CS 3650 Computer Systems – Summer 2025

File I/O

Unit 6

* Acknowledgements: created based on Christo Wilson, Ferdinand Vesely, Ji-Yong Shin, and Alden Jackson’s lecture slides for the same course.

POSIX File I/O

Everything is a file, until it isn't.

POSIX File System Basics

- We’ve been introduced to two types of virtualization:
- The process, which virtualizes the CPU
- The address space, which virtualizes memory (more details on this later)
- Together, they allow a program to run as if it had its own private processor and its own memory

- Persistent storage, i.e., disk drives, which keep data intact when power is lost, is one more element in the virtualization model
- Two major abstractions: files and directories

Files and Directories

- File
    - Linear array of bytes that can be written or read
    - Name
        - Low-level: inode number, an non-zero integer, used by the OS
        - User-readable

- Directory
    - File containing list of (low-level name, user-readable name) pairs
    - Can contain other directories, as a directory is a file
        - Root directory: /
        - Current directory: .
        - Parent directory: ..

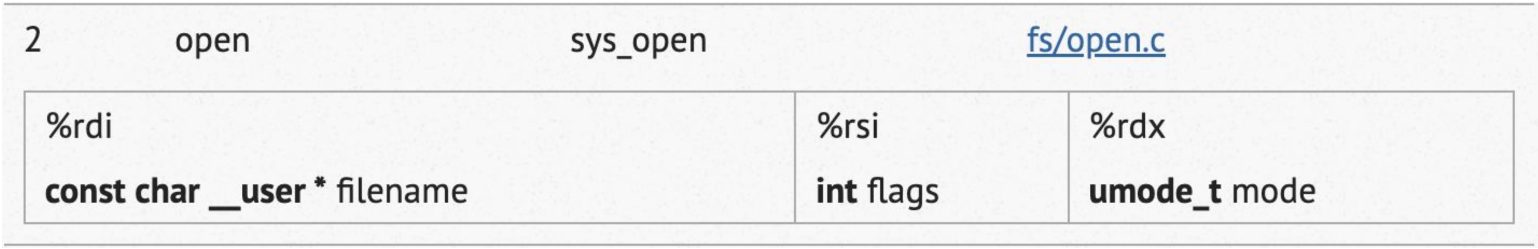
Path

- Absolute path
    - Starts from the root directory
        - /home/ben/courses/cs3650/assignment.txt

- Relative path
    - Starts from current directory location
        - Assume current directory is /home/ben/
            - ./courses/cs3650/assignment.txt

open / close

- Opening an existing or creating a new file is with the open() system call



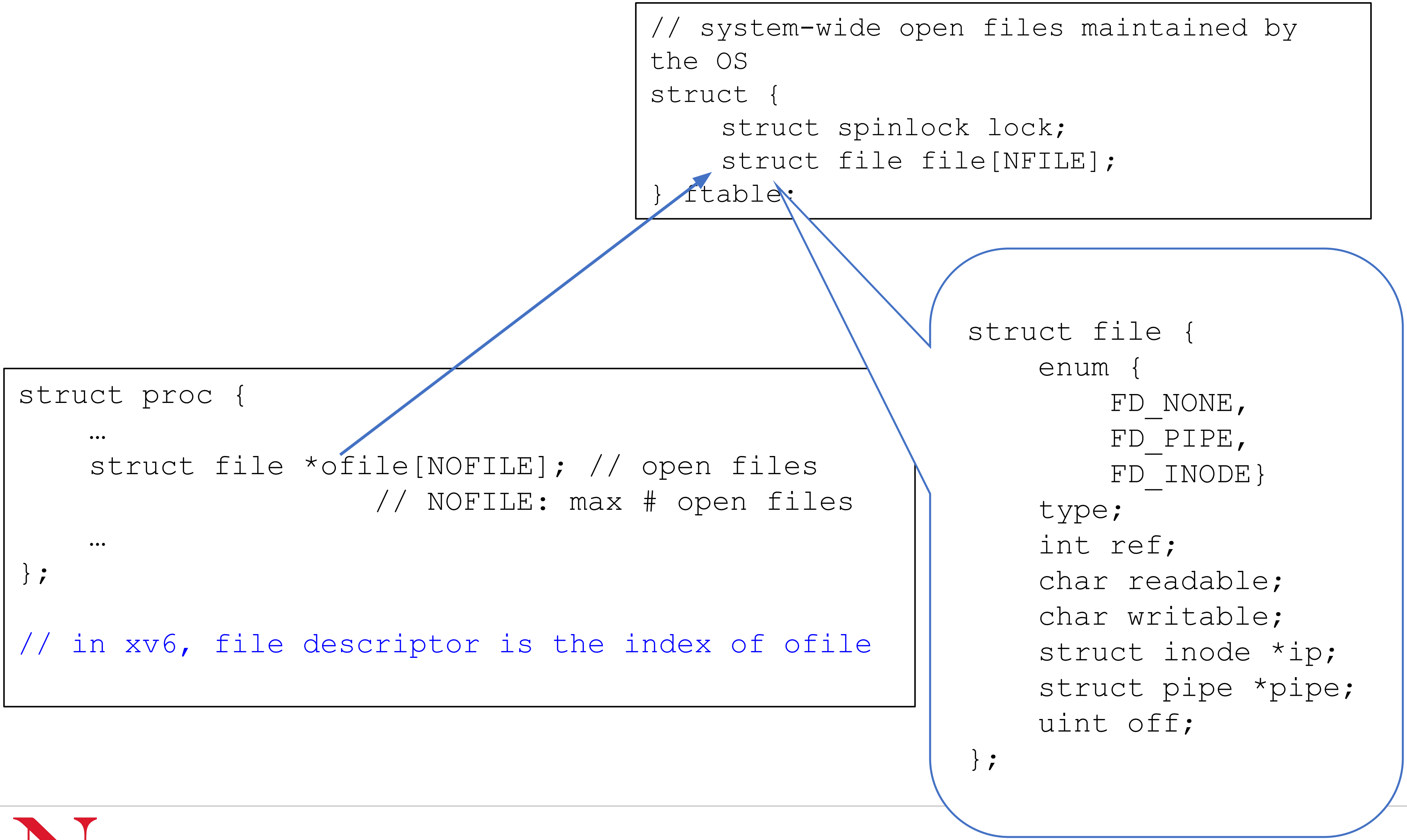
2	open	sys_open	fs/open.c

%rdi	%rsi	%rdx
const char __user * filename	**int** flags	**umode_t** mode

```
// Create file “foo” and return a file descriptor
int fd = open("foo",
        O_CREAT|O_WRONLY|O_TRUNC, // create write-only
        S_IRUSR|S_IWUSR);         // set permissions
```

- File descriptor, fd:
  - An integer, private per process, used by OS to access files
  - Use fd to read or write the file.
  - stdin = 0, stdout = 1, stderr = 2
  - Open returns lowest-numbered fd that is not currently open

Struct file in xv6

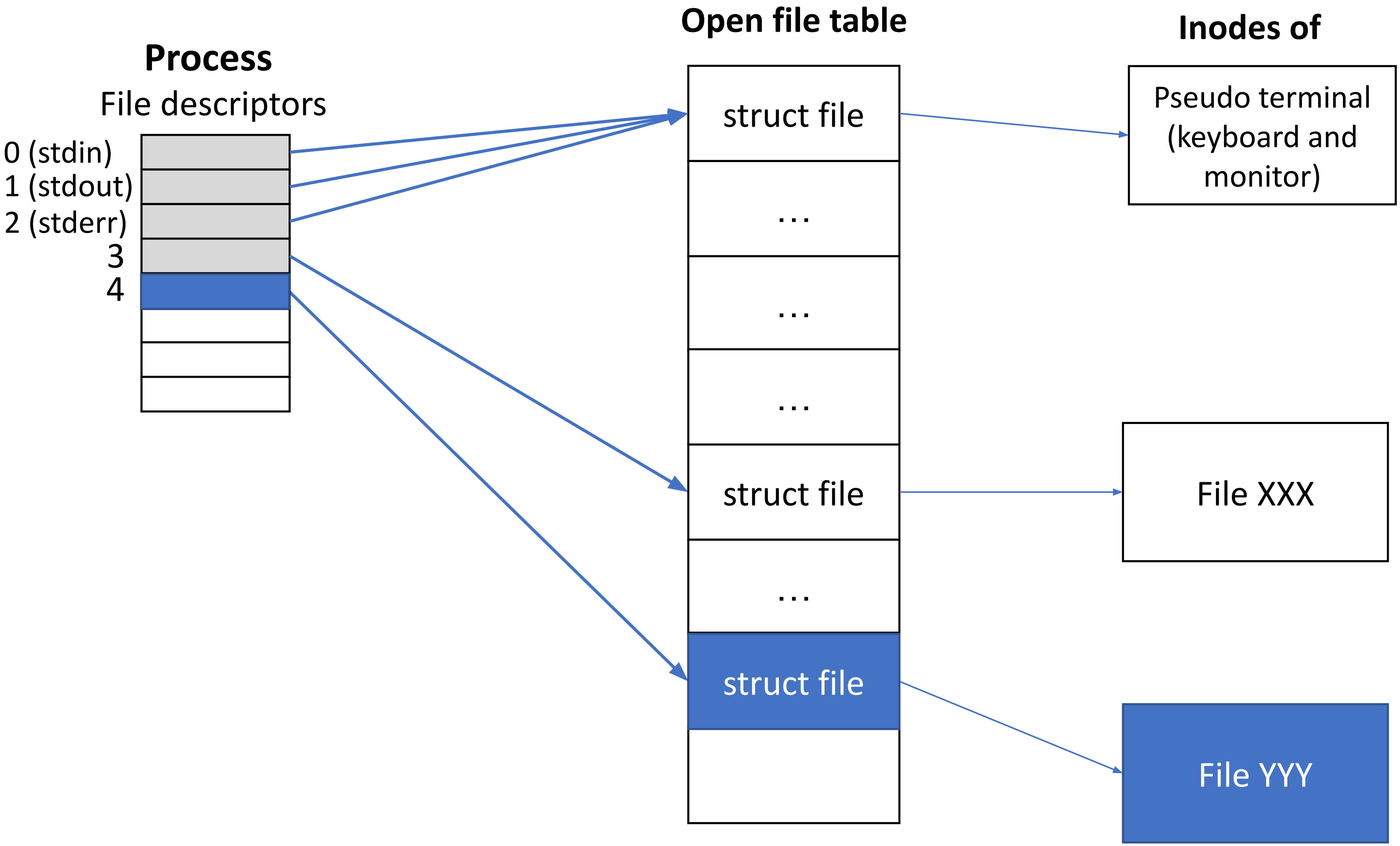


```
// system-wide open files maintained by
the OS
struct {
    struct spinlock lock;
    struct file file[NFILE];
} ftable;
```

```
struct proc {
    …
    struct file *ofile[NOFILE]; // open files
                    // NOFILE: max # open files
    …
};

// in xv6, file descriptor is the index of ofile
```

```
struct file {
    enum {
        FD_NONE,
        FD_PIPE,
        FD_INODE}
    type;
    int ref;
    char readable;
    char writable;
    struct inode *ip;
    struct pipe *pipe;
    uint off;
};
```

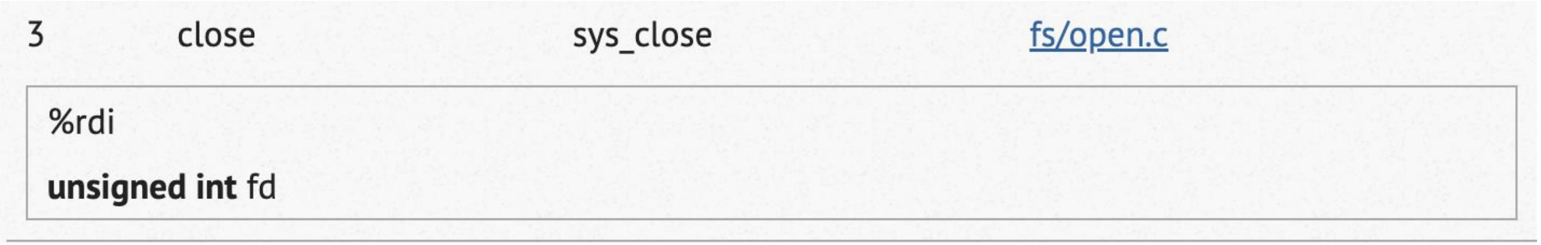
Struct file in xv6



open / close

- To close the file:

```
// Close an open file descriptor
int close(int fd);  // returns 0 on success
```



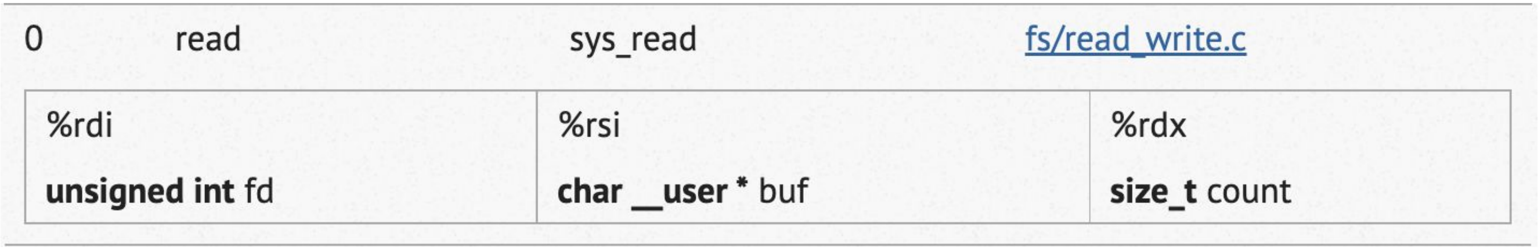
3	close	sys_close	fs/open.c
%rdi **unsigned int** fd			

read / write

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

read() attempts to read up to **count** bytes from file descriptor **fd** into the buffer starting at **buf**.

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.



0	read	sys_read	fs/read_write.c

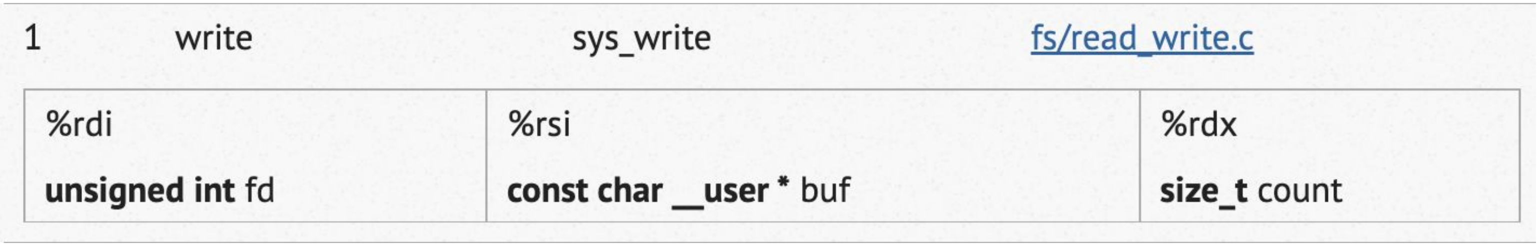
%rdi	%rsi	%rdx
unsigned int fd	**char __user *** buf	**size_t** count

read / write

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

`write()` writes up to **`count`** bytes from the buffer starting at **`buf`** to the file referred to by the file descriptor **`fd`**.

On success, the number of bytes written is returned. On error, -1 is returned and errno is set to indicate the cause of the error.



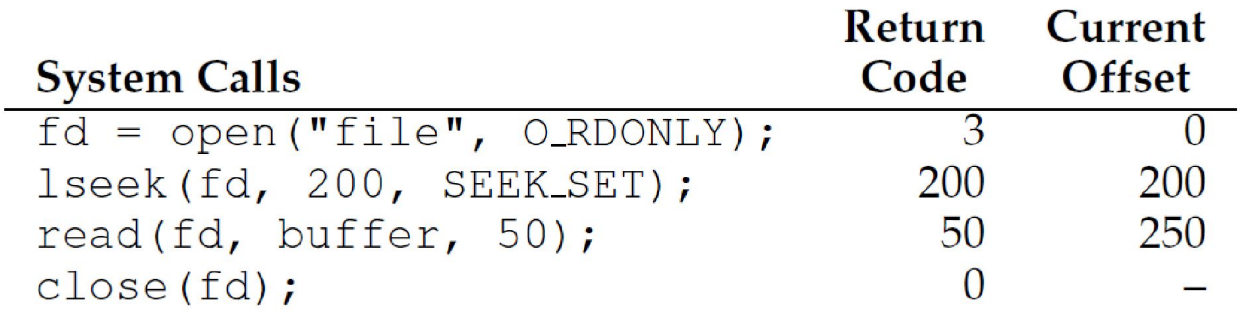
1	write	sys_write	fs/read_write.c

%rdi	%rsi	%rdx
unsigned int fd	**const char __user *** buf	**size_t** count

lseek

- Setting offset of the file for data accesses

- `off_t lseek(int fd, off_t offset, int whence)`
  - Fd: file descriptor
  - Offset: resulting offset location
  - Whence: tells us how to compute the location using the offset
    - SEEK_SET: offset = given offset
    - SEEK_CUR: offset = current offset + given offset
    - SEEK_END: offset = end of file + given offset



System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`lseek(fd, 200, SEEK_SET);`	200	200
`read(fd, buffer, 50);`	50	250
`close(fd);`	0	–

Example: using `strace`

```
$ echo "hello cs3650" > foo
$ strace cat foo
…
openat(AT_FDCWD, "foo", O_RDONLY)       = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=13, ...}) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap(NULL, 1056768, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f8f66844000
read(3, "hello cs3650\n", 1048576)      = 13
write(1, "hello cs3650\n", 13)          = 13
read(3, "", 1048576)                    = 0
munmap(0x7f8f66844000, 1056768)         = 0
close(3)                                = 0
close(1)                                = 0
close(2)                                = 0
…
$
```

stdin = 0, stdout = 1, stderr = 2

openat() returns file descriptor = 3
fstat() returns status information on 3,
in particular length of file (13 bytes)

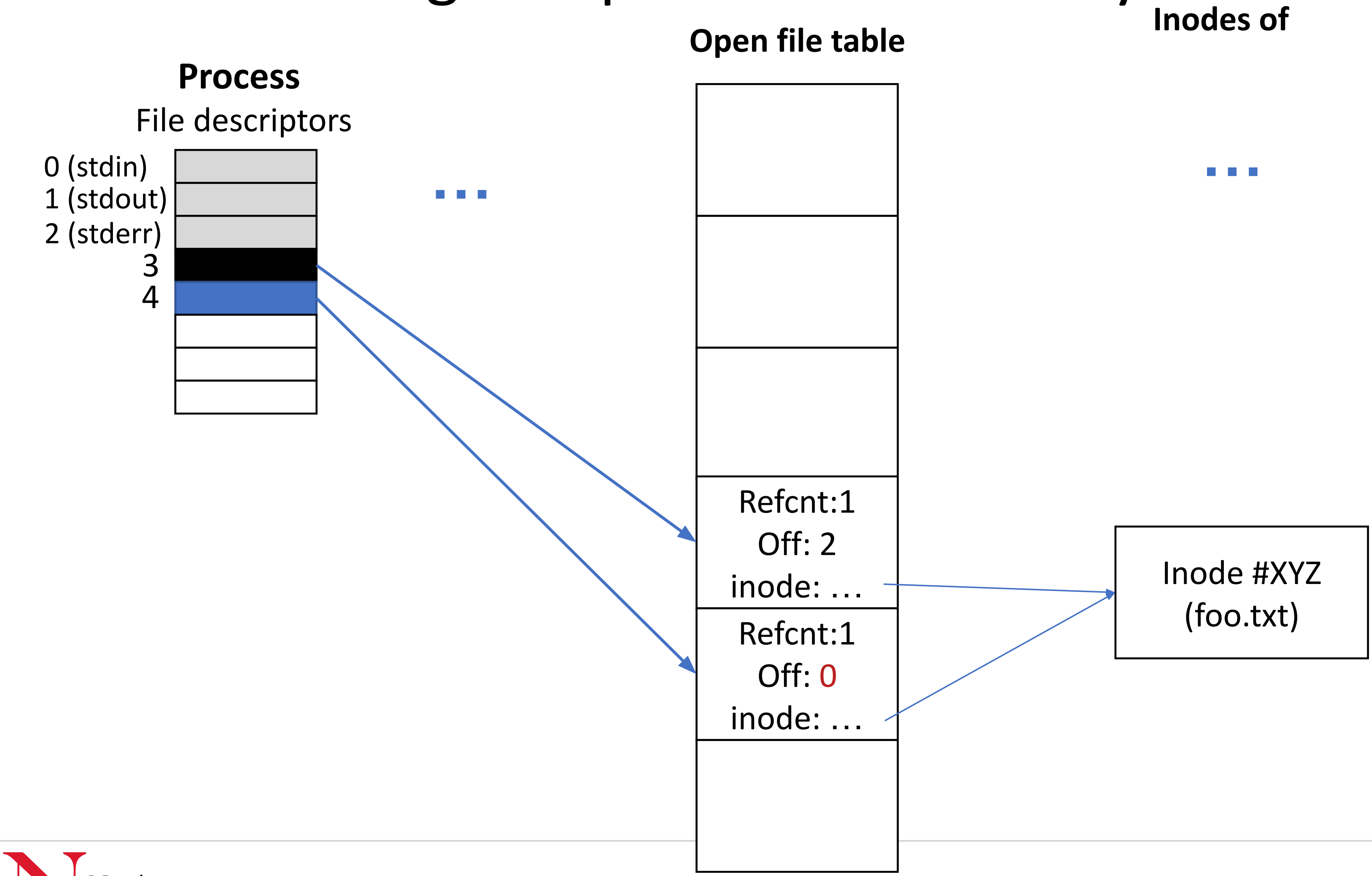
read(13 bytes from 3)
write(13 bytes to 1)

read(0 bytes from 3)

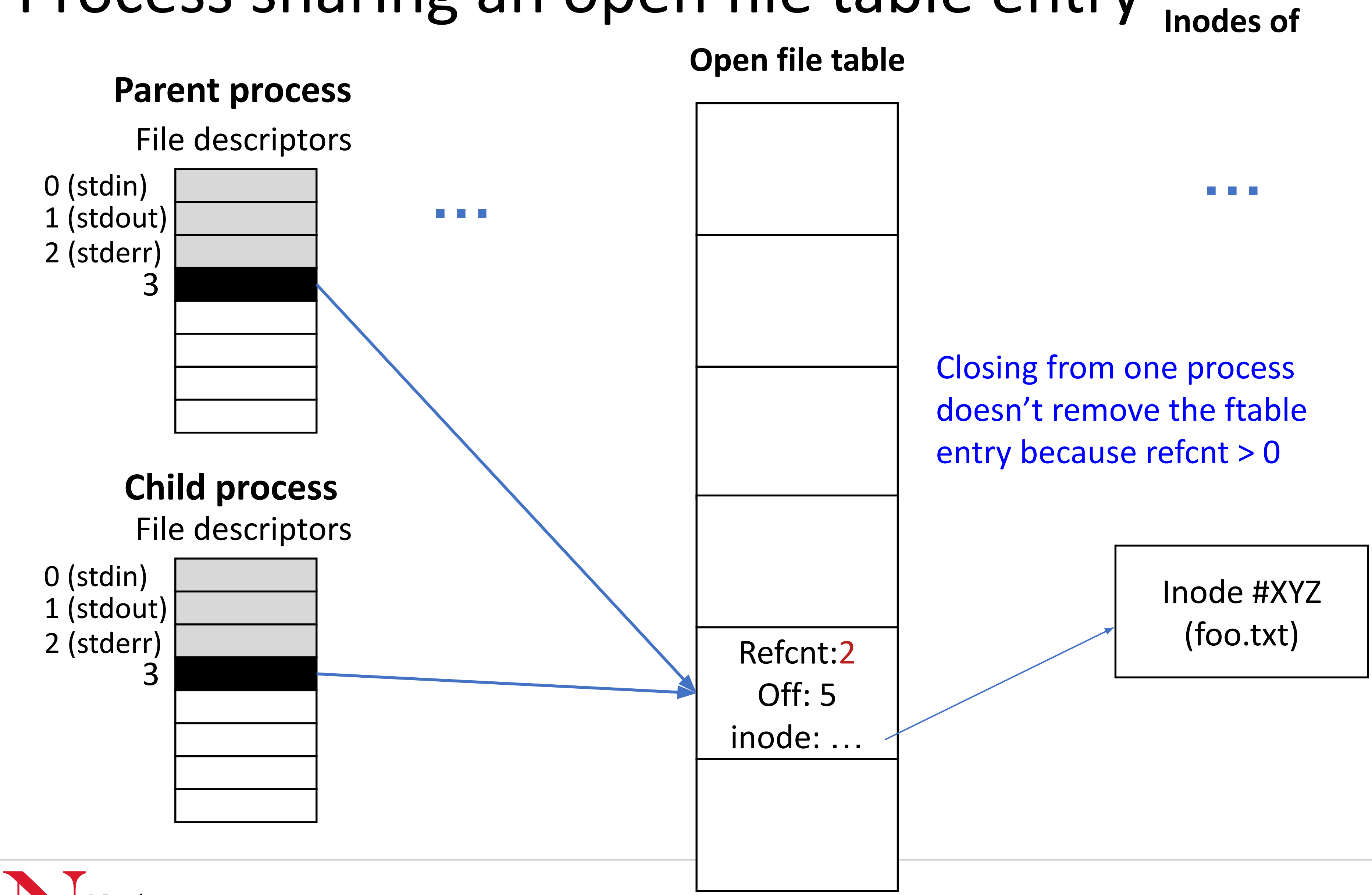
close() all open fds

Open/Read/Write/lseek Demo

Process sharing an open file table entry



Process sharing an open file table entry



Closing from one process doesn’t remove the ftable entry because refcnt > 0

Redirecting I/O

All running programs have 3 default I/O streams:

- Standard Input: `stdin` (0)
- Standard Output: `stdout` (1)
- Standard Error: `stderr` (2)

By default,

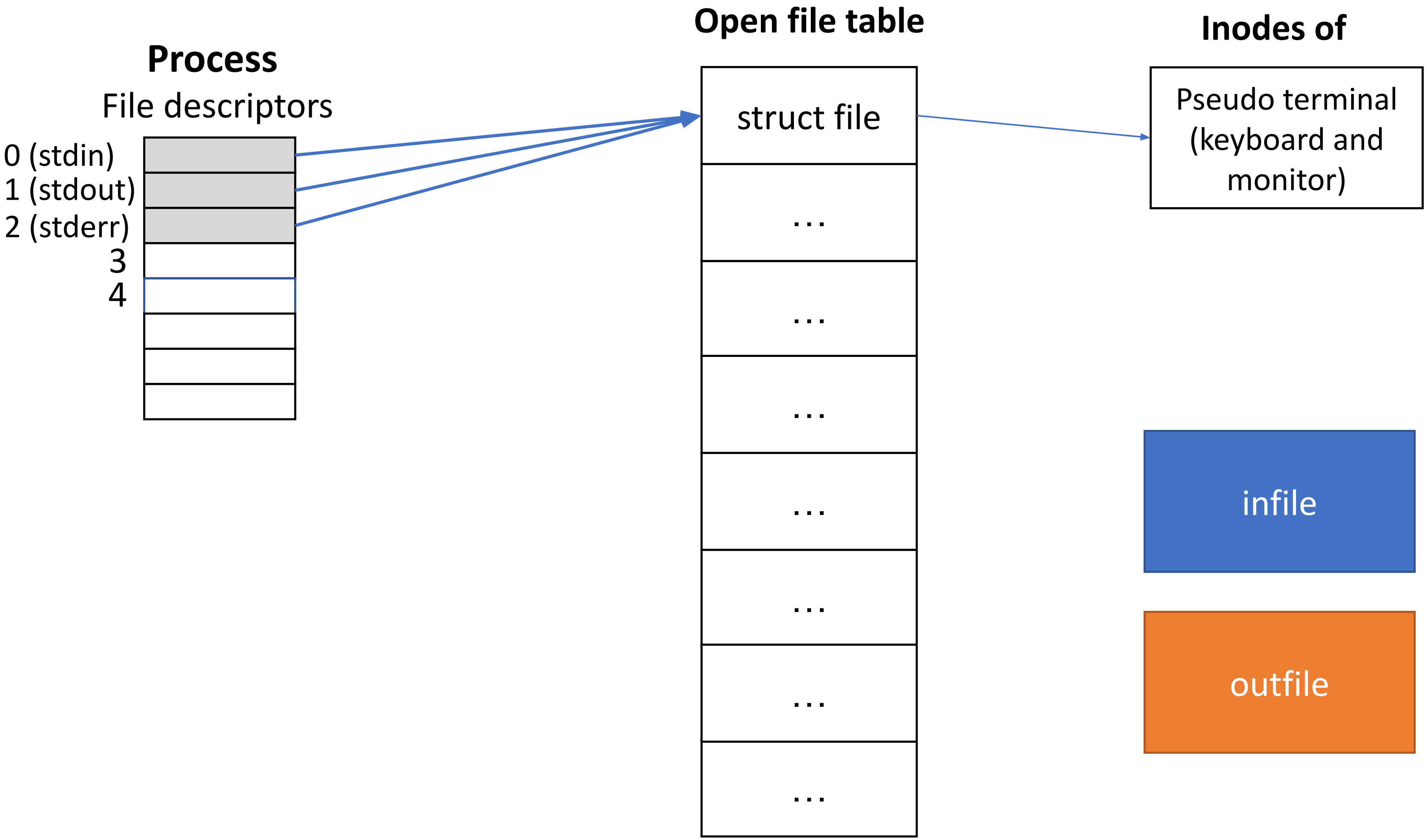
- **`stdin` is the keyboard**
- **`stdout` and `stderr` are the terminal**

But these can be redirected…

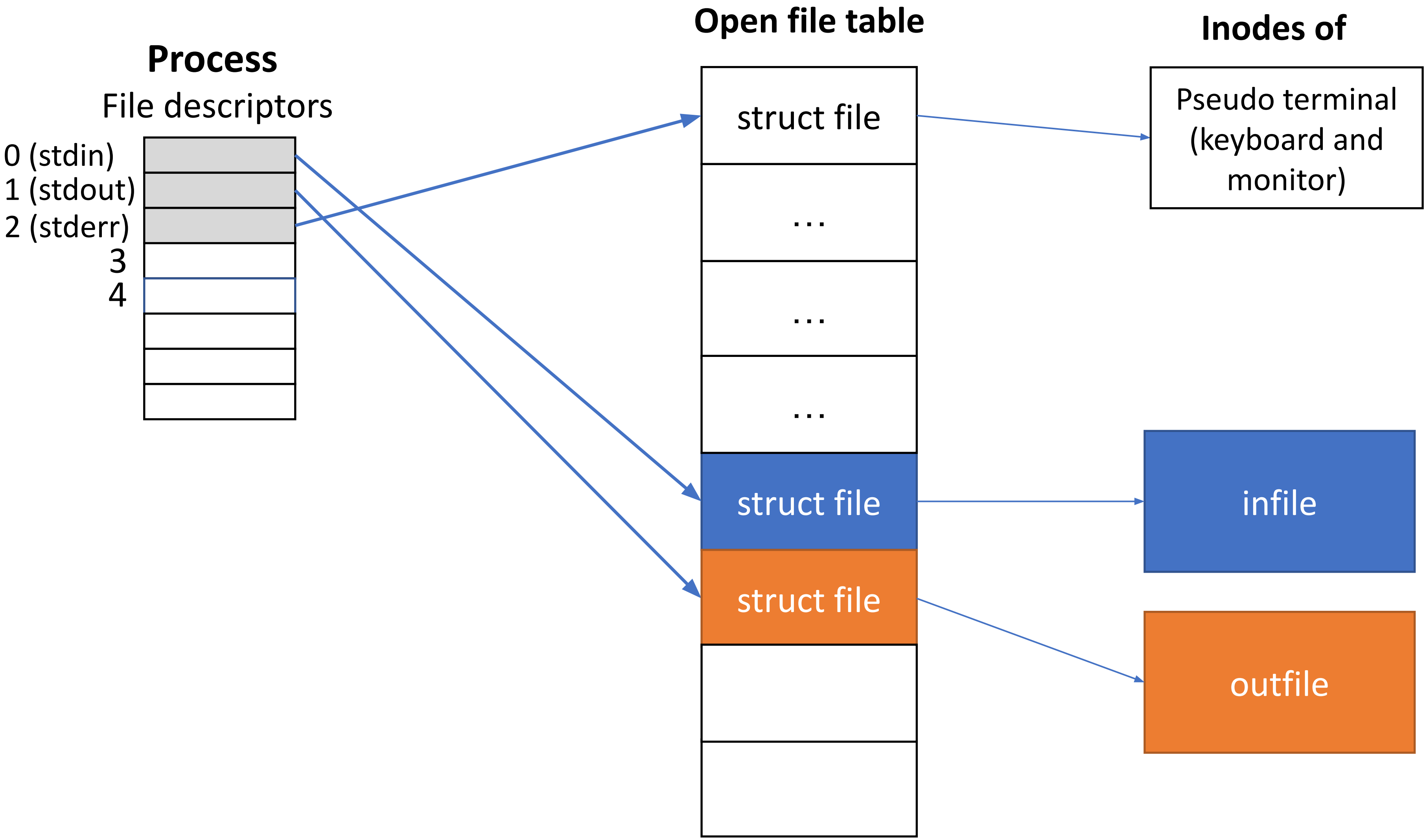
```
#  redirect a.out's stdin to read from file infile.txt:

$ ./a.out < infile.txt


#  redirect a.out's stdout to print to file outfile.txt:

$ ./a.out > outfile.txt


# redirect a.out's stdout and stderr to a file out.txt

$ ./a.out &> outfile.txt


# redirect all three to different files:

#   (< redirects stdin, 1> stdout, and 2> stderr):

  $ ./a.out < infile.txt 1> outfile.txt 2> errorfile.txt
```

https://diveintosystems.org/singlepage/#_io_in_c

Implementing redirection



Implementing redirection



Redirection demo

Pipes

- At its simplest, a pipe is a unidirectional data channel
- Typical use is to connect the ‘output’ of a process to the ‘input’ of another process
- In the shell (see right) or in a program

```
# find the number of processes
# option 1
$ ps axu > output.txt
$ wc -l output.txt
  120  output.txt
# option 2 using a pipe ‘|’
$ ps axu | wc -l
    121
```

Creating pipes in C

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

Creates a unidirectional data channel.

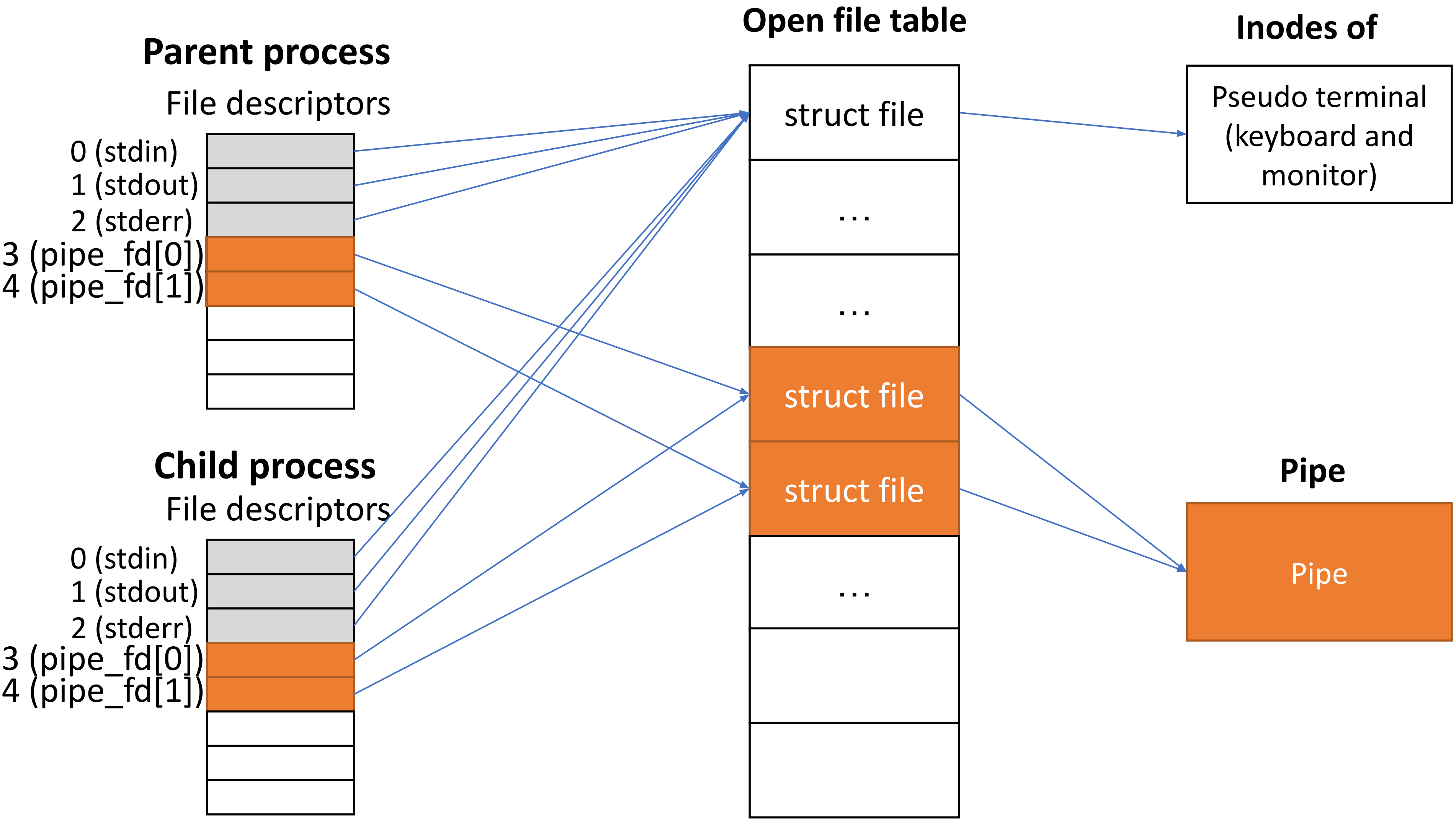
`int pipefd[2]`: contains the newly created file descriptors

- `pipefd[0]` is the 'read' end
- `pipefd[1]` is the 'write' end

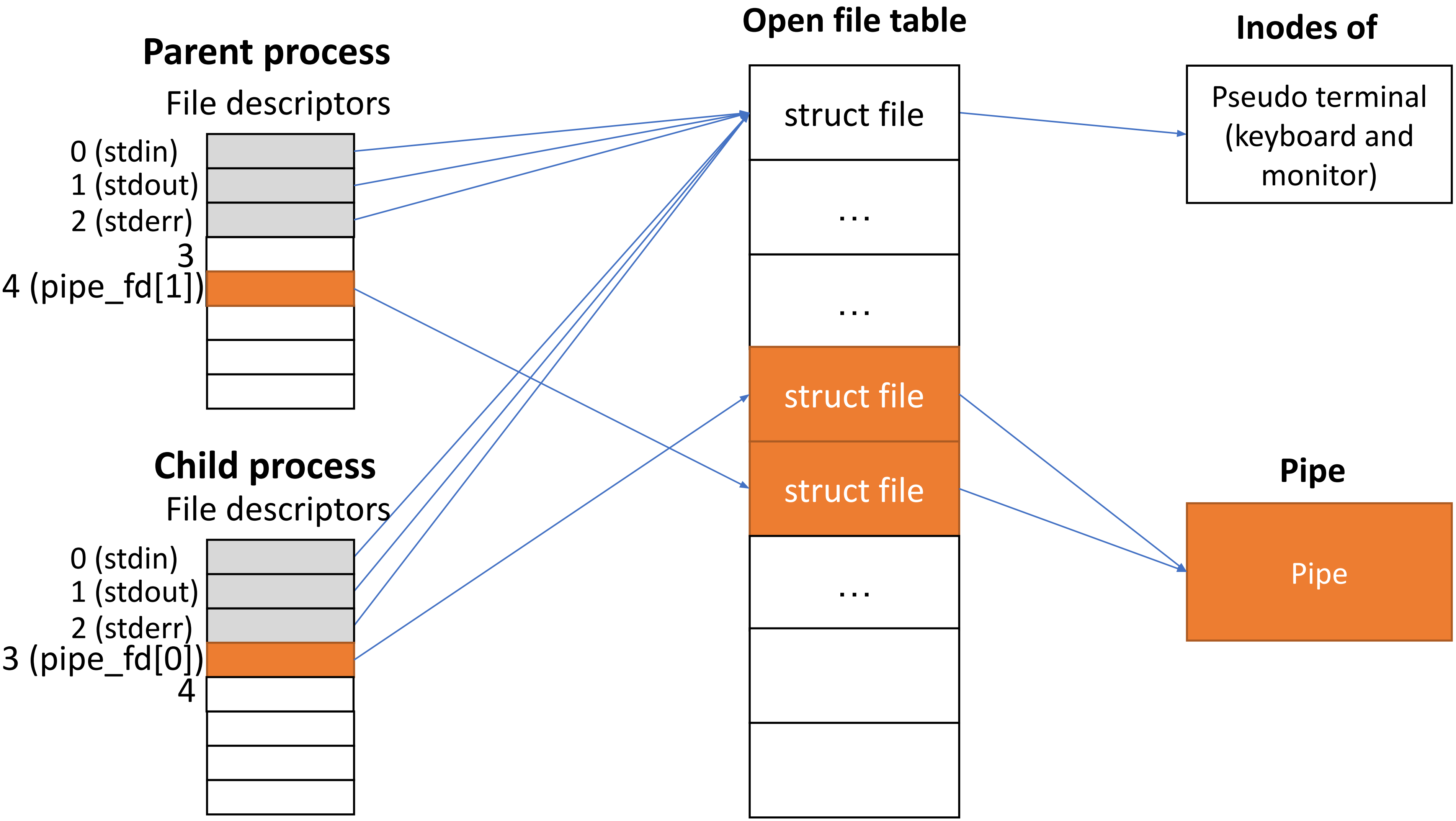
Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

Basic pipe demo

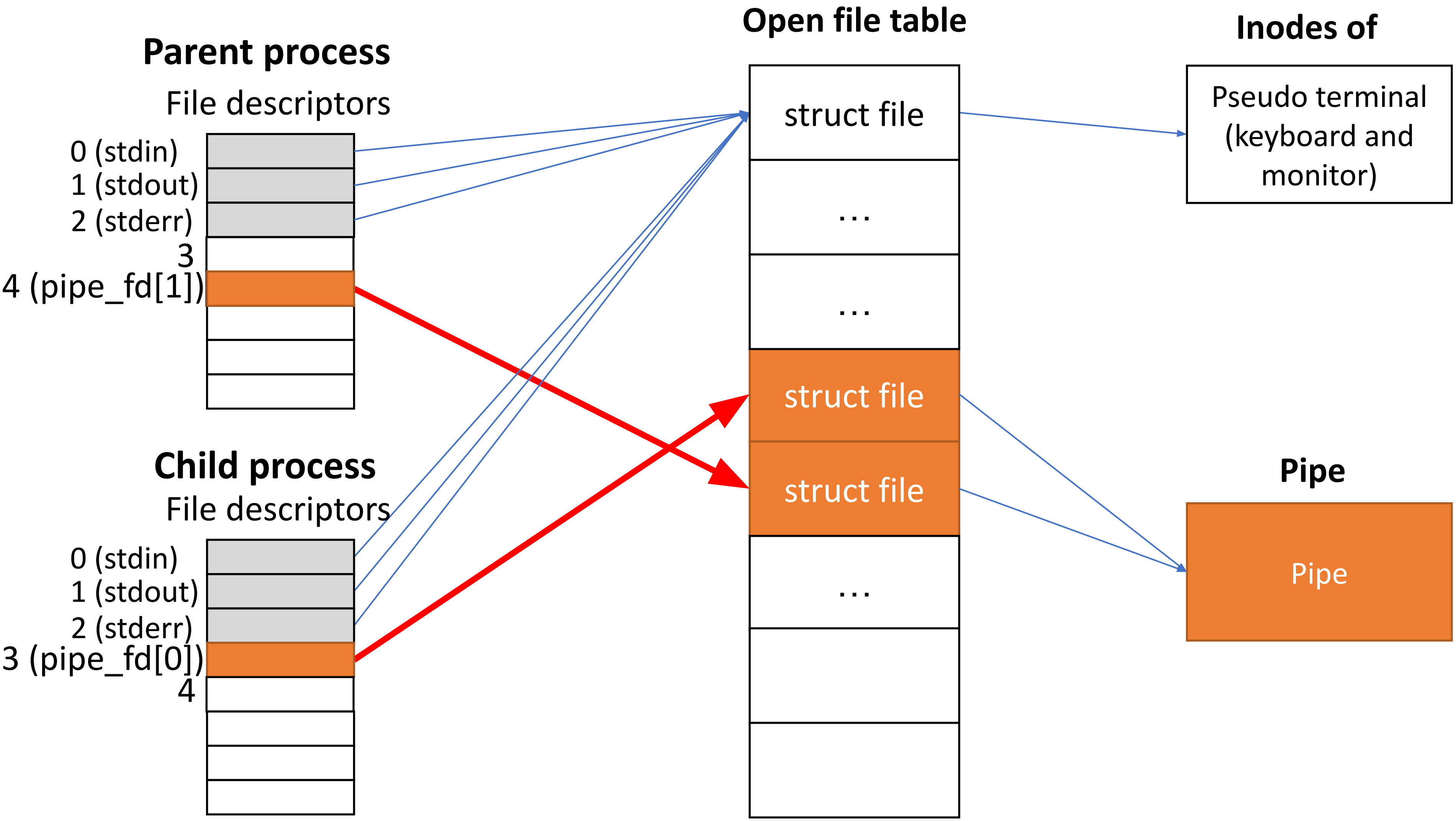
basic_pipe.c illustration



basic_pipe.c illustration



basic_pipe.c illustration



How can we relate pipe with stdin/stdout?

- We know how to create a channel/pipe between two processes

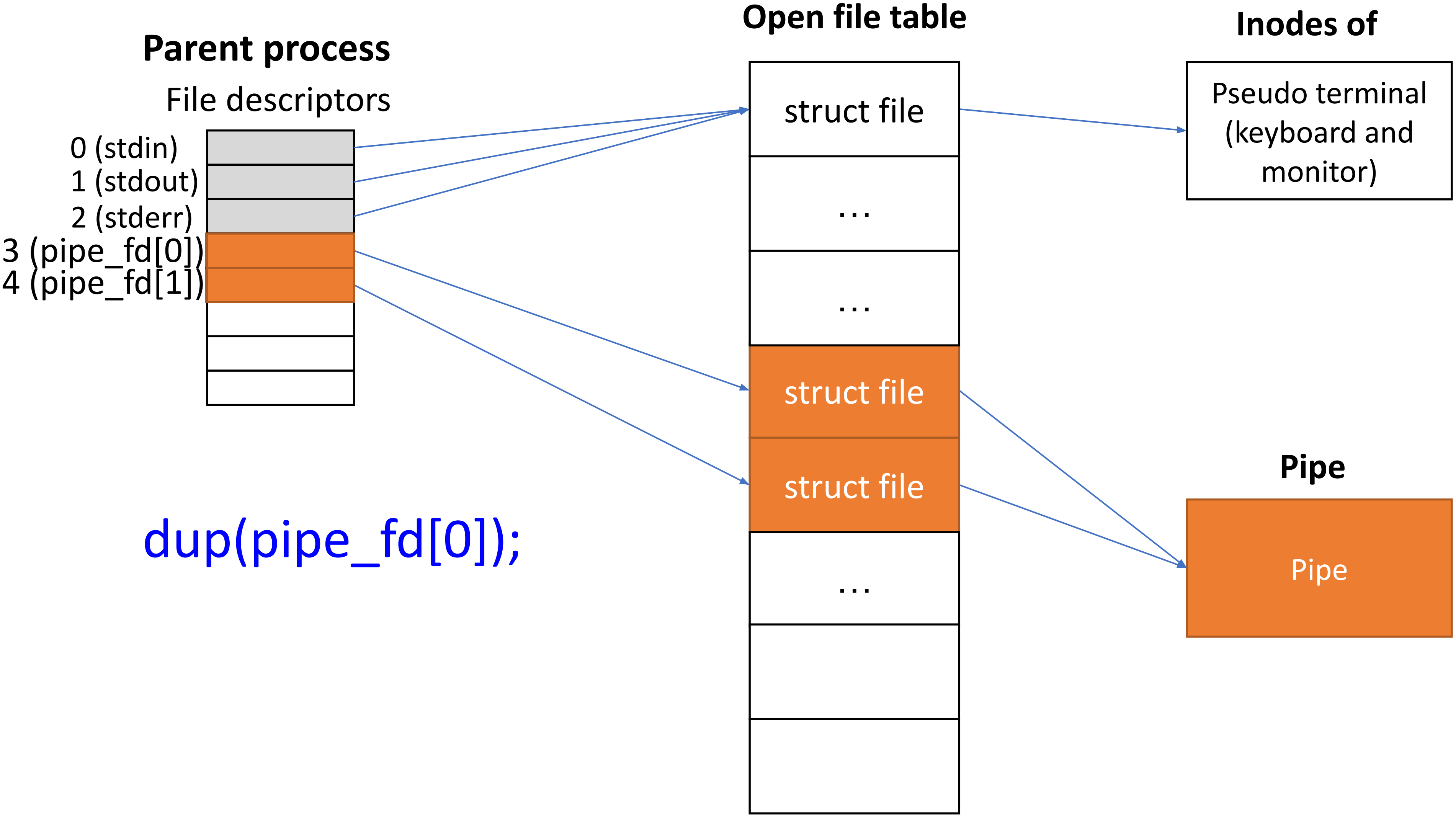
- How can we make what goes to stdout to be written to pipe[1]?
- How can we make what comes from stdin to be read from pipe[0]?

Dup

- int dup(int oldfd);
  - Creates a copy of the file descriptor
  - Assigns the copy to the lowest unassigned fd number

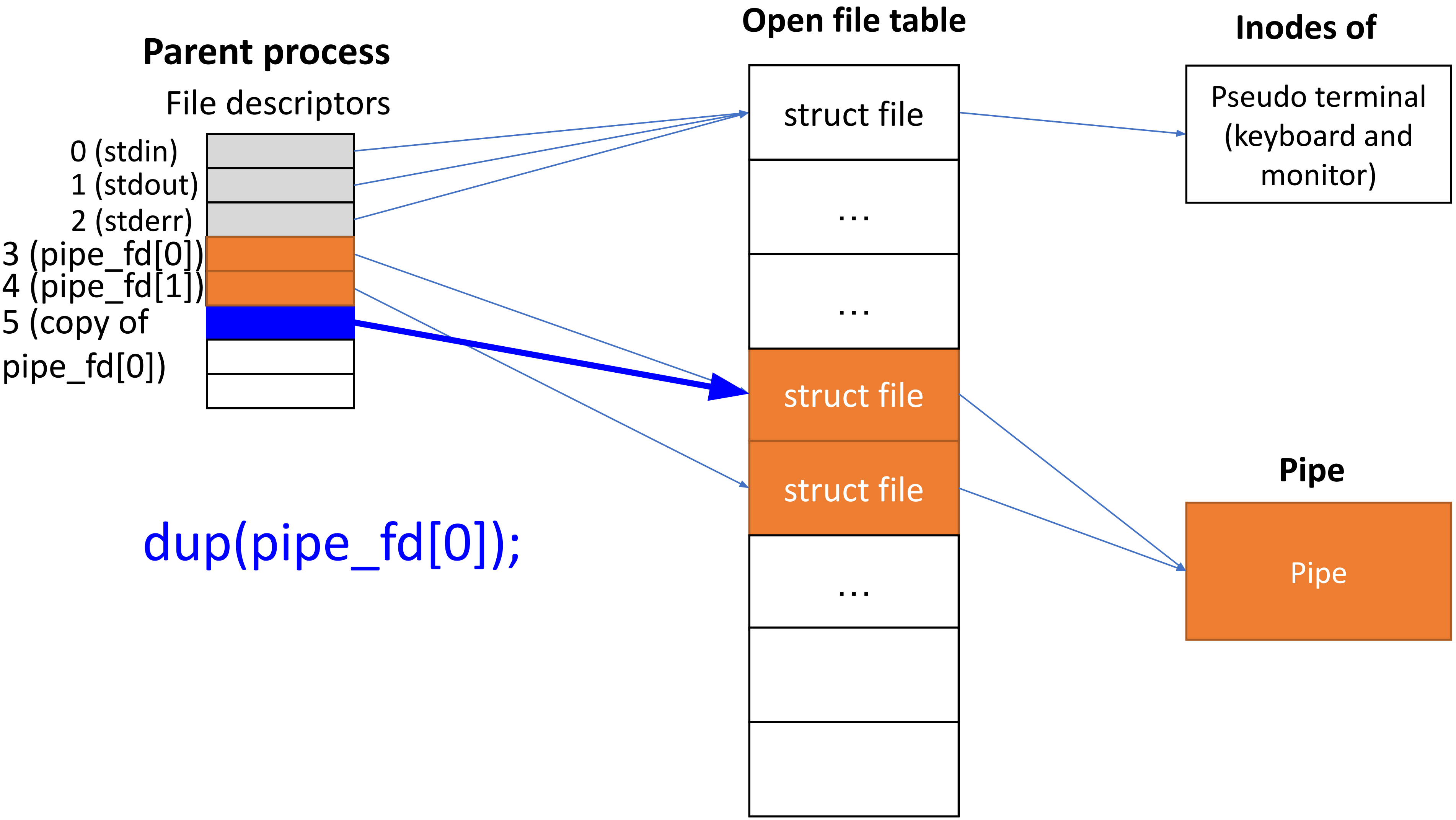
- int dup2(int oldfd, int newfd);
  - Creates a copy of the oldfd file descriptor and assigns it to newfd
  - If newfd is already open, it will silently close (need to watch out!)

Dup example



```
dup(pipe_fd[0]);
```

Dup example



Parent process

File descriptors

0 (stdin)
1 (stdout)
2 (stderr)
3 (pipe_fd[0])
4 (pipe_fd[1])
5 (copy of
pipe_fd[0])

Open file table

struct file
…
…
struct file
struct file
…

Inodes of

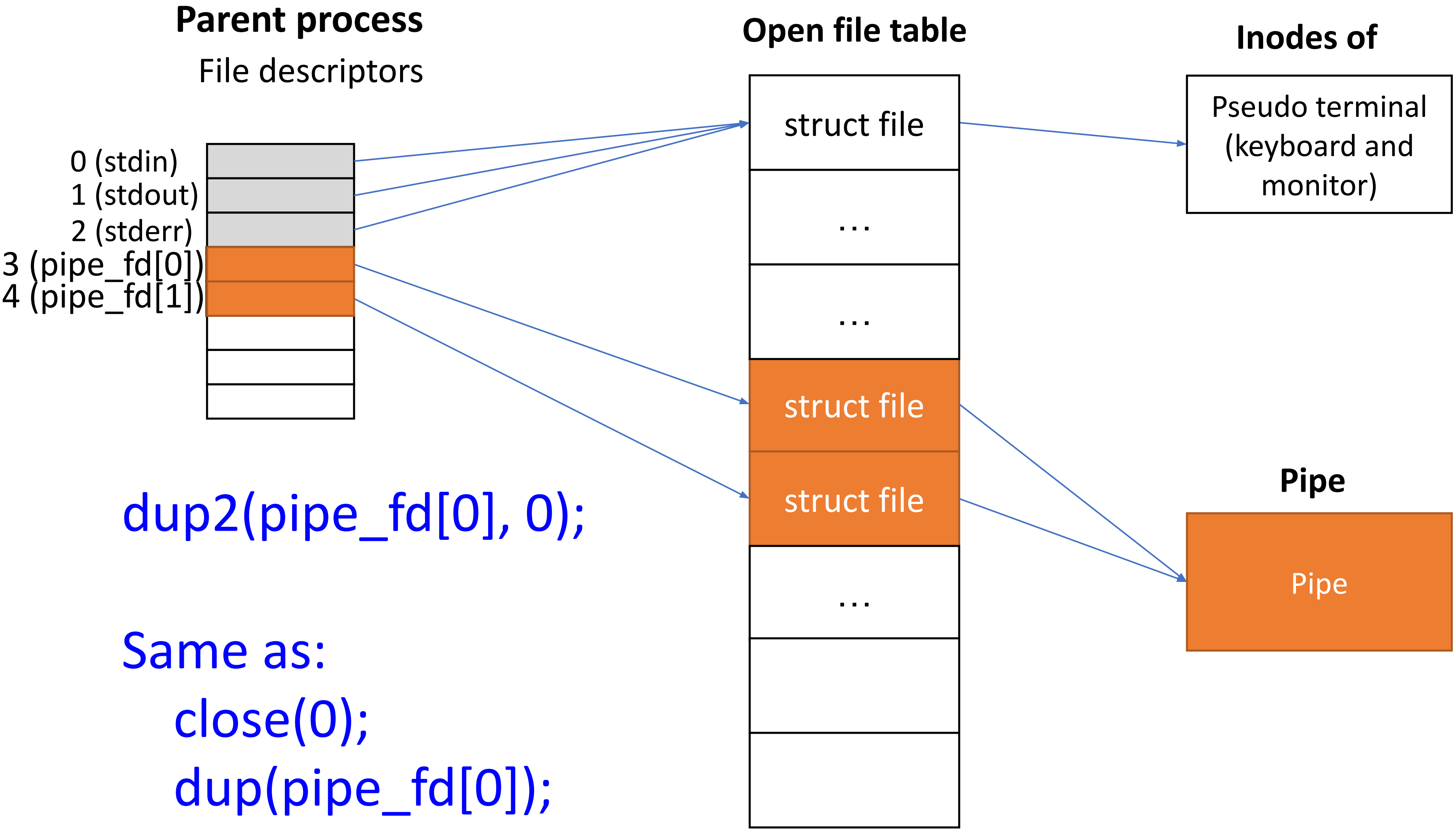
Pseudo terminal
(keyboard and
monitor)

Pipe

Pipe

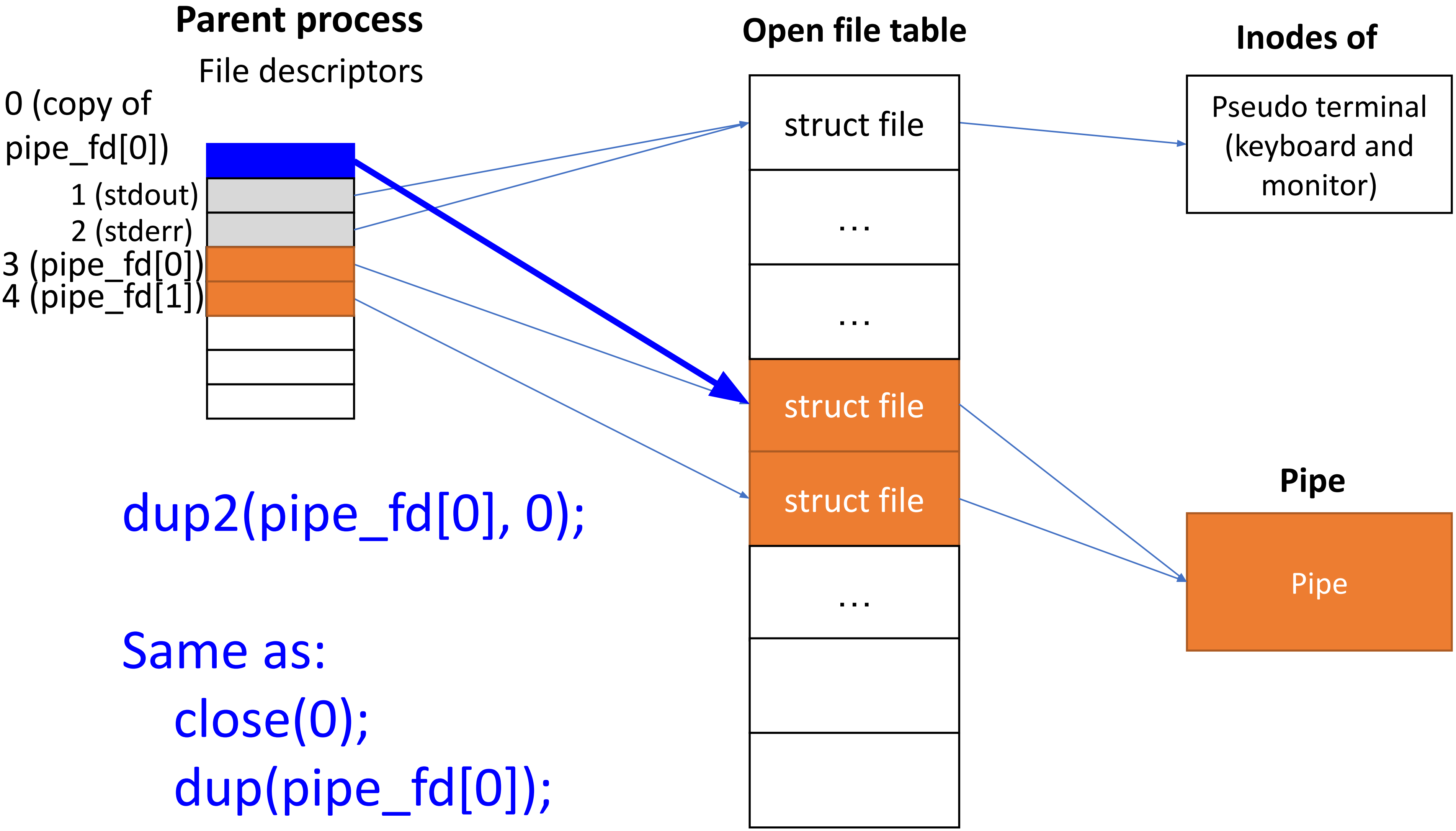
```
dup(pipe_fd[0]);
```

Dup2 example



```
dup2(pipe_fd[0], 0);

Same as:
    close(0);
    dup(pipe_fd[0]);
```

Dup2 example



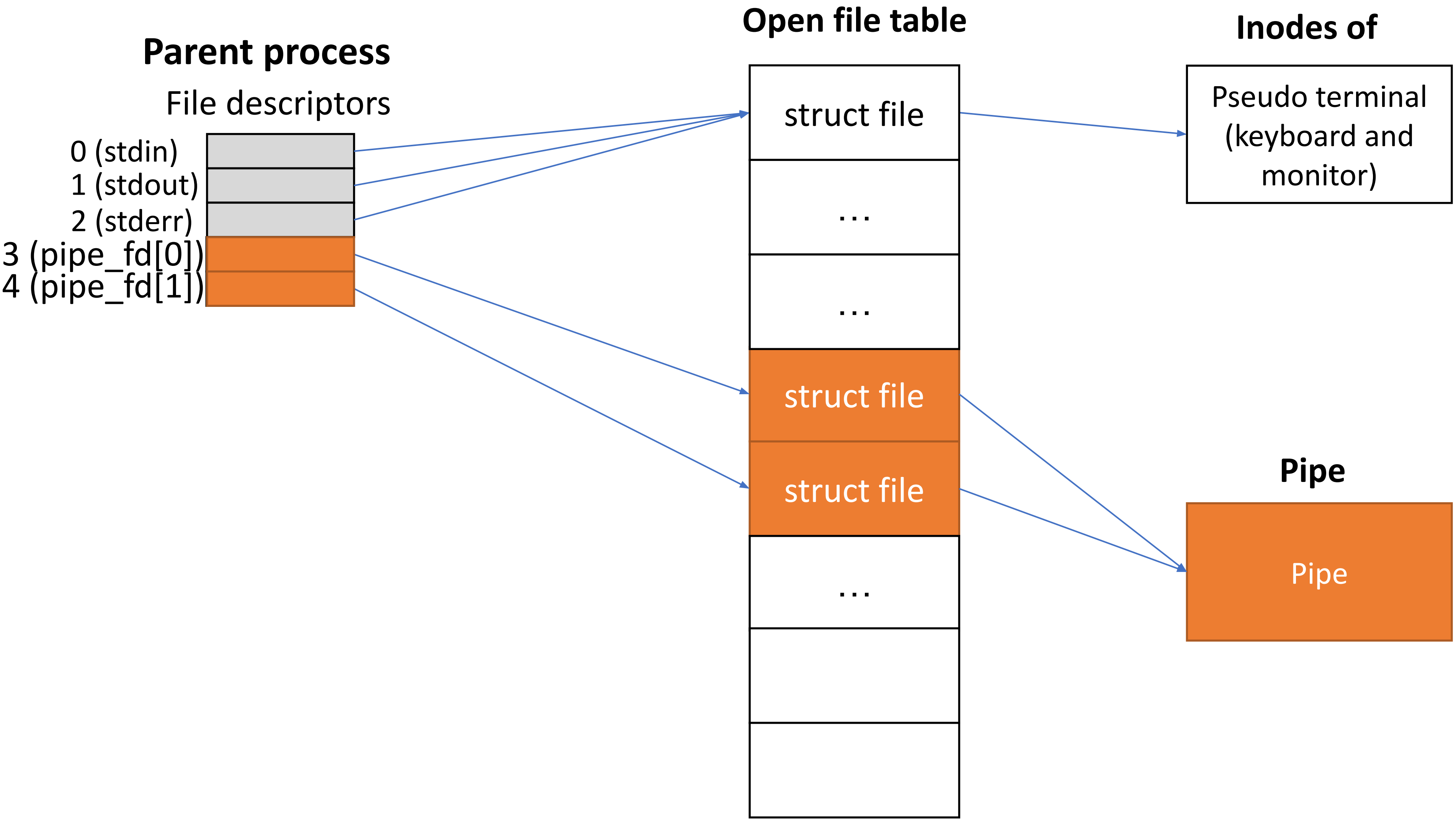
```
dup2(pipe_fd[0], 0);
```

Same as:

```
    close(0);
    dup(pipe_fd[0]);
```

Pipe.c demo

pipe.c illustration



Parent process

File descriptors

0 (stdin)
1 (stdout)
2 (stderr)
3 (pipe_fd[0])
4 (pipe_fd[1])

Open file table

struct file
…
…
struct file
struct file
…

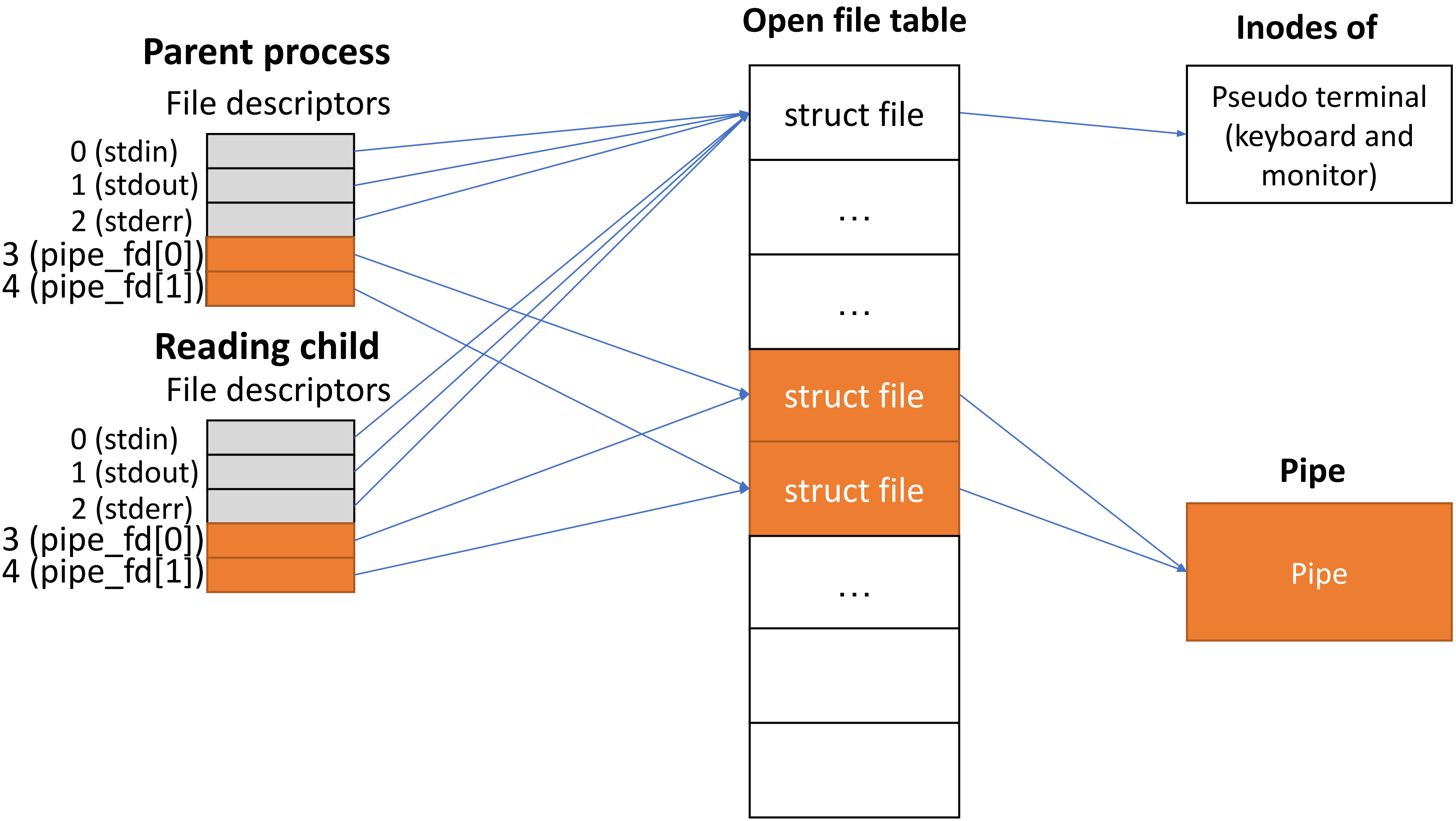
Inodes of

Pseudo terminal (keyboard and monitor)

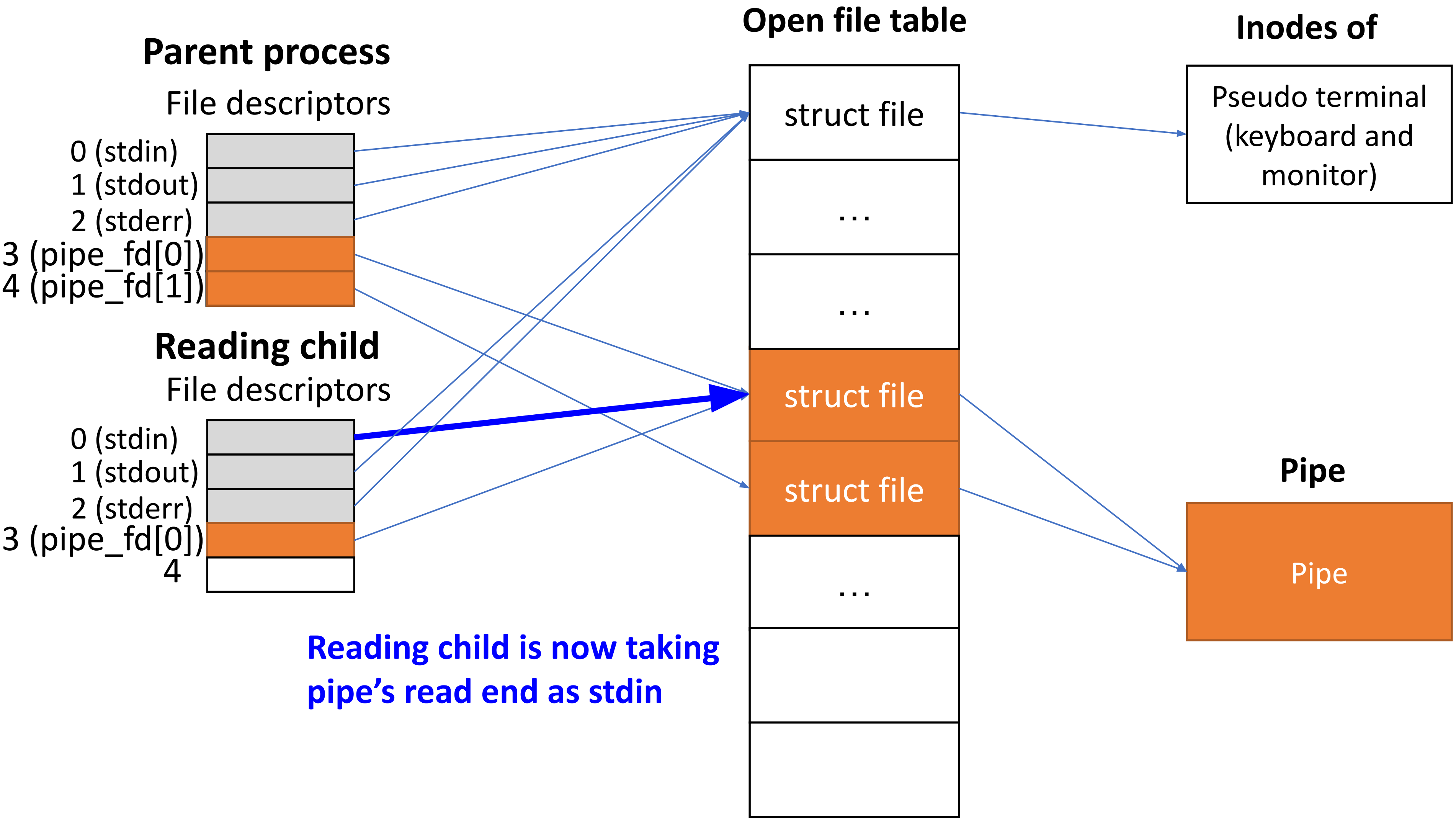
Pipe

Pipe

pipe.c illustration

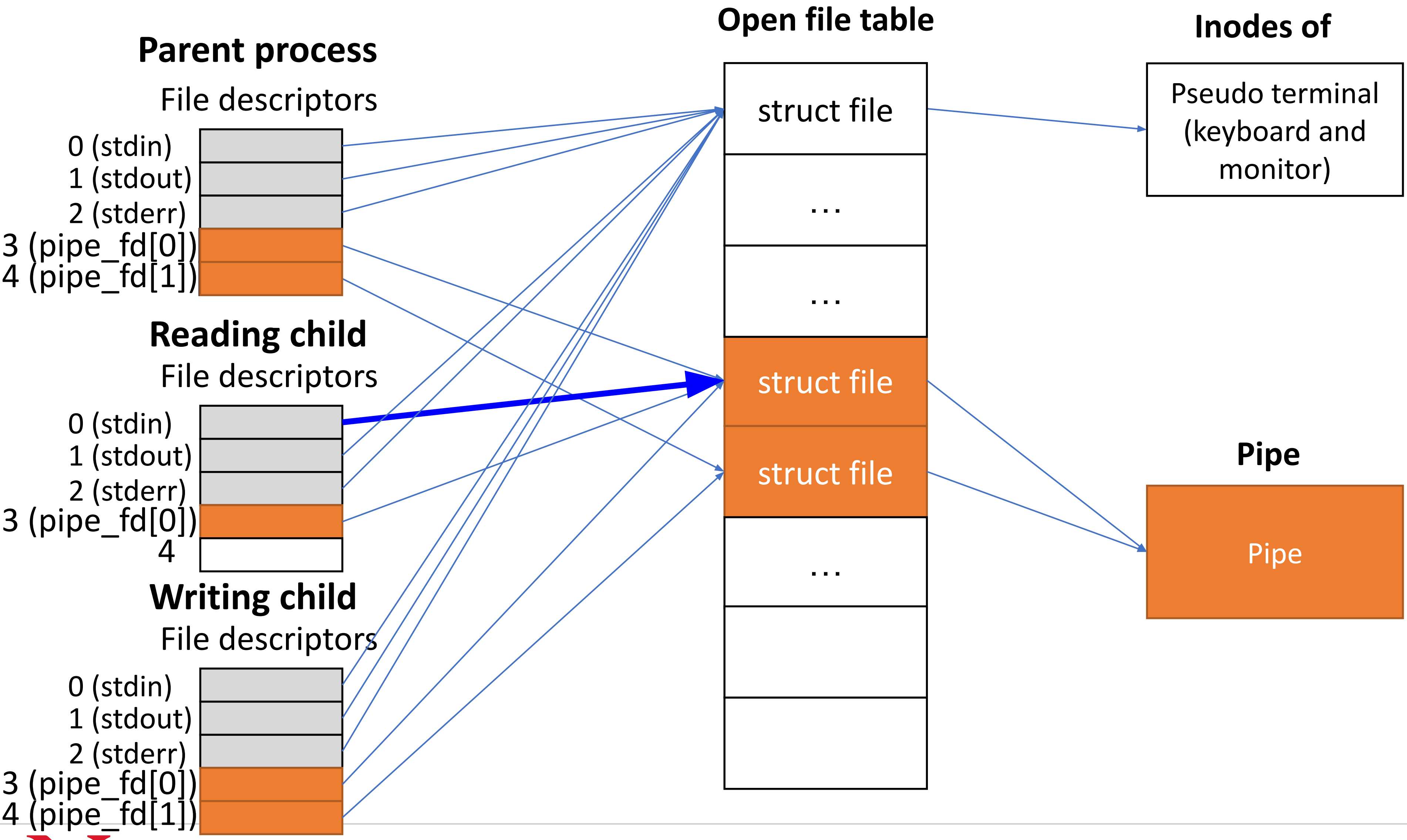


pipe.c illustration

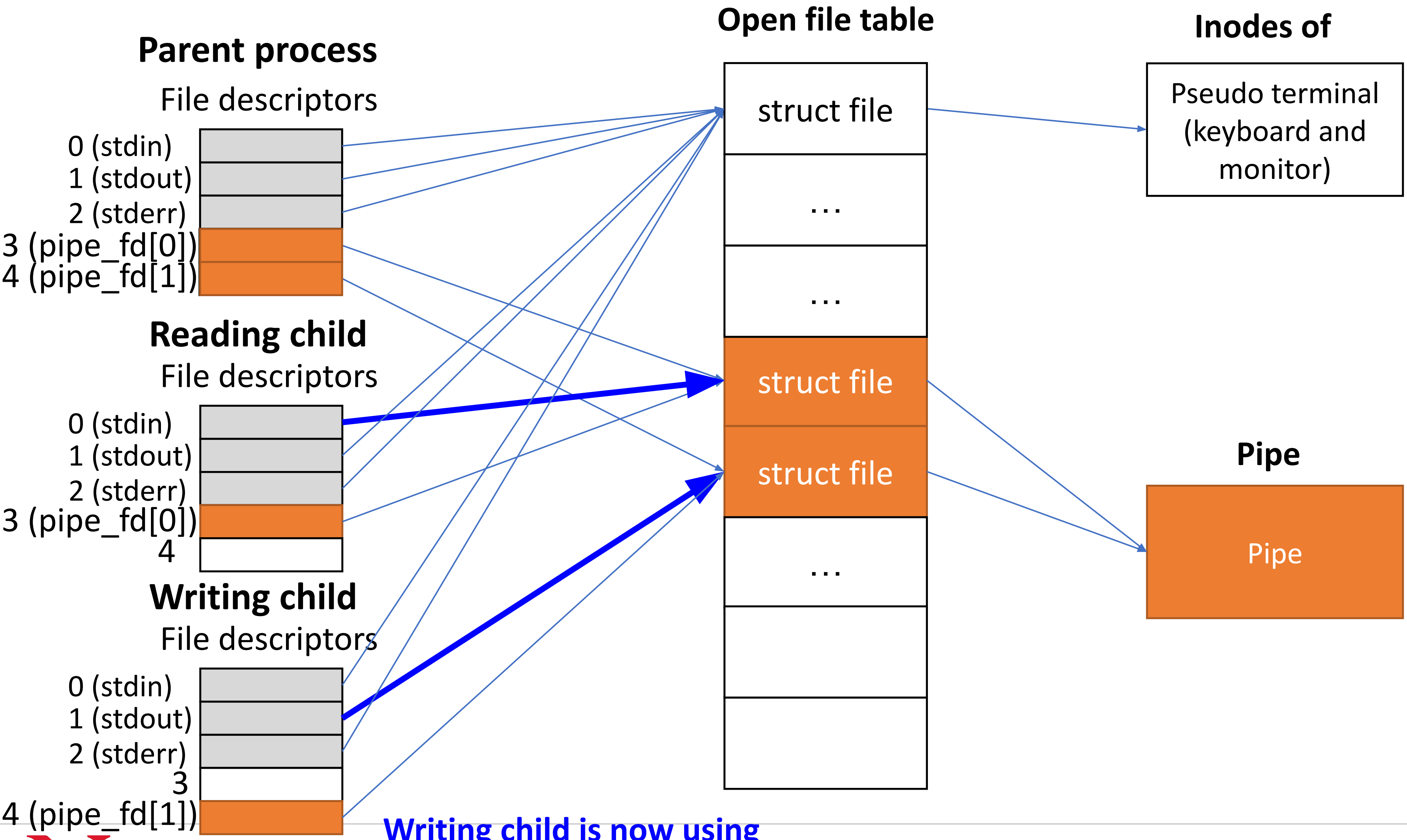


Reading child is now taking pipe’s read end as stdin

pipe.c illustration

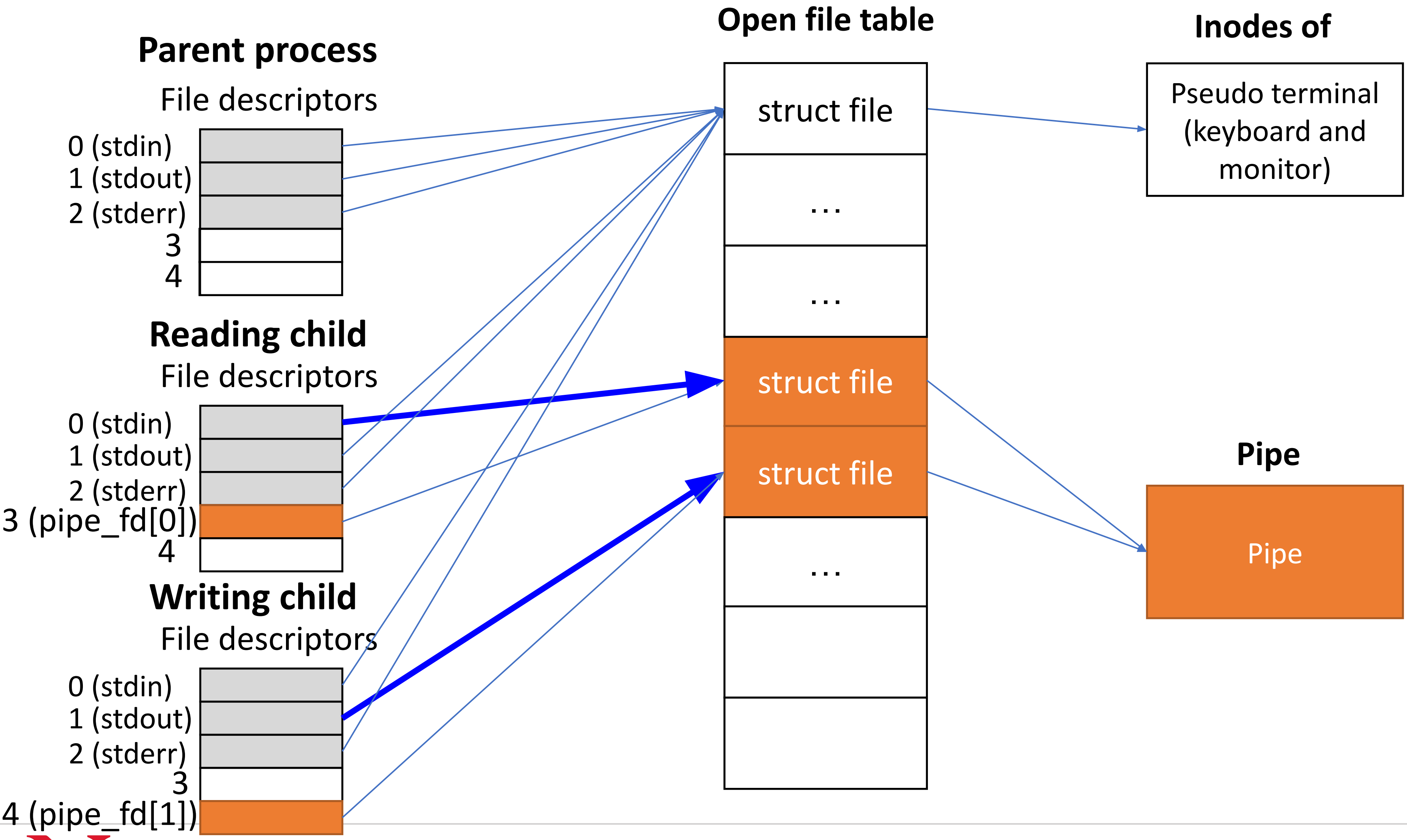


pipe.c illustration



Writing child is now using pipe's write end as stdout

pipe.c illustration



Fsync

- File system buffers writes in memory for performance
  - If power goes out writes can be lost
- Fsync() tells the file system to write data to the disk/ssd.

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
              S_IRUSR|S_IWUSR);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

Stat

- Stat returns file information

```
prompt> echo hello > file

prompt> stat file
File: ‘file’
Size: 6 Blocks: 8 IO Block: 4096 regular file
Device: 811h/2065d Inode: 67158084 Links: 1
Access: (0640/-rw-r-----) Uid: (30686/remzi)
   Gid: (30686/remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

Rename

- Renaming a file
  - mv moves or renames a file
    - mv foo bar

  - Rename function can rename the file

```
int fd = open("foo.txt.tmp",
         O_WRONLY|O_CREAT|O_TRUNC,
         S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

Link

- Hard link
  - Creating another human readable name of the file
  - Removing/unlinking one does not remove the actual file

```
prompt> echo hello > file
prompt> cat file hello
prompt> ln file file2
prompt> cat file2
hello

prompt> ls -i file file2
67158084 file
67158084 file2

prompt> rm file
removed ‘file’
prompt> cat file2
hello
```

Link

- Symbolic link
  - This is like a pointer to a file
  - Deleting/renaming the source file will create a dangling reference

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello

prompt> ls -al
drwxr-x--- 2        remzi remzi 29          May 3 19:10 ./
drwxr-x--- 27       remzi remzi 4096        May 3 15:14 ../
-rw-r----- 1        remzi remzi 6           May 3 19:10 file
lrwxrwxrwx 1        remzi remzi 4           May 3 19:10 file2 -> file

prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

Unlink

- Unlink removes/deletes a file

```
prompt> strace rm foo
...
unlink("foo") = 0
...
```