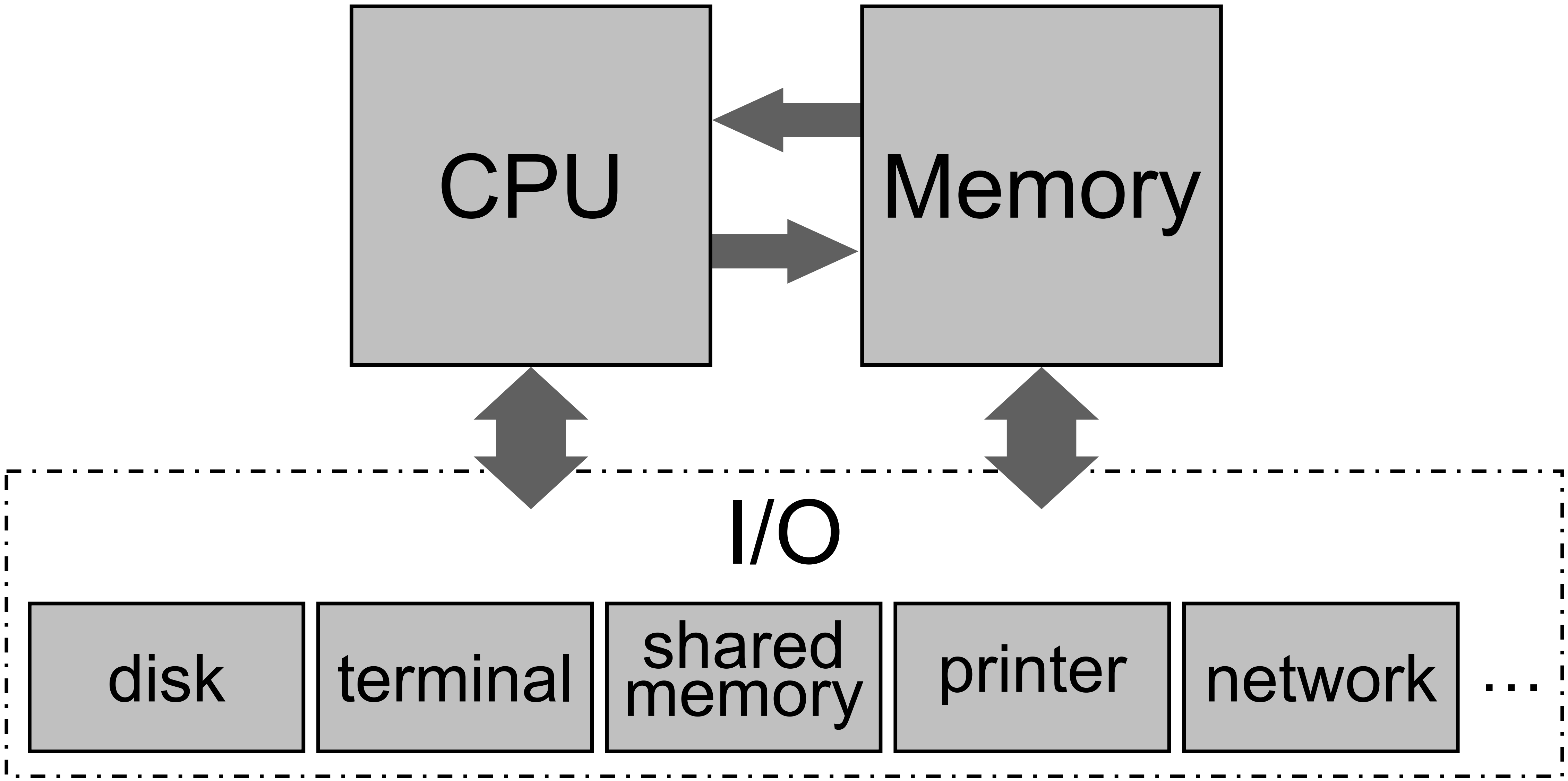
Input/Output

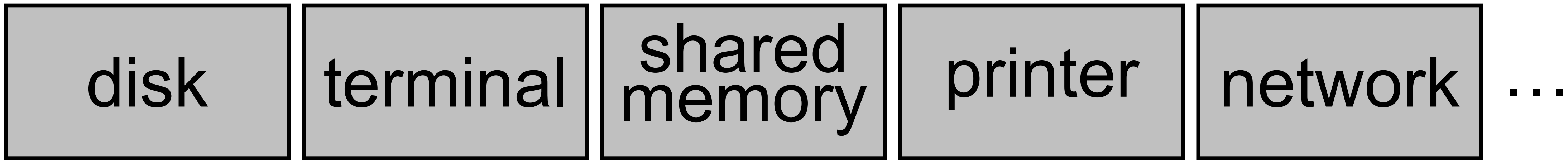
Science | Computer Science

CS 351: Systems Programming
Melanie Cornelius

Slides and course content obtained with permission
from Prof. Michael Lee, <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY


CPU
Memory
I/O
disk
terminal
shared memory
printer
network
...



- vast number of different mechanisms
- but overlapping requirements:
  - read/write operations
  - metadata (e.g., name, position)
  - robustness, thread-safety

programming concerns:

- how are I/O endpoints represented?
- how to perform I/O?

…*efficiently*?

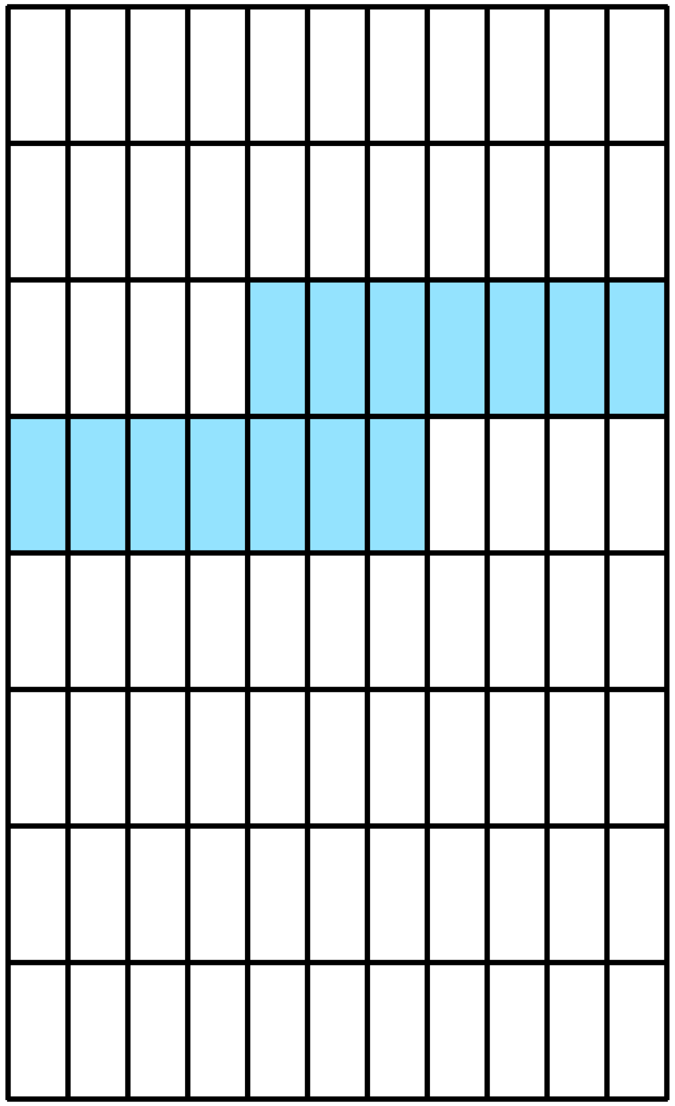
focus on **Unix system-level I/O**

- Unix I/O & Filesystem Architecture Brief

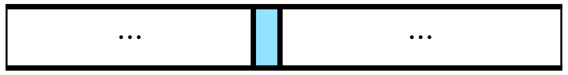
2 general classes of I/O devices:

- *block*: accessed in fixed-size chunks; support for seeking & random access
- *character*: char-by-char streaming access; no seeking / random access

block device



char device



2 general classes of I/O devices:

- *block*: e.g., disk, memory
- *character*: e.g., network, mouse

the **filesystem** acts as a *namespace* for data residing on different devices

- *regular files* consist of ASCII or binary data, stored on a block device
- *special files* may represent directories, in-memory structures, sockets, or raw devices

“Files” are a *general OS abstraction* for arbitrary data objects!

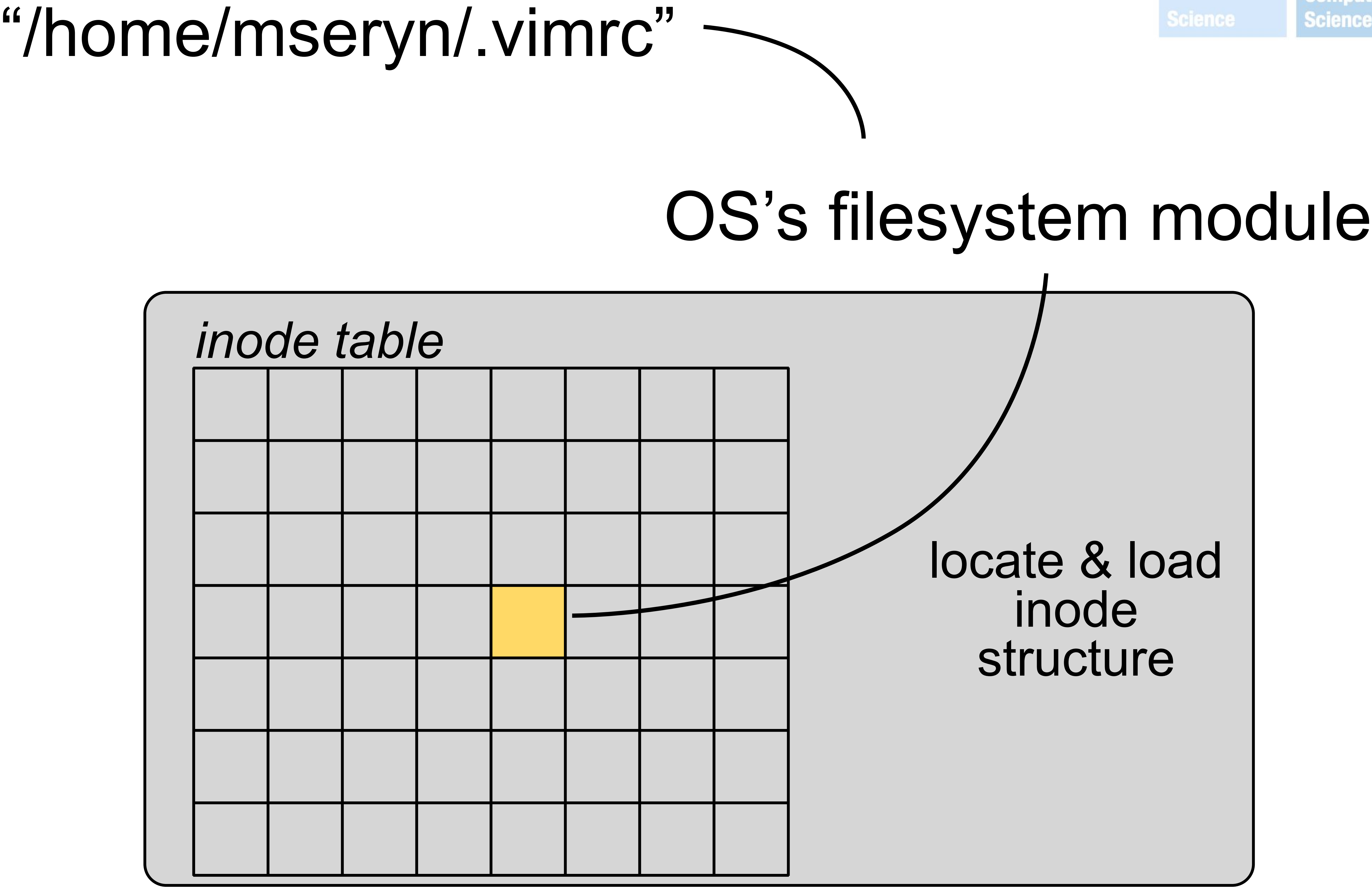
each file has a unique **inode** data structure in the filesystem, tracking:

- ownership & permissions
- size, type, and location
- number of *links*

a given inode can be referenced using one or more *fully qualified path(s)*, e.g.,

-/proc/sys/kernel/version

-/dev/tty

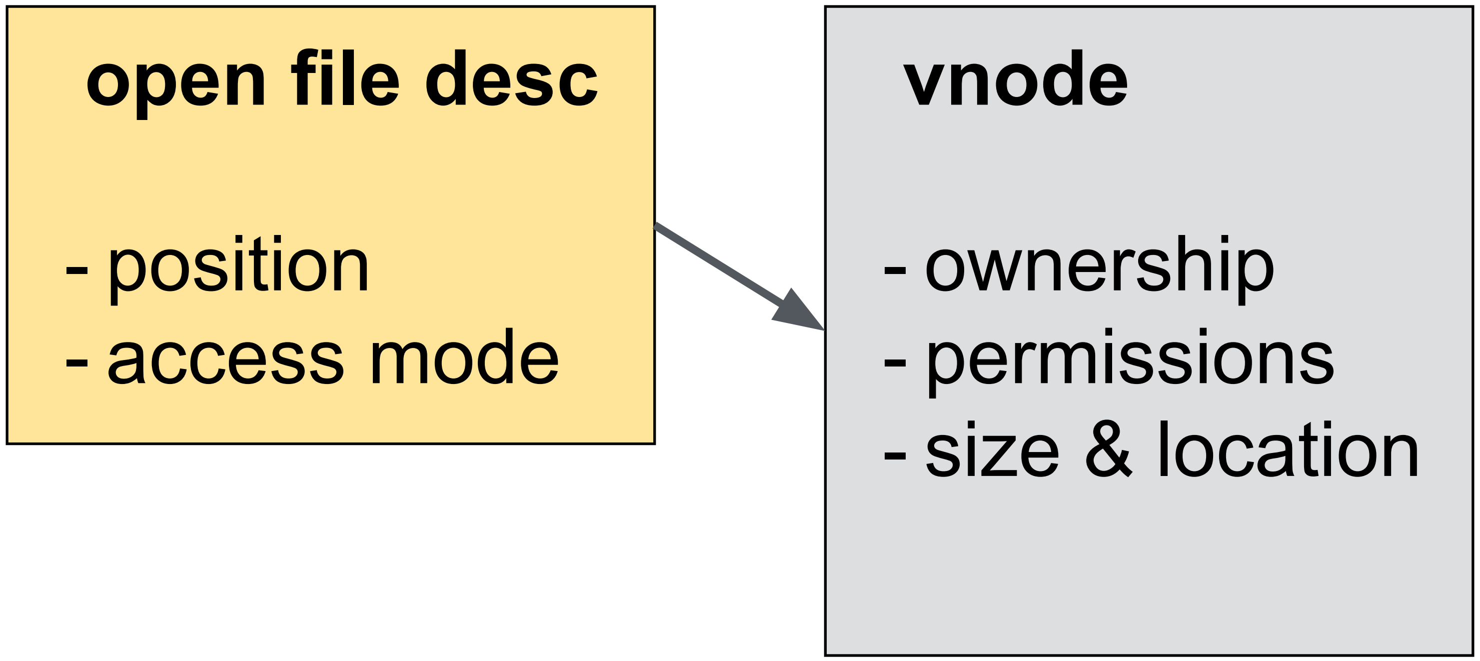

"/home/mseryn/.vimrc"
OS's filesystem module
inode table
locate & load inode structure

every currently open file has a *single* in-memory inode, aka. “vnode”

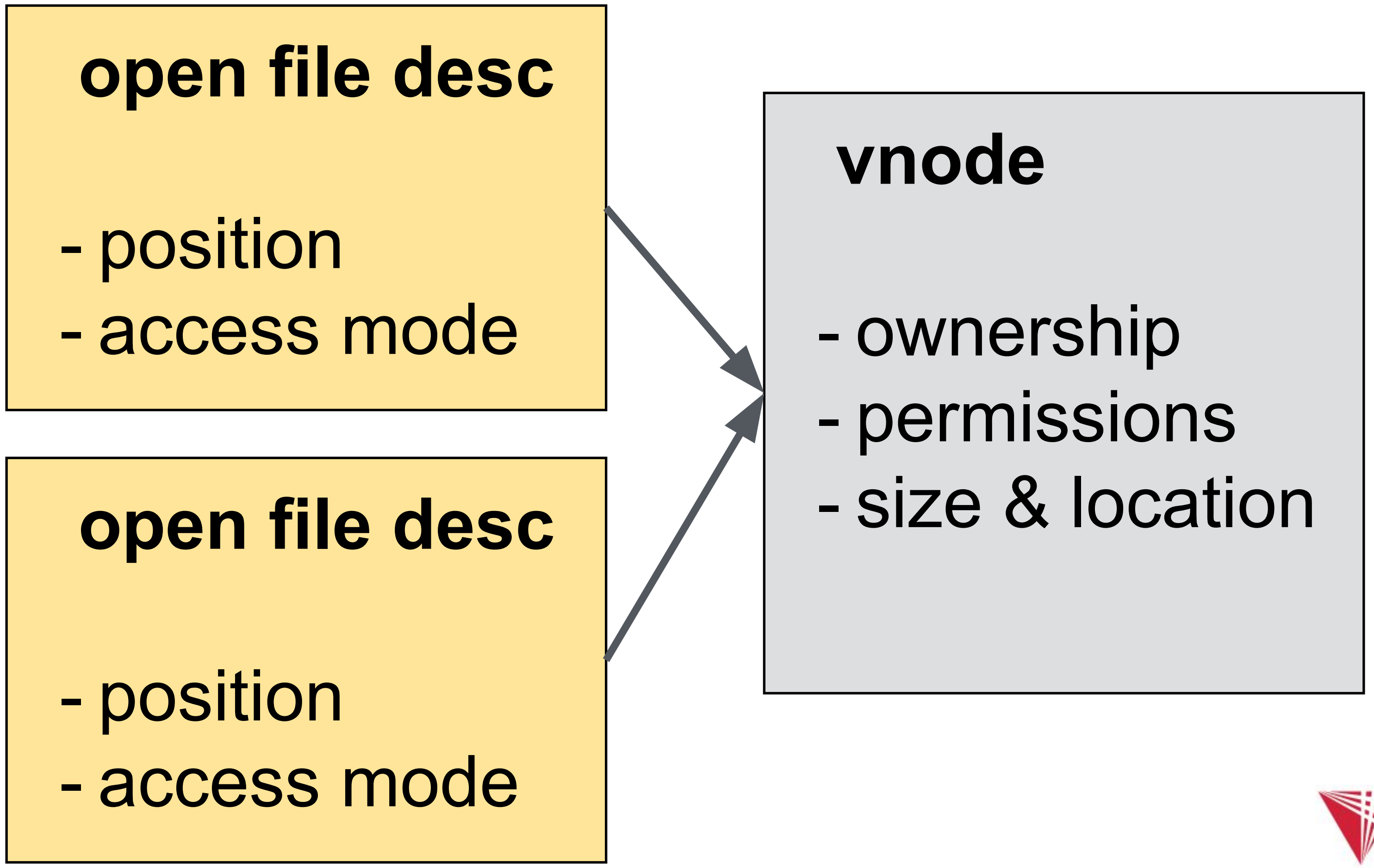
vnode

- ownership
- permissions
- size & location

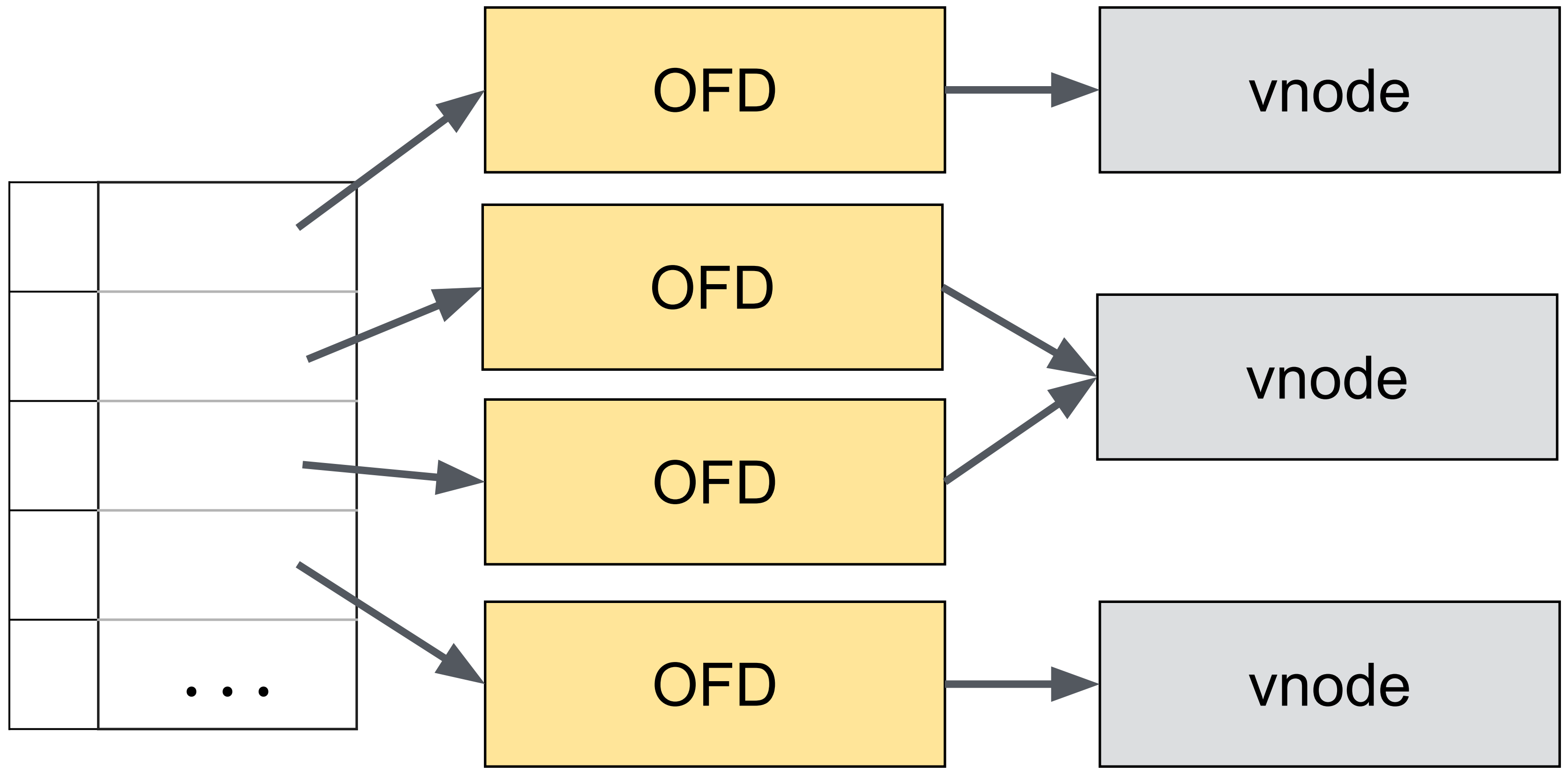
each open file is also tracked by the kernel using an *open file description* structure



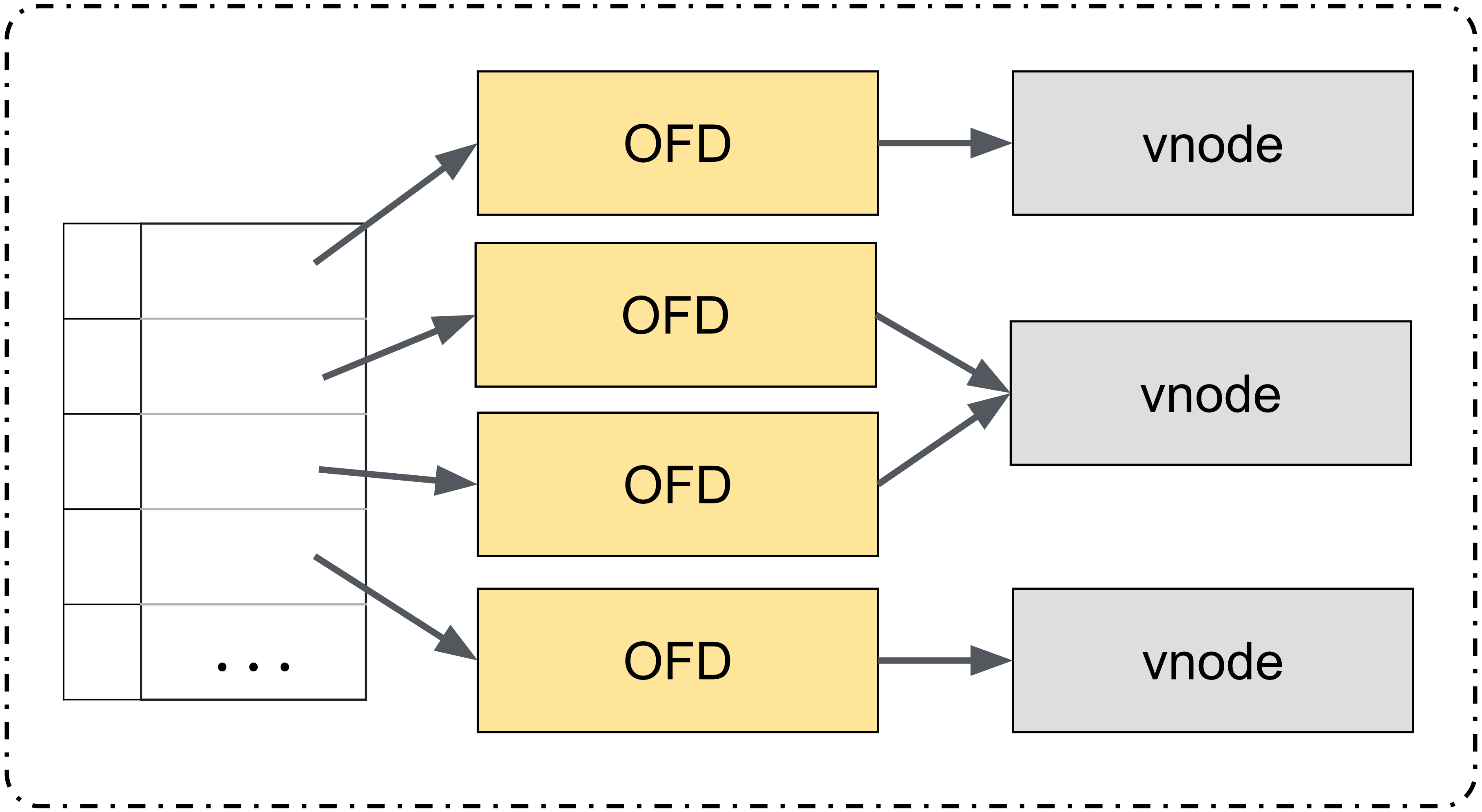
can have *multiple open file descriptions* referencing a *single vnode* (e.g., to track separate read/write positions)



for ***each process***, the kernel maintains a table of pointers to its open file structures

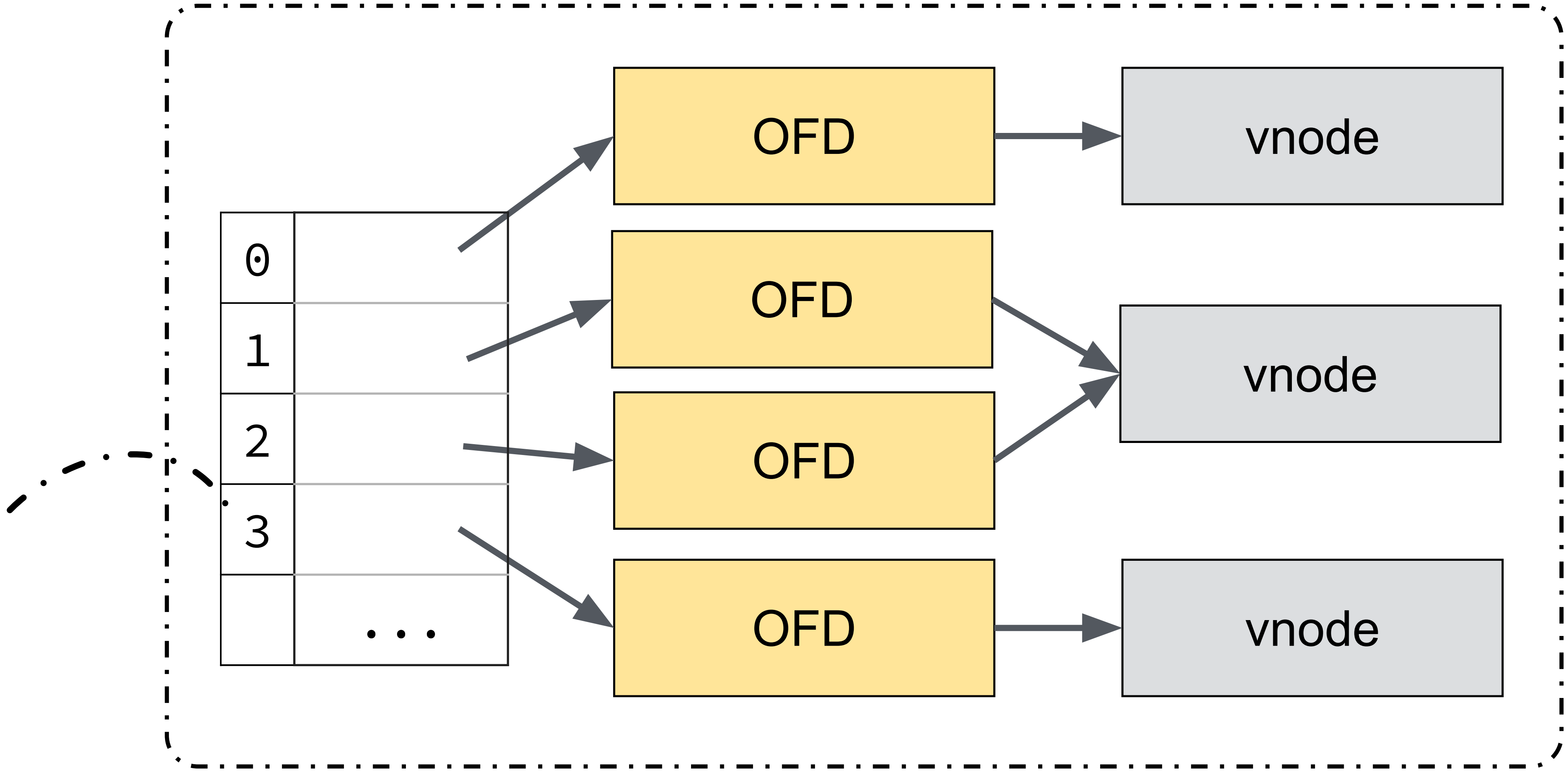


all these structures reside in *kernel memory* (off-limits to user processes)!

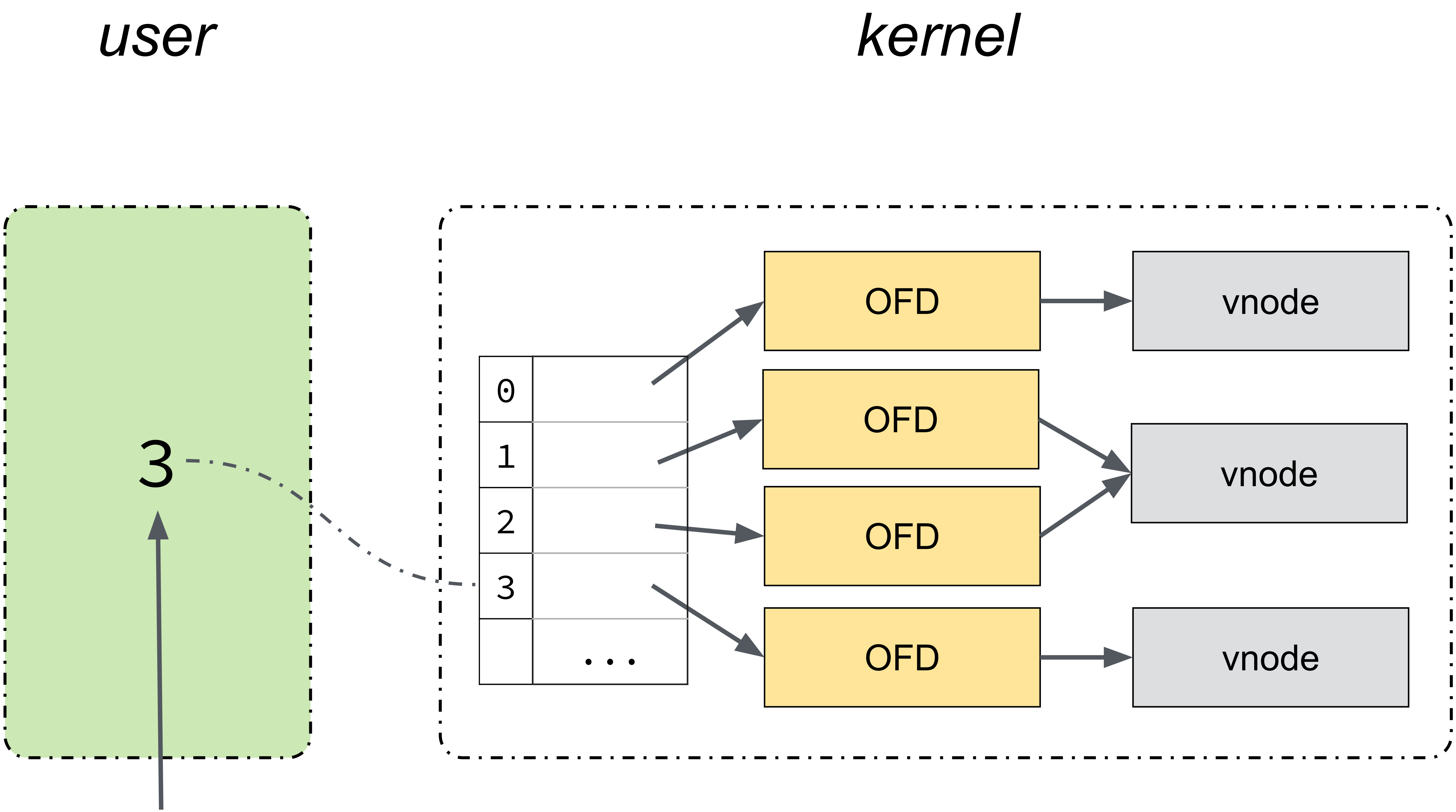


protected memory

to let a process reference an open file, the kernel returns *an index into* the table



protected memory



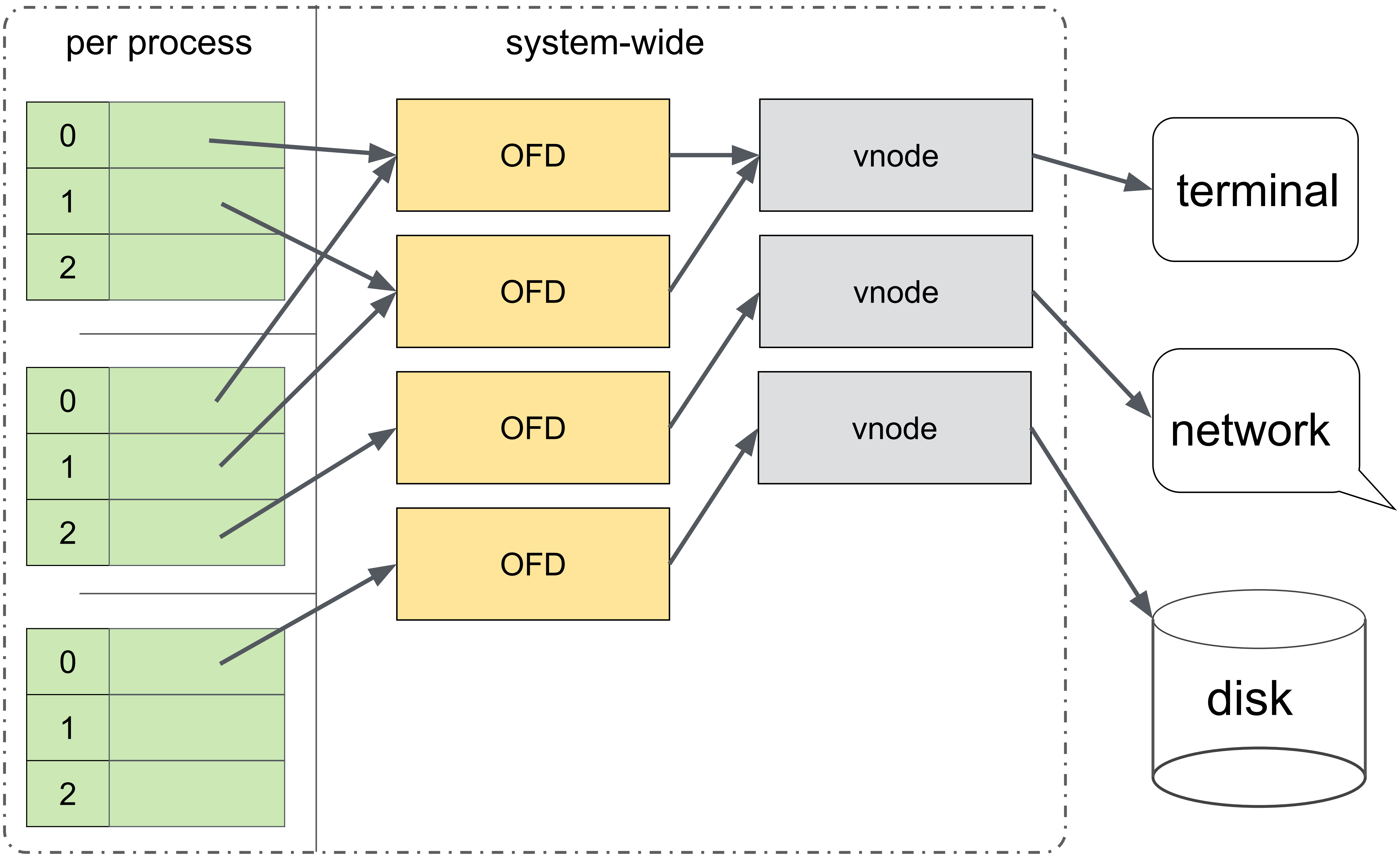
call this a **file descriptor** (FD)

by convention, processes …

- read from FD 0 for *standard input*
- write to FD 1 for *standard output*
- write to FD 2 for *standard error*

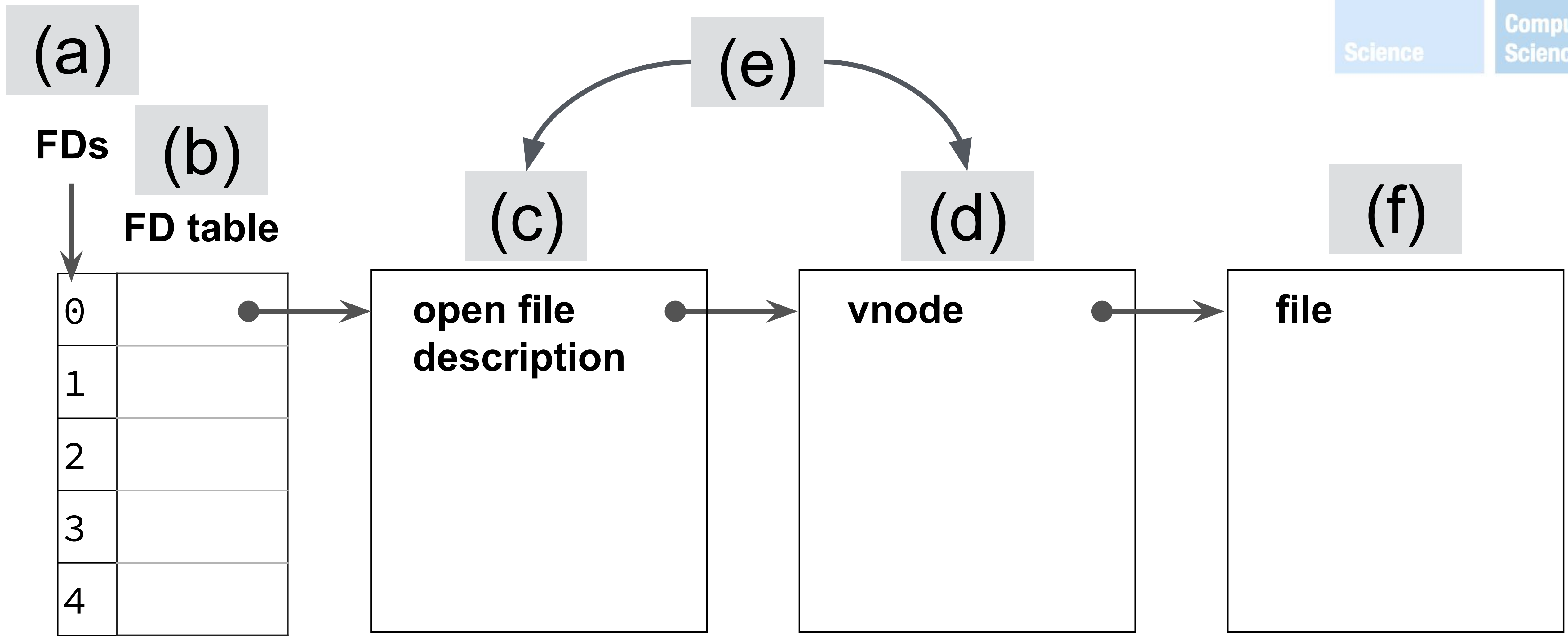
after opening a file, ***all*** *file operations* are performed using file descriptors!

kernel space

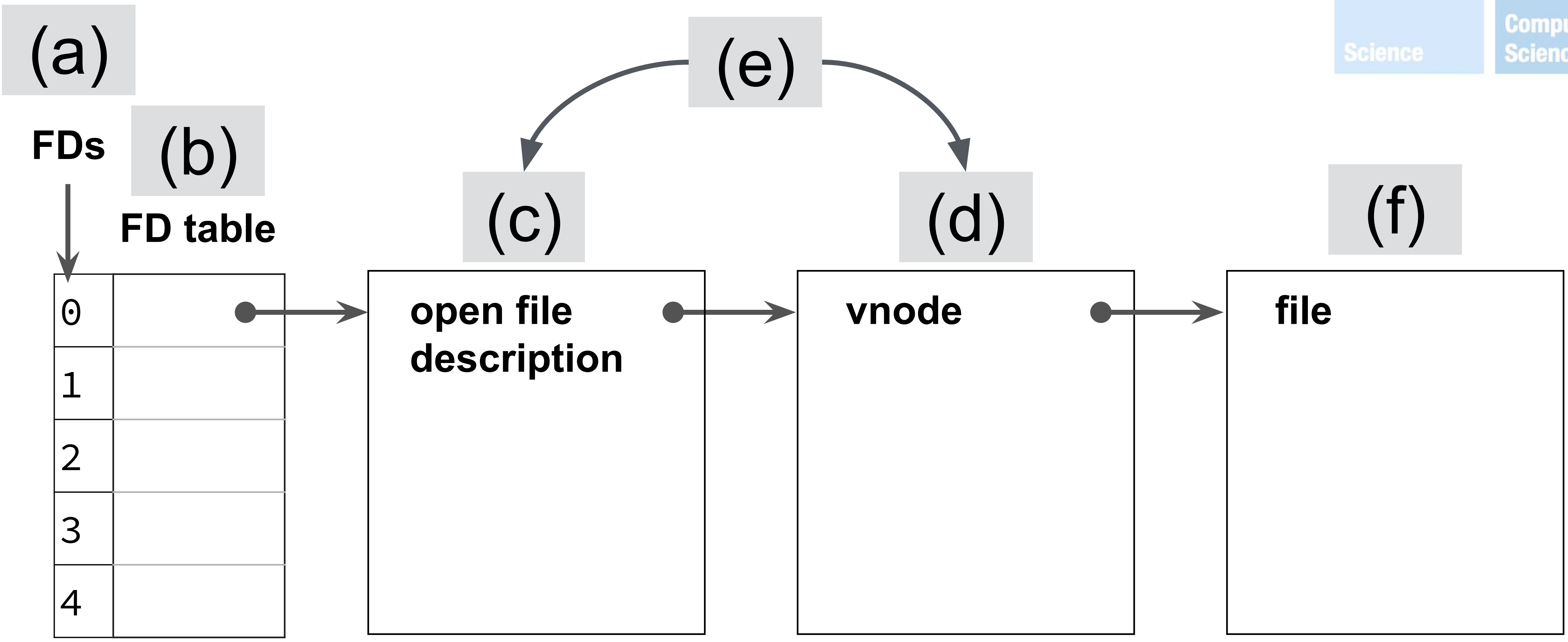


FDs *obscure* kernel I/O & FS implementation details from the user, and enable an *elegant, abstract* I/O API

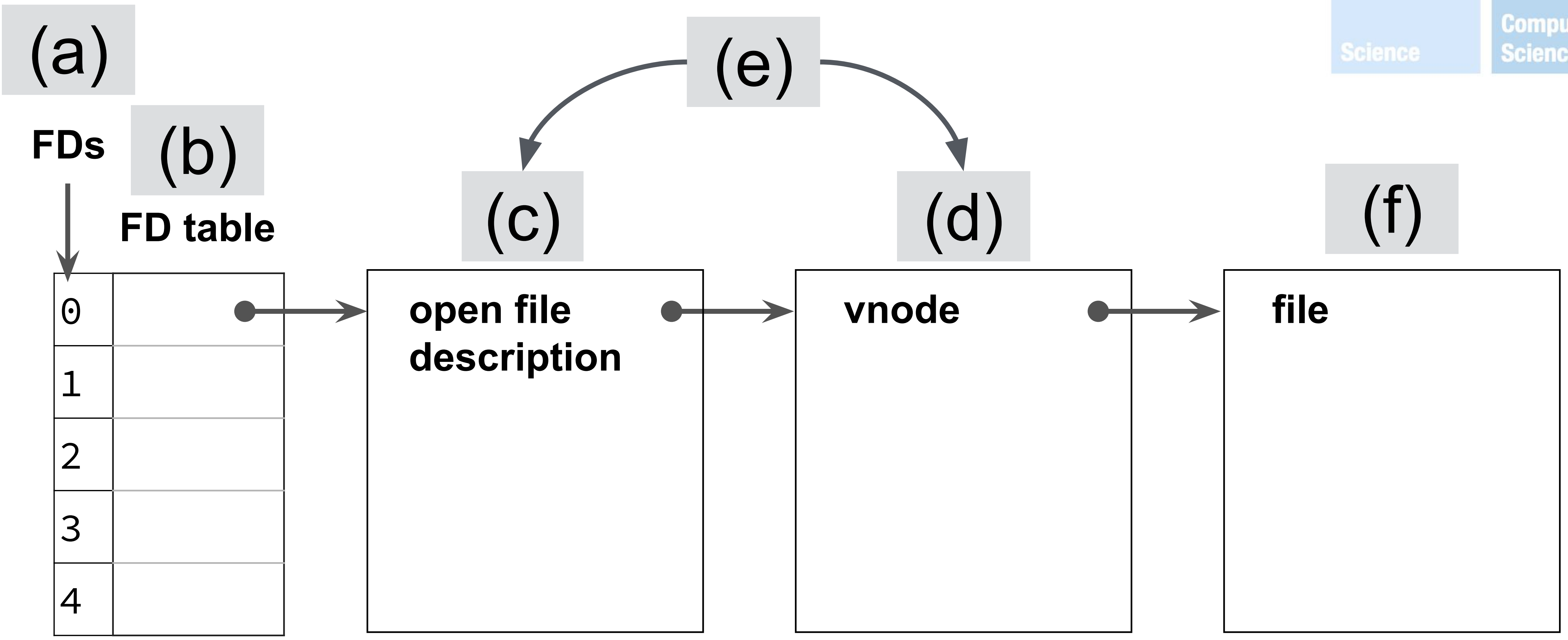
Some mini-quizzes



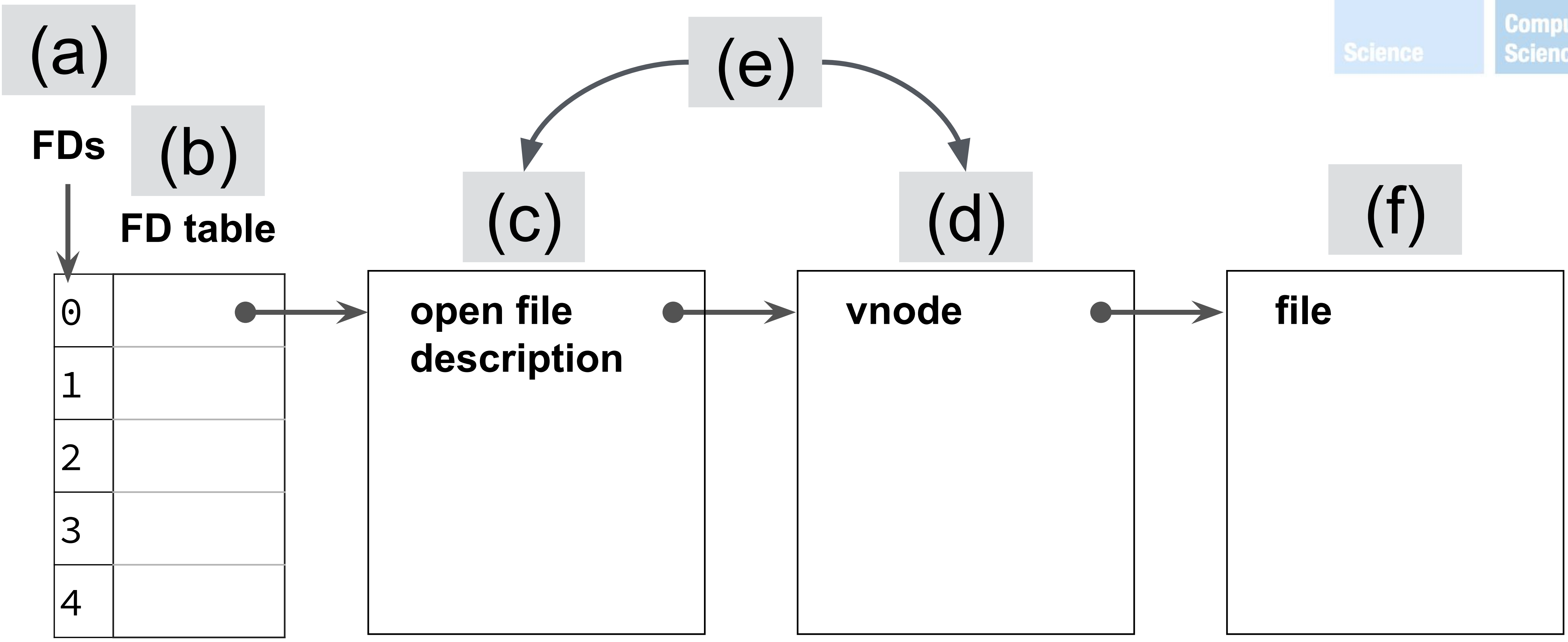
Where is the file position stored?



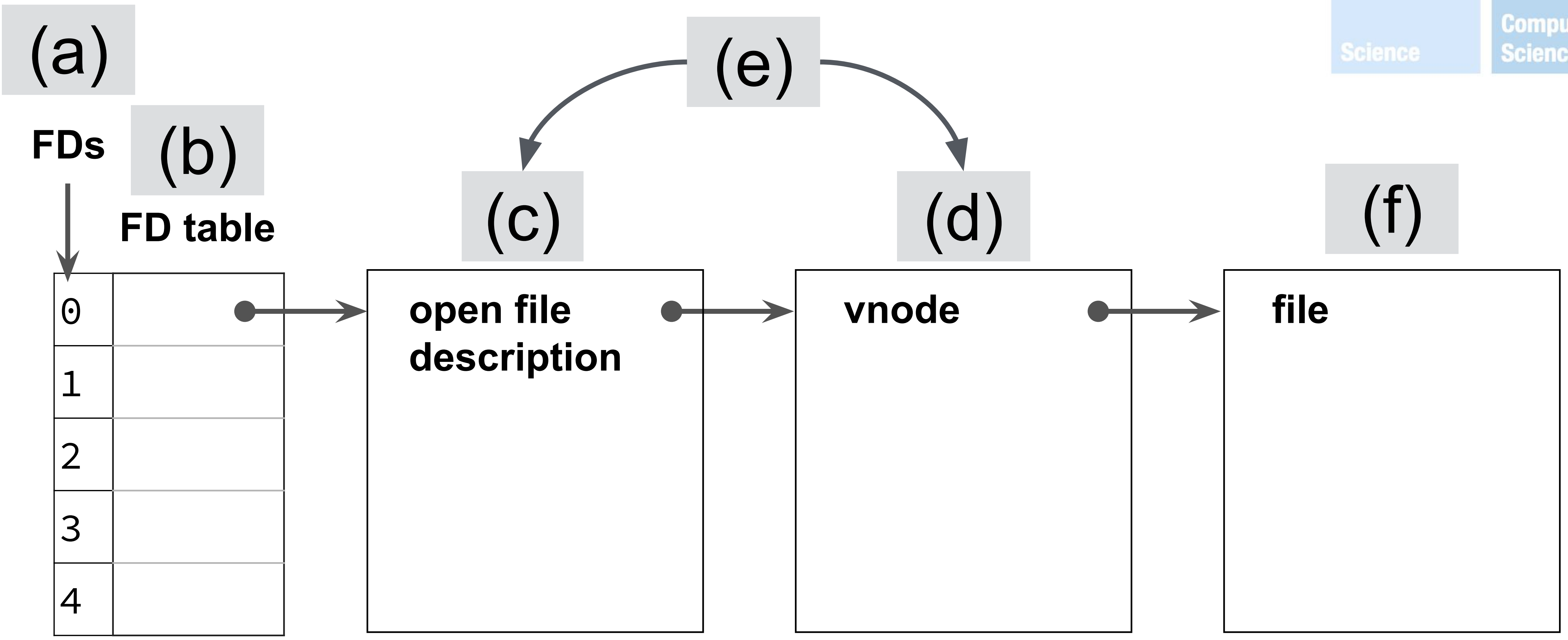
Which can be directly accessed by the user?



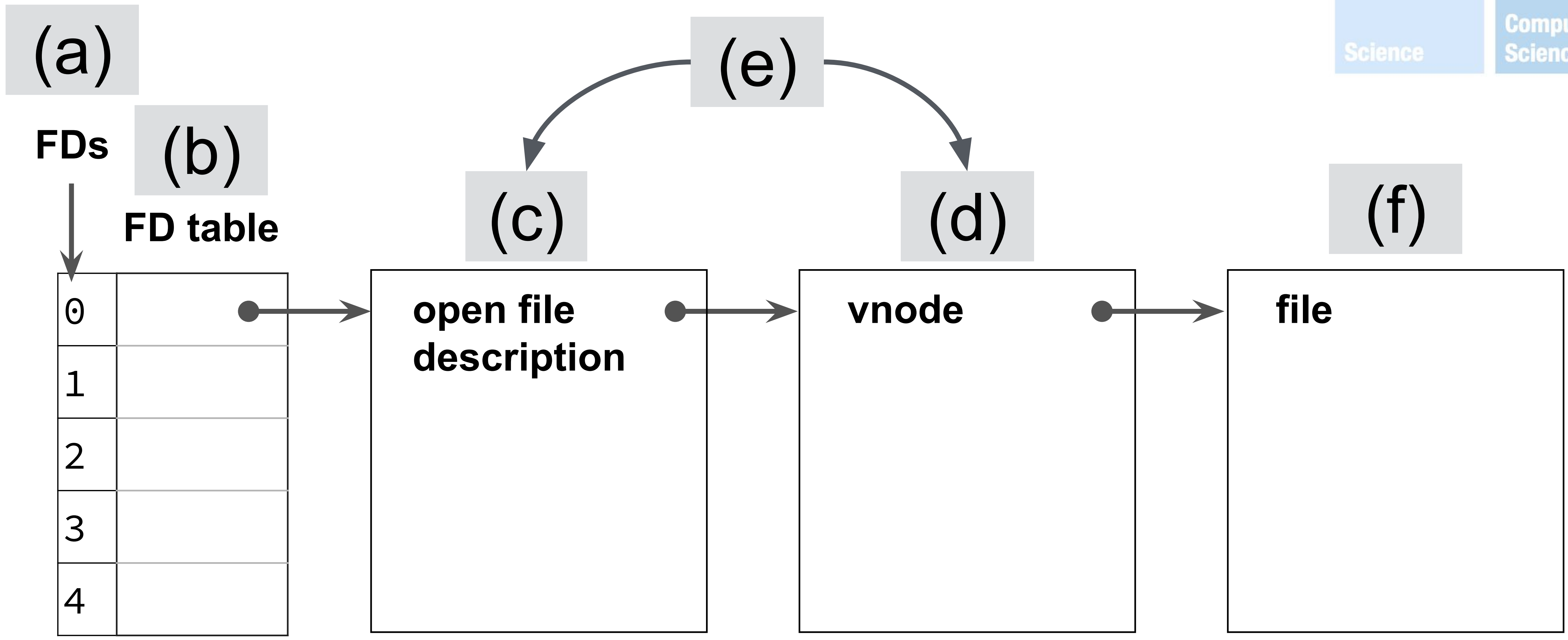
Where are permissions/ownership info stored?



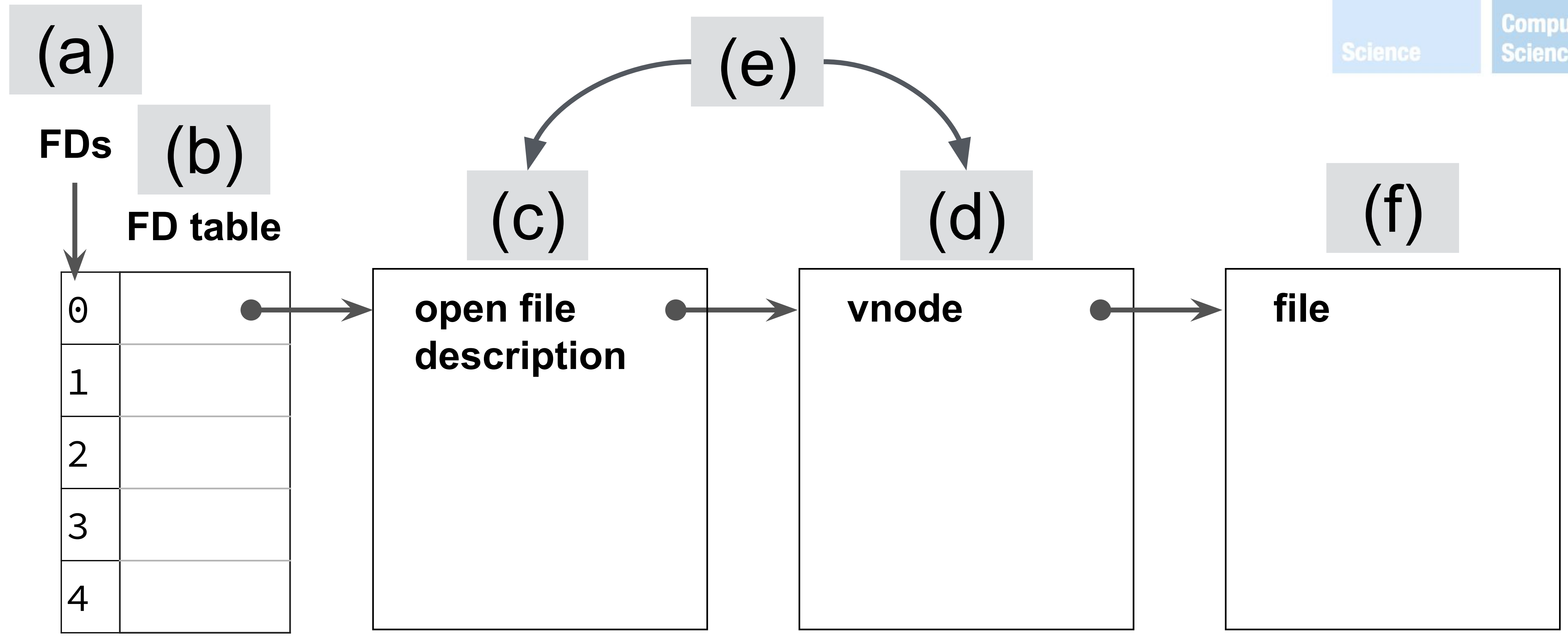
Where is data ultimately read from/written to?



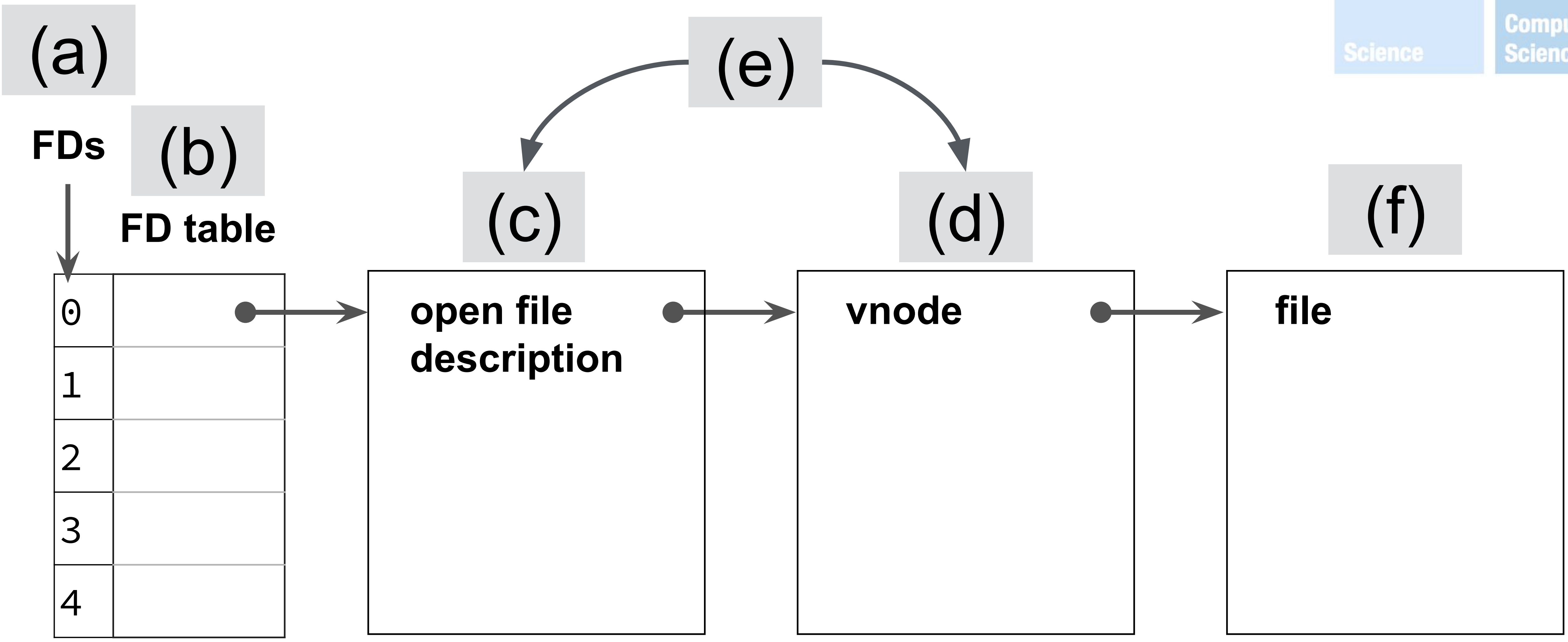
Which establish the stdin/out/err conventions?



Which are per-process?



Which are cloned on fork?



Which have a one-to-one mapping to open files?

- System-level I/O API

```
int     open ( const char *path, int oflag, ... );
int     fstat( int fd, struct stat *buf );
int     dup  ( int fd );
int     dup2 ( int fd1, int fd2 );
int     close( int fd );
off_t   lseek( int fd, off_t offset, int whence );
ssize_t read ( int fd, void *buf, size_t nbytes );
ssize_t write( int fd, const void *buf, size_t nbytes );
```

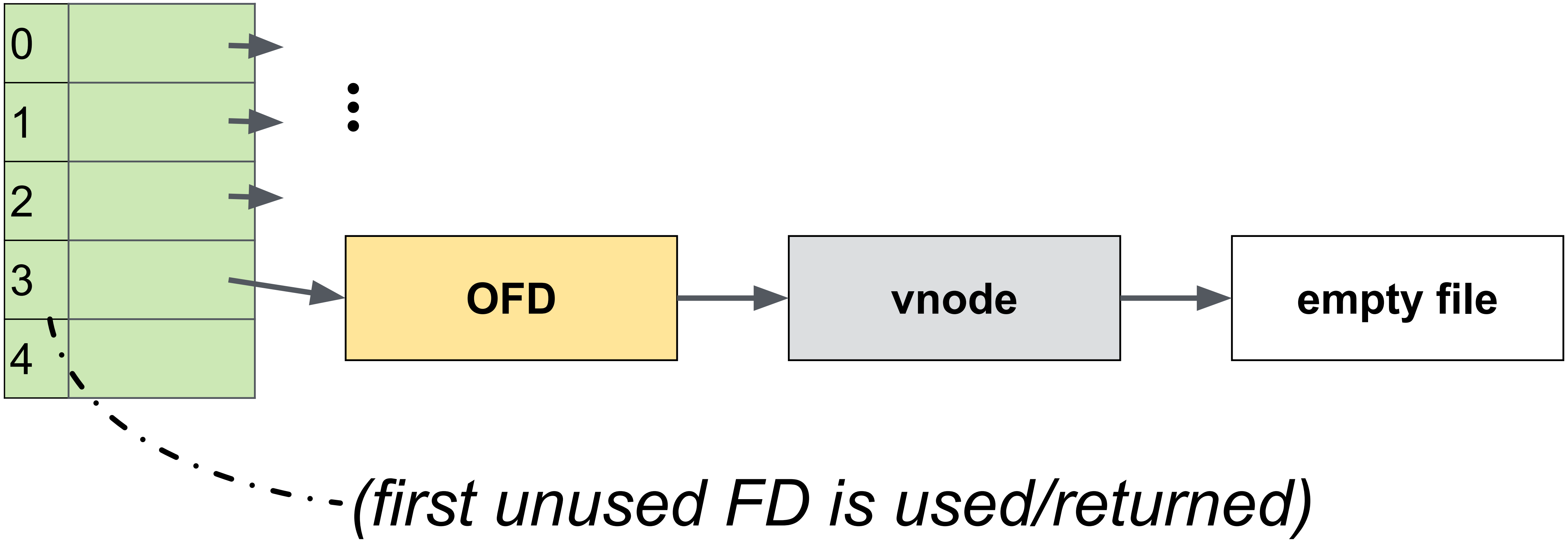
```
int open(const char *path,
         int oflag, ...);
```

- loads *vnode* for file at path (if not already loaded)
- creates & inits a new OFD
- returns a **FD** referring to the new OFD

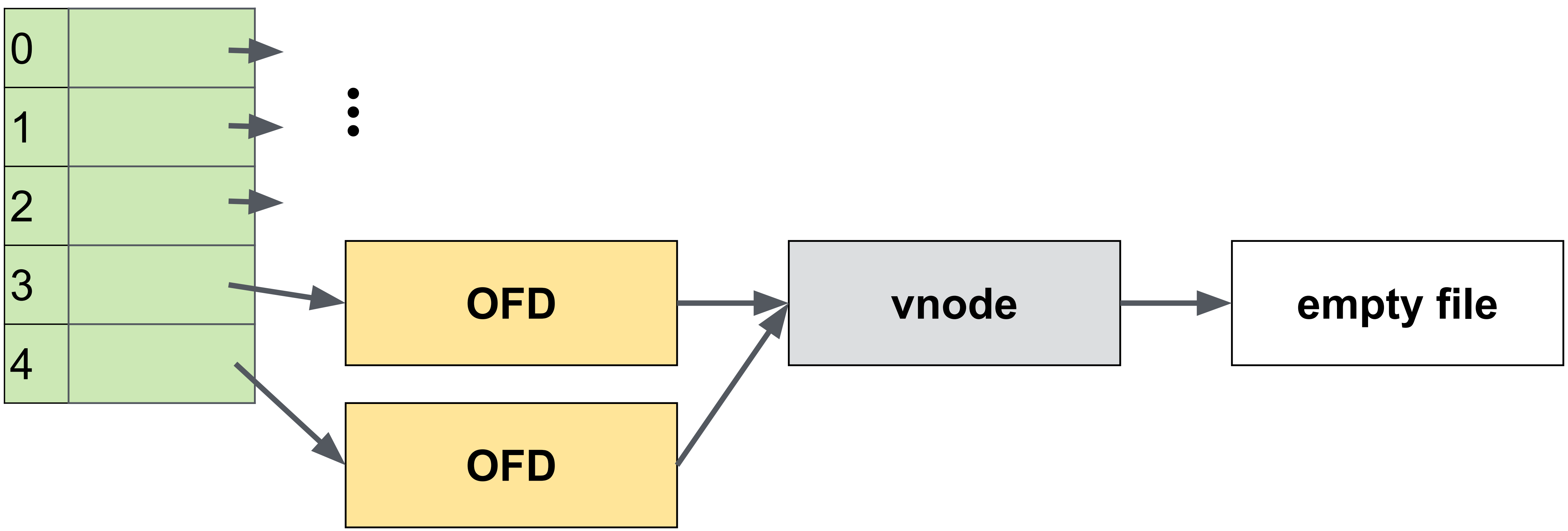
```
int open(const char *path,
         int oflag, ...);
```

- `oflag` is an *or*-ing of `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, `O_TRUNC`, etc.
- if `O_CREAT`, must specify access permissions of new file (“rwx” flags)

```
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
```



```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = open("foo.txt", O_RDONLY);
```

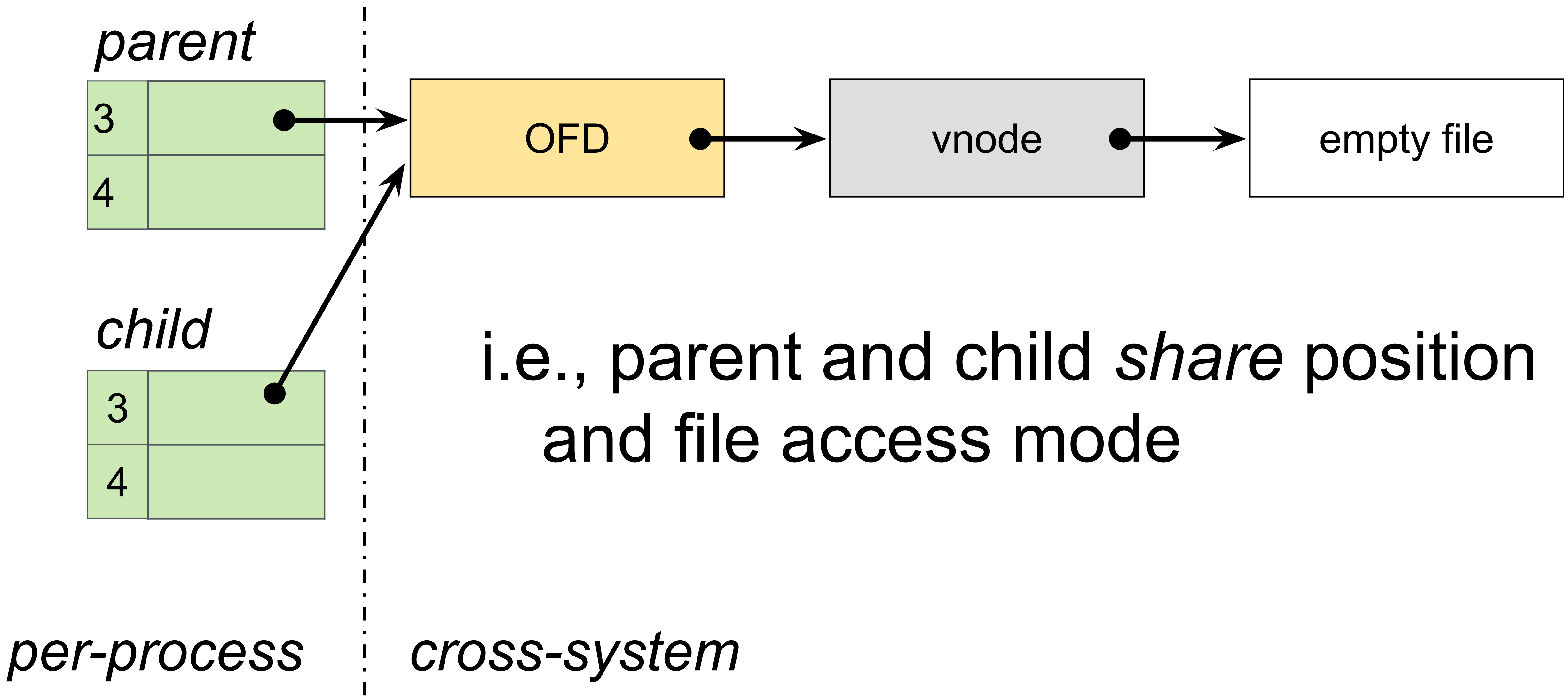


```
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);

struct stat stat;

/* query file metadata */
fstat(fd1, &stat);

printf("Inode # : %lu\n", stat.st_ino);
printf("Size    : %lu\n", stat.st_size);
printf("Links   : %lu\n", stat.st_nlink);
```

```
Inode # : 19603149
Size    : 0
Links   : 1
```

a process inherits its parent’s open files across a fork, and *retains them post*-exec!

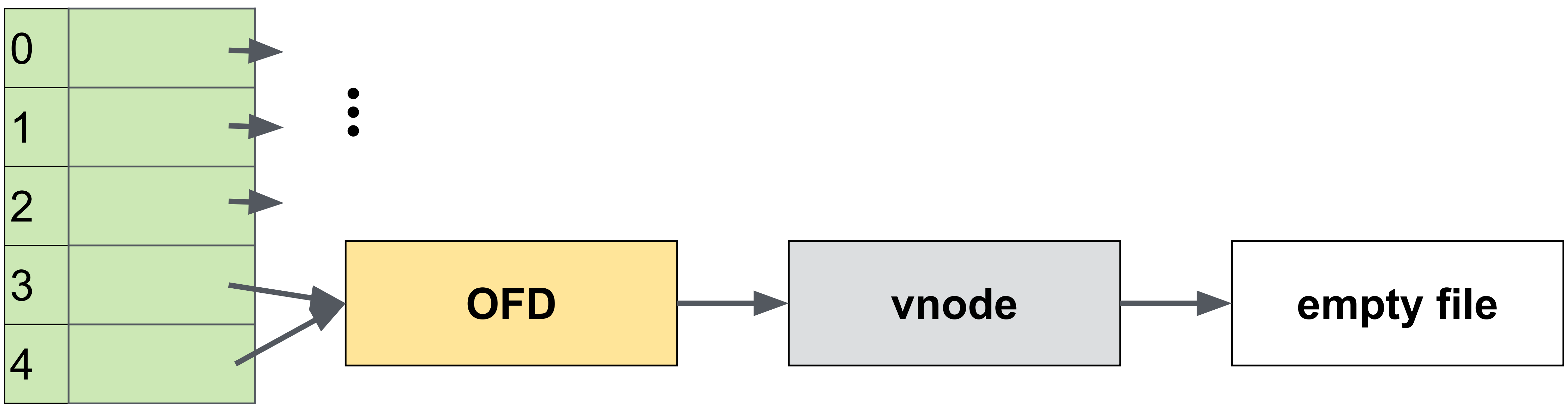
```
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
fork();
```



sharing an OFD can be very handy — e.g., for coordinating output to terminal

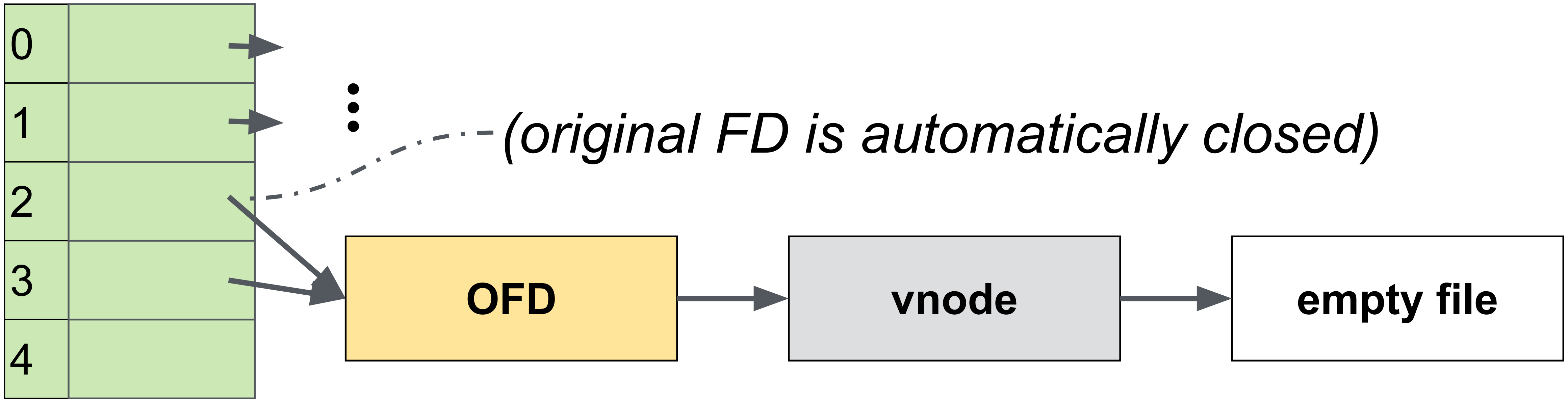
can also explicitly “share” position from separate FDs using dup syscalls

```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = dup(fd1);
```



i.e., reading/writing FD 4 is equivalent to doing so with FD 3

```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
dup2(fd1, 2); /* second arg is "destination" fd */
```

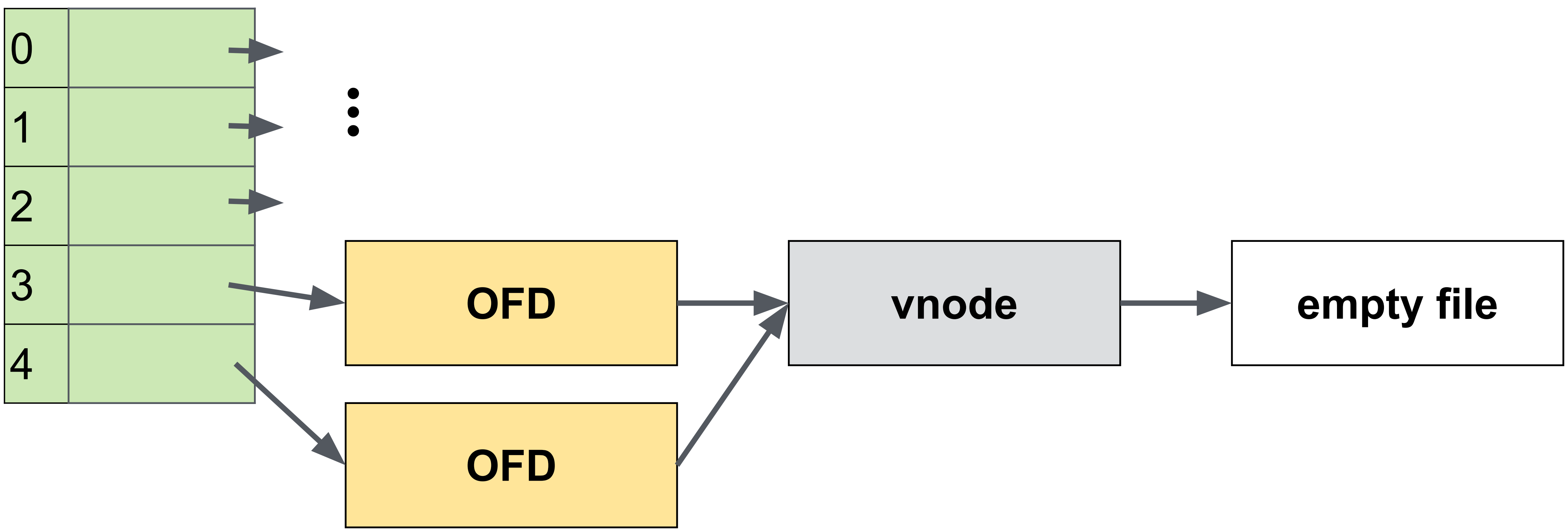


i.e., reading/writing FD 2 (*stderr*) is equivalent to doing so with FD 3

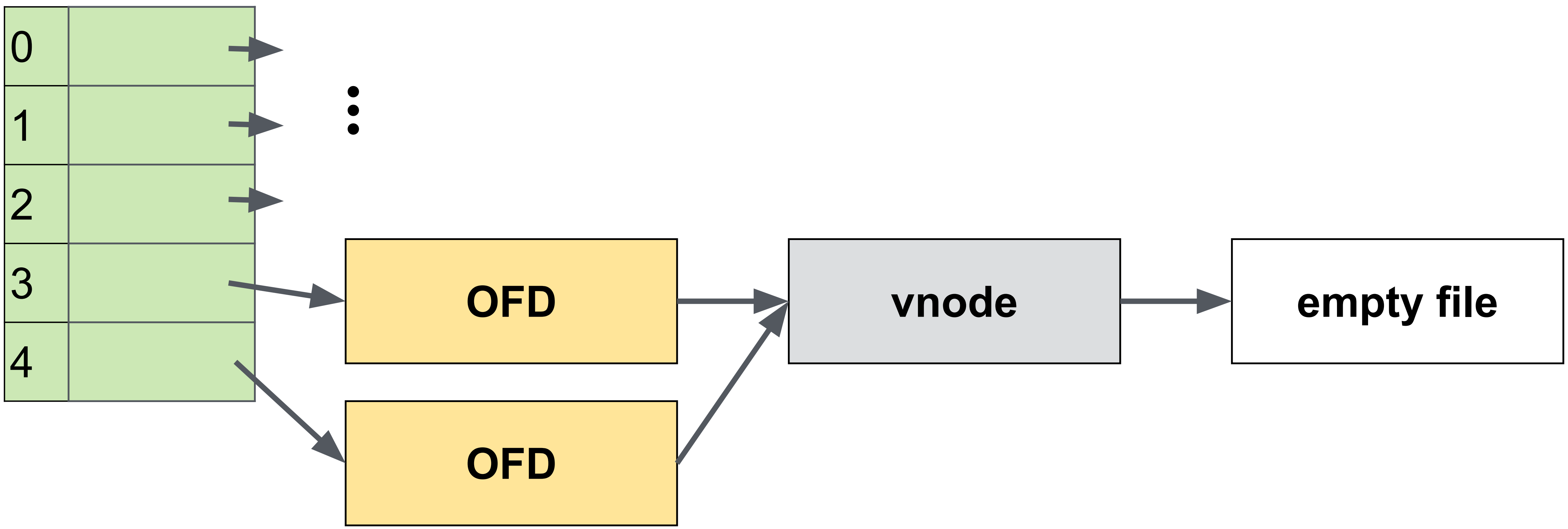
```
int close(int fd);
```

- delete OFD pointer in file table for fd
- if the OFD has no referring FDs (in any process), deallocate it

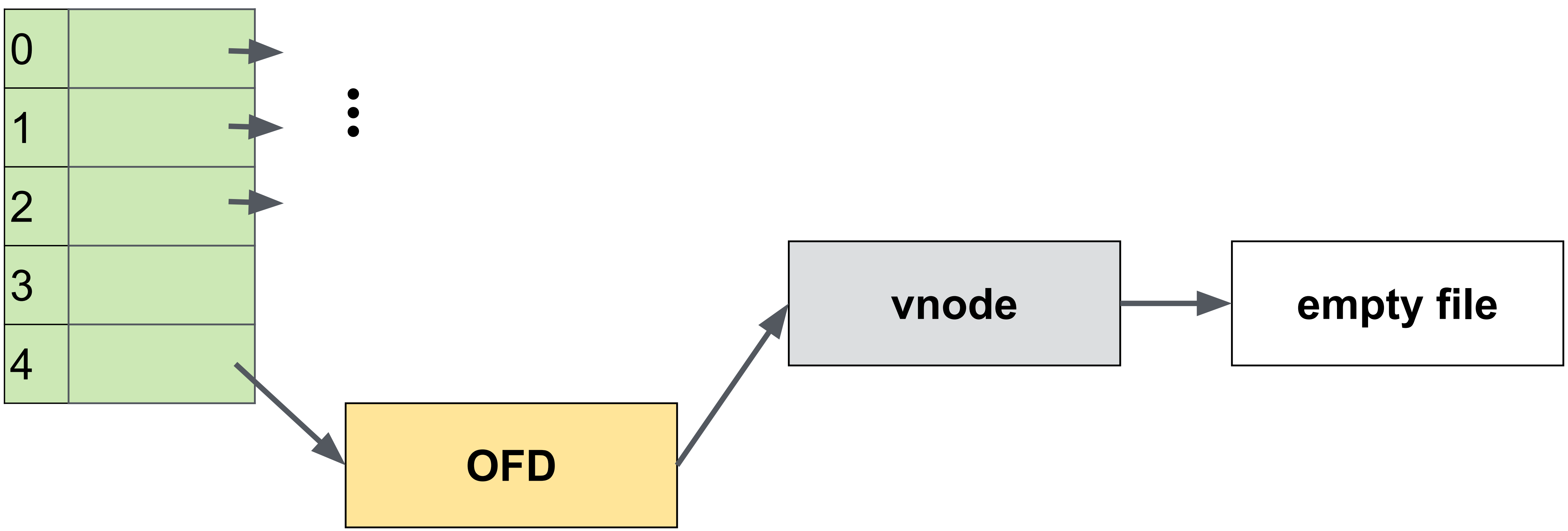
```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = open("foo.txt", O_RDONLY);
```



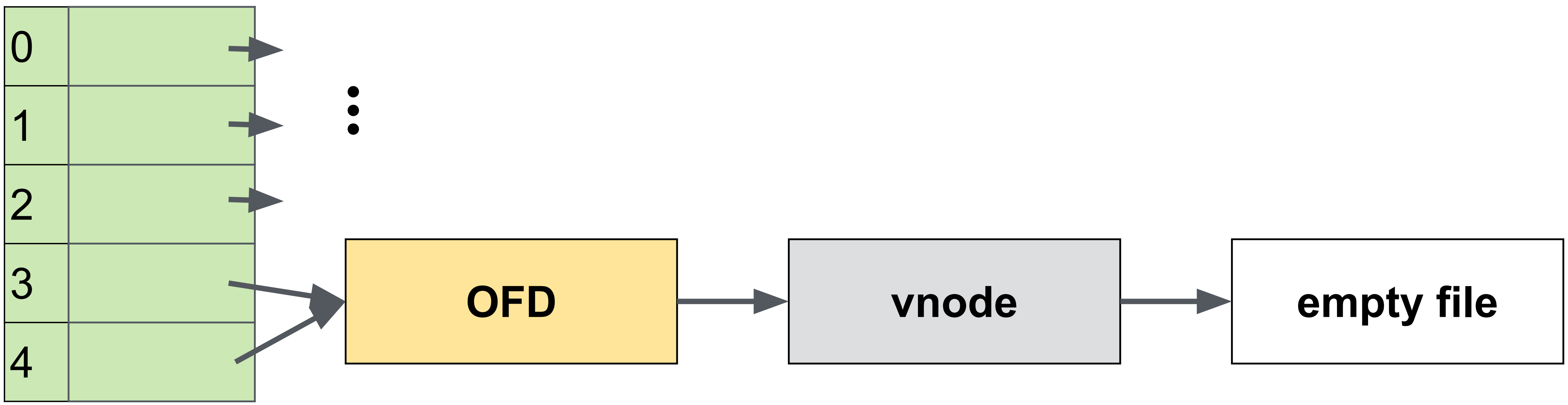
```
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = open("foo.txt", O_RDONLY);
close(fd1);
```



```
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = open("foo.txt", O_RDONLY);
close(fd1);
close(fd2);
```



```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = dup(fd1);
close(fd1);
```



```
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = dup(fd1);
close(fd1);
close(fd2);
```