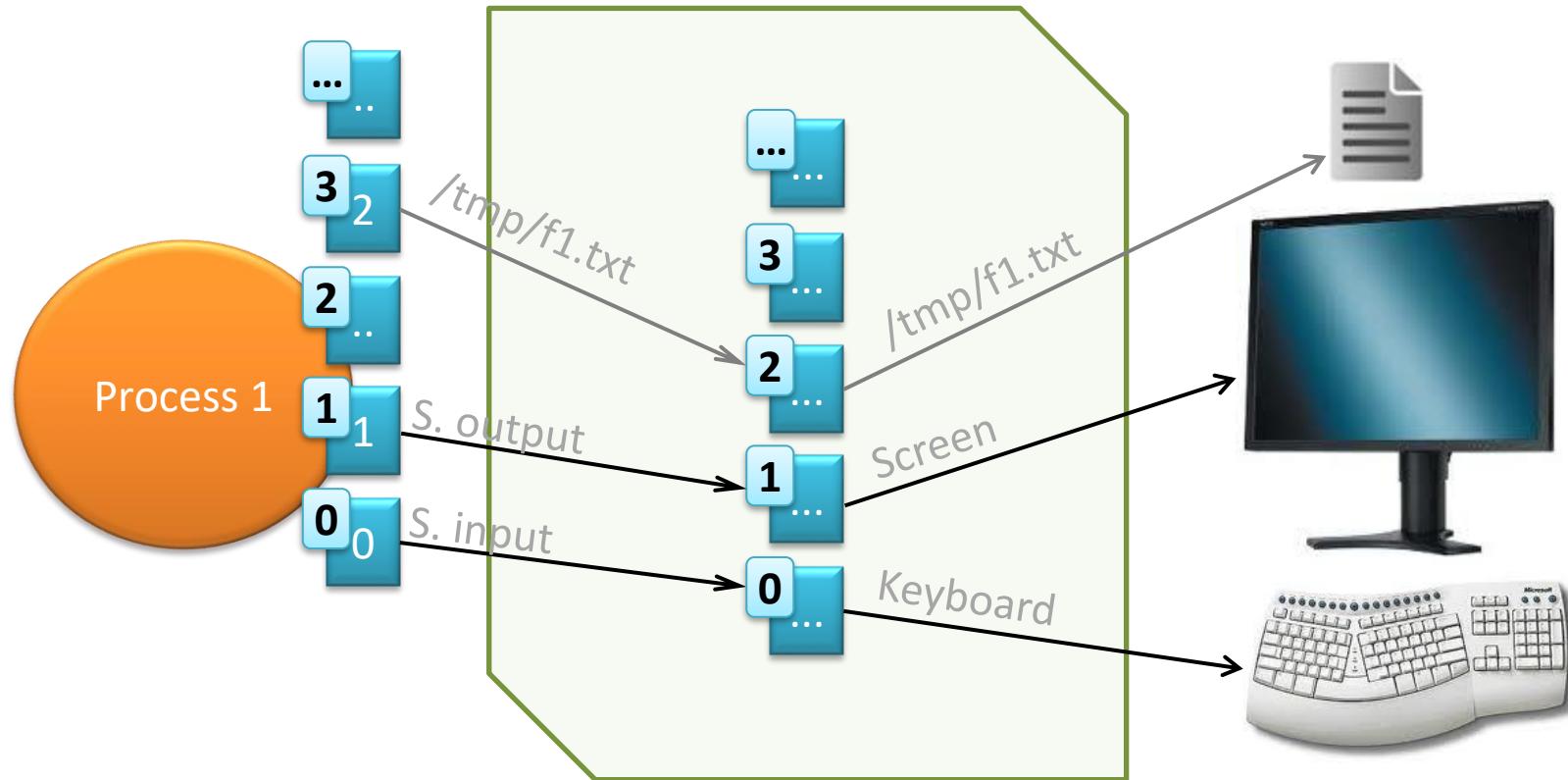


1. `close(2);`
2. `open("/tmp/errors.txt");`

?

# File descriptors

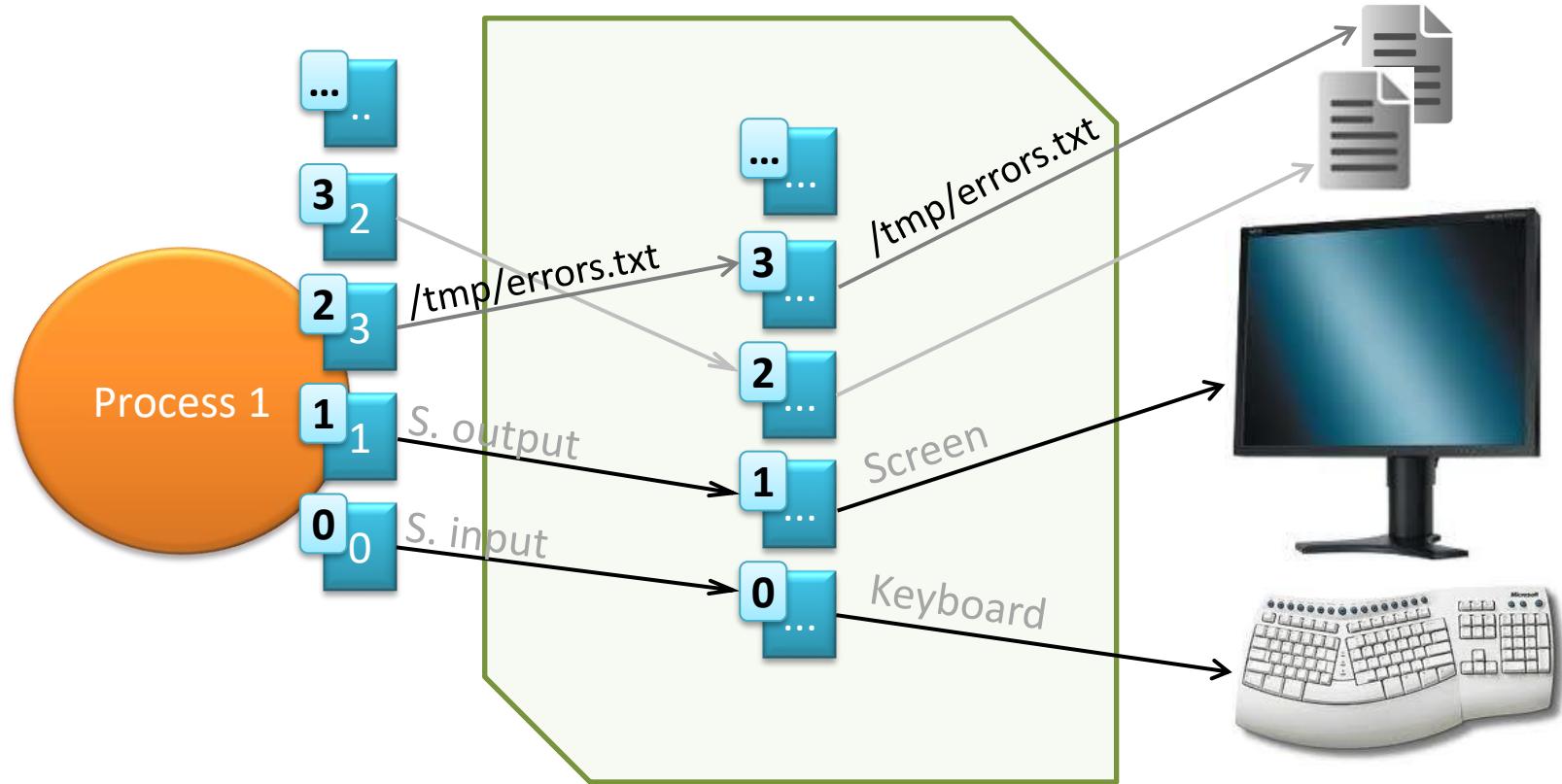
model | redirection | duplication | *fork()*



1. **close(2);**
2. **open("/tmp/errors.txt");**

# File descriptors

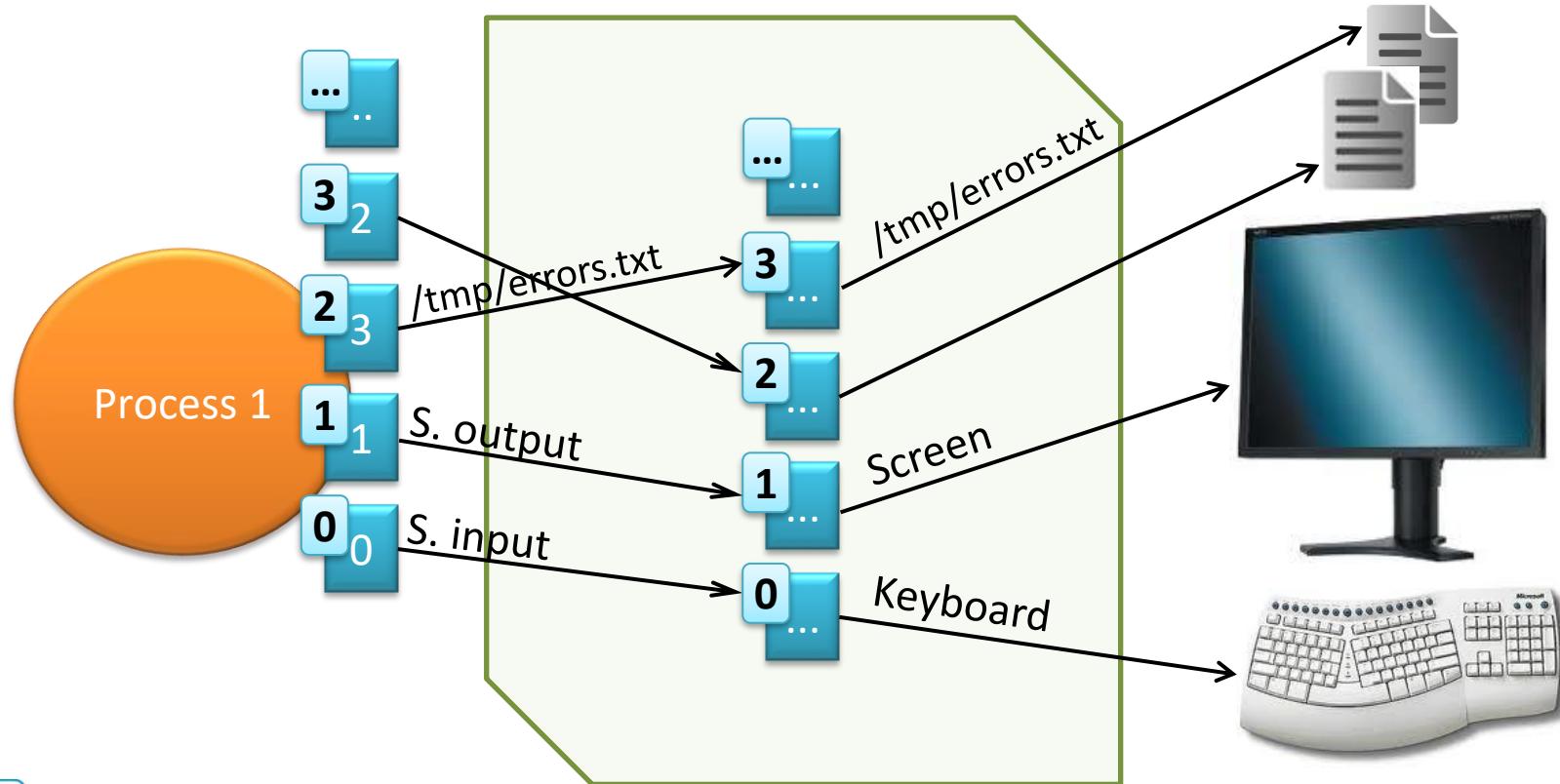
model | redirection | duplication | *fork()*



- 1. `close(2);`
- 2. `open("/tmp/errors.txt");`

# File descriptors

model | redirection | duplication | *fork()*

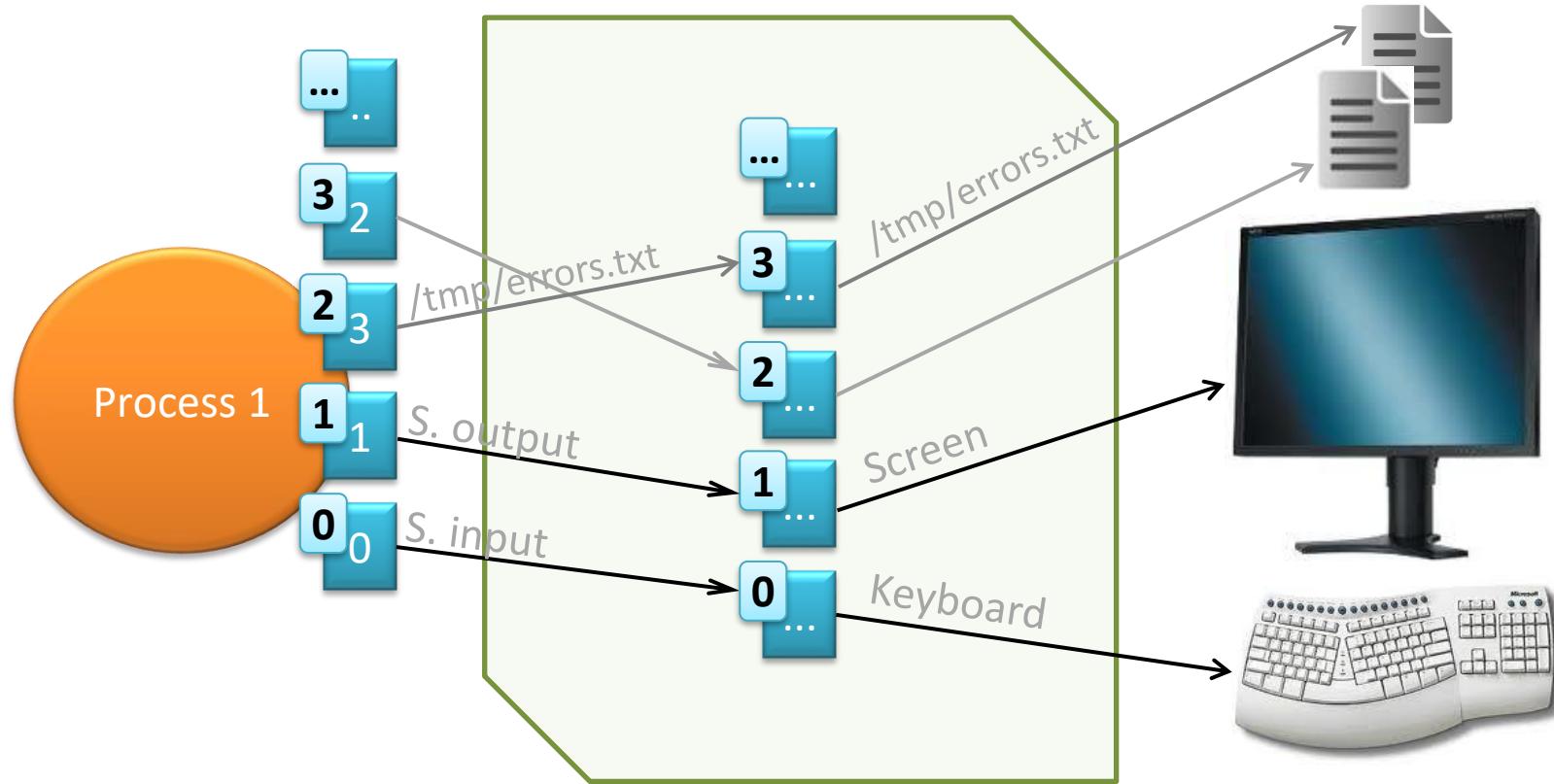


`close(2) + open("/tmp/errors.txt")`

It is possible to change the file associated with a descriptor (redirection).

# File descriptors

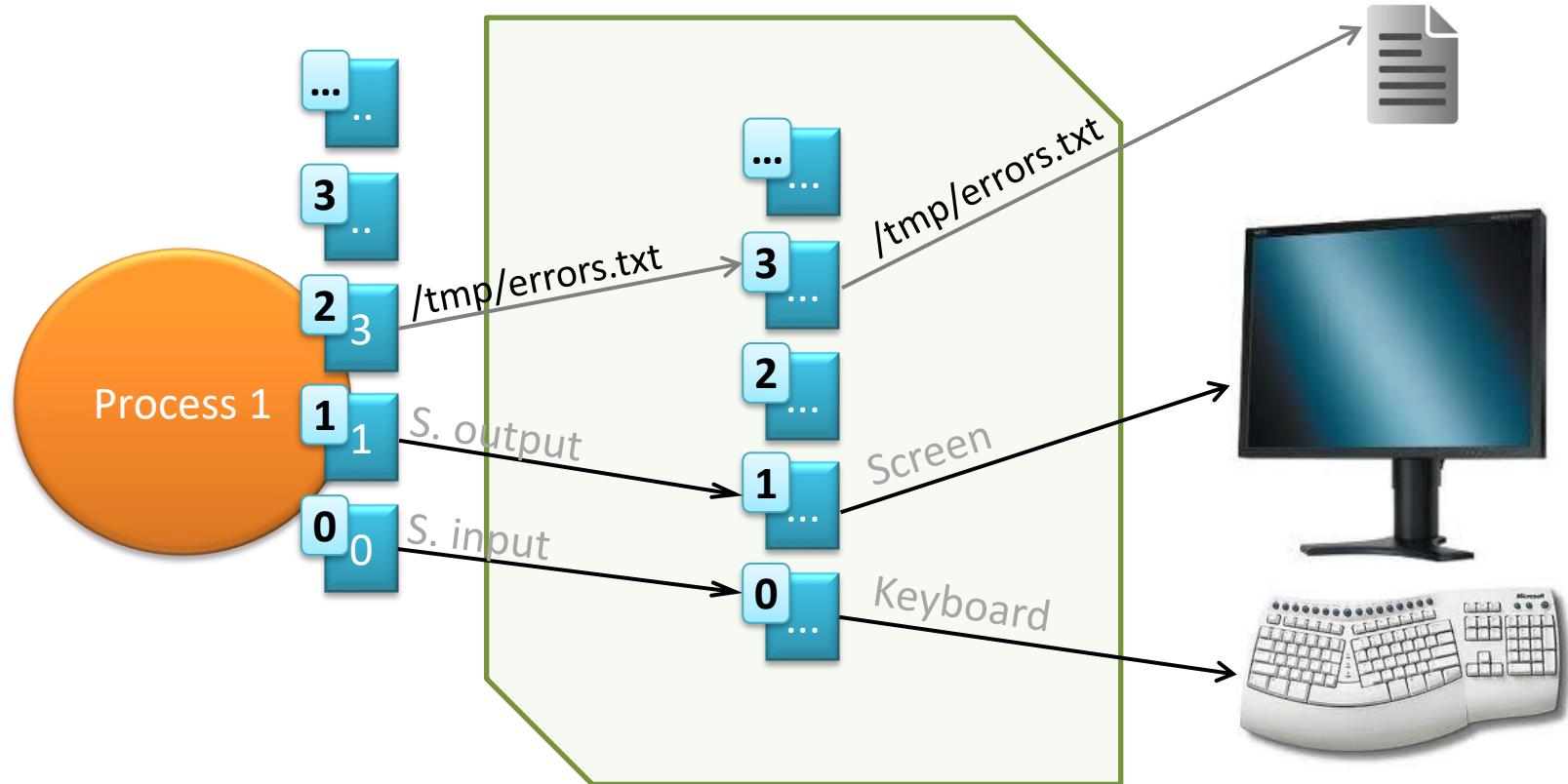
model | redirection | duplication | *fork()*



1. `close(3);`
2. `dup(2);` ?

# File descriptors

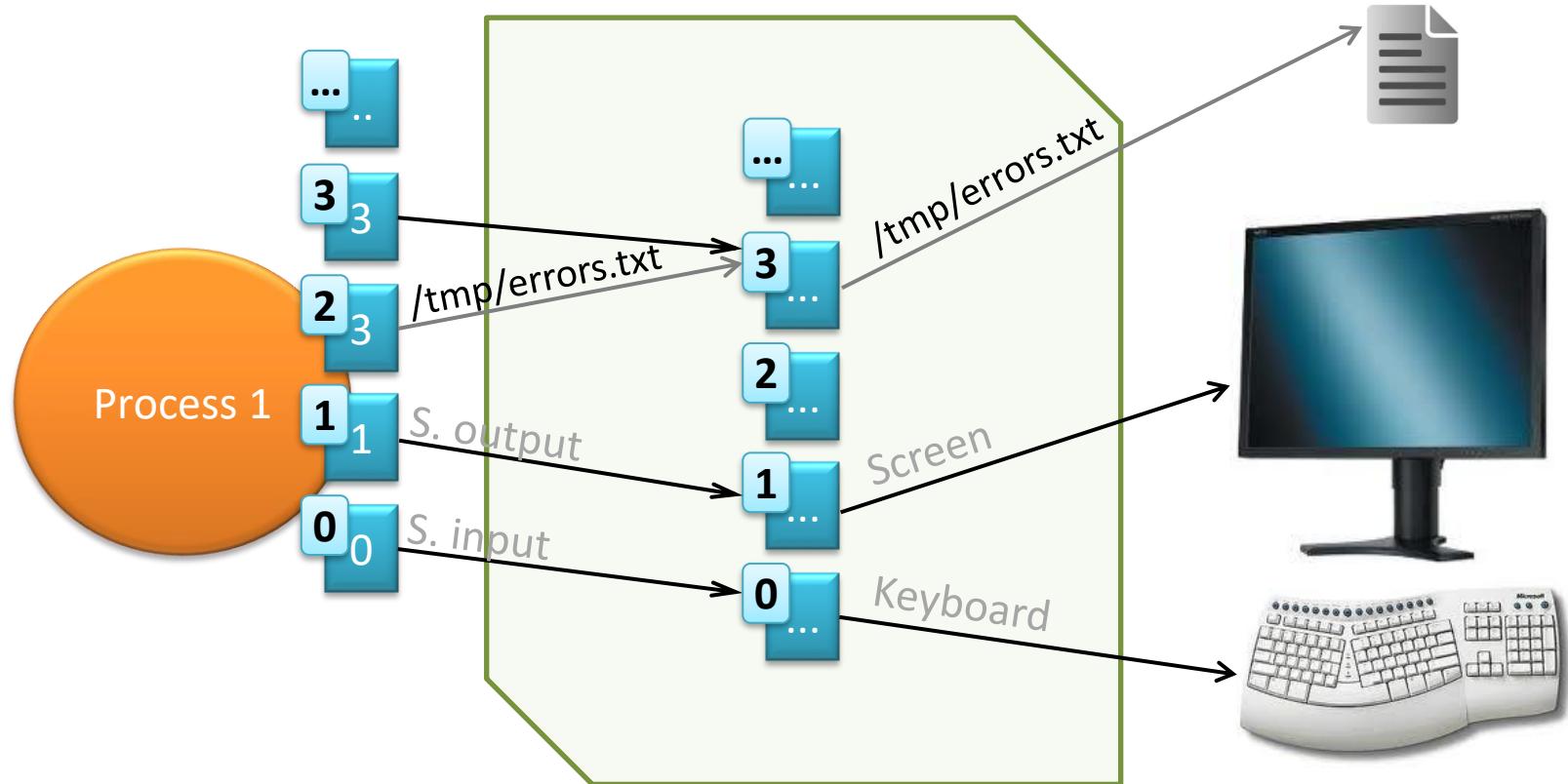
model | redirection | duplication | *fork()*



1. **close(3);**
2. **dup(2);**

# File descriptors

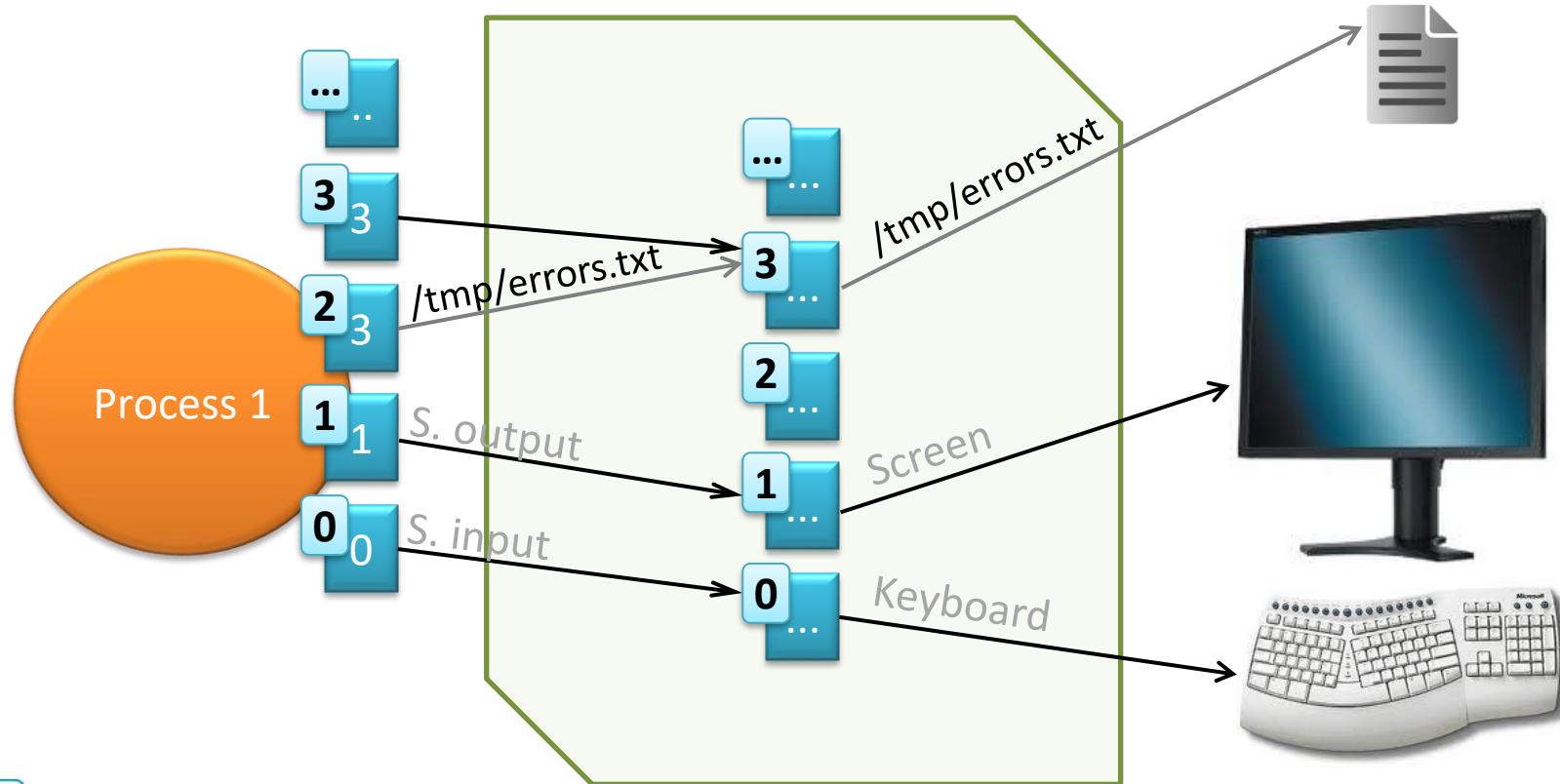
model | redirection | duplication | *fork()*



- 1. `close(3);`
- 2. **`dup(2);`**

# File descriptors

model | redirection | duplication | *fork()*

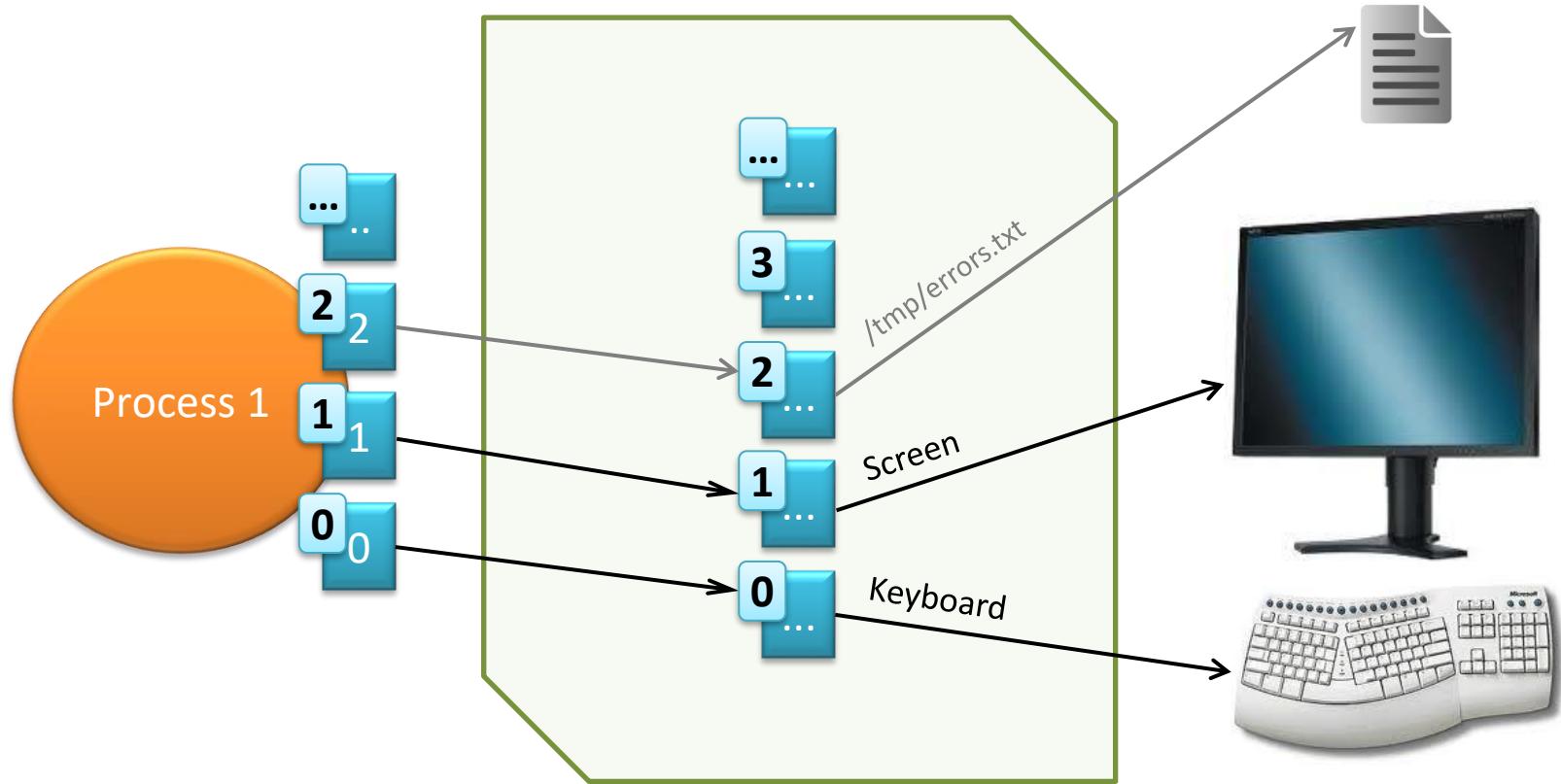


`close(3) + dup(2)`

Allows access to the same file from two different descriptors (duplication).

# File descriptors

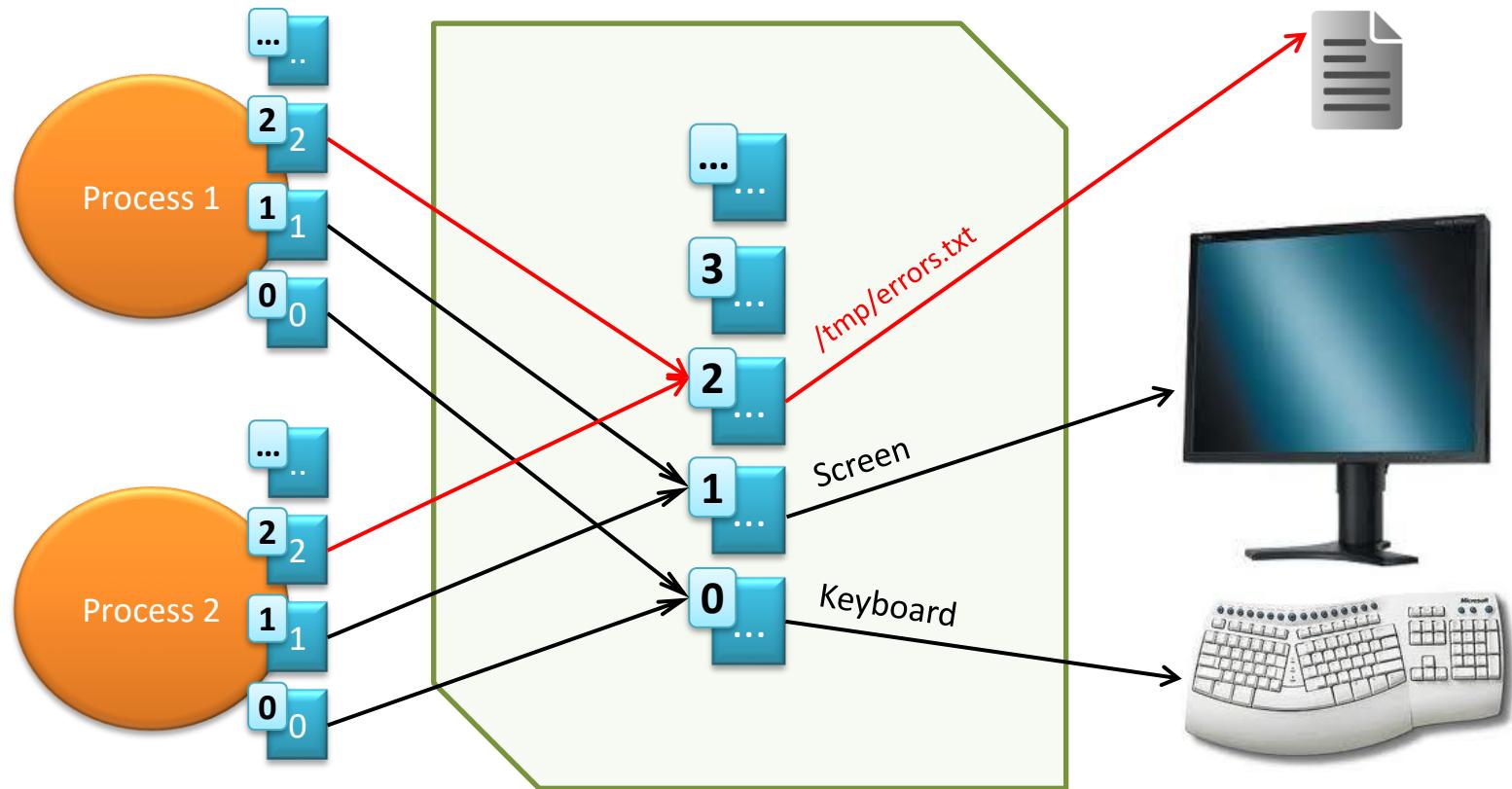
model | redirection | duplication | ***fork()***



`fork()` creates a duplicate of the process (child process) that calls the function (parent process)

# File descriptors

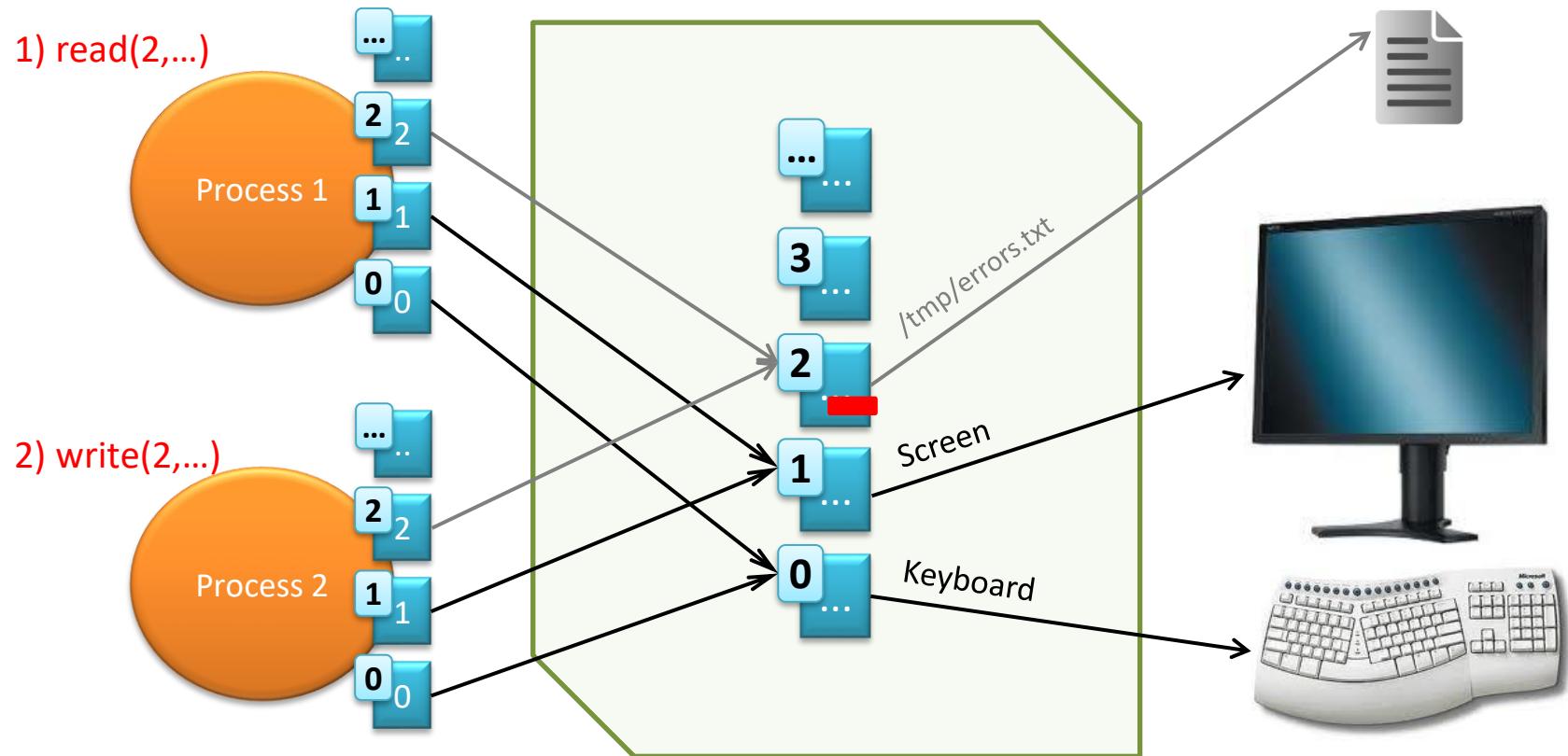
model | redirection | duplication | ***fork()***



- Both have the same descriptors (redirects before `fork()` are inherited).
- Both reference the same elements (common R/W position after `fork()`)

# File descriptors

model | redirection | duplication | **fork()**

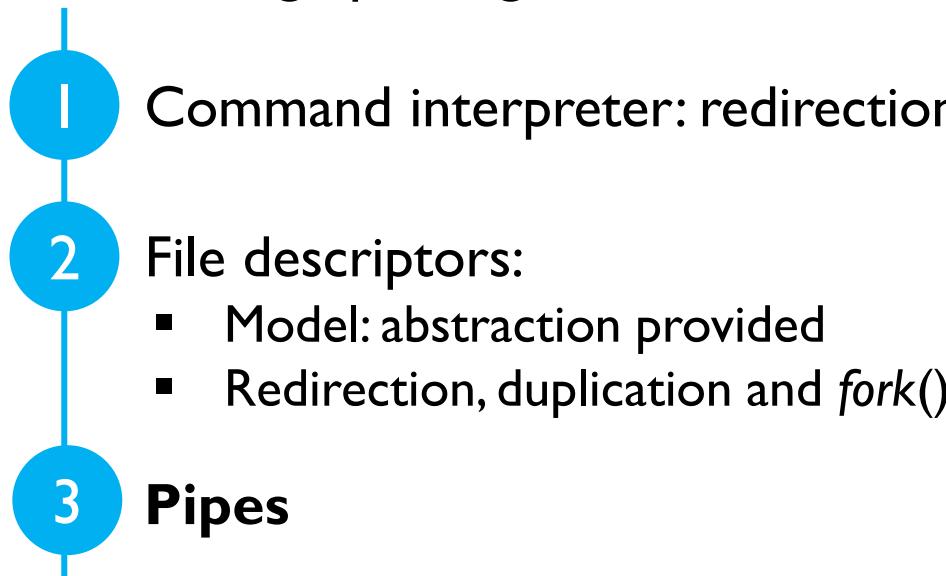


- Both have the same descriptors (redirects before `fork()` are inherited).
- **Both reference the same elements (common R/W position after `fork()`)**

# Contents

---

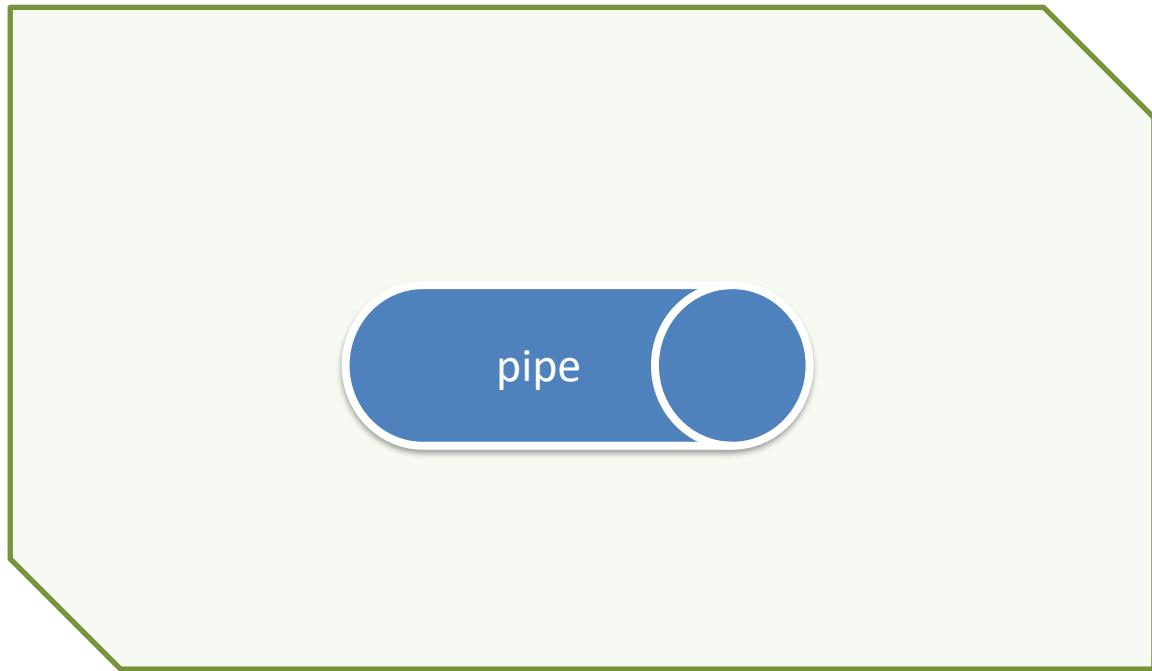
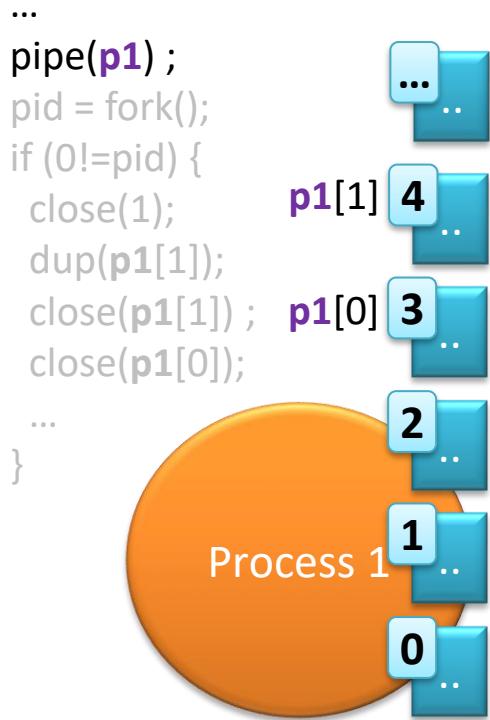
1. Signals and exceptions.
2. Timers.
3. Process environment.
4. Process communication with pipes,  
local message passing



# Pipes

(1) creation + (2) fork() + (3) redirection + (4) cleanup

```
int p1[2] ;
```



A pipe is a special file that is created with the *pipe()* system call.

# Pipes

(1) creation + (2) fork() + (3) redirection + (4) cleanup

```
int p1[2] ;
```

```
...
```

```
pipe(p1) ;
```

```
pid = fork();  
if (0!=pid) {
```

```
close(1);
```

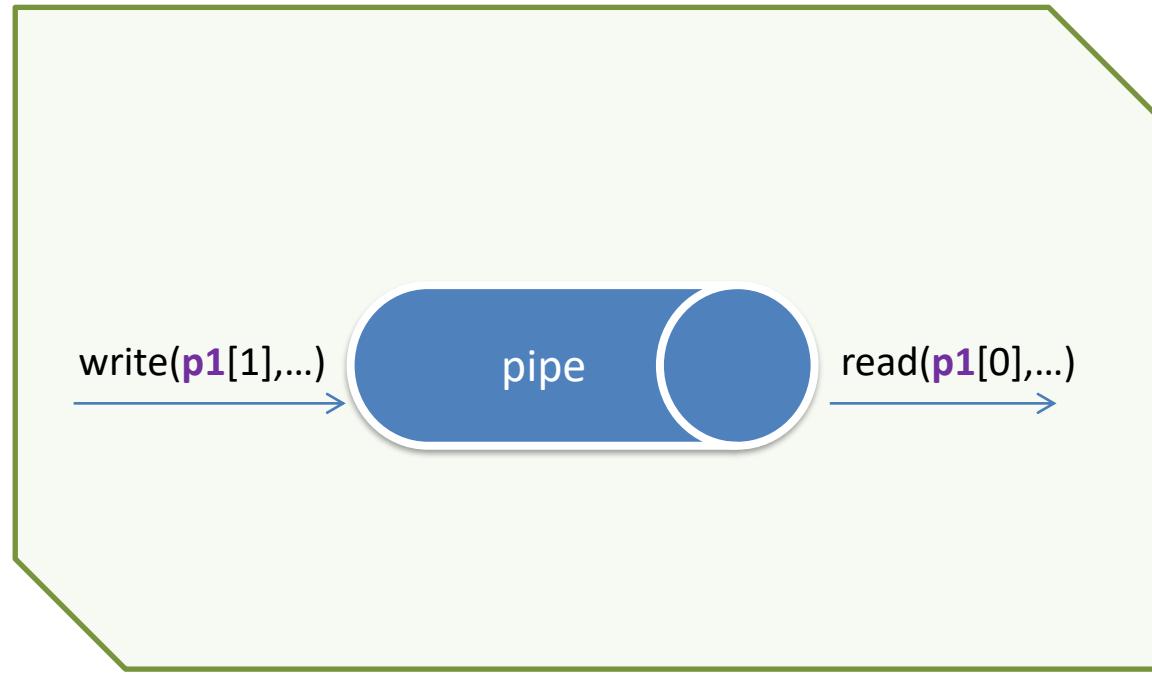
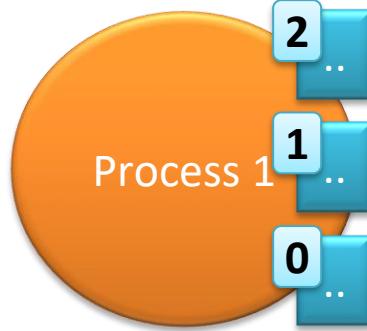
```
dup(p1[1]);
```

```
close(p1[1]);
```

```
close(p1[0]);
```

```
...
```

```
}
```



A pipe is a special file that is created with the *pipe()* system call.  
This call creates the pipe and reserves two file descriptors: read and write.

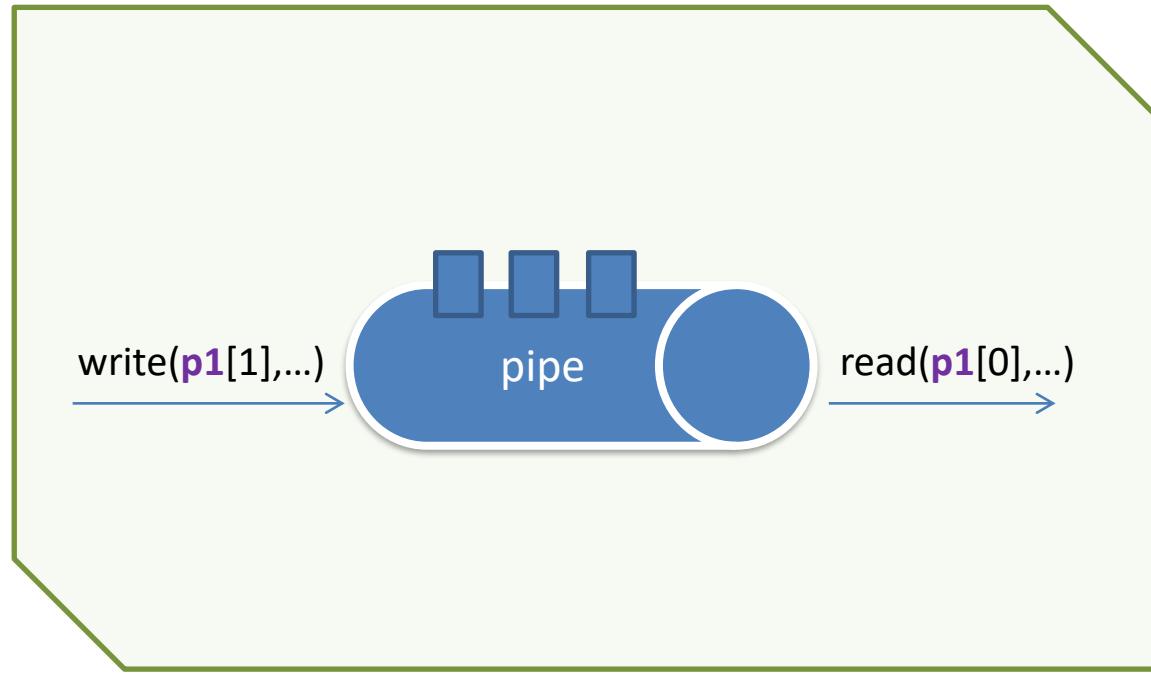
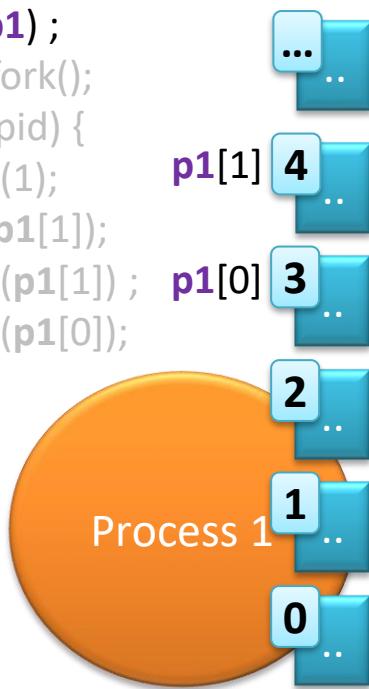
# Pipes

(1) creation + (2) fork() + (3) redirection + (4) cleanup

```
int p1[2] ;
```

```
...  
pipe(p1);  
pid = fork();  
if (0!=pid) {  
    close(1);  
    dup(p1[1]);  
    close(p1[1]);  
    close(p1[0]);  
}
```

```
}
```



- If you write to a full pipe, the execution of the process is blocked until you can write.
- If you read from an empty pipe, process execution is blocked until you can read something.

# Pipes

(1) creation + (2) fork() + (3) redirection + (4) cleanup

```
int p1[2] ;
```

```
...
```

```
pipe(p1) ;
```

```
pid = fork();
```

```
if (0!=pid) {
```

```
close(1);
```

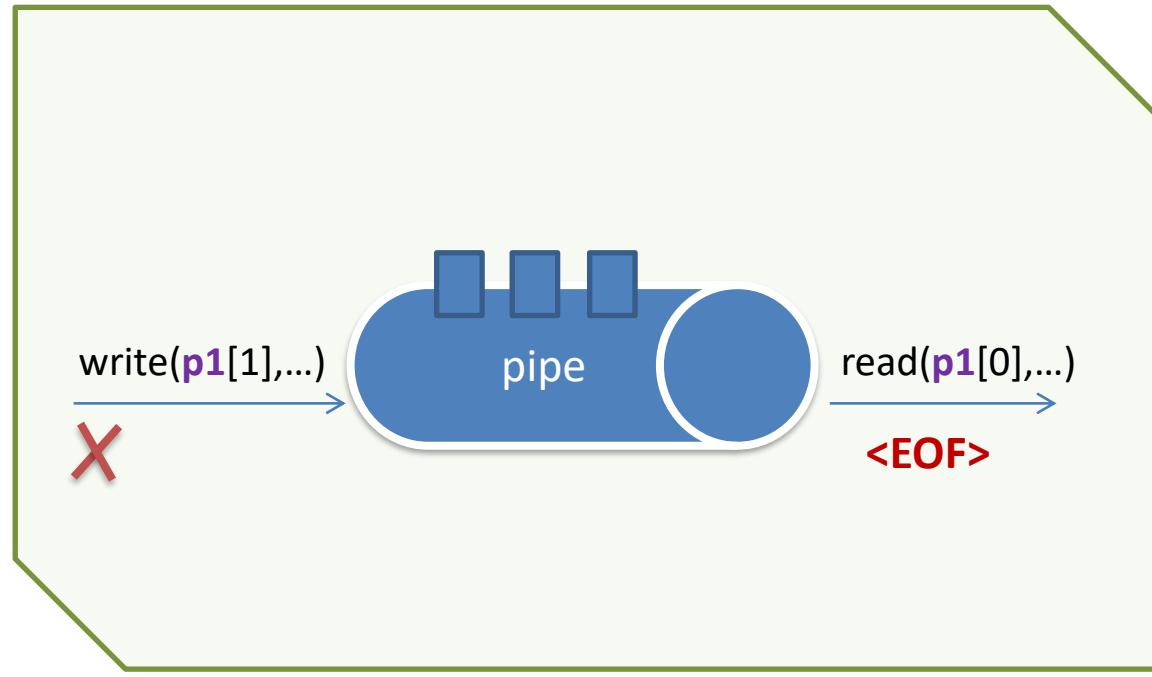
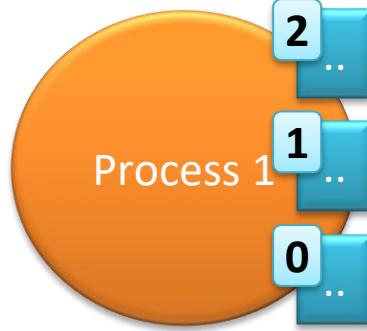
```
dup(p1[1]);
```

```
close(p1[1]);
```

```
close(p1[0]);
```

```
...
```

```
}
```



- When all possible writing processes in the pipe close the writing part, then an end-of-file (EOF) is sent to the readers.

# Pipes

(1) creation + (2) **fork()** + (3) redirection + (4) cleanup

```
int p1[2] ;
```

```
...
```

```
pipe(p1) ;
```

```
pid = fork();
```

```
if (0!=pid) {
```

```
close(1);
```



```
p1[1]
```

```
dup(p1[1]);
```

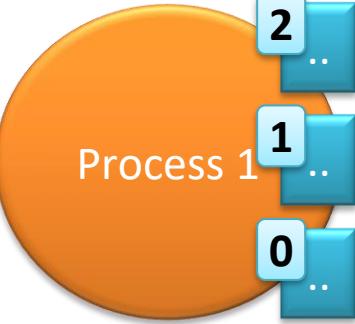


```
p1[0]
```

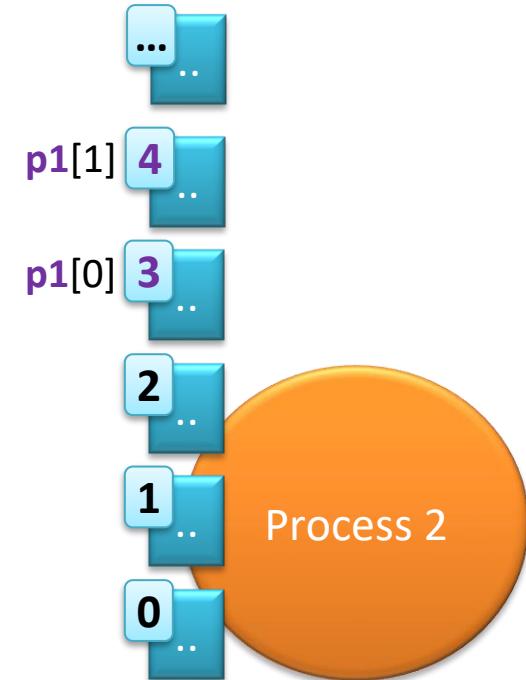
```
close(p1[1]);
```

```
close(p1[0]);
```

```
}
```



Process 1



pipe() + fork() -> father and son view the same pipe

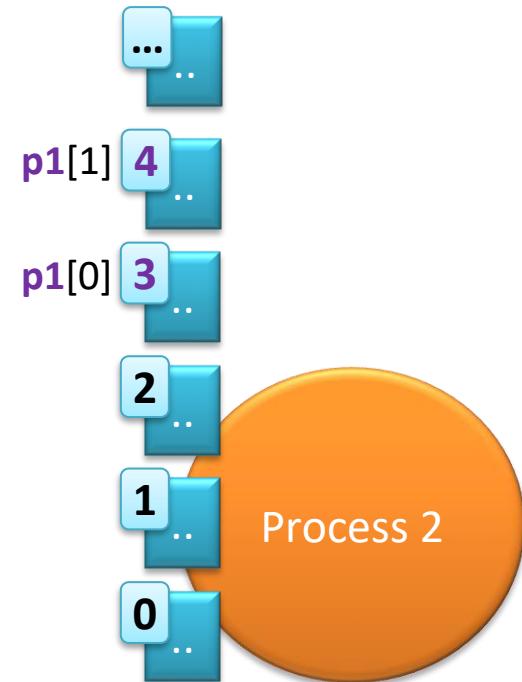
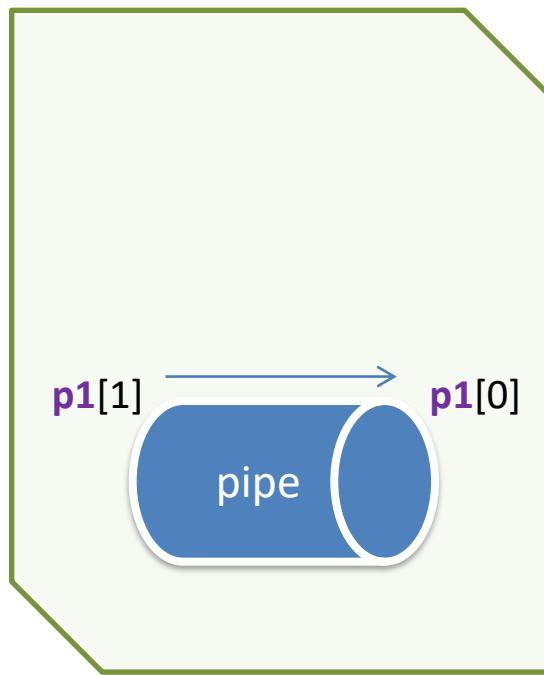
# Pipes

(1) creation + (2) **fork()** + (3) redirection + (4) cleanup

```
int p1[2] ;
```

```
...
```

```
pipe(p1) ;  
pid = fork();  
if (0!=pid) {  
    close(1);  
    dup(p1[1]);  
    close(p1[1]);  
    p1[0] = ...  
    close(p1[0]);  
}  
...  
}
```



pipe() + fork() -> father and son view the same pipe  
-> both could read and write on it

# Pipes

(1) creation + (2) fork() + (3) redirection + (4) cleanup

```
int p1[2] ;
```

```
...
```

```
pipe(p1) ;  
pid = fork();
```

```
if (0!=pid) {
```

```
close(1);
```

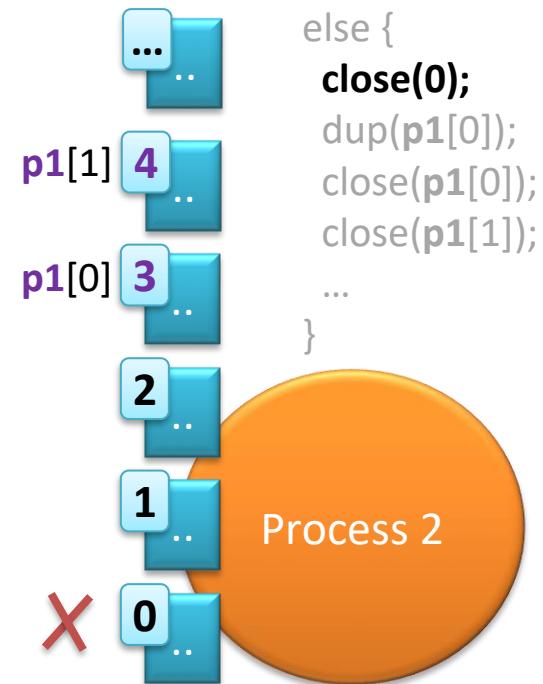
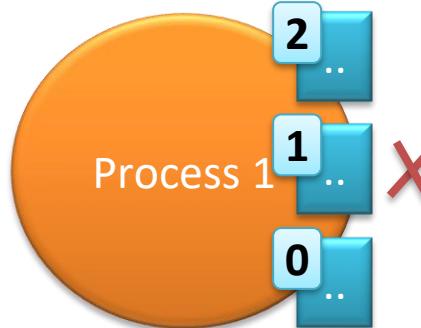
```
dup(p1[1]);
```

```
p1[0] 3  
close(p1[1]) ;
```

```
close(p1[0]);
```

```
...
```

```
}
```



Redirection of the standard output on the parent...

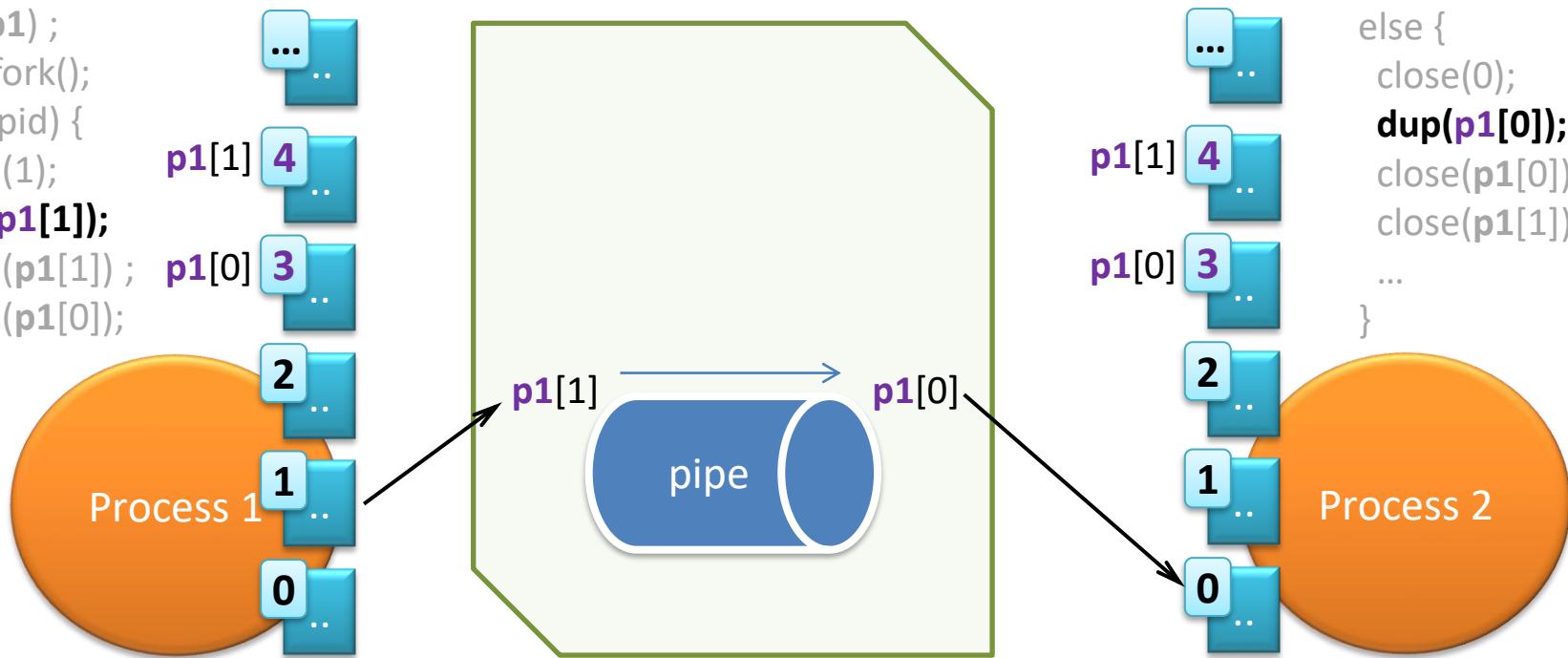
Redirection of the standard input on the child...

# Pipes

(1) creation + (2) fork() + (3) redirection + (4) cleanup

```
int p1[2] ;
```

```
...  
pipe(p1) ;  
pid = fork();  
if (0!=pid) {  
    close(1);  
    dup(p1[1]);  
    close(p1[1]) ;  
    close(p1[0]);  
}  
...  
}
```



Redirection of the standard output on the parent...

Redirection of the standard input on the child...

# Pipes

(1) creation + (2) fork() + (3) redirection + **(4) cleanup**

```
int p1[2] ;
```

```
...
```

```
pipe(p1) ;
```

```
pid = fork();
```

```
if (0!=pid) {
```

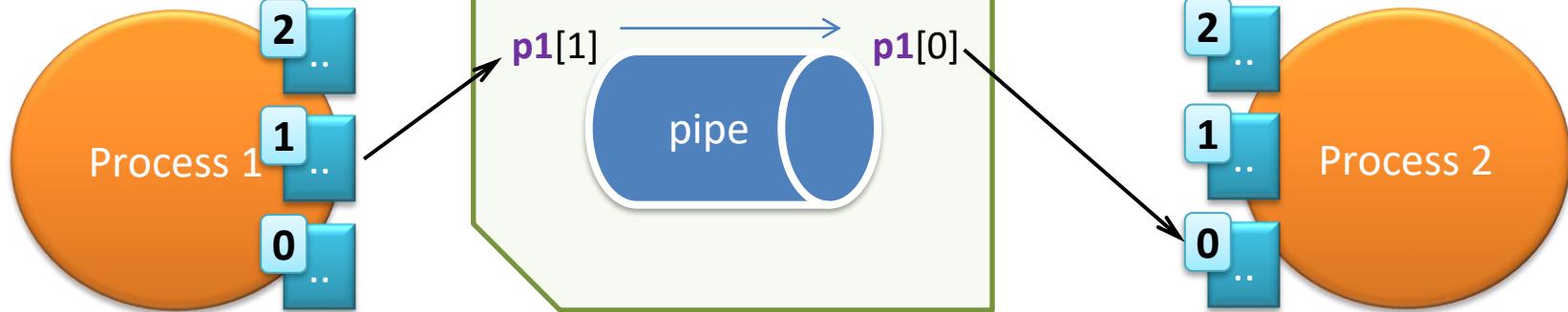
```
close(1);
```

```
dup(p1[1]);
```

```
close(p1[1]);
```

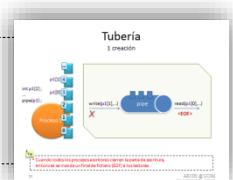
```
close(p1[0]);
```

```
}
```



Closing of descriptors that are not used in the parent...

Closure of descriptors that are not used in the child...



# File descriptors

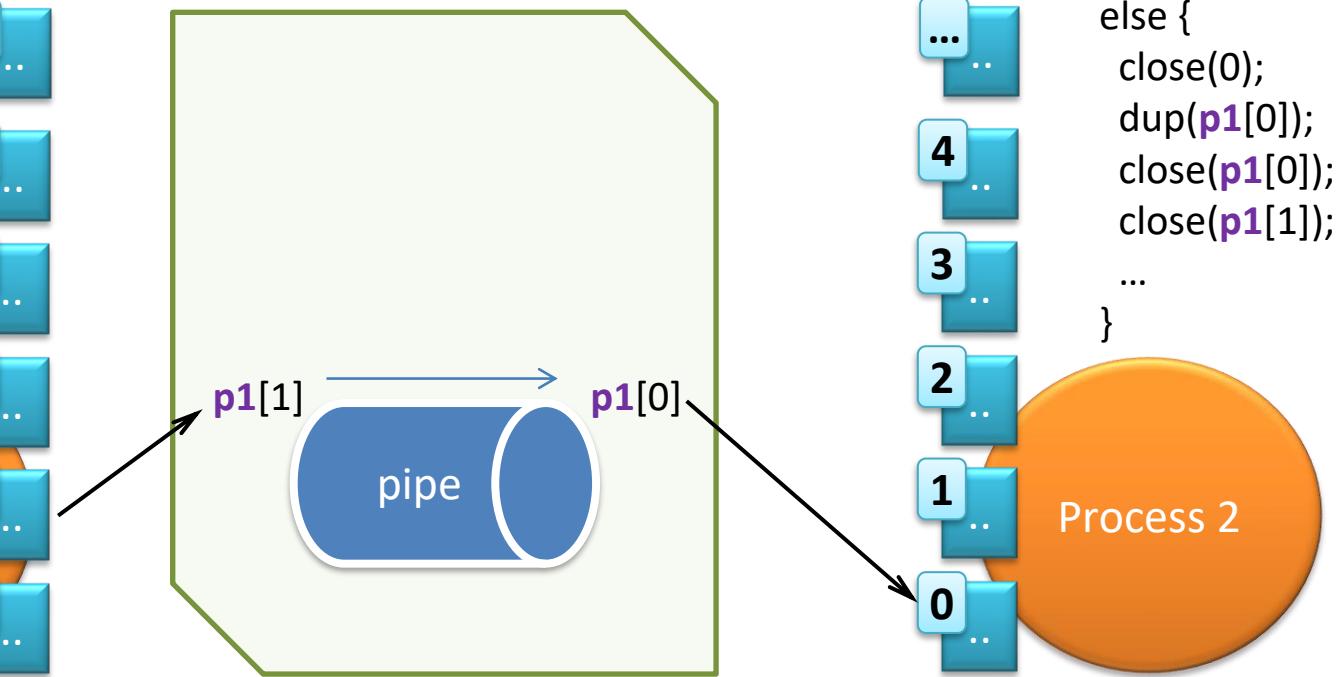
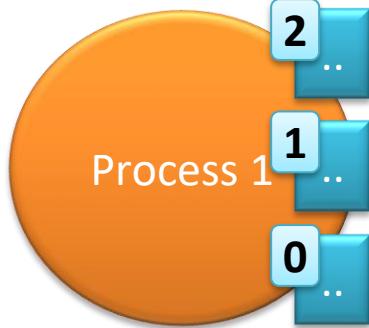
## summary

```
int p1[2] ;
```

```
...
```

```
pipe(p1) ;  
pid = fork();  
if (0!=pid) {  
    close(1);  
    dup(p1[1]);  
    close(p1[1]) ;  
    close(p1[0]);  
}
```

```
}
```



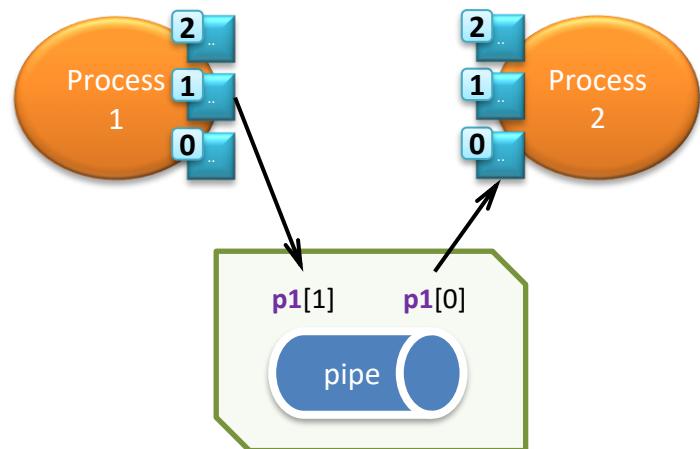
# File descriptors

## summary

```
int p1[2] ;  
...  
pipe(p1) ;  
pid = fork();  
if (0!=pid) {  
    close(1);  
    dup(p1[1]);  
    close(p1[1]);  
    close(p1[0]);  
    ...  
}  
} else {  
    close(0);  
    dup(p1[0]);  
    close(p1[0]);  
    close(p1[1]);  
    ...  
}  
}
```

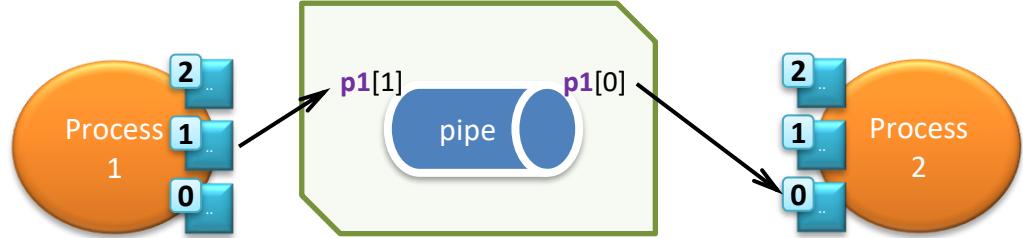
The code illustrates the creation and manipulation of file descriptors for a pipe. It shows the following steps:

- 1) Creation: The pipe is created using `pipe(p1)`.
- 2) `fork()`: The process is forked.
- 3) Redirection (father): The father process performs redirection operations:
  - Closes descriptor 1.
  - Duplicates descriptor `p1[1]` to descriptor 1.
  - Closes descriptor `p1[1]`.
  - Closes descriptor `p1[0]`.
- 4) Cleanup (father): The father process performs cleanup operations:
  - Closes descriptor `p1[1]`.
  - Closes descriptor `p1[0]`.
- 3) Redirection (son): The son process performs redirection operations:
  - Closes descriptor 0.
  - Duplicates descriptor `p1[0]` to descriptor 0.
  - Closes descriptor `p1[0]`.
  - Closes descriptor `p1[1]`.
- 4) Cleanup (son): The son process performs cleanup operations:
  - Closes descriptor `p1[1]`.
  - Closes descriptor `p1[0]`.



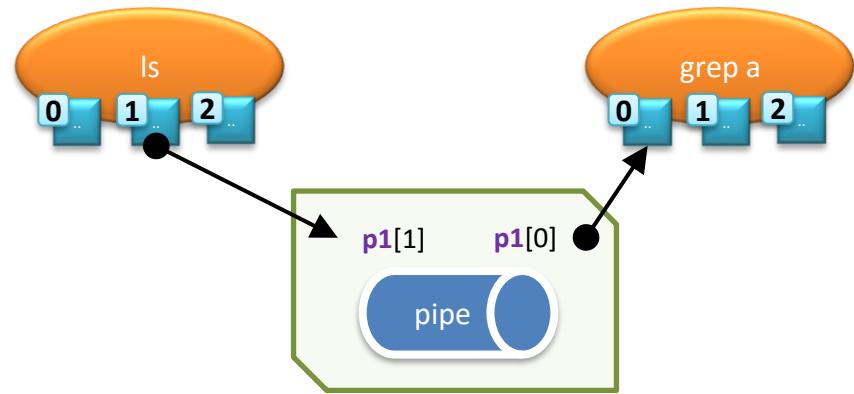
# Pipes

## limitations



- **Semi-duplex:**
  - One way: data is written by one process at one end of the pipe and read by another process from the other end of the pipe.
- They can only be used between **related processes** that have a common ancestor.
- **Reading is destructive.**

# Example 1: "ls | grep a"

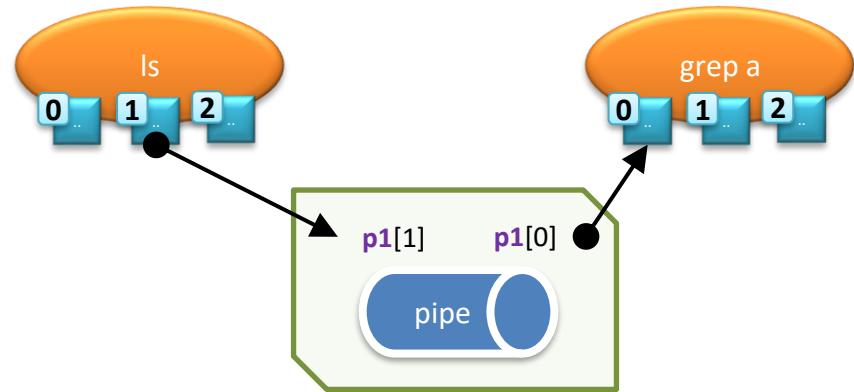


# Example 1: "ls | grep a"

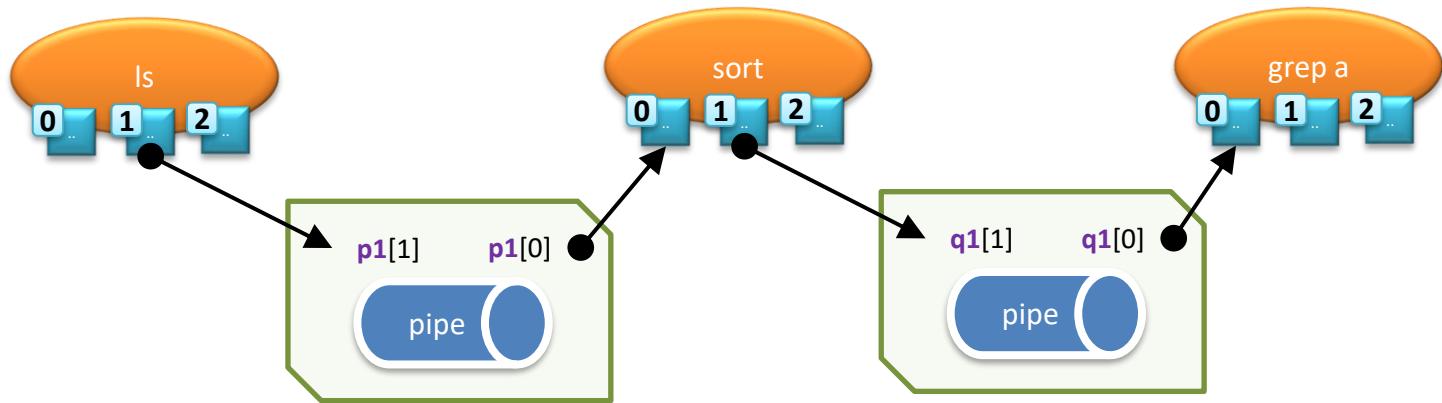
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char *argv[])
{
    int fd[2];

    pipe(fd);
    if (fork() !=0) { /* code for father */
        close(STDIN_FILENO);
        dup(fd[STDIN_FILENO]);
        close(fd[STDIN_FILENO]);
        close(fd[STDOUT_FILENO]);
        execvp("grep", "grep", "a", NULL);
    } else { /* code for son */
        close(STDOUT_FILENO);
        dup(fd[STDOUT_FILENO]);
        close(fd[STDOUT_FILENO]);
        close(fd[STDIN_FILENO]);
        execvp("ls", "ls", NULL);
    }
    return 0;
}
```

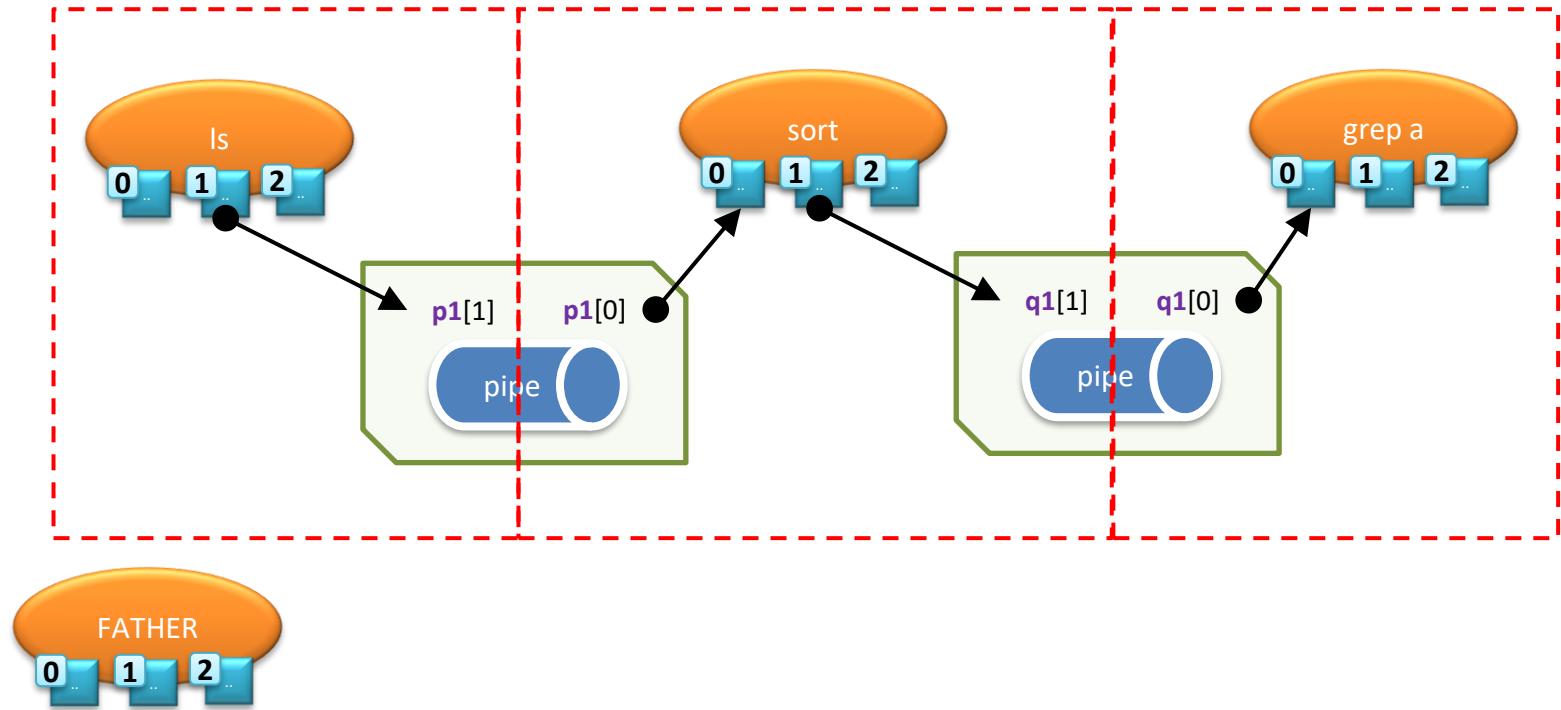


## Example 2: "ls | sort | grep a"



It is possible to work with several pipes in a similar way.  
A "FATHER" process is used to create the structure.

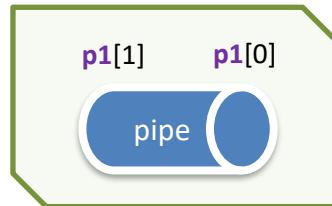
# Example 2: "ls | sort | grep a"



The idea is to apply a "divide and conquer" approach on each pipe and its associated processes. Three types of cases: first process, intermediate processes and last process.

# Example 2: "ls | sort | grep a"

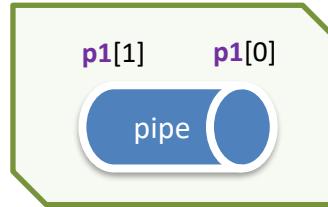
## (1/4) creation



The 4 steps should be done for each pipe (**pipe**, fork, redirection and cleanup)

# Example 2: "ls | sort | grep a"

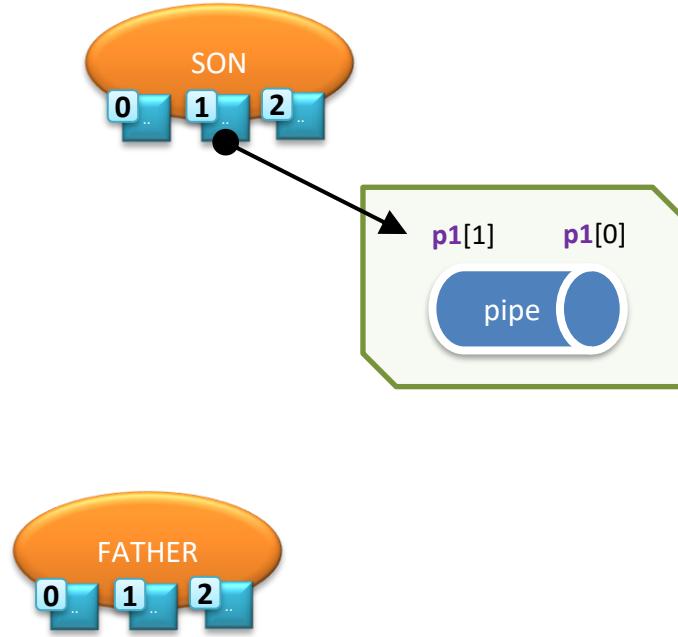
(2/4) fork()



The 4 steps should be done for each pipe (pipe, **fork**, redirection and cleanup)

# Example 2: "ls | sort | grep a"

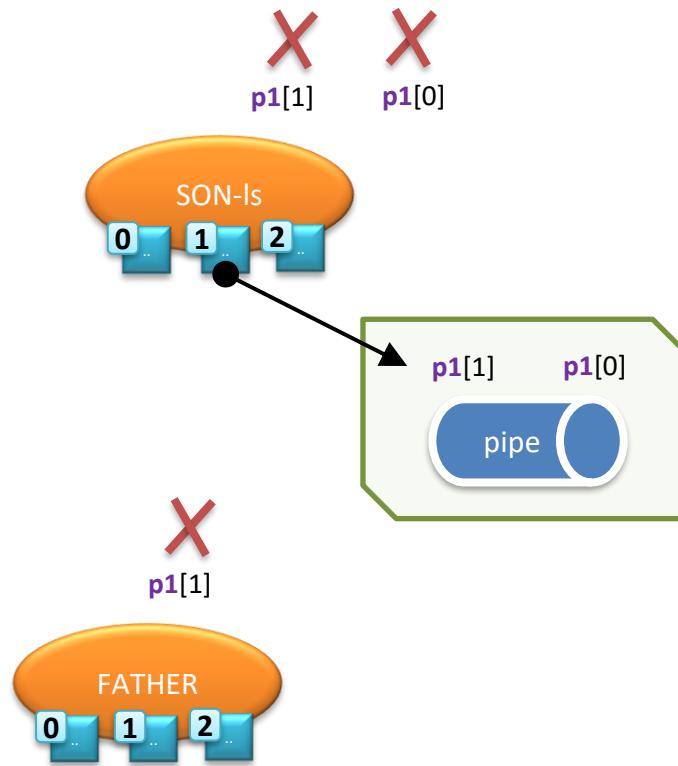
## (3/4) redirection



The 4 steps should be done for each pipe (pipe, fork, **redirection** and cleanup)  
**The first process only uses the pipe in the output redirection.**

# Example 2: "ls | sort | grep a"

## (4/4) cleanup (+exec in son)

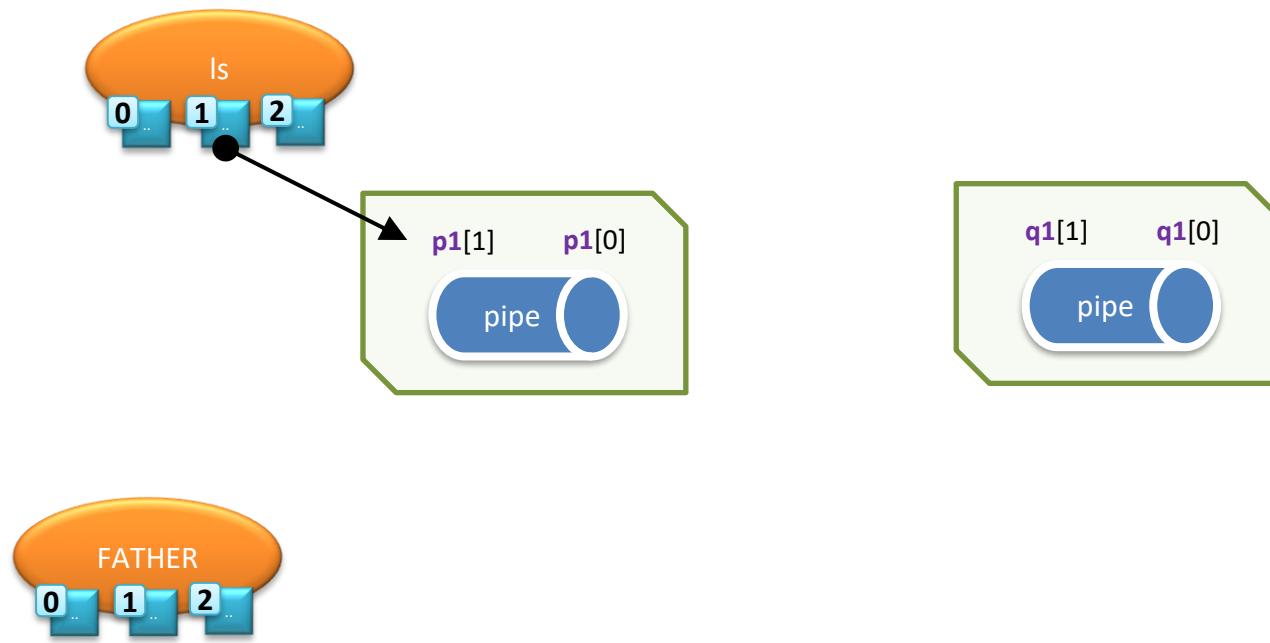


The 4 steps should be done for each pipe (pipe, fork, redirection and **cleanup**)

**The father would still not do a total cleanup for the next child to have access to "p1[0]".**

# Example 2: "ls | sort | grep a"

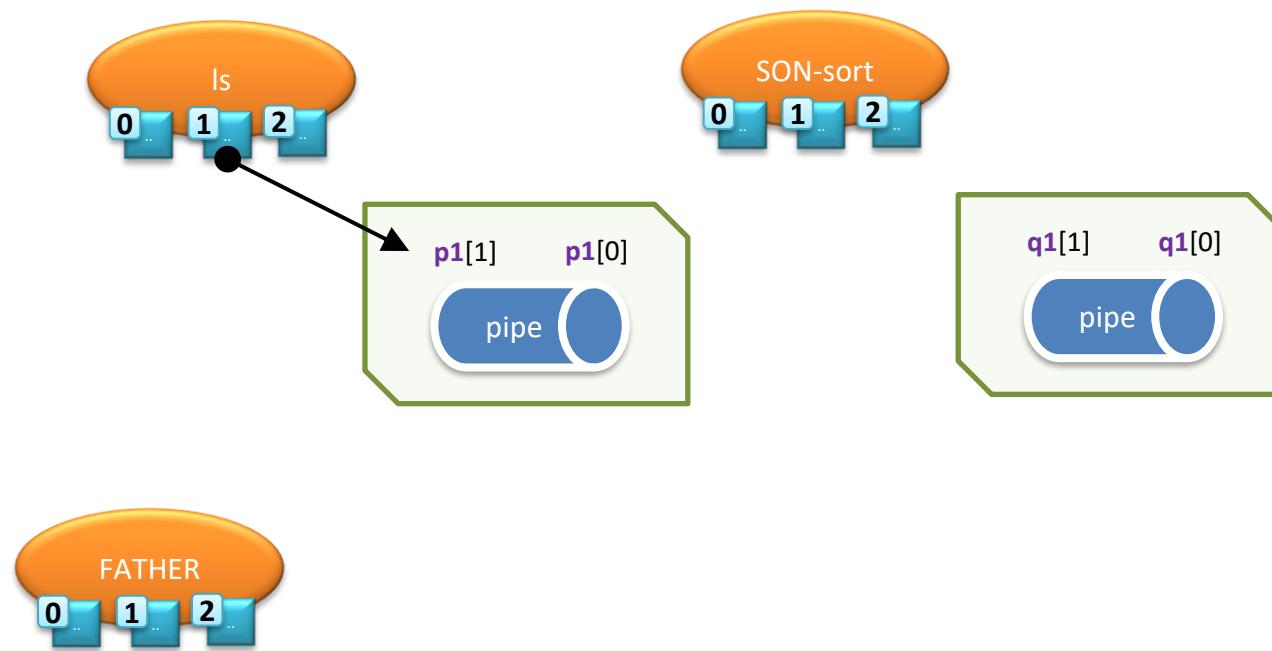
## (1/4) creation



The 4 steps should be done for each pipe (**creation**, fork, redirection and cleanup)

# Example 2: "ls | sort | grep a"

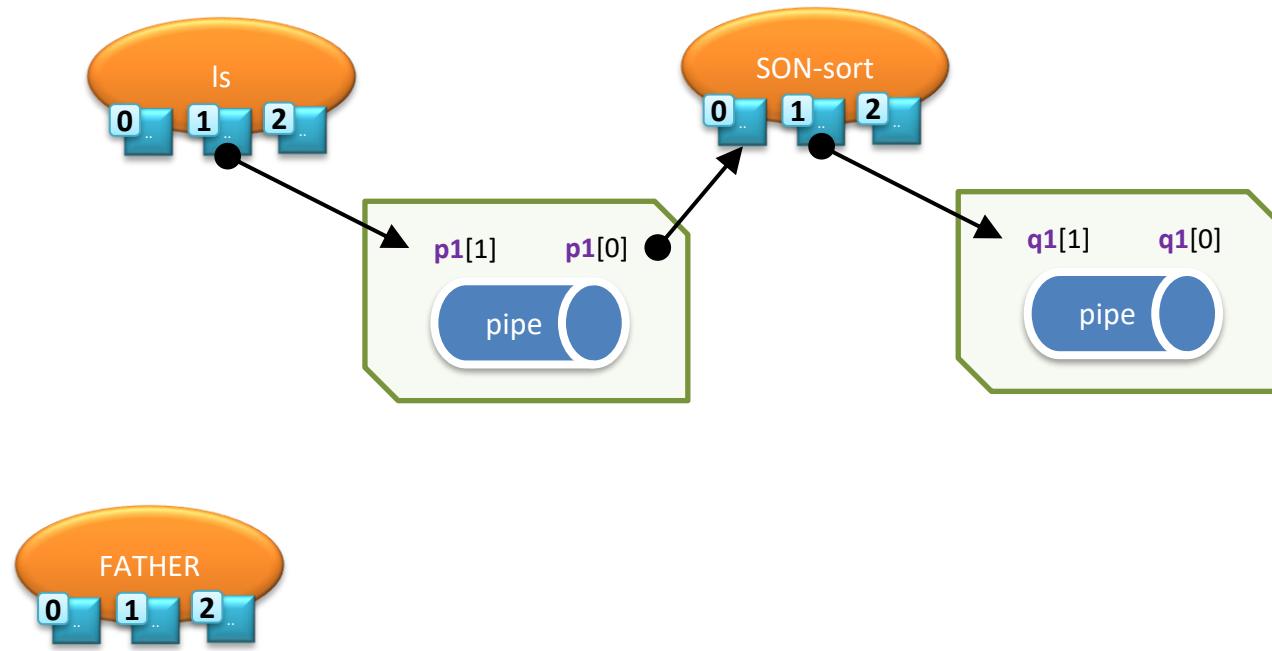
(2/4) fork()



The 4 steps should be done for each pipe (creation, **fork**, redirection and cleanup)

# Example 2: "ls | sort | grep a"

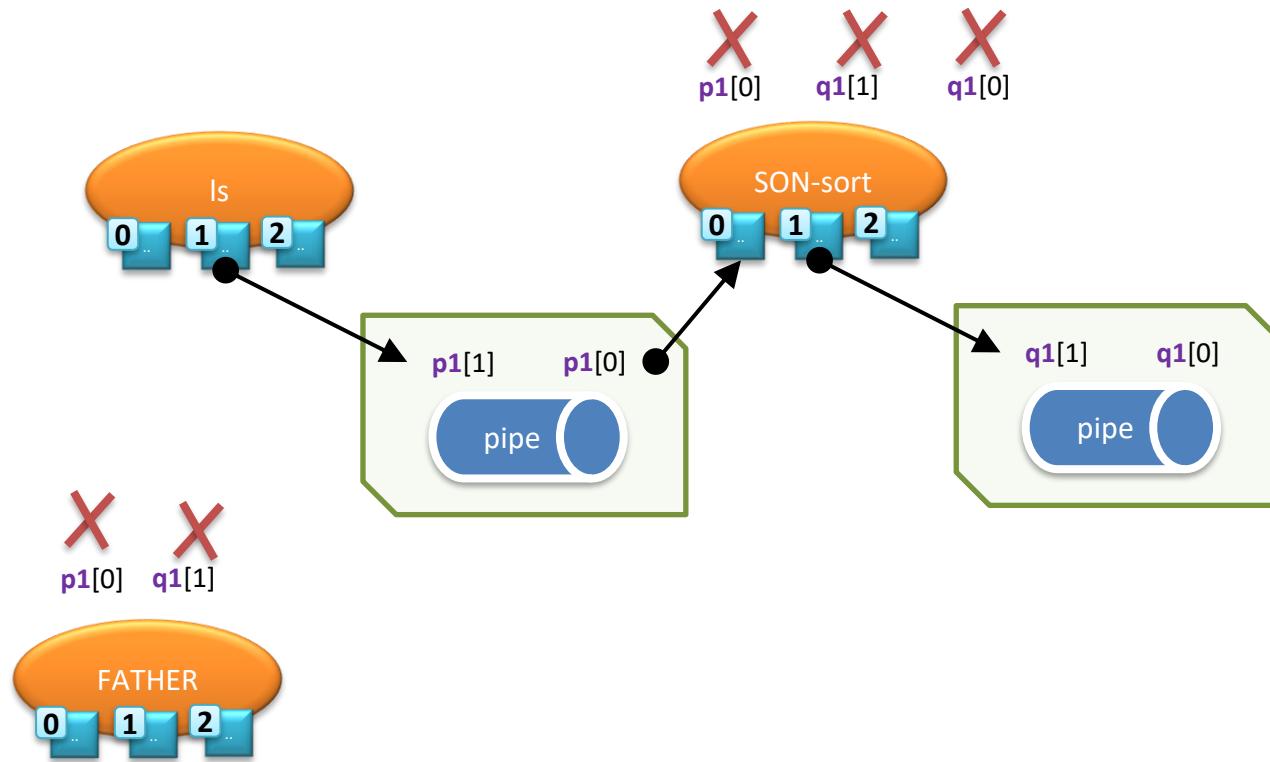
## (3/4) redirection



The 4 steps should be done for each pipe (creation, fork, **redirection** and cleanup)  
**Intermediate processes redirect their standard input and output to pipe.**

# Example 2: "ls | sort | grep a"

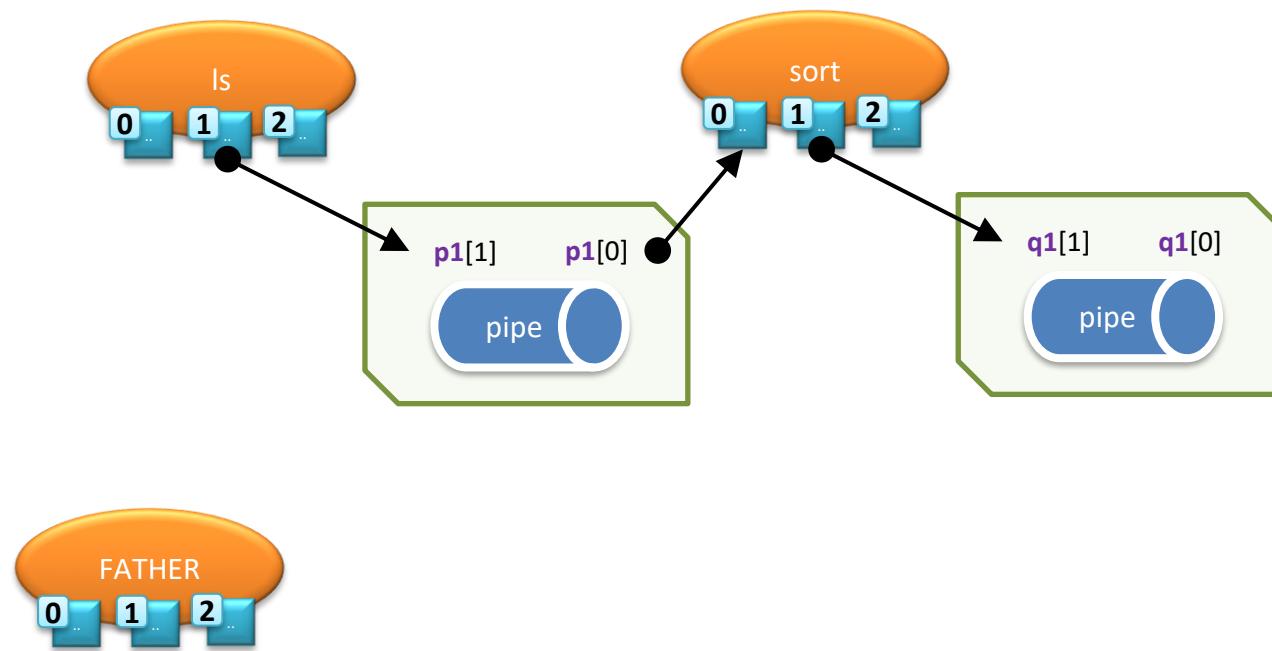
## (4/4) cleanup (+exec in son)



The 4 steps should be done for each pipe (creation, fork, redirection and cleanup)  
The father would still not do total a cleanup so next child would have access to "q1[0]"

# Example 2: "ls | sort | grep a"

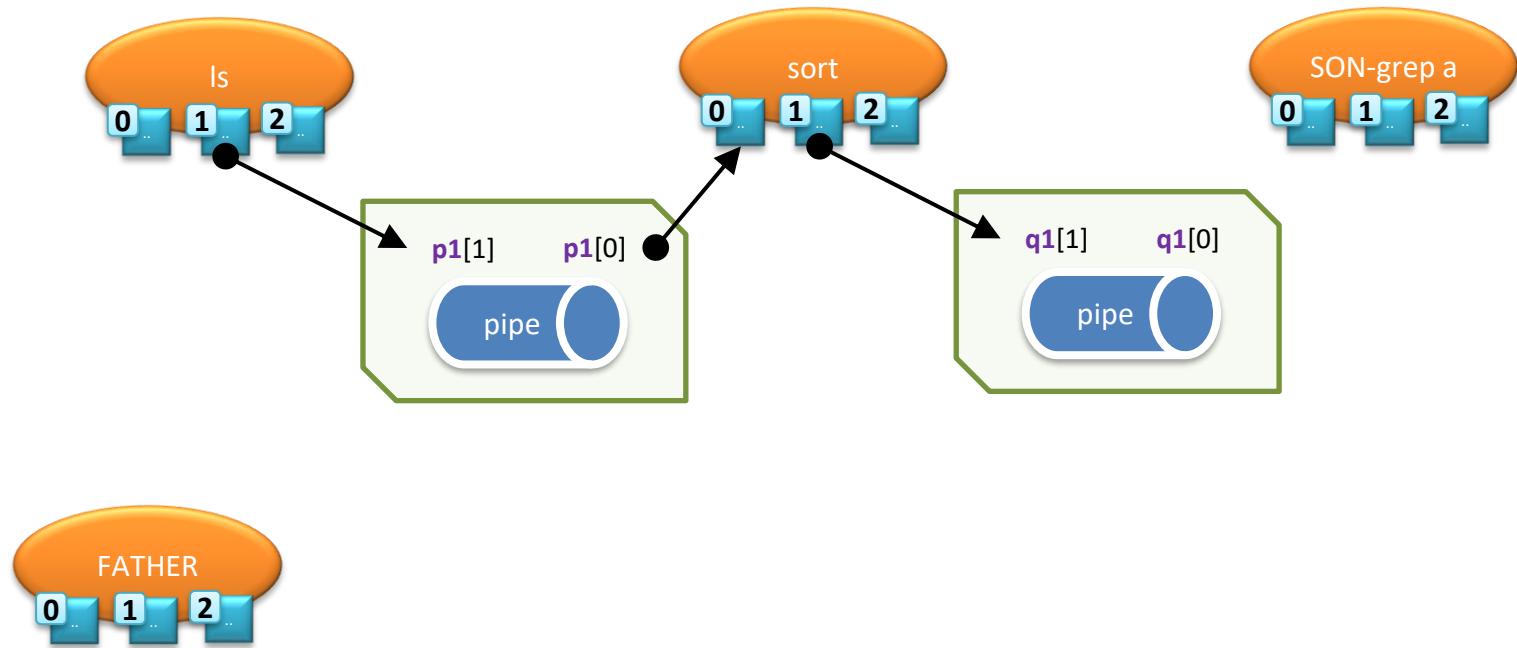
## (1/4) creation



The 4 steps should be done for each pipe (**creation**, fork, redirection and cleanup)  
The last process uses the pipe already created (there are n processes and n-1 pipes)

# Example 2: "ls | sort | grep a"

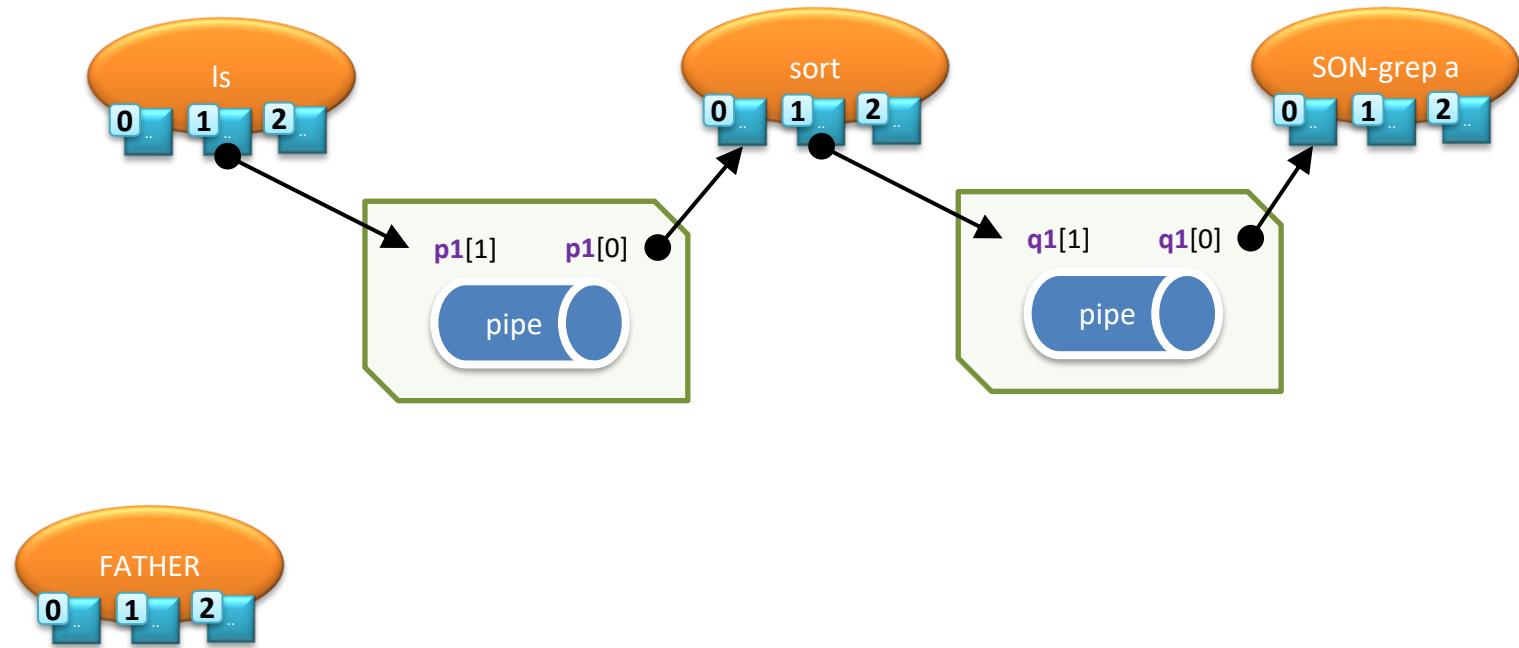
(2/4) fork()



The 4 steps should be done for each pipe (creation, **fork**, redirection and cleanup)

# Example 2: "ls | sort | grep a"

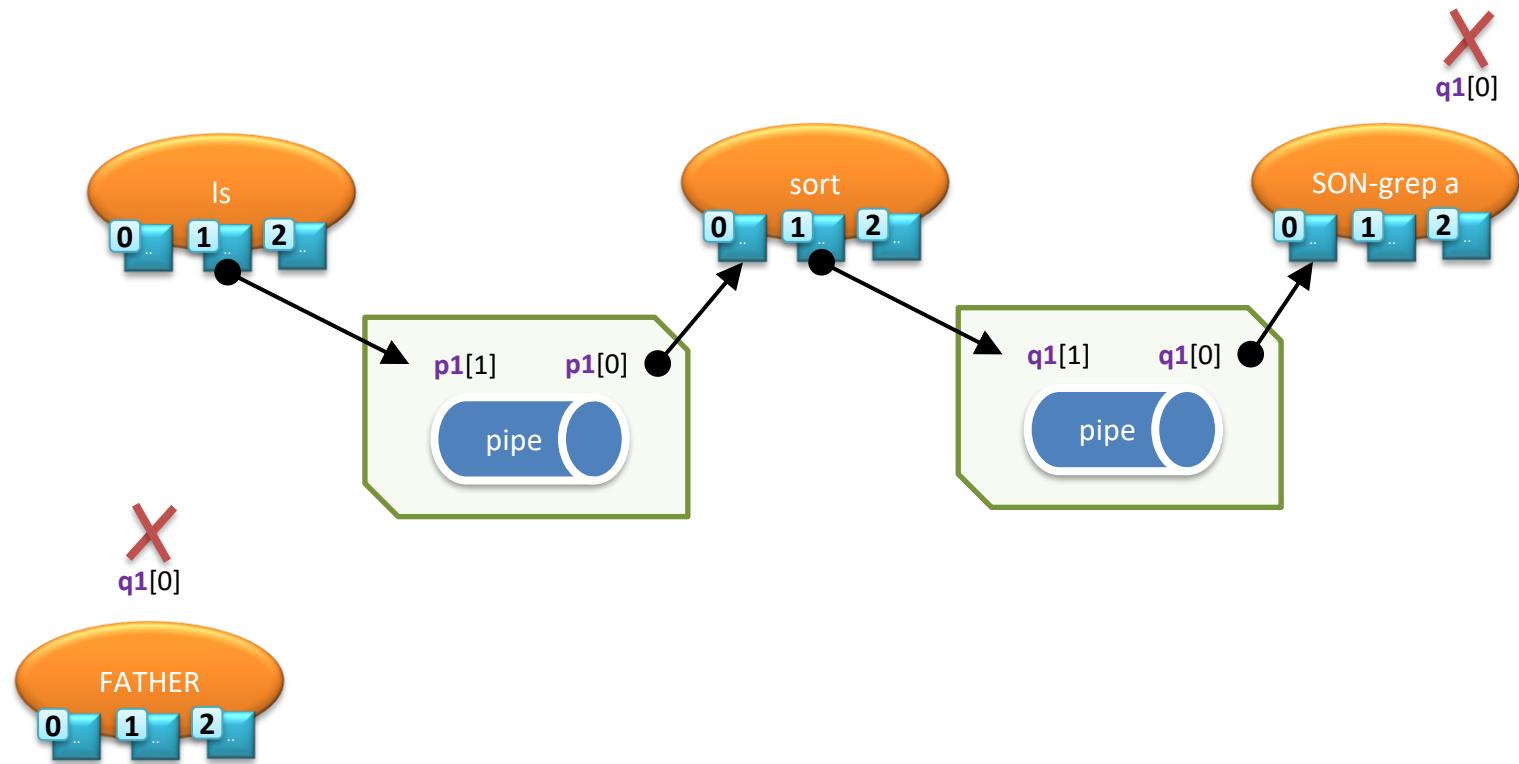
## (3/4) redirection



The 4 steps should be done for each pipe (creation, fork, **redirection** and cleanup)  
**The last process uses the pipe for redirection on its standard input.**

# Example 2: "ls | sort | grep a"

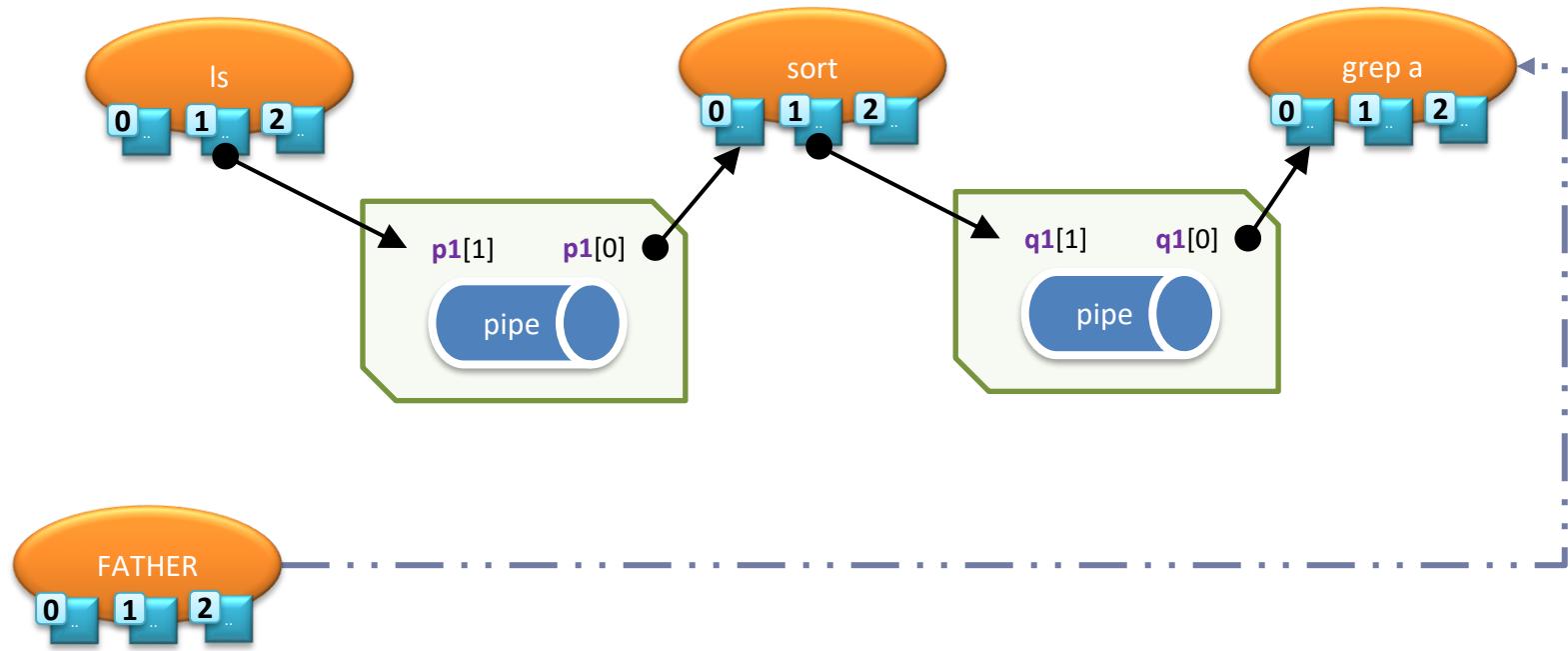
## (4/4) cleanup (+exec in son)



The 4 steps should be done for each pipe (creation, fork, redirection and cleanup)  
The parent process closes all remaining descriptors.

## Example 2: "ls | sort | grep a"

**if (! bg) then wait(...) for the last son created**



If it is not a *background task*, then the parent process does a *wait(...)* waiting for the last process (which becomes "grep a")

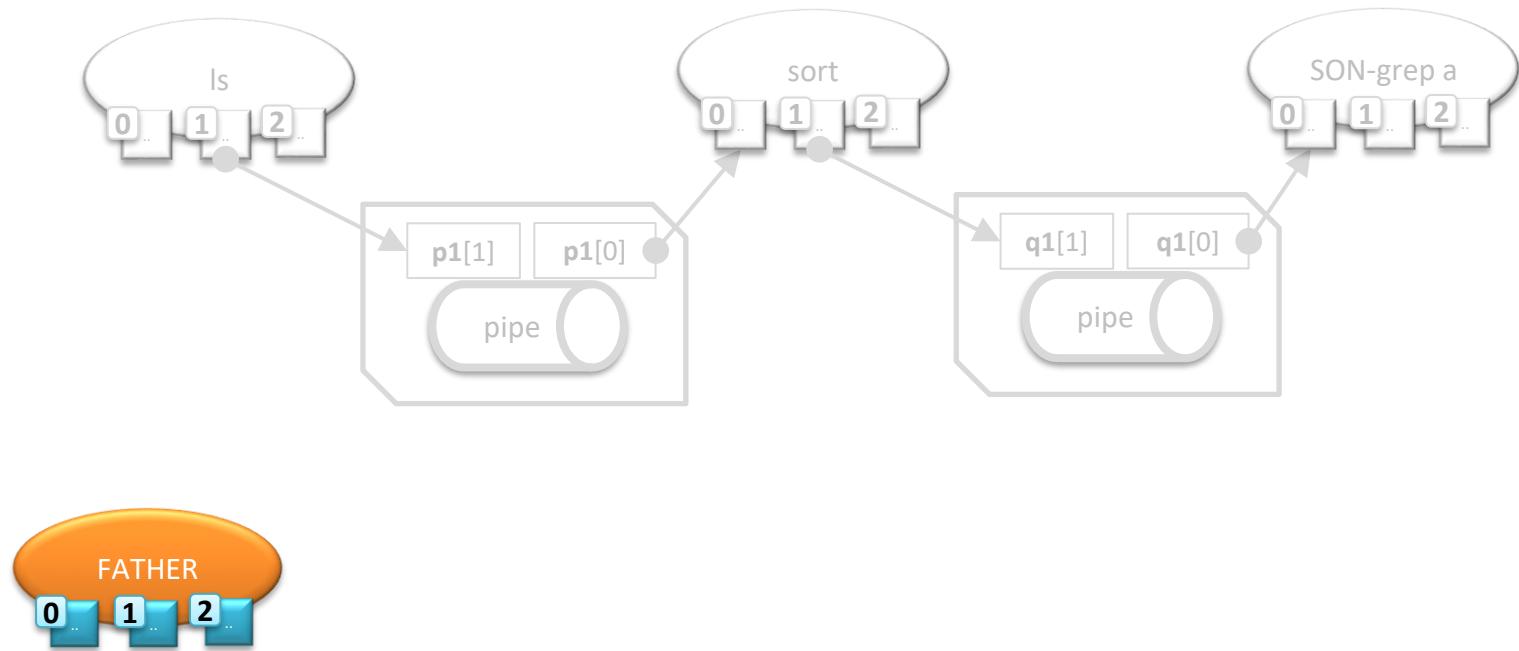
# Example 2: "ls | sort | grep a"



The 4 steps should be done for each pipe (pipe, fork, redirection and cleanup)

**Take 2: generalization for a number of processes  $\geq 3$ .**

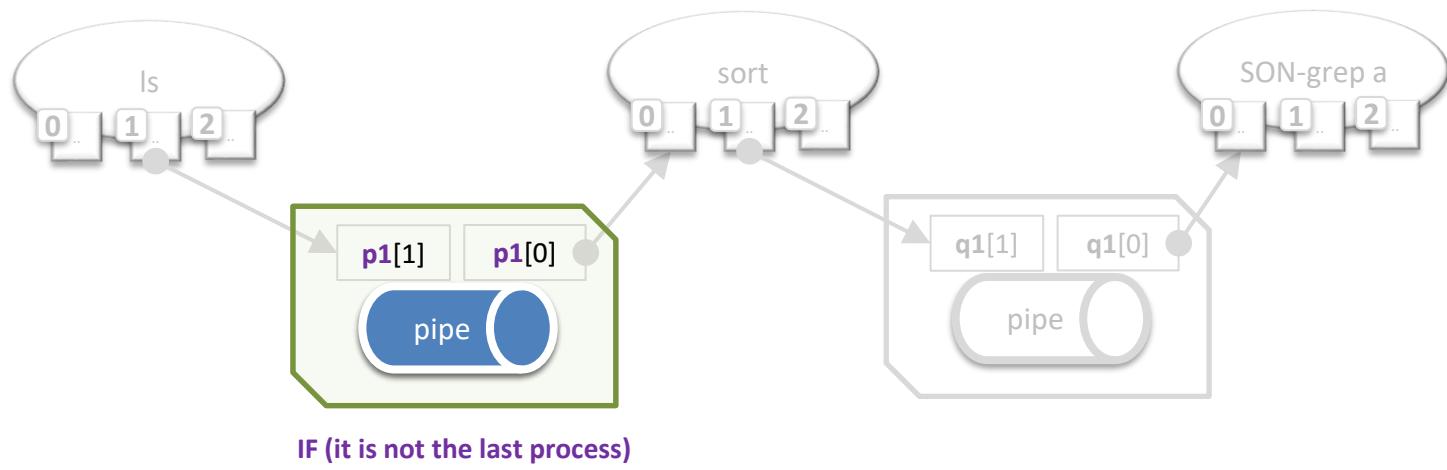
# Example 2: "ls | sort | grep a"



The 4 steps should be done for each pipe (pipe, fork, redirection and cleanup)

# Example 2: "ls | sort | grep a"

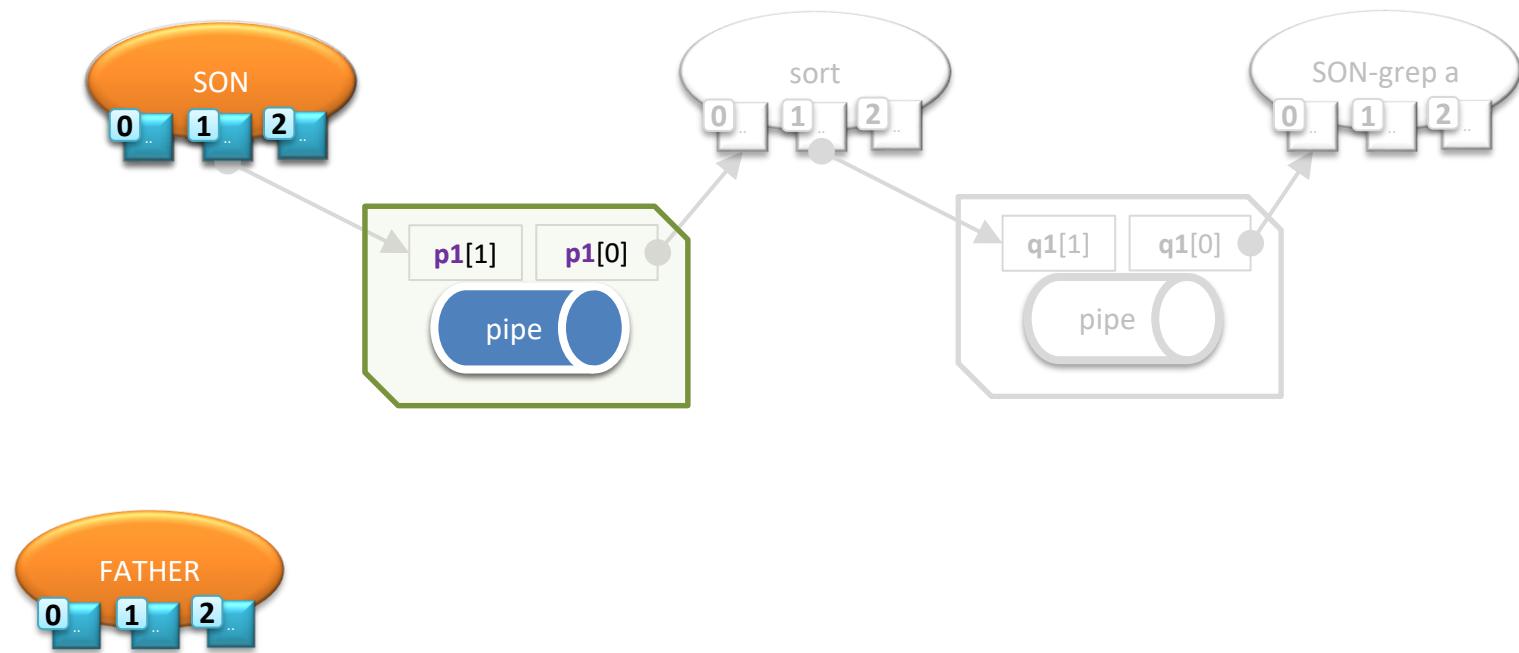
## (1/4) creation



The 4 steps should be done for each pipe (**pipe**, fork, redirection and cleanup)

# Example 2: "ls | sort | grep a"

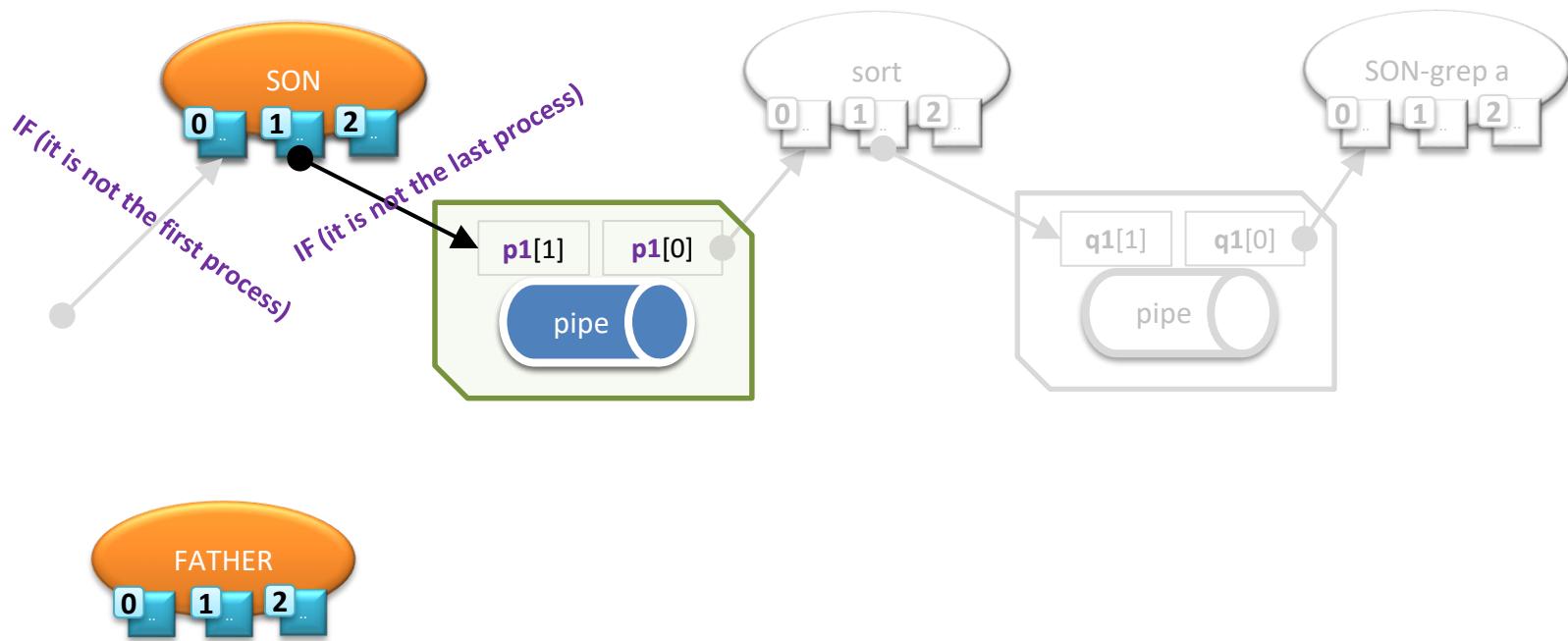
(2/4) fork()



The 4 steps should be done for each pipe (pipe, **fork**, redirection and cleanup)

# Example 2: "ls | sort | grep a"

## (3/4) redirection

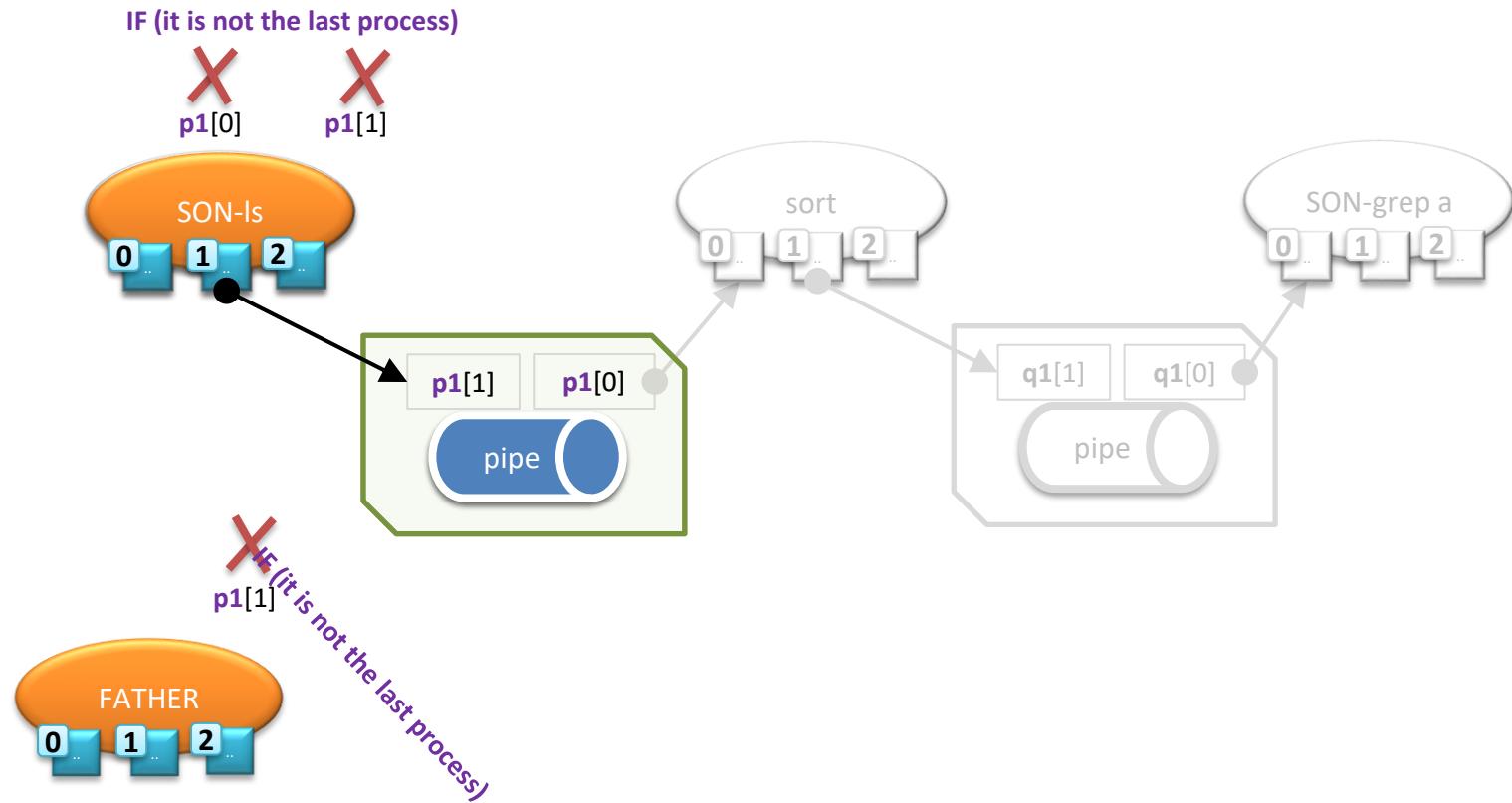


The 4 steps should be done for each pipe (pipe, fork, **redirection** and cleanup)

**The first process only uses the pipe in the output redirection.**

# Example 2: "ls | sort | grep a"

## (4/4) cleanup (+exec in son)

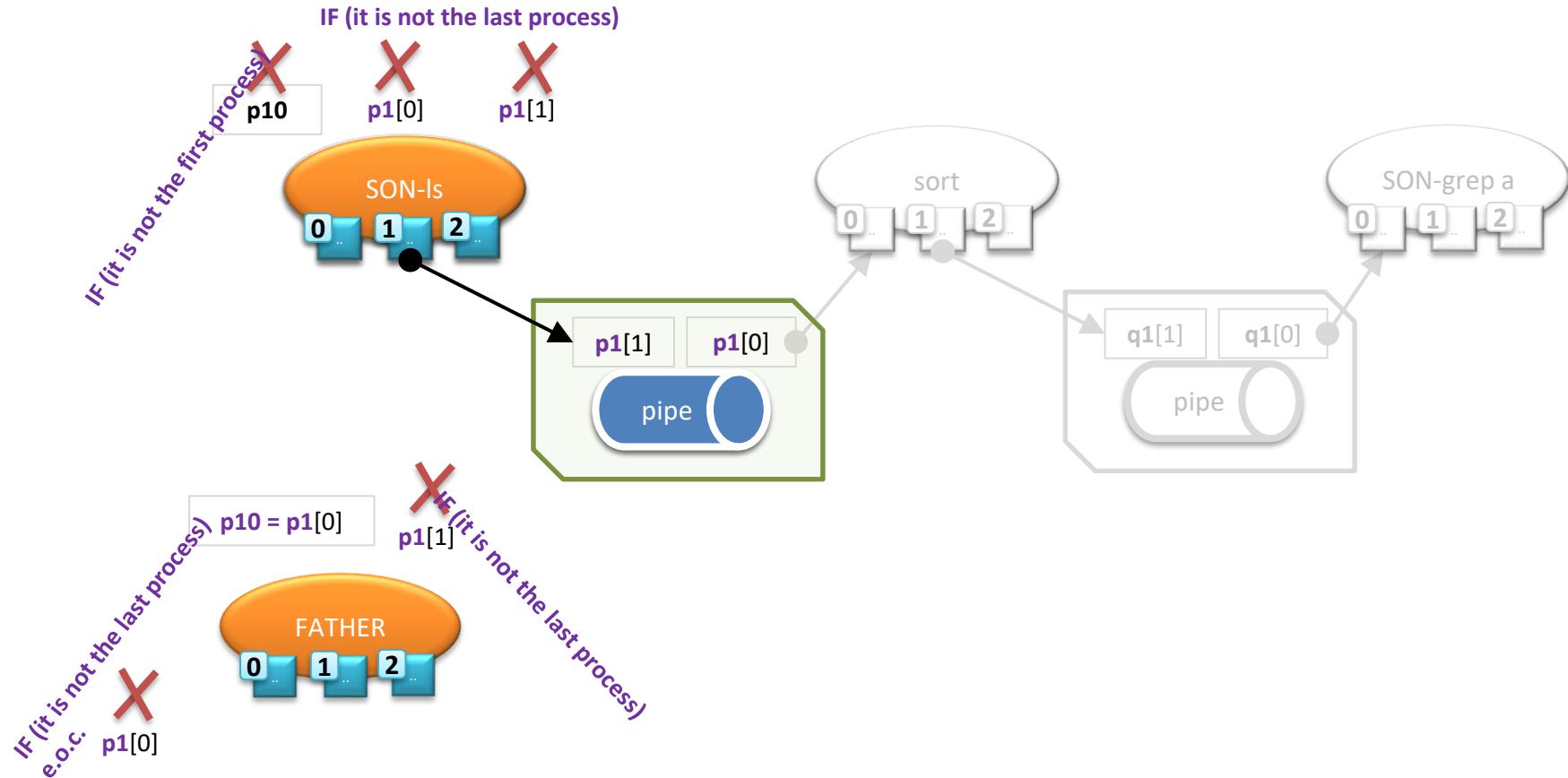


The 4 steps should be done for each pipe (pipe, fork, redirection and **cleanup**)

**The father would still not do a total cleanup for the next child to have access to "p1[0]"**

# Example 2: "ls | sort | grep a"

## (4/4) cleanup (+exec in son)

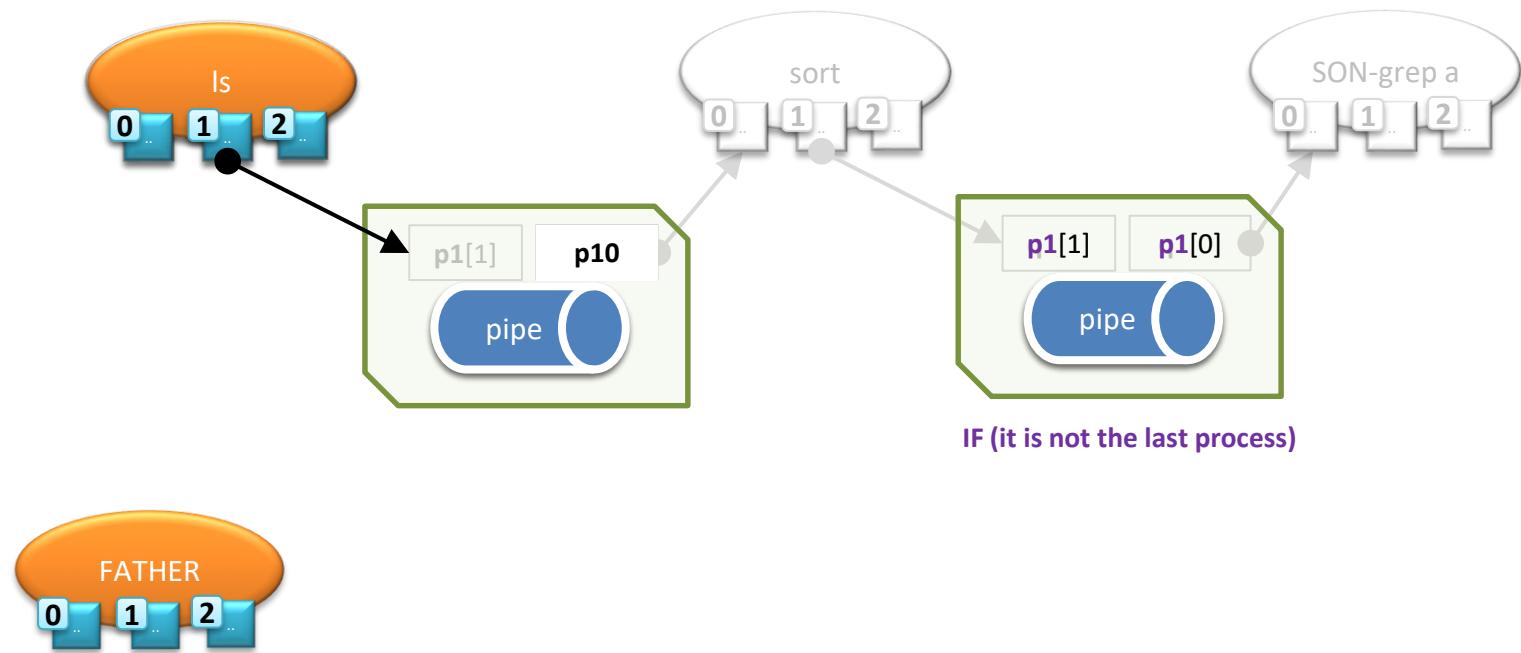


The 4 steps should be done for each pipe (pipe, fork, redirection and **cleanup**)

**The parent stores in  $p10$  the value of " $p1[0]$ " for reusing the same pipe (if not the last)**

# Example 2: "ls | sort | grep a"

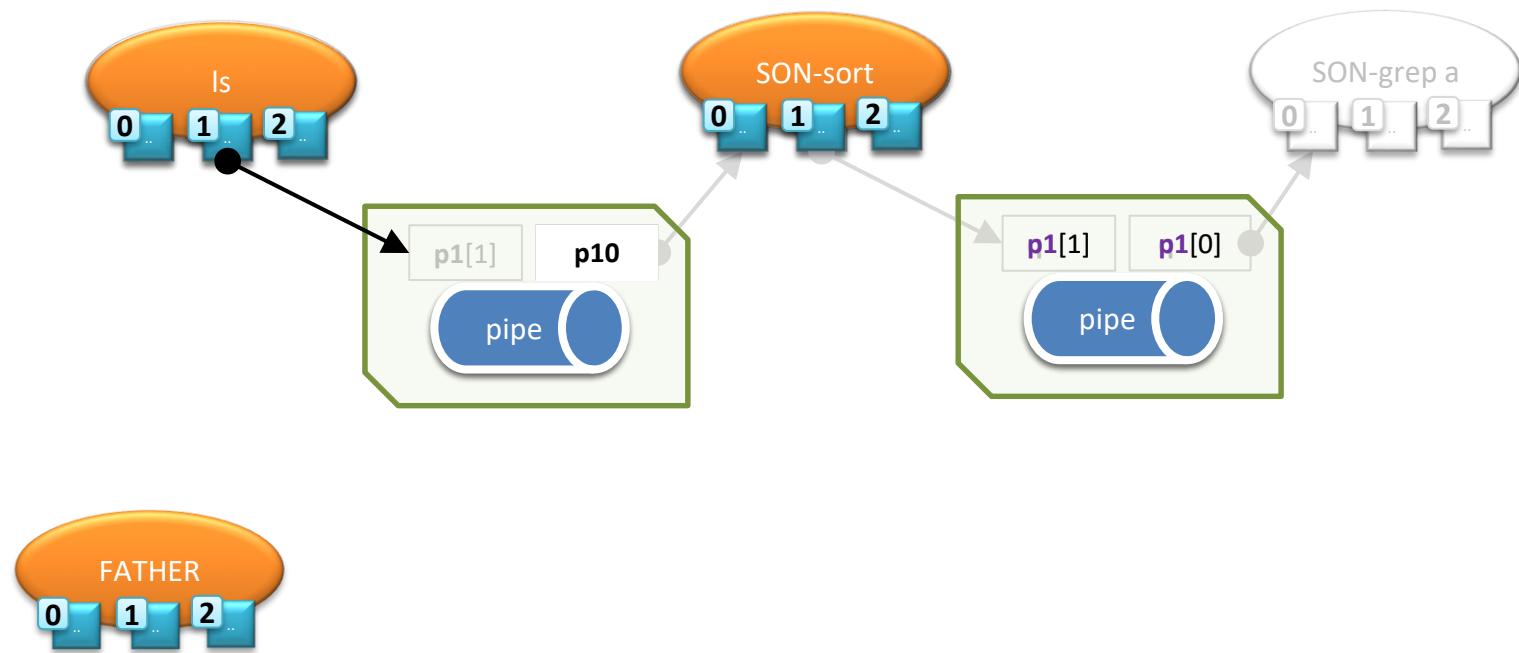
## (1/4) creation



The 4 steps should be done for each pipe (**creation**, fork, redirection and cleanup)

# Example 2: "ls | sort | grep a"

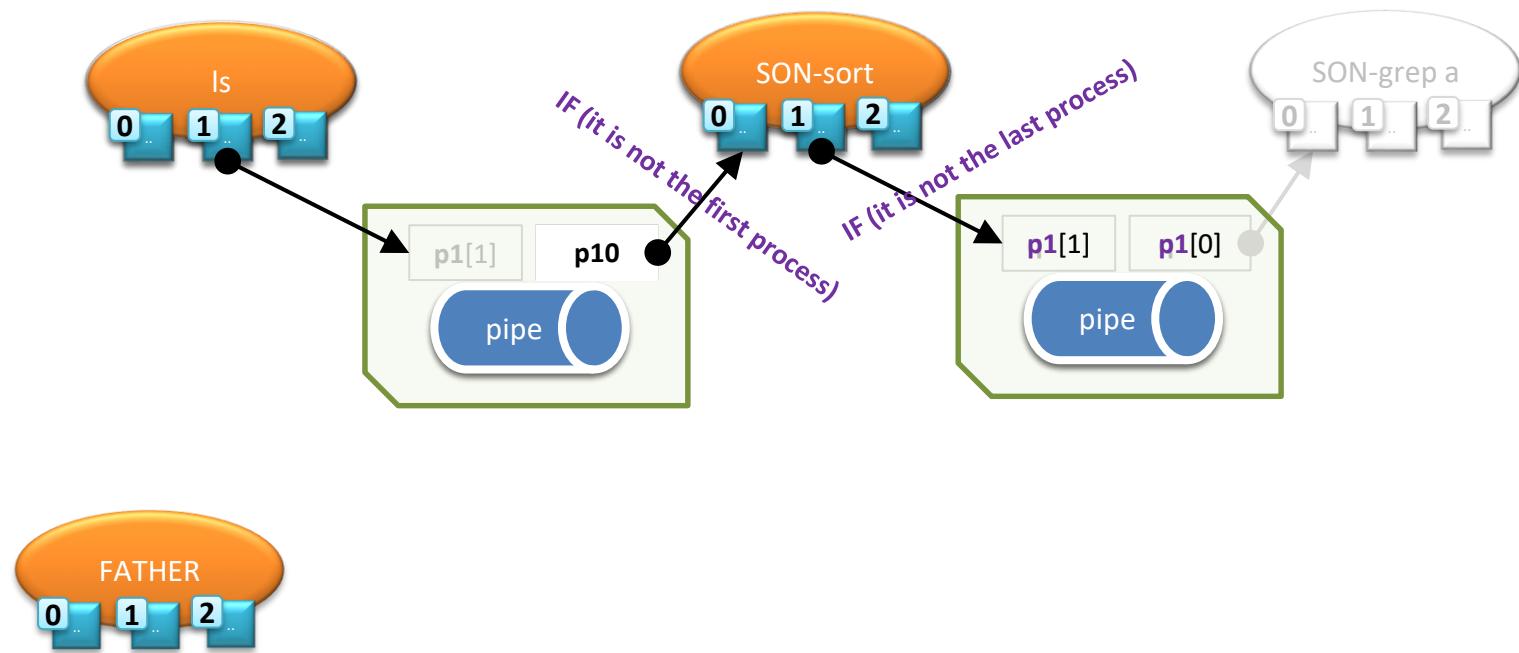
(2/4) fork()



The 4 steps should be done for each pipe (creation, **fork**, redirection and cleanup)

# Example 2: "ls | sort | grep a"

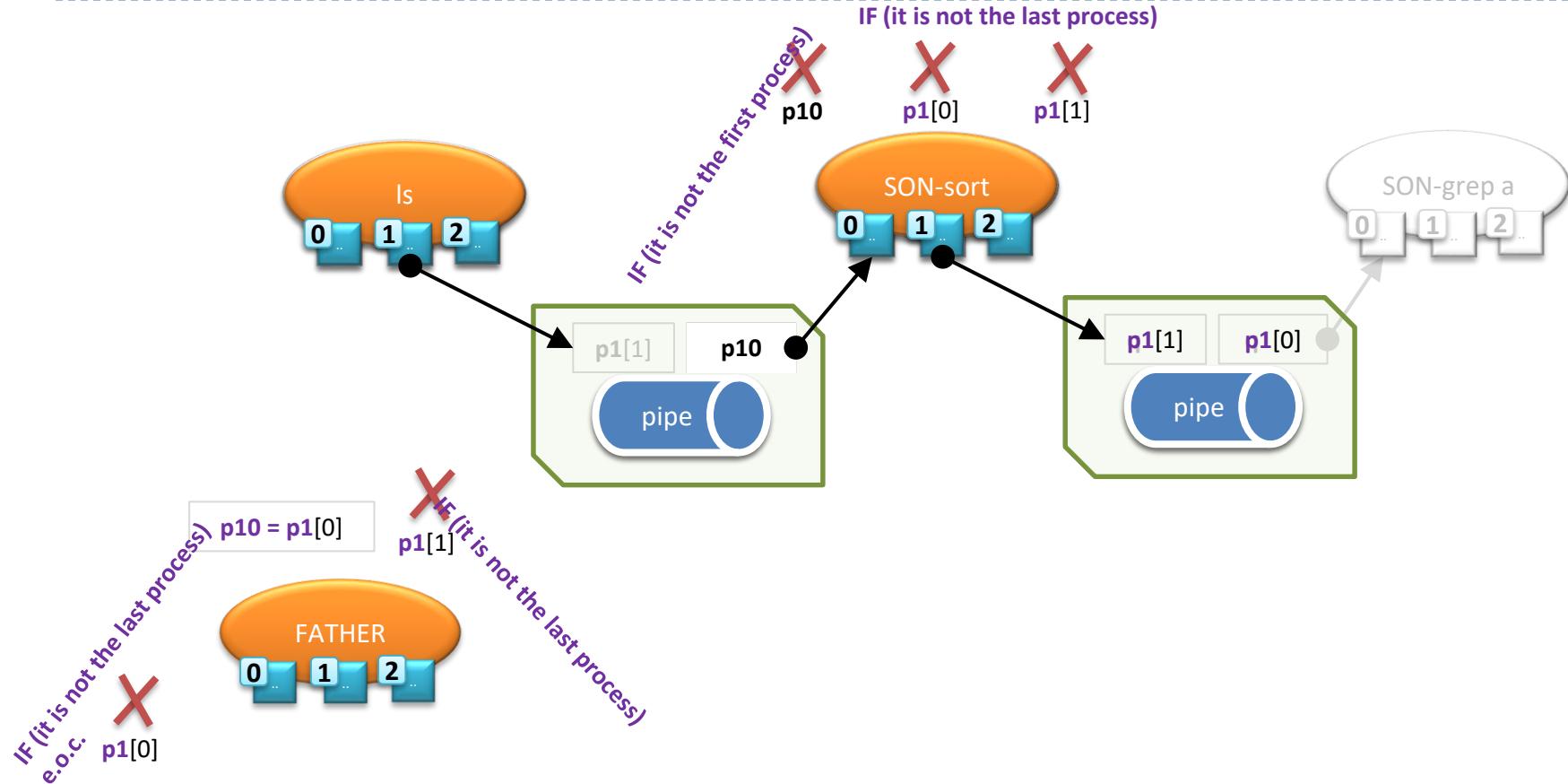
## (3/4) redirection



The 4 steps should be done for each pipe (creation, fork, **redirection** and cleanup)  
**Intermediate processes redirect their standard input and output to the pipe.**

# Example 2: "ls | sort | grep a"

## (4/4) cleanup (+exec in son)

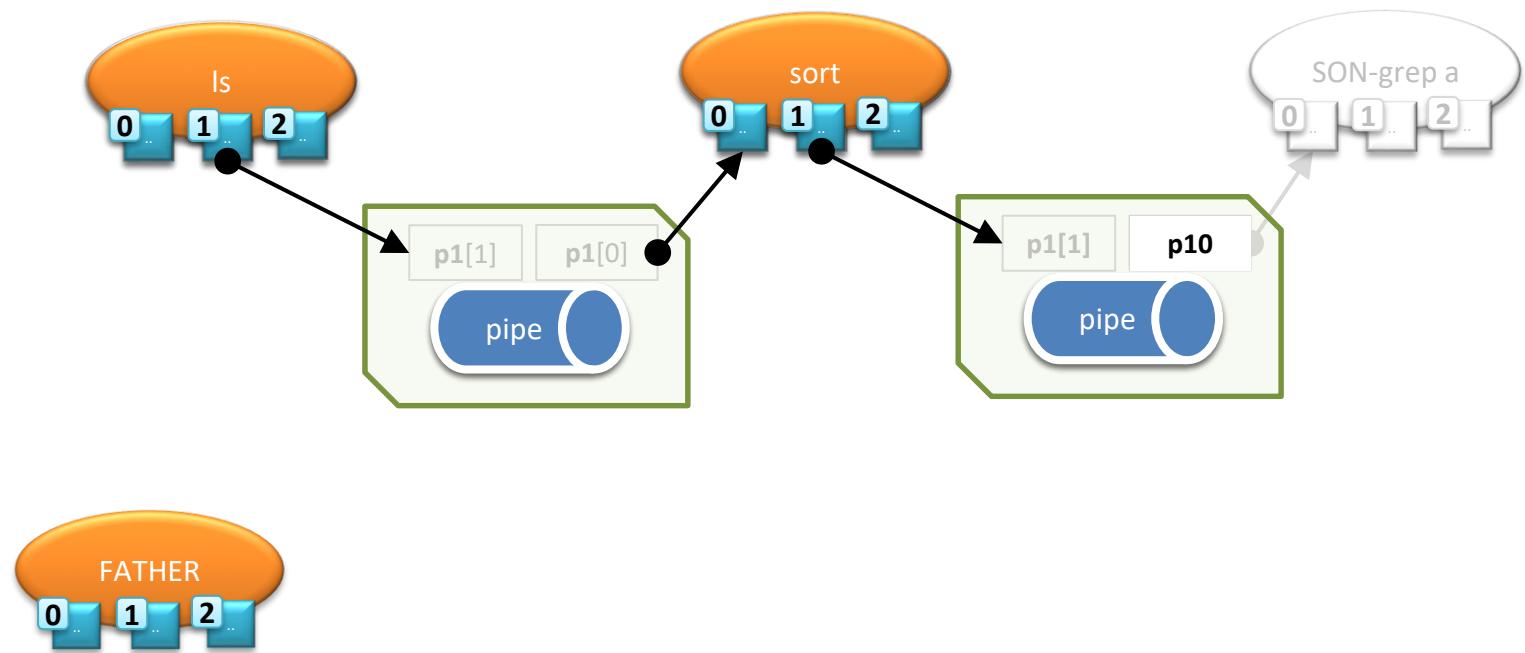


The 4 steps should be done for each pipe (pipe, fork, redirection and **cleanup**)

**The father stores in p10 the value of "p1[0]" for reusing the same pipe (if not last one)**

# Example 2: "ls | sort | grep a"

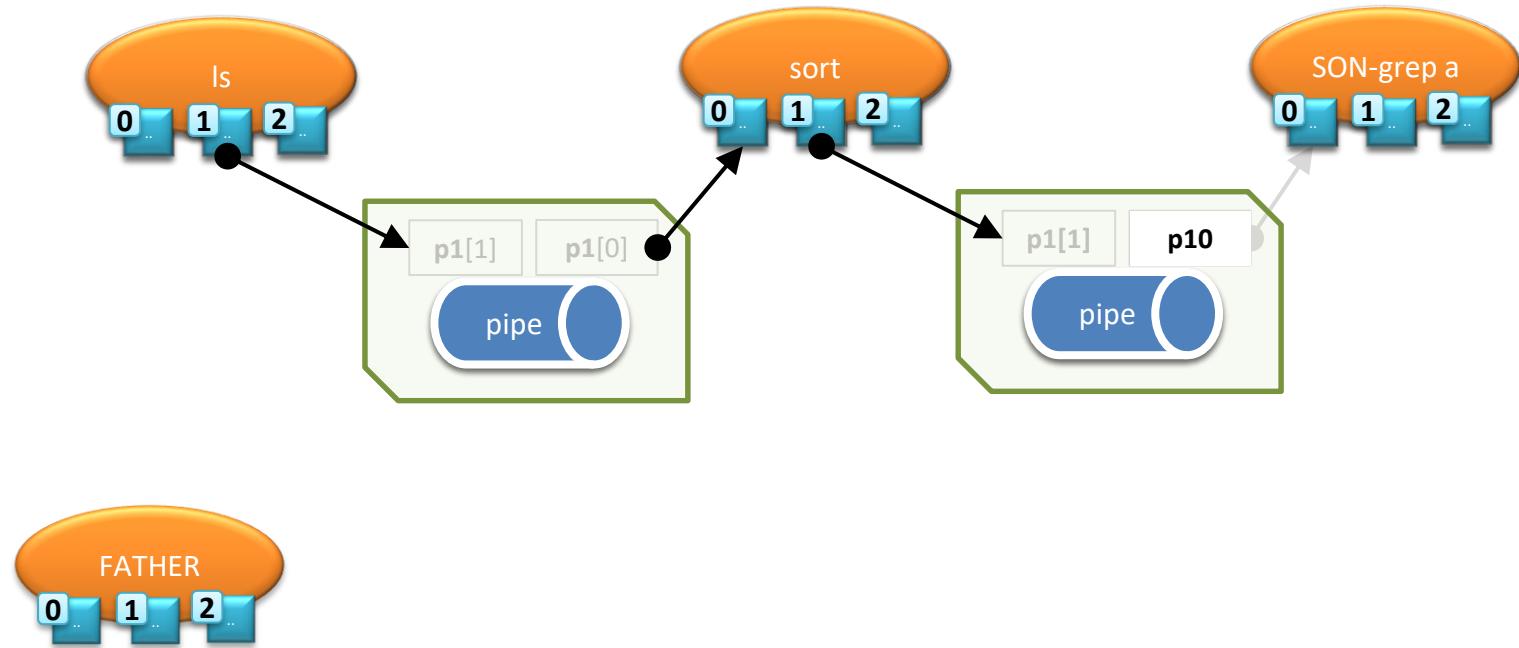
## (1/4) creation



The 4 steps should be done for each pipe (**creation**, fork, redirection and cleanup)  
The last process uses the pipe already created (there are n processes and n-1 pipes)

# Example 2: "ls | sort | grep a"

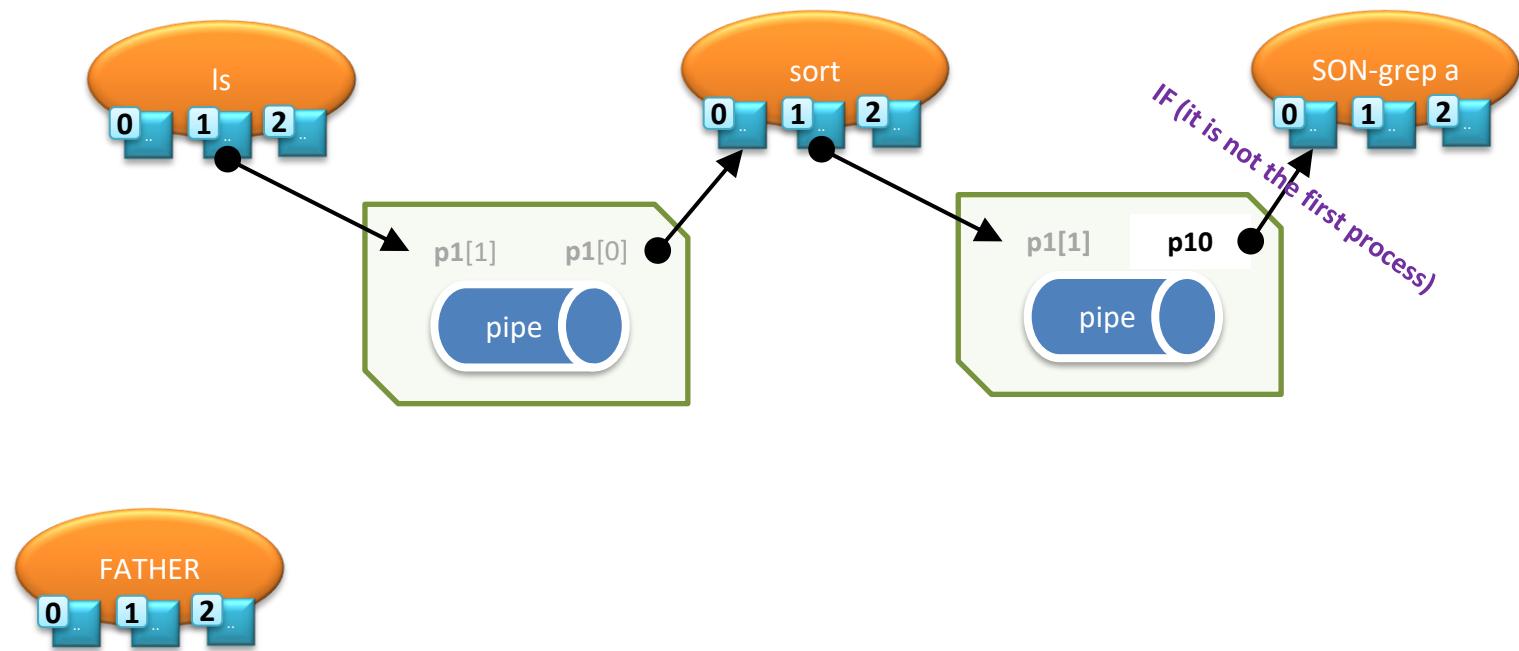
(2/4) fork()



The 4 steps should be done for each pipe (creation, **fork**, redirection and cleanup)

# Example 2: "ls | sort | grep a"

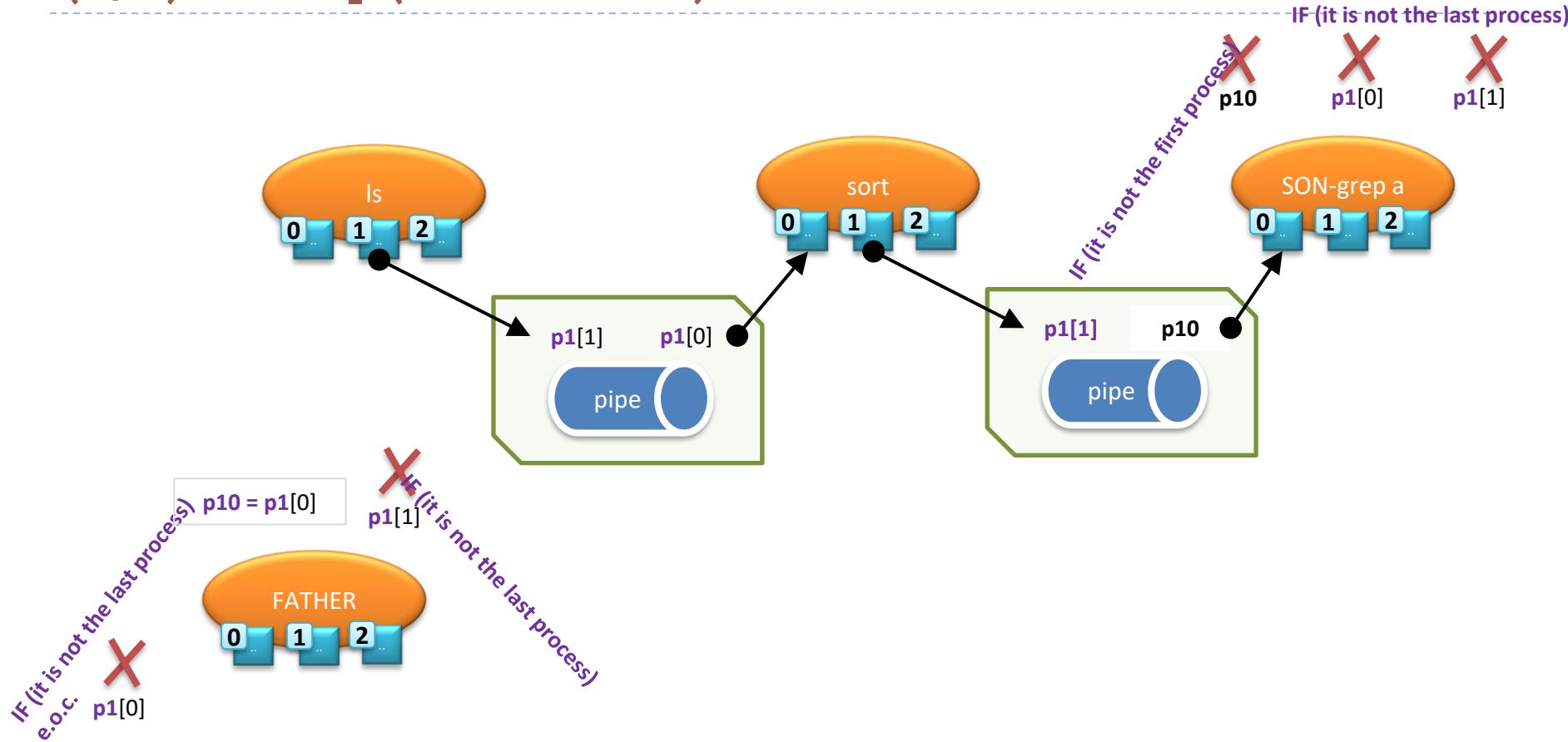
## (3/4) redirection



The 4 steps should be done for each pipe (creation, fork, **redirection** and cleanup)  
**The last process uses the pipe for redirection on its standard input.**

# Example 2: "ls | sort | grep a"

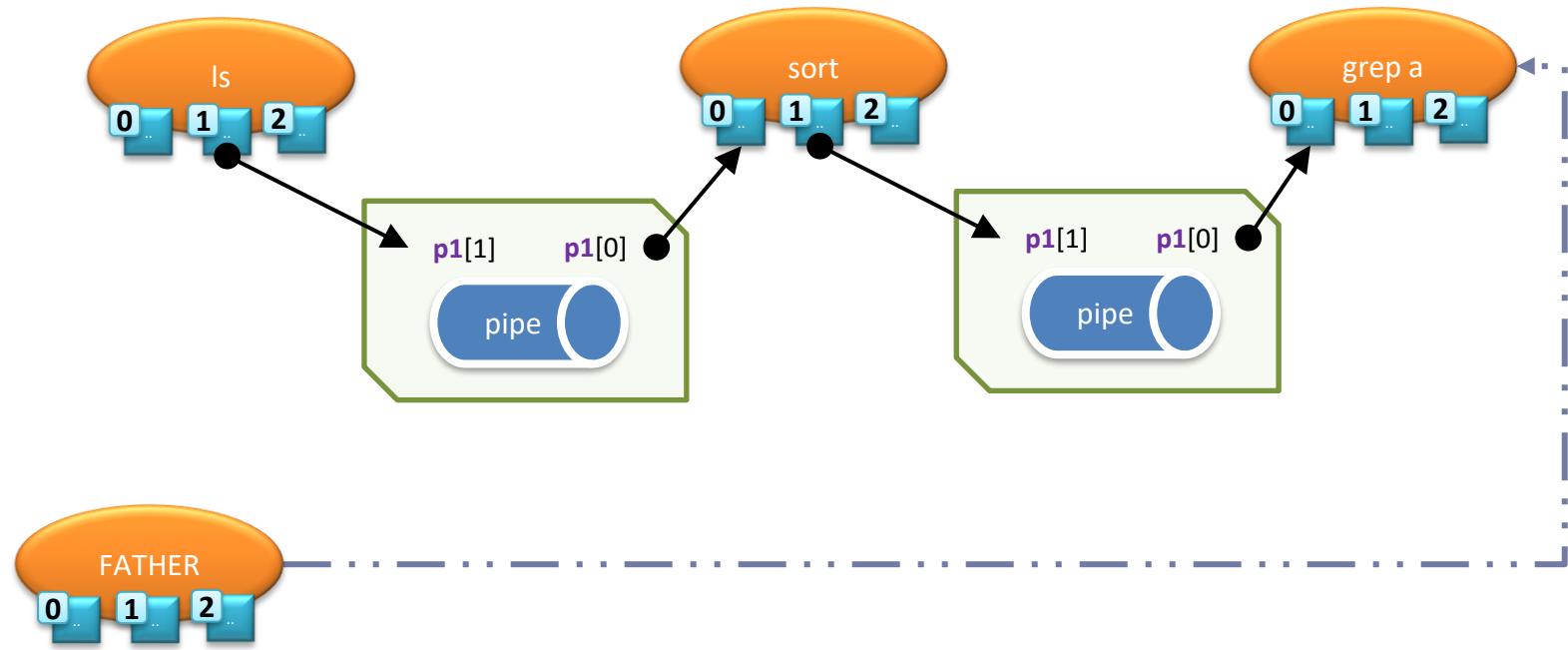
## (4/4) cleanup (+exec in son)



The 4 steps should be done for each pipe (creation, fork, redirection and **cleanup**)  
**The parent process closes all remaining descriptors.**

# Example 2: "ls | sort | grep a"

**if (! bg) then wait(...) for the last son created**



If it is not a *background task*, then the parent process does a *wait(...)* waiting for the last process (which becomes "grep a")

## Lesson 3B

### Signals, exceptions and pipes

Operating Systems  
Computer Science and Engineering

