CMSC 216
Introduction to Computer Systems
Lecture 18
Process Control and System-Level I/O

Administrivia

- Start reading Chapter 10, start with Sections 10.1-10.4

Chapter 11, Bryant and O'Hallaron

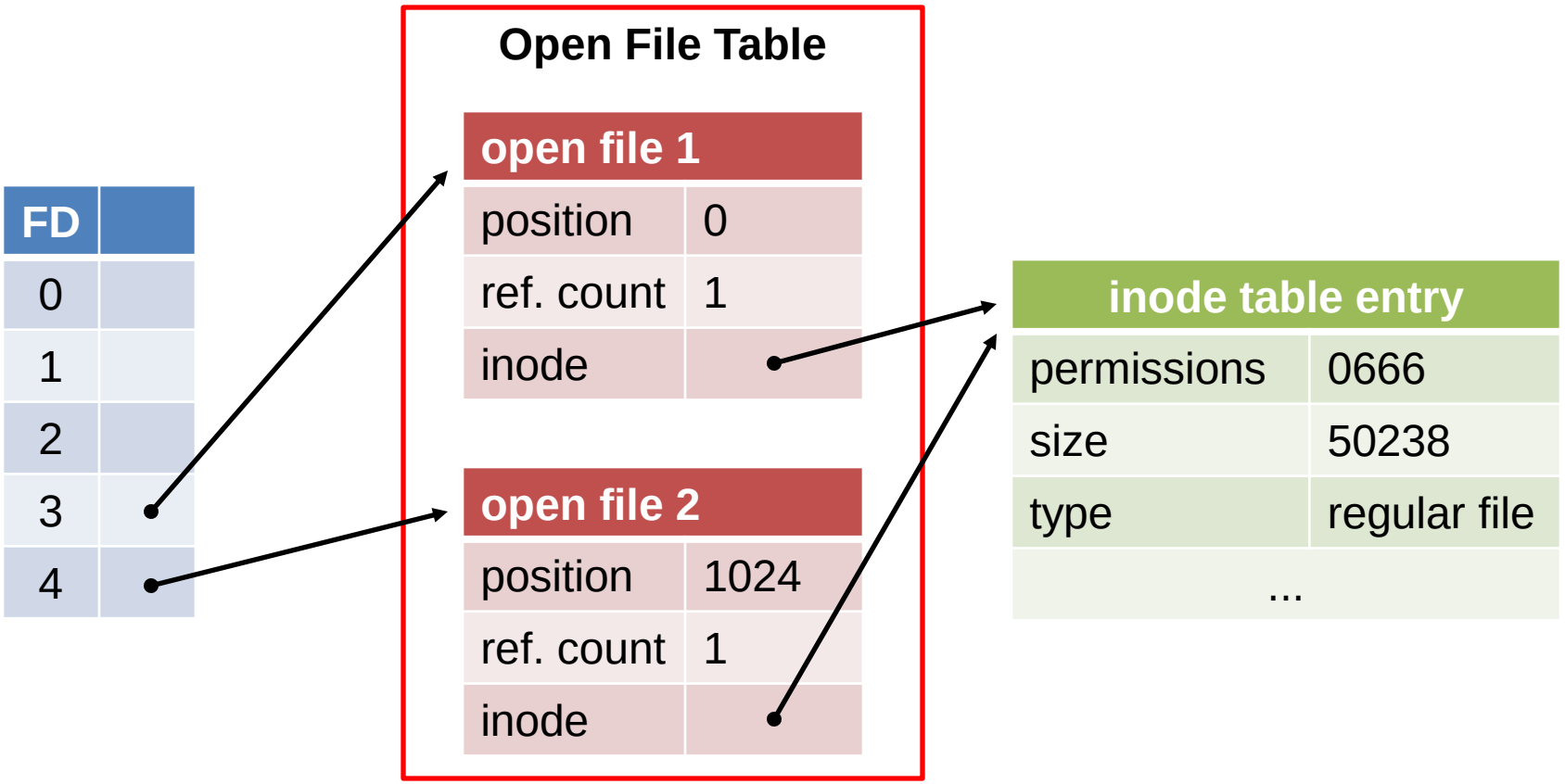
System-Level I/O (cont.)

Kernel file data structures

- Each process has its own file descriptor table, mapping integers to open files
- The kernel keeps an open file table to keep track of all open files in the system
  - shared by all processes
  - the same file can be opened multiple times
  - contains current file position, reference count (how many descriptors point to the entry), and inode pointer (among other things)
  - entries are removed from here once the reference count is 0
- The kernel also has an inode table; each inode contains information about a file
  - e.g., mode (permissions + type of file), size
  - also shared by all processes

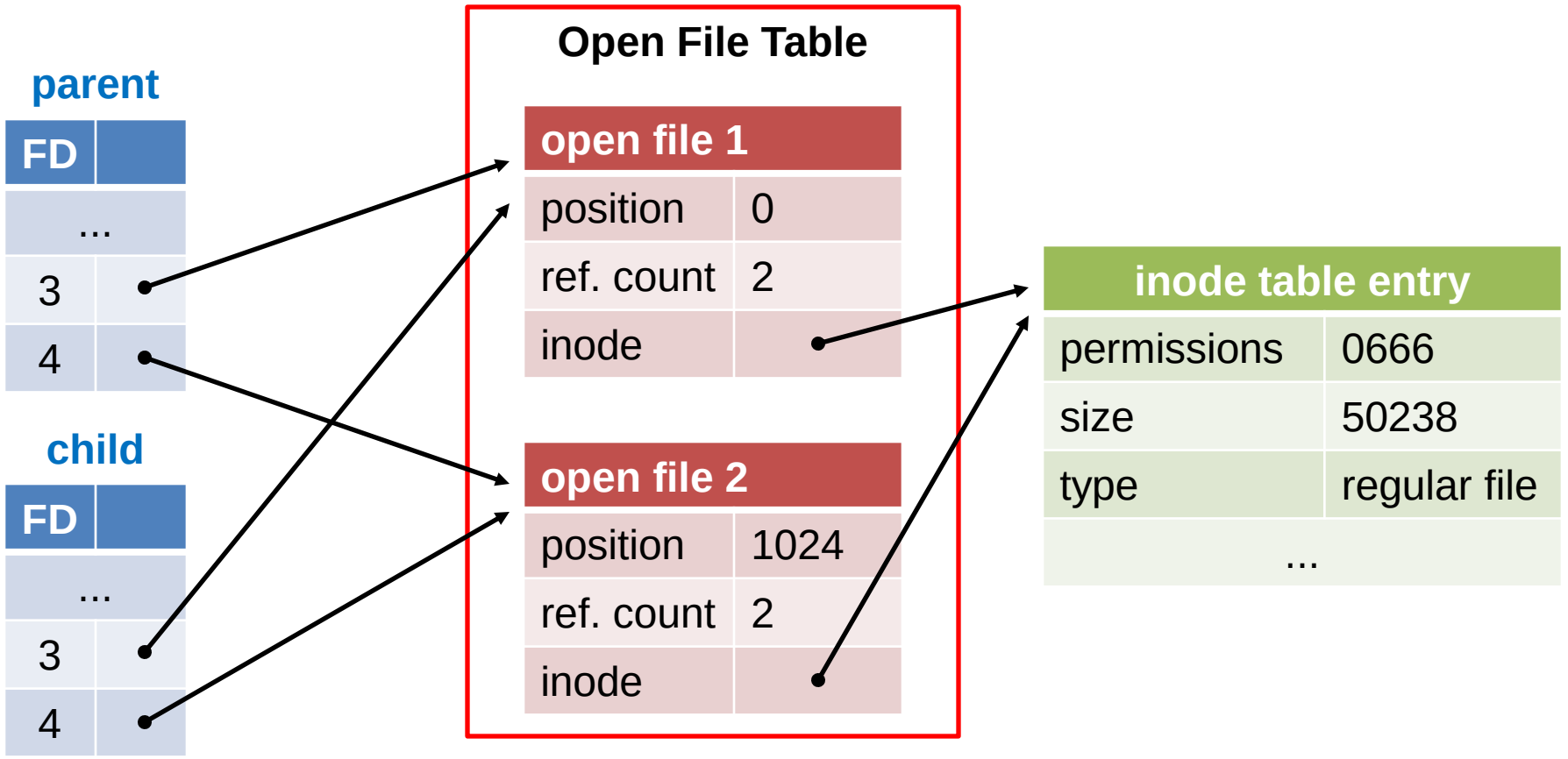
Opening the same file twice

```
fd1 = open("file.txt", O_RDONLY);
fd2 = open("file.txt", O_RDONLY);
read(fd2, buffer, 1024);
```



After a fork

```
fd1 = open("file.txt", O_RDONLY);
fd2 = open("file.txt", O_RDONLY);
read(fd2, buffer, 1024);
fork();
```



Interprocess Communication

- Processes can communicate with each other via various means, including signals and pipes
- Signals have a limited "vocabulary" - only 64 signals on some machines; other limitations as well
- Pipes allow arbitrary strings to be written from one process to another
- Two kinds of pipes: FIFOs (or "named pipes", created by **`mkfifo()`**), and anonymous pipes (often just called "pipes", created by **`pipe()`**)

Pipes

- Note: much of this information is from the pipe man page, in section 7 ("**`man 7 pipe`**")
  - the **`pipe()`** function discussed below is in section 2
- Provide a unidirectional IPC channel, with a read end and a write end
- Created with the **`pipe()`** function:

```
#include <unistd.h>
int pipe(int fd[2]);
```

  - **`fd`** needs to be an array of 2 elements
  - return value is 0 on success, -1 on error
  - On success, a new pipe is created, **`fd[0]`** will be the file descriptor for the read end, and **`fd[1]`** will be the file descriptor for the write end

Pipes, cont.

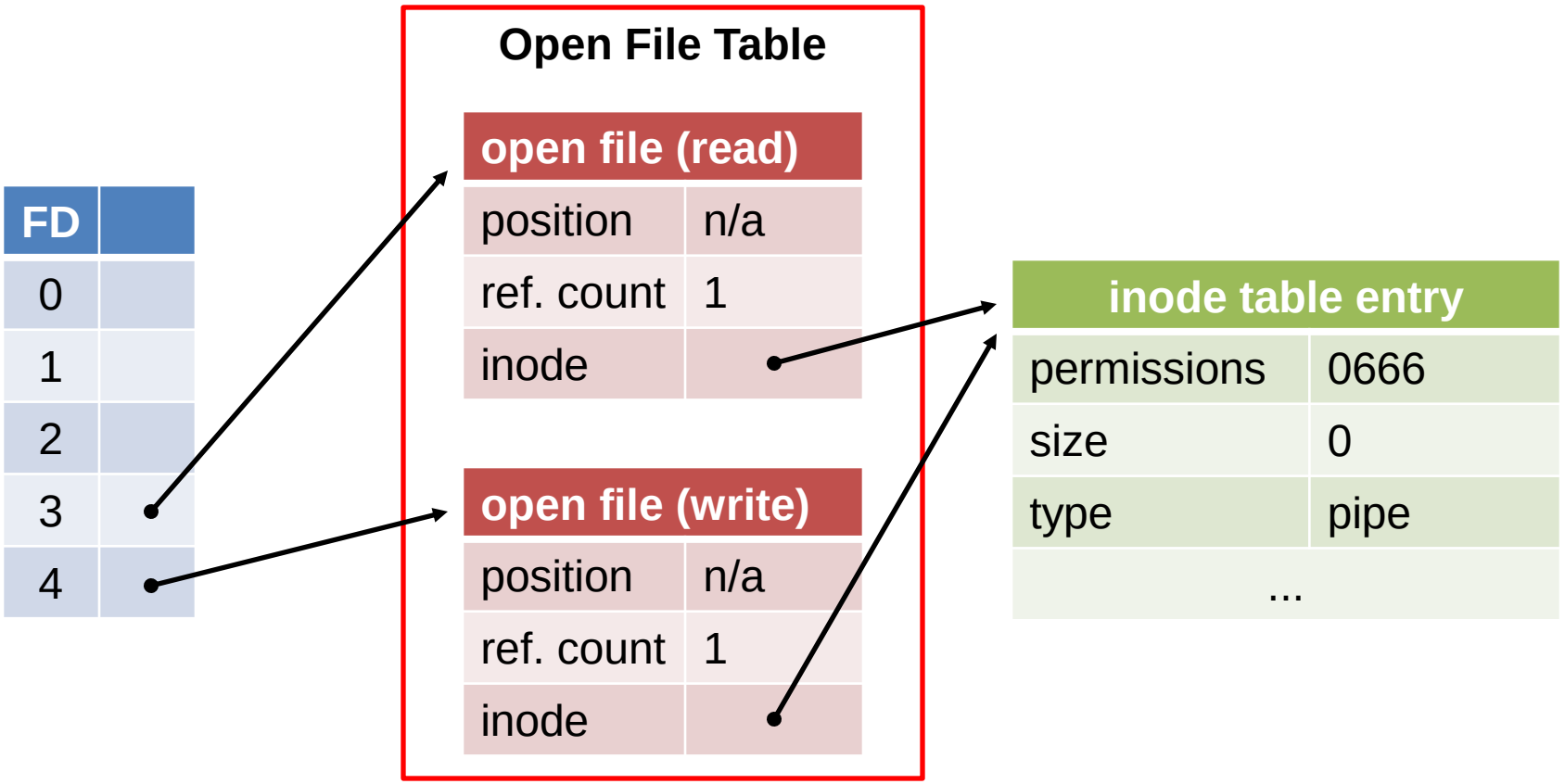
- An anonymous pipe's file descriptors are only visible in the process that created the pipe (and its children)
- I/O is blocking – read() from an empty pipe does not return until data is in the pipe, and write() to a full pipe does not return until sufficient space exists to complete the write
- Reads on the pipe will return EOF if no process has the write end open; writes to a pipe that has no readers cause an error
- Can't move the file position around (no seeking)

Pipes, cont.

- Generally, we create a pipe, then fork()
  - parent and child can then communicate via the pipe
- Because of the blocking I/O, we need to close ends of the pipe that we're not using, both immediately after the fork and once we're finished reading/writing
  - otherwise, read won't signal EOF appropriately or cause errors when we want to

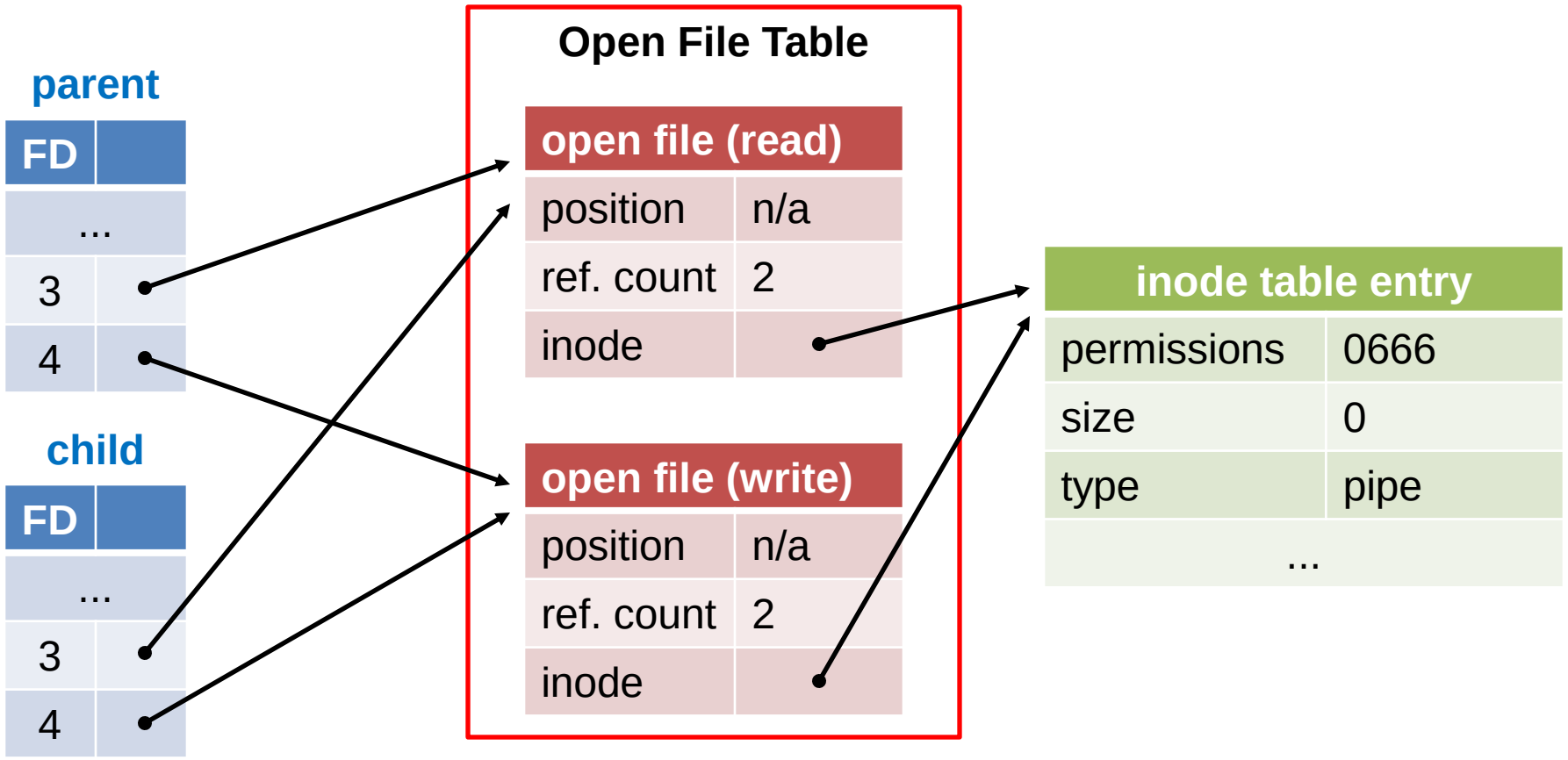
Opening a pipe

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



After a fork

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



An example with `pipe()`

```
/* #include statements and most error checking omitted */

int main() {
  int fd[2];
  char buf[BUFSIZ];  /* BUFSIZ is defined in stdio.h */

  if (pipe(fd) < 0)
    err(EX_OSERR, "pipe error");
  if (fork()) {  /* parent code */
    close(fd[1]);
    read(fd[0], buf, BUFSIZ);
    printf("Message from child: %s\n", buf);
  }
  else {  /* child code */
    char msg[] = "Hi there!";
    close(fd[0]);
    write(fd[1], msg, sizeof(msg));
  }
  return 0;
}
```

I/O redirection

- The **`dup2()`** function copies an open file table entry from one file descriptor to another

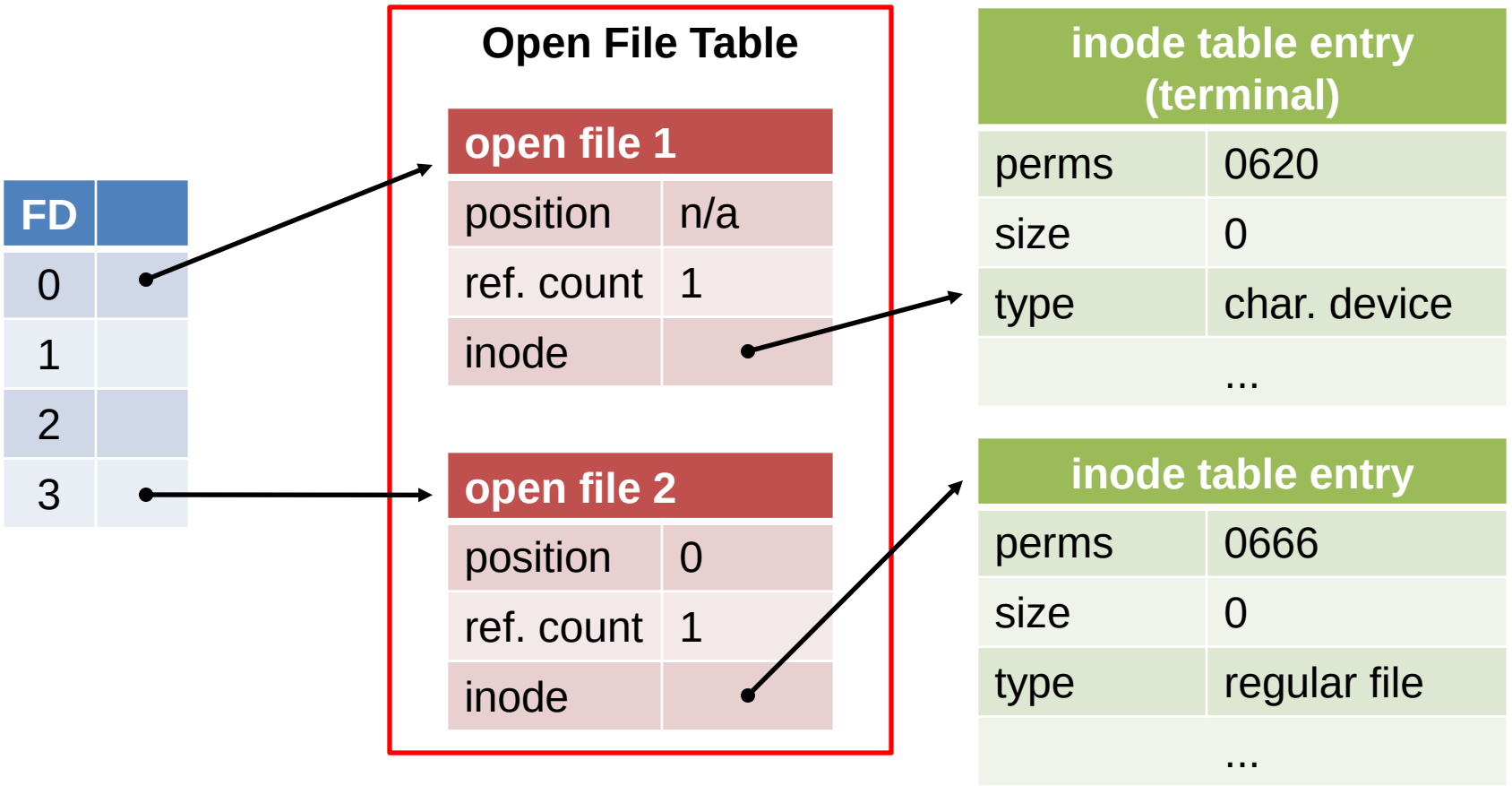
```
#include <unistd.h>

int dup2(int old_fd, int new_fd);
```

  - returns -1 on error, **`new_fd`** on success
  - **`new_fd`** is closed first if necessary (if it's already open)
- This can be used for things like I/O redirection, or to allow reading from a file named on the command line (instead of stdin)

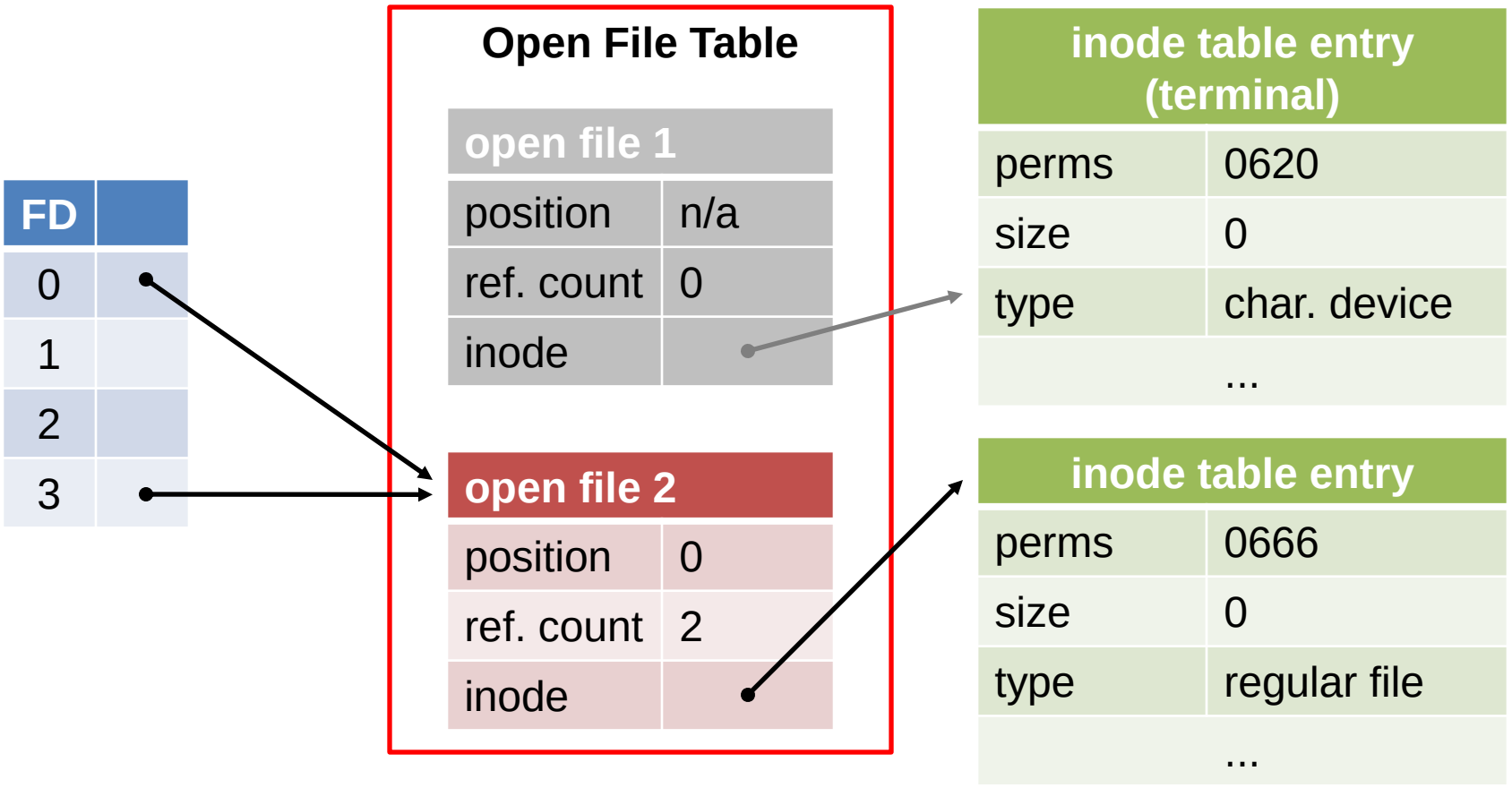
Before `dup2()`

```
int fd = open(filename, O_RDONLY);
```



After dup2()

```
int fd = open(filename, O_RDONLY);
dup2(fd, STDIN_FILENO);
```



Example using **dup2()**

- Allow reading from stdin or named file:

```
int main(int argc, char **argv) {
   ...
   if (argc > 1) {
     int fd = open(argv[1], O_RDONLY);
     dup2(fd, STDIN_FILENO);
     close(fd);
   }
   /* read from stdin from here on */
   ...
}
```

Section 16.3, Reek

Time Measurement

Time

- On a computer there are many ways to measure time
- Conceptual difference:
  - *wall time* is always running
  - *process time* is the time your program was running
    - process time doesn't count:
      - the time when your program was stopped for others
      - the time when your program stopped to wait for I/O
    - is composed of:
      - user time: when the OS is running your code
      - system time: when the OS is running system code (handling system calls)
- Difference in how to measure
  - interval time
  - clock cycles
- What time to use depends on what you are measuring

Timing a program

- To time an entire program in the shell:
  - **`time`** *program-name program-arguments*
  - runs the program and prints its execution time in the form (tcsh)

    `2.230u 0.260s 0:06.52 38.1% 0+0k 0+0io 80pf+0w`
    - the first two numbers are user and system time
    - the third number is wall time
    - the fourth is percentage of wall time: (user+system)/wall
    - the remainder are paging and I/O statistics
- This provides some idea about timing
  - but it's hard to know what to do about it - what functions are taking most of the time?

Representing time-of-day

- How to deal with
  - time zones
  - daylight saving time rules
  - computers moving between time zones
  - leap seconds?
- Answer:
  - UNIX keeps time internally as seconds and fractions of a second
    - 2^{32} bit values work nicely for 1ns accuracy for > 100 years
  - use a reference starting point
    - called the epoch - midnight Jan. 1, 1970
  - keep all time in UTC form until printed
    - no time zones or daylight saving time to deal with

Date and time functions

- There are several functions in **<time.h>** for working with times.
  - most use a type **time_t** that contains an encoded representation of a time
  - several use the following **tm** structure defined in **<time.h>** that has fields that can be extracted from a **time_t** variable:

```
struct tm {
  int tm_sec;   /* seconds */
  int tm_min;   /* minutes */
  int tm_hour;  /* hours */
  int tm_mday;  /* day of the month */
  int tm_mon;   /* month */
  int tm_year;  /* year */
  int tm_wday;  /* day of the week */
  int tm_yday;  /* day in the year */
  int tm_isdst; /* daylight saving time */
};
```