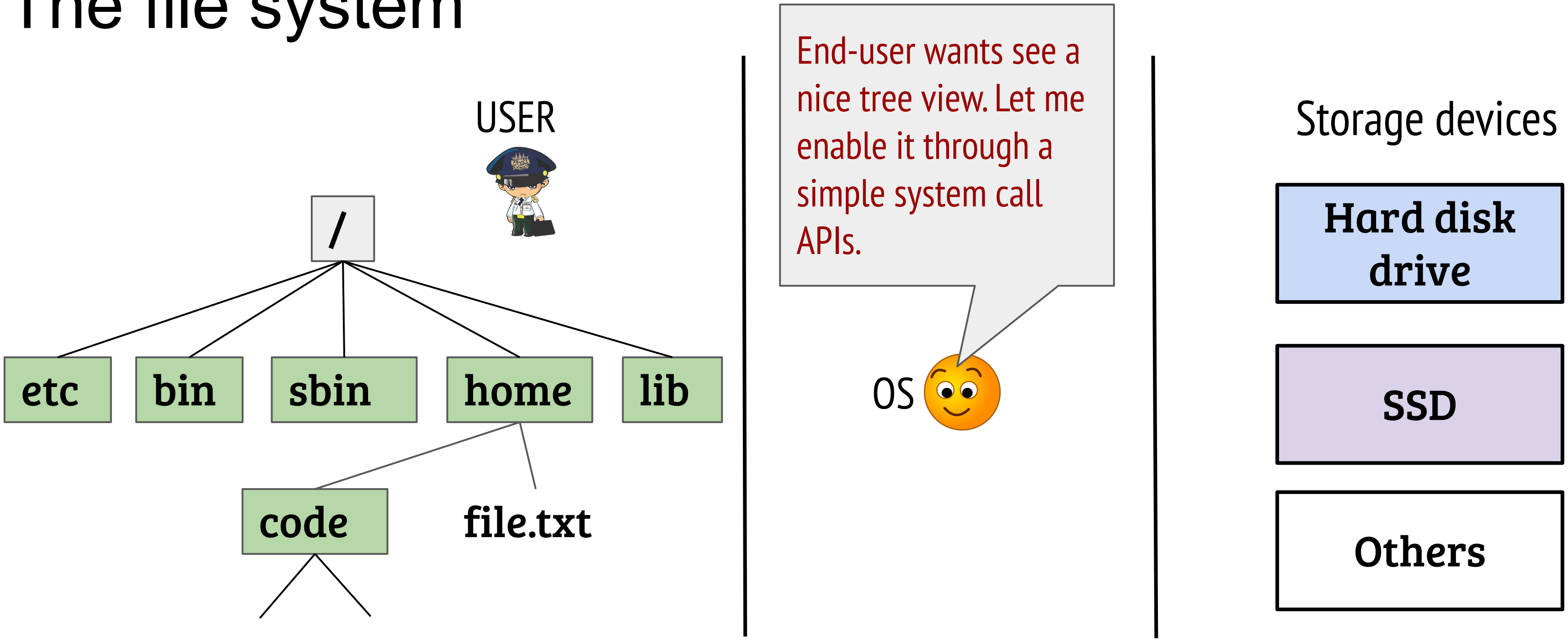
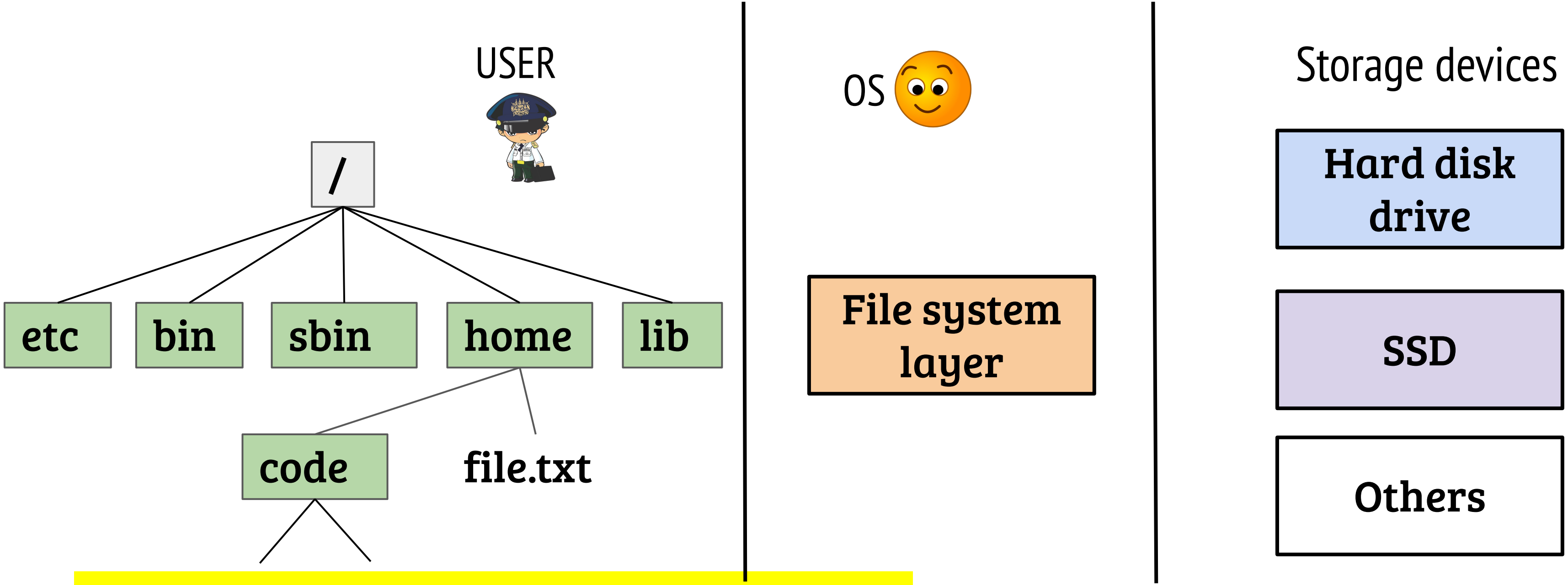
CS330: Operating Systems

Files

The file system

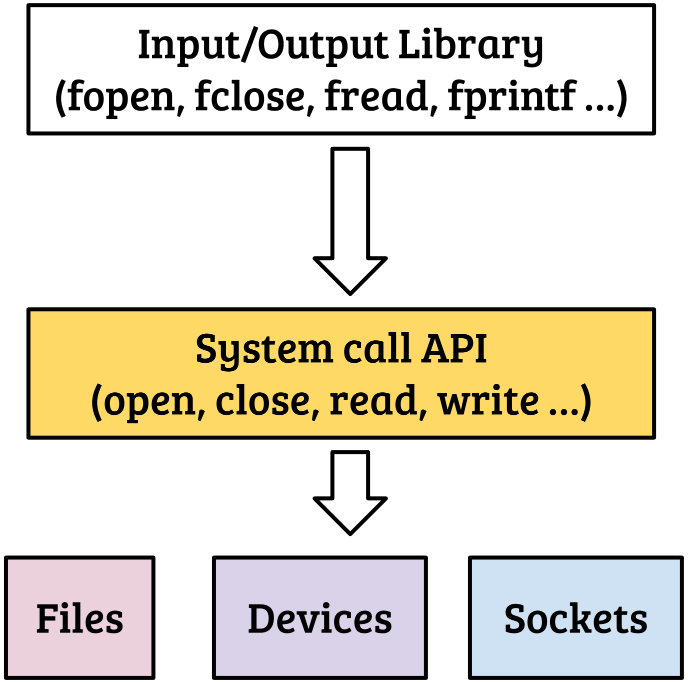


The file system



- File system is an important OS subsystem
    - Provides abstractions like files and directories
    - Hides the complexity of underlying storage devices

File system interfacing



- Processes identify files through a file handle a.k.a. file descriptors
- In UNIX, the POSIX file API is used to access files, devices, sockets etc.
- What is the mapping between library functions and system calls?

open: getting a handle

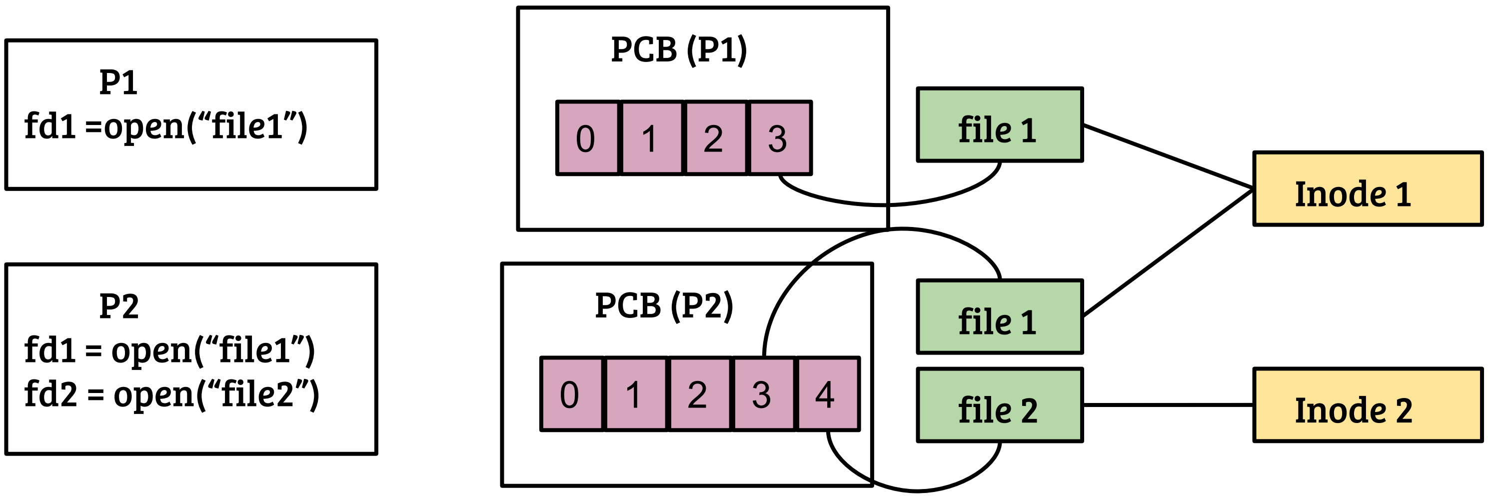
```
int open (char *path, int flags, mode_t mode)
```

open: getting a handle

int open (char *path, int flags, mode_t mode)

- Access mode specified in flags : O_RDONLY, O_RDWR, O_WRONLY
- Access permissions check performed by the OS
- On success, a file descriptor (integer) is returned
- If flags contain O_CREAT, mode specifies the file creation mode
- Refer man page (“man 2 open")

Process view of file



- Per-process file descriptor table with pointer to a "file" object
- file object ⟶ inode is many-to-one

Process view of file

- What do file descriptors 0, 1 and 2 represent?
- What happens to the FD table and the file objects across fork( )?
    - What happens in exec( )?
- Can multiple FDs point to the same file object?

Read and Write

ssize_t read (int fd, void *buf, size_t count);

- fd ⟶ file handle
- buf ⟶ user buffer as read destination
- count ⟶ #of bytes to read
- read ( ) returns #of bytes actually read, can be smaller than count

Read and Write

ssize_t read (int fd, void *buf, size_t count);

- fd ⟶ file handle
- buf ⟶ user buffer as read destination
- count ⟶ #of bytes to read
- read ( ) returns #of bytes actually read, can be smaller than count

ssize_t write (int fd, void *buf, size_t count);

- Similar to read

Process view of file

- What do file descriptors 0, 1 and 2 represent?
- 0 → STDIN, 1 → STDOUT and 2 → STDERR
- What happens to the FD table and the file objects across fork( )?
  - What happens in exec( )?
- Can multiple FDs point to the same file object?

lseek

```
off_t lseek(int fd, off_t offset, int whence);
```

- fd ⟶ file handle
- offset ⟶ target offset
- whence ⟶ SEEK_SET, SEEK_CUR, SEEK_END
- On success, returns offset from *the starting of the file*

lseek

off_t lseek(int fd, off_t offset, int whence);

- fd ⟶ file handle
- offset ⟶ target offset
- whence ⟶ SEEK_SET, SEEK_CUR, SEEK_END
- On success, returns offset from *the starting of the file*
- Examples
  - lseek(fd, 100, SEEK_CUR) ⟶ forwards the file position by 100 bytes
  - lseek(fd, 0, SEEK_END) ⟶ file pos at EOF, returns the file size
  - lseek(fd, 0, SEEK_SET) ⟶ file pos at beginning of file

File information (stat, fstat)

```
int stat(const char *path, struct stat *sbuf);
```

- Returns the information about file/dir in the argument path
- The information is filled up in structure called stat

File information (stat, fstat)

```
int stat(const char *path, struct stat *sbuf);
```

- Returns the information about file/dir in the argument path
- The information is filled up in structure called stat

```
struct stat sbuf;
stat("/home/user/tmp.txt", &sbuf);
printf("inode = %d size = %ld\n", sbuf.st_ino, sbuf.st_size);
```

- Other useful fields in *struct stat* : st_uid, st_mode (Refer stat man page)

Process view of file

PCB (P1)

- What do file descriptors 0, 1 and 2 represent?
- 0 → STDIN, 1 → STDOUT and 2 → STDERR
- What happens to the FD table and the file objects across fork( )?
    - What happens in exec( )?
- The FD table is copied across fork( ) ⇒ File objects are shared
- On exec, open files remain shared by default
- Can multiple FDs point to the same file object?

Duplicate file handles (dup and dup2)

```
int dup(int oldfd);
```

- The dup() system call creates a "copy" of the file descriptor oldfd
- Returns the lowest-numbered unused descriptor as the new descriptor
- The old and new file descriptors represent the same file

Duplicate file handles (dup and dup2)

```
int fd, dupfd;
fd = open("tmp.txt");
close(1);
dupfd = dup(fd);      //What will be the value of dupfd?
printf("Hello world\n");  // Where will be the output?
```

Duplicate file handles (dup and dup2)

```
int fd, dupfd;
fd = open("tmp.txt");
close(1);
dupfd = dup(fd);        //What will be the value of dupfd?
printf("Hello world\n"); // Where will be the output?
```

- Value of dupfd = 1 (assuming STDIN is open)
- "Hello world" will be written to tmp.txt file

Duplicate file handles (dup and dup2)

```
int dup2(int oldfd, int newfd);
```

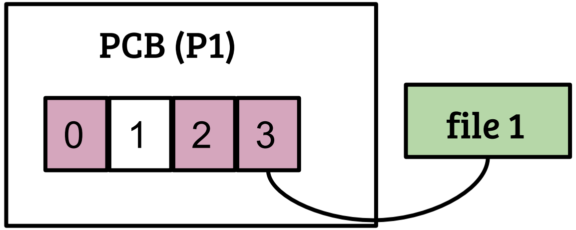
- Close newfd before duping the file descriptor oldfd
- dup2 (fd, 1) equivalent to
  - close(1);
  - dup(fd);

Duplicate file handles (dup and dup2)

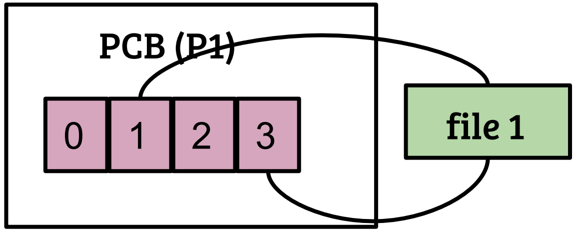
P1
fd1 =open(“file1”)

+ dup(fd1)

Before dup( )



After dup( )



- Lowest numbered unused fd (i.e., 1) is used (Assume STDOUT is closed before)
- Duplicate descriptors share the same file state
- Closing one file descriptor *does not* close the file

Use of dup: shell redirection

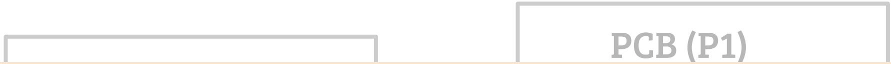
- Example: ls > tmp.txt
- How implemented?

Use of dup: shell redirection

- Example: ls > tmp.txt
- How implemented?

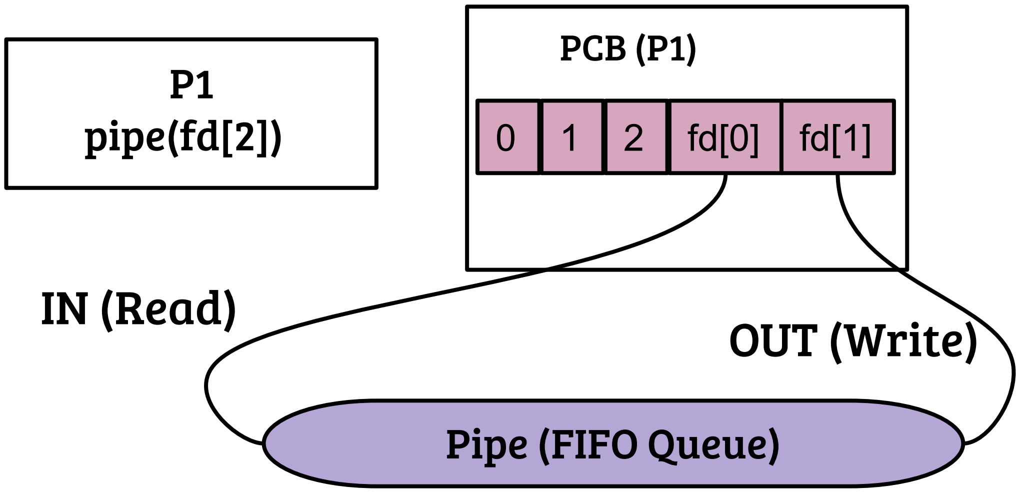
```
fd = open ("tmp.txt")

close( 1); close(2);      // close STDOUT and STDERR

dup(fd); dup(fd)          // 1→ fd, 2 → fd

exec(ls )
```

Process view of file



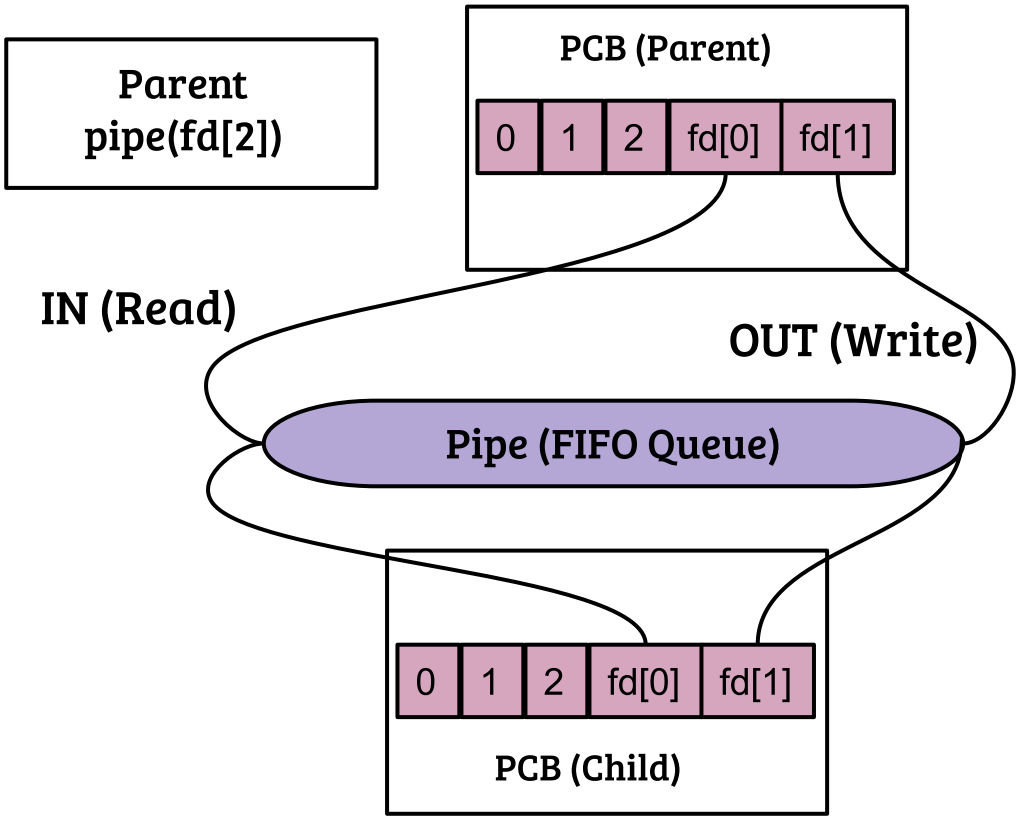
- What do file descriptors 0, 1 and 2 represent?
- 0 → STDIN, 1 → STDOUT and 2 → STDERR
- What happens to the FD table and the file objects across fork( )?
    - What happens in exec( )?
- The FD table is copied across fork( ) ⇒ File objects are shared
- On exec, open files remain shared by default
- Can multiple FDs point to the same file object?
- Yes, duped FDs share the same file object (within a process)

UNIX pipe( ) system call



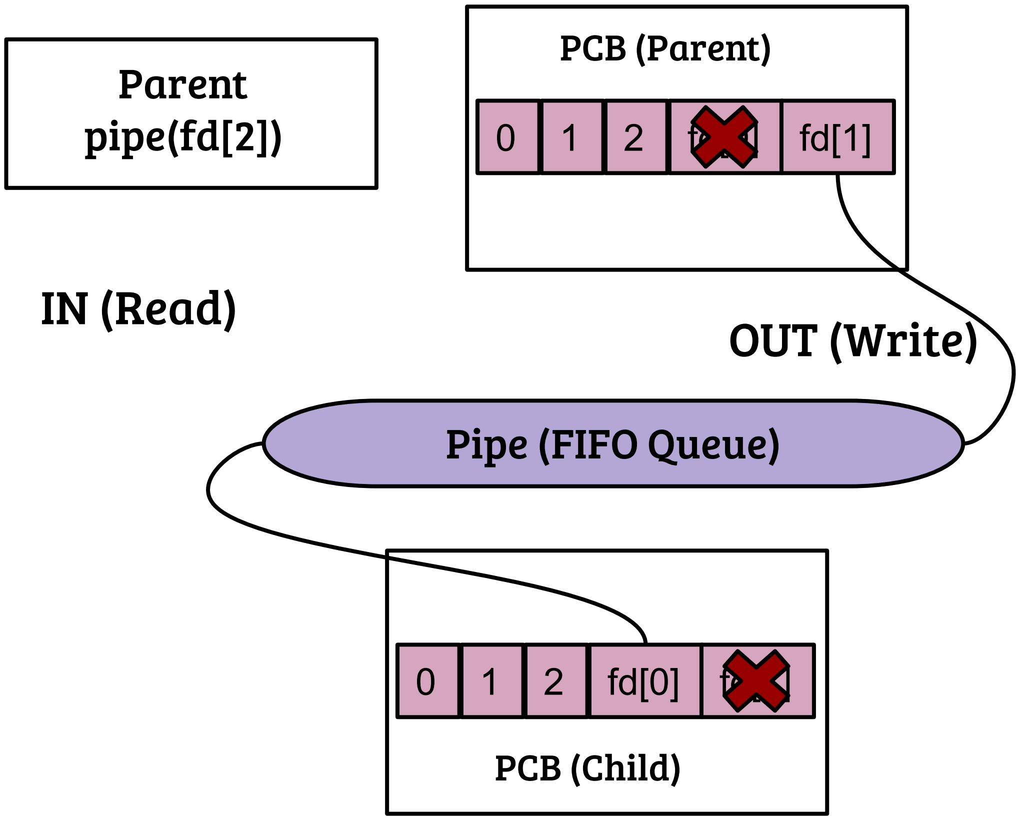
- pipe( ) takes array of two FDs as input
- *fd[0]* is the read end of the pipe
- *fd[1]* is the write end of the pipe
- Implemented as a FIFO queue in OS

UNIX pipe( ) with fork( )



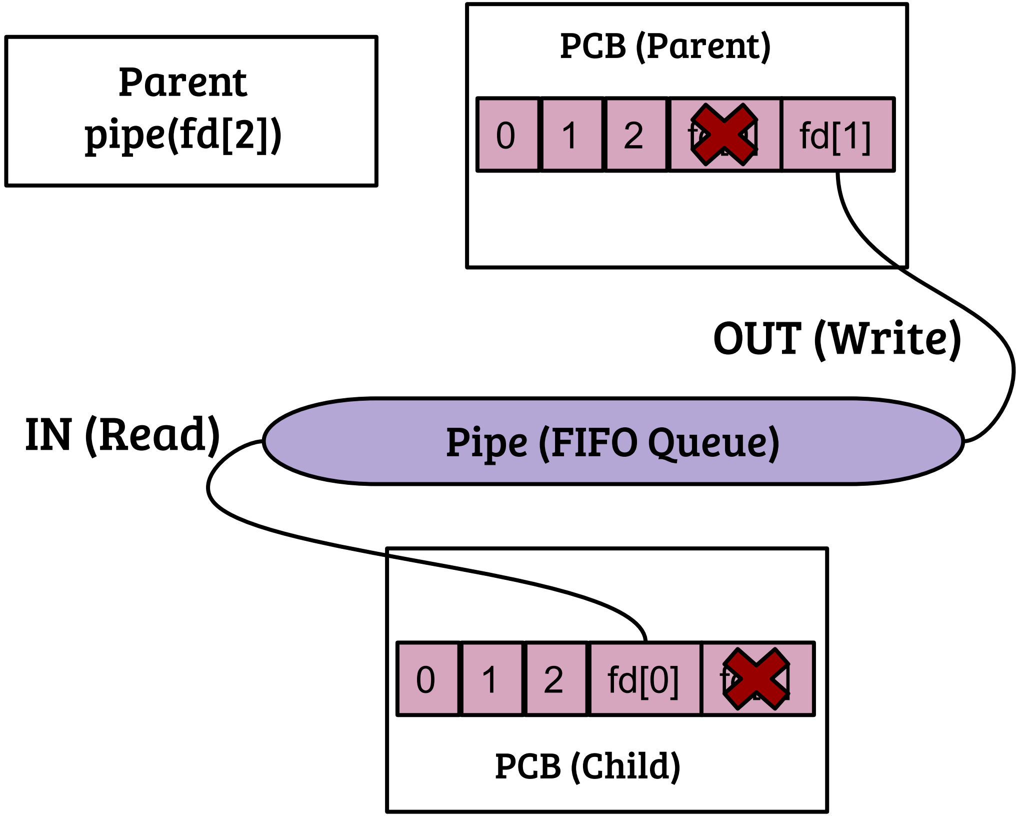
- fork( ) duplicates the file descriptors
- At this point, both the parent and the child processes can read/write to the pipe

UNIX pipe( ) with fork( )



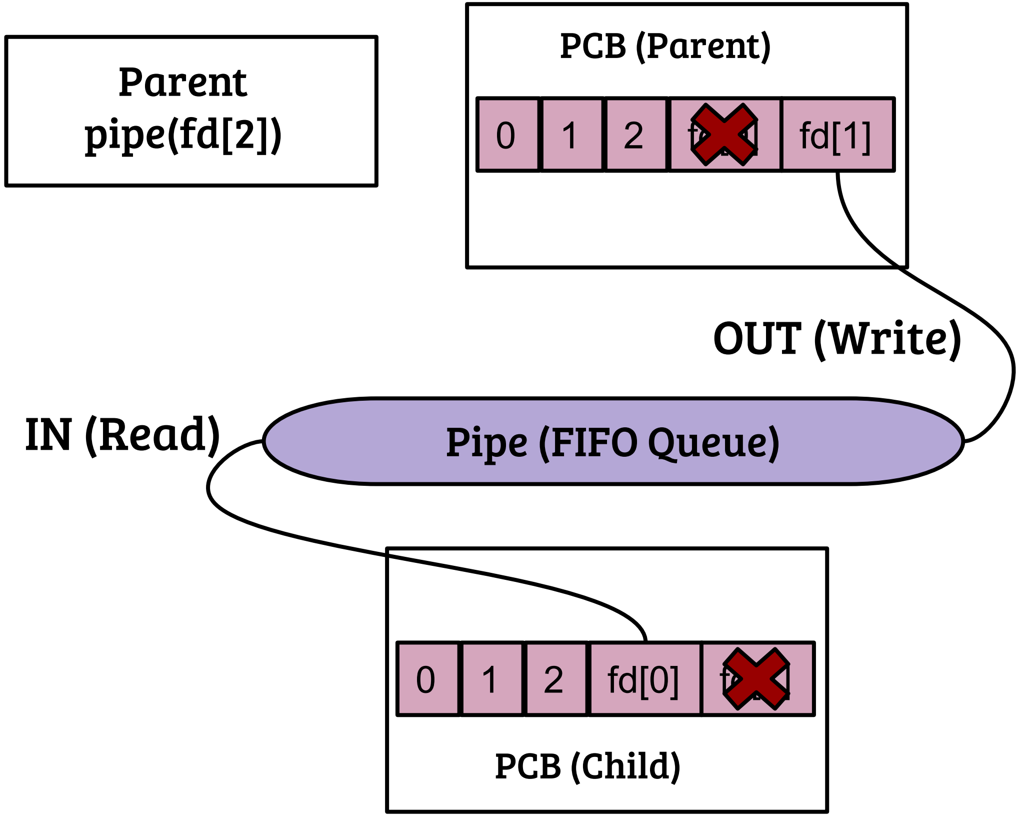
- fork( ) duplicates the file descriptors
- close( ) one end of the pipe, both in child and parent
- Result
  - A queue between parent and child

Shell piping : ls | wc -l



- pipe( ) followed by fork( )
- Parent: exec("ls") after making STDOUT ⟶ out fd of the pipe (using dup)

Shell piping : ls | wc -l



- pipe( ) followed by fork( )
- Parent: exec("ls") after making STDOUT ⟶ out fd of the pipe (using dup)
- Child: exec("wc") after closing STDIN and duping in fd of pipe
- Result: input of "wc" is connected to output of "ls"