CPSC313: Computer Hardware and Operating Systems

Unit 4: File Systems
Fun with File Descriptors

Admin

- Final examination
  - Reserve your time of PrairieTest if you haven't already done it.
- Tutorial 8 is this week
- Lab 8 has been released and is due Sunday November 17th.
- Code for today is in the course code repository: 4.2-fun-with-file-descriptors

Where we are

- Unit Map:
  - P18: File Systems APIs and How disks work
  - 4.1. Using File Systems APIs
  - P19: File descriptors
  - 4.2. File descriptors management
  - P20: File Systems implementation overview
  - 4.3. How we represent files

Today

- Learning Outcomes:
  - Review of file descriptor management data structures.
  - Redirect standard in, standard out, and standard error from within a program.
  - Describe how the shell implements redirection to implement commands that do the following:
    - ./cmd > something
    - ./cmd >& something
    - ./cmd < something
    - ./cmd >> something
  - Map redirection into operations on file descriptors, the file descriptor table, open file structures and vnodes.

Questions about how a file system should behave (all answered in 1969)*

- If you and I are both allowed to read a file in the file system, should we be able to share the file's data? Yes!
- If two processes are reading (writing) the same file, should they be using the same file offset? No!
- If a process opens the same file twice, should it have one file descriptor or two? Two! In either case, should the two opens share an offset? No!

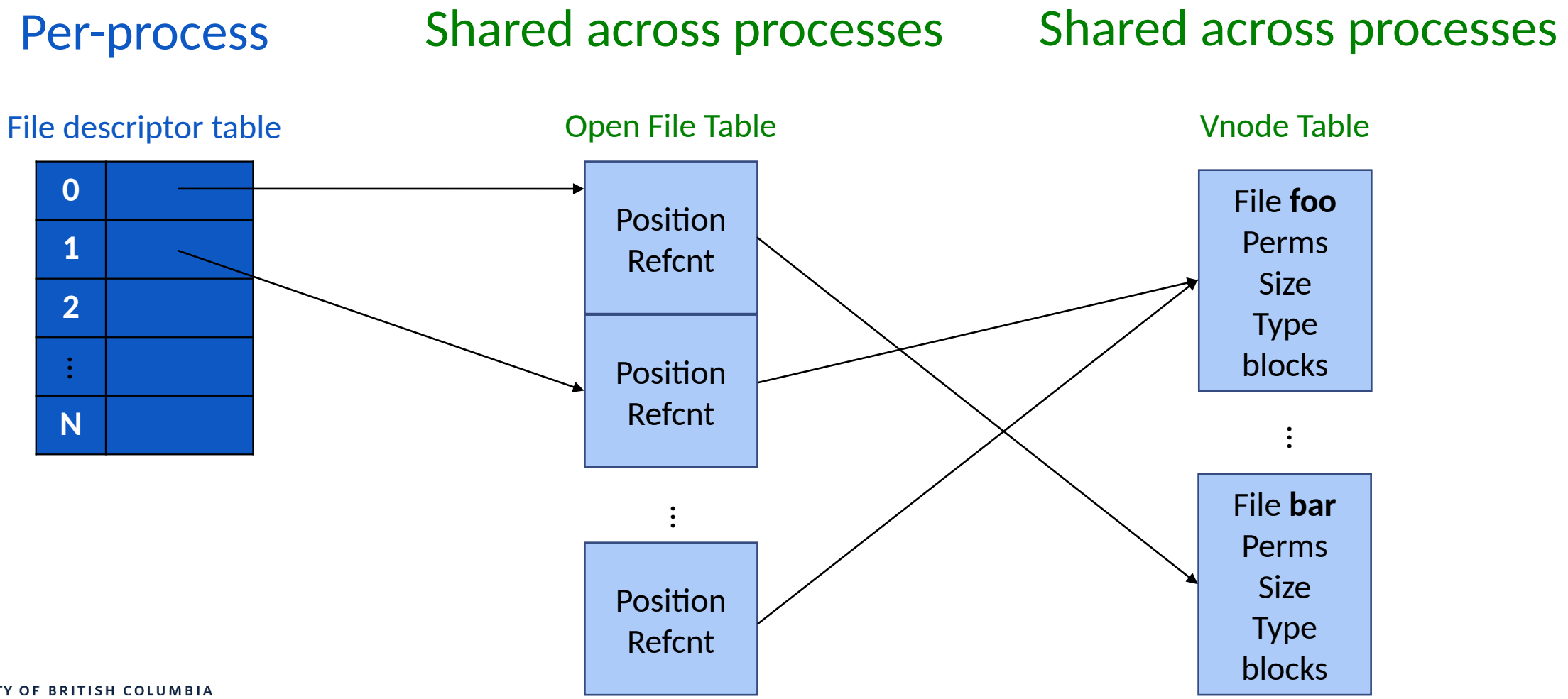
*Except the one with threads

Questions about how a file system should behave (all answered in 1969)*

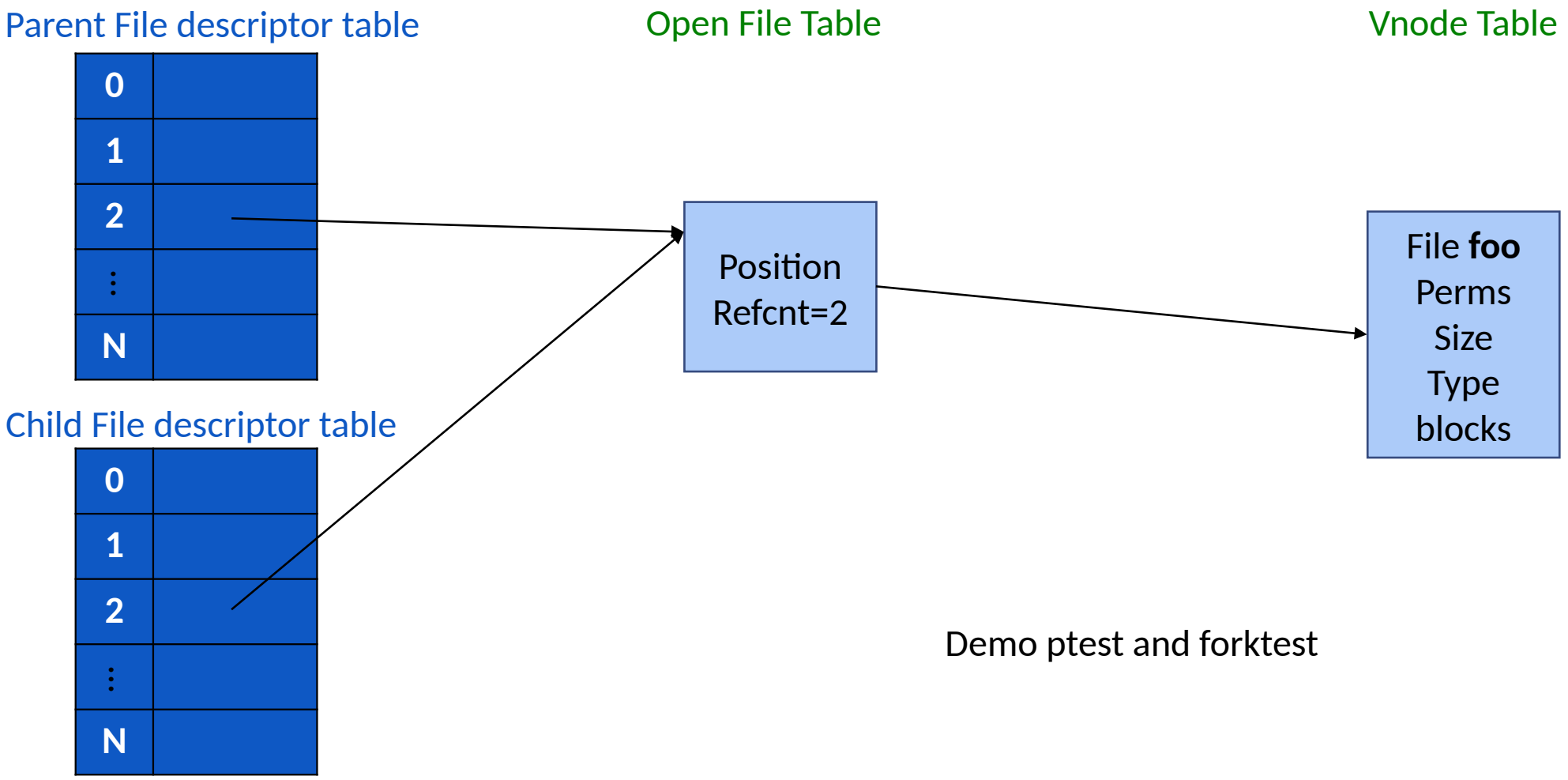
- If two threads are using the same file descriptor, should they be using the same file offset? Yes!
- If one process (the parent) creates another process (a child), should the child inherit the parent's file descriptors? Will they use the same offset? Yes! Yes!

*Except the one with threads

Putting this all together



Recall: Parent and Child have the same FDs



Demo ptest and forktest

Fork: `pid_t fork(void)`

- Creates a new process almost identical to the process that called it.
  - The parent's return value will be the pid of the child process.
  - The child process's return value will be 0.
- Immediately after the fork call, both processes should check their return values and proceed accordingly.
  - The child process often calls a member of the exec family.
  - The parent might wait on the child or do something else.

Exec and friends

- `int execve(const char *path, char *const argv[], char *const envp[])`
  - Typically `path` is the pathname of a command you want to execute, e.g., `./myprog`, `/bin/ls`
  - `argv` is an argument vector -- it is what is passed to main, e.g.,

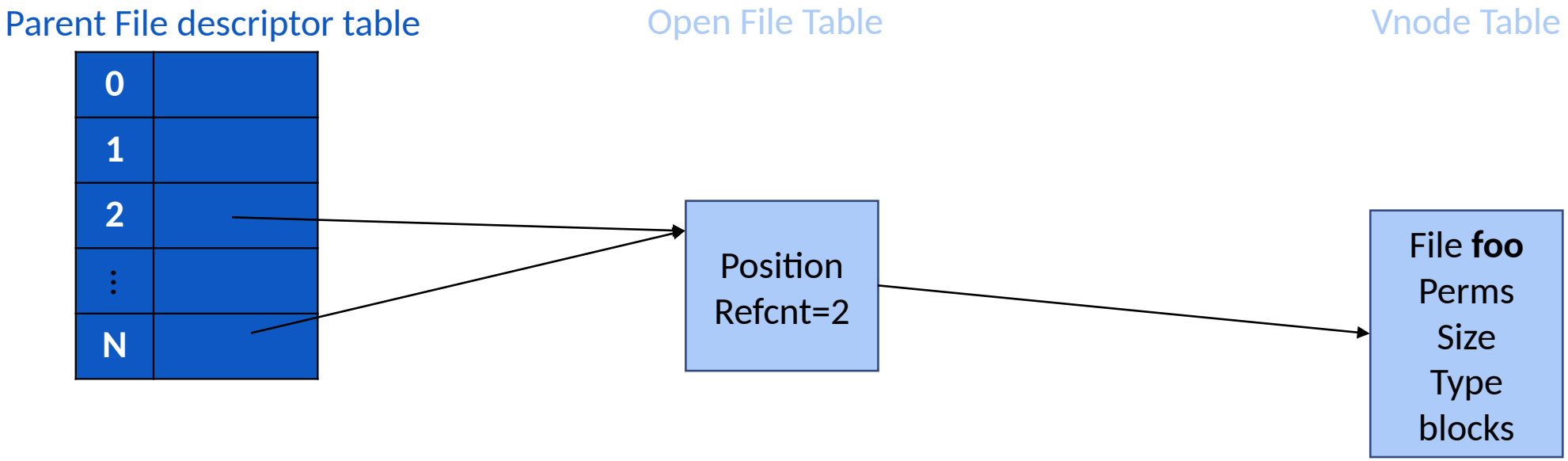
    ```
    int main(int argc, char *argv[])
    ```

  - `envp` is an environment: the environment is a set of name/value pairs that are frequently used to communicate 'environmental' information to processes: where should the process look for commands, what OS are we running, etc.

Exec and friends

- `int execvp(const char *path, char *const argv[])`
  - If the parameter path does not begin with “/”, or “.”, or “..”, then execvp searches for parameter path in each directory listed in the PATH environment variable (just like the shell does).

Recall: Another way to get a refcount of 2



The dup (dup2) system call(s) "duplicates a file descriptor"

```
int dup(int fd);
int dup2(int fd1, int fd2);
```

Demo fdtest and duptest

When might you use dup?

Modes are traditionally written as OCTAL (base 8) digits to correspond to the 3 bits for each of owner, group, and world. A leading 0 (zero) means the number is octal

- Redirecting standard out:
  - When you want to capture a program's output, you can use:

```
ls > my-stdout-file.txt
```

- How does this work?
  - The ugly brute force way (building into a program):

```
int fd = open("my-stdout-file.txt", O_WRONLY|
O_CREATE|O_TRUNC, 0644);
dup2(fd, STDOUT_FILENO);  // Closes old standard out
```

When might you use dup?

- Fun fact about dup:
  - If you dup into an `fd` that is open (e.g., `STDOUT_FILENO`), this automatically closes the file that had been using that `fd`.
  - Even though standard IO (e.g., `printf`) is not a system call, when we change where stdout writes, it also changes where `printf` output goes.

The shell implements redirection for you

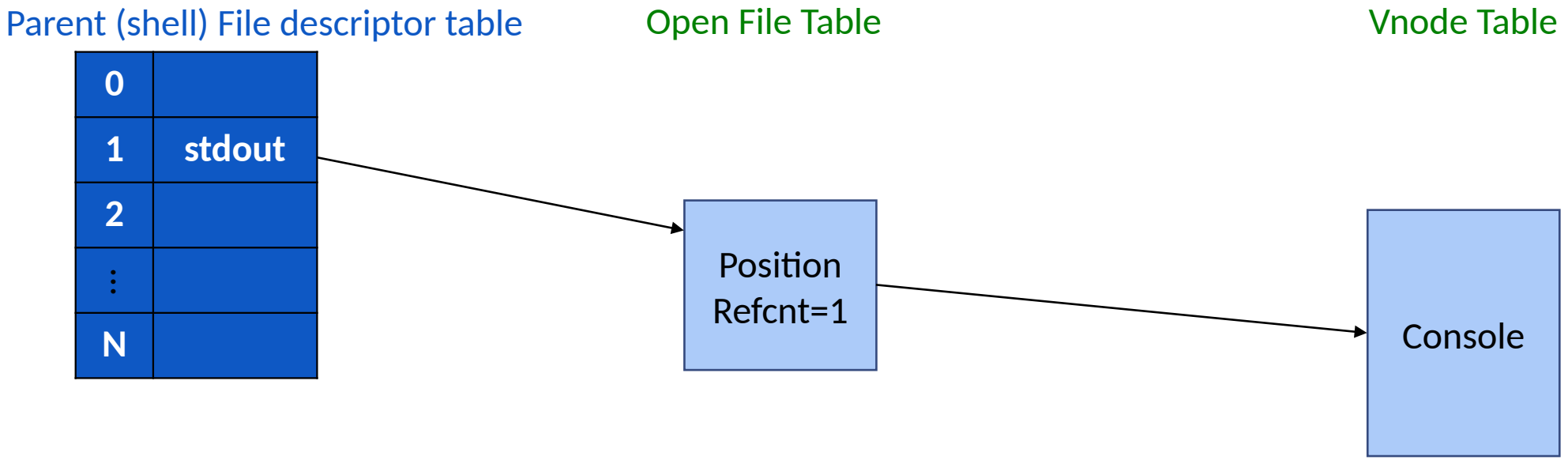
- Redirecting standard out:
  - When you want to capture a program's output, you can use:

```
ls > my-stdout-file.txt
```

```
pid_t child_pid = fork();
if (child_pid == 0) {
        /* In the child. */
        int fd = open("my-stdout-file.txt", O_WRONLY|O_CREATE|O_TRUNC, 0644);
        dup2(fd, STDOUT_FILENO);
        close(fd);
        execv("ls", argv); // Changes child to execute 'ls' instead of shell
} else if (child_pid > 0) {
        This is the parent
} else {
        This is an error
}
```

Redirecting a command in the shell (1)

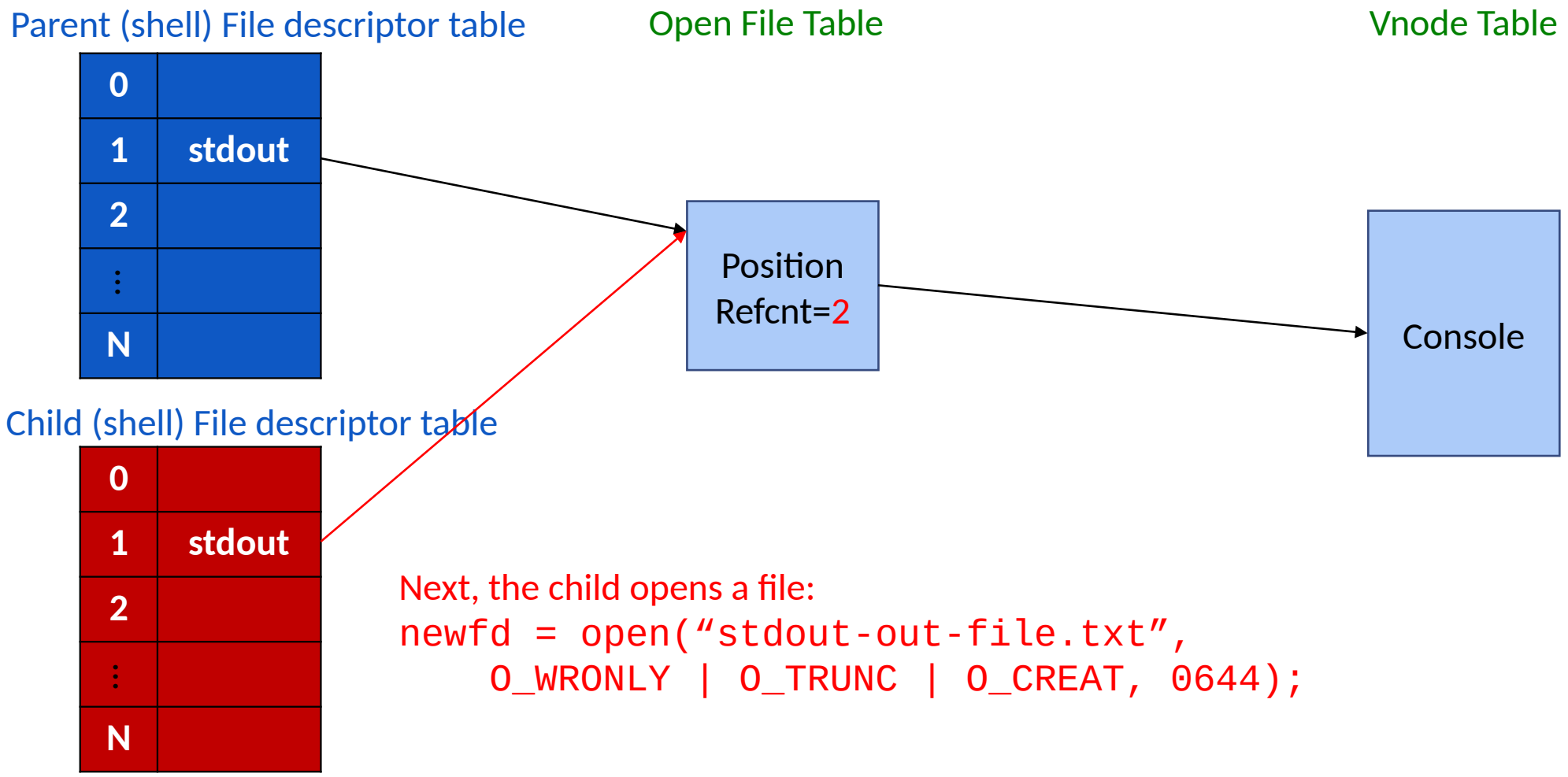
The red font tells you what happens between this slide and the next one.



Now, what happens when the parent forks?

```
child_pid = fork();
```

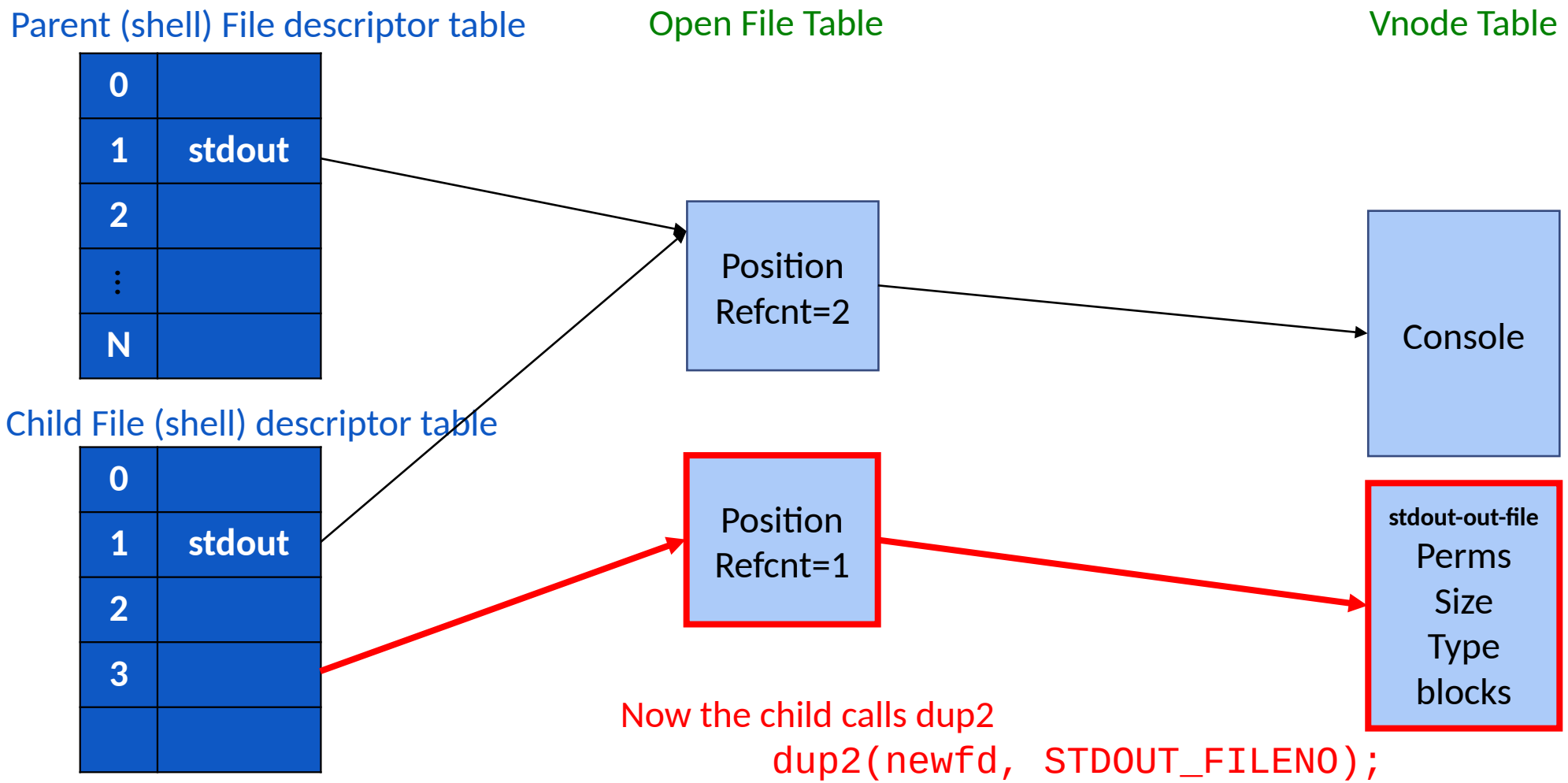
Redirecting a command in the shell (2)



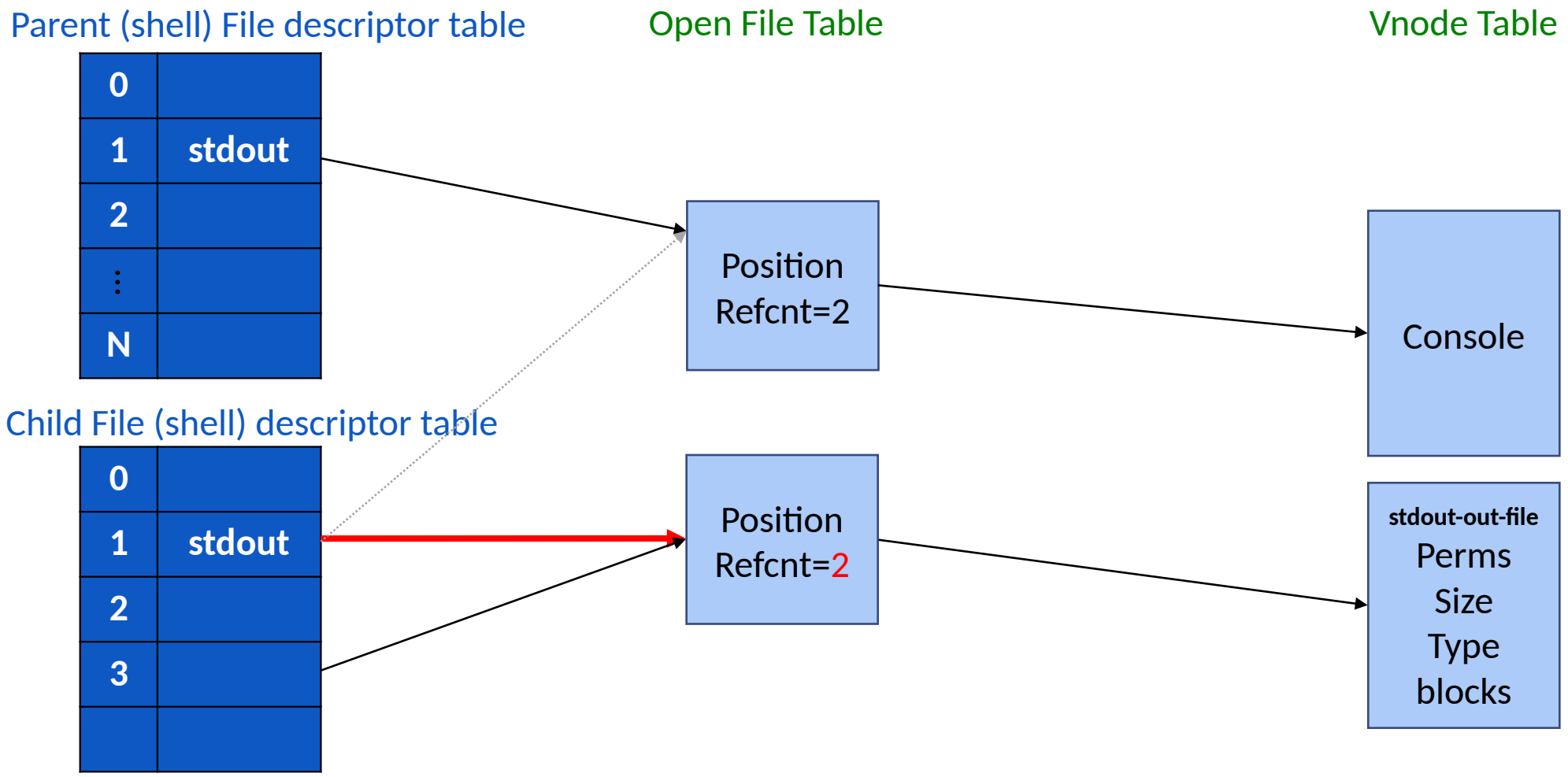
Next, the child opens a file:

```
newfd = open("stdout-out-file.txt",
    O_WRONLY | O_TRUNC | O_CREAT, 0644);
```

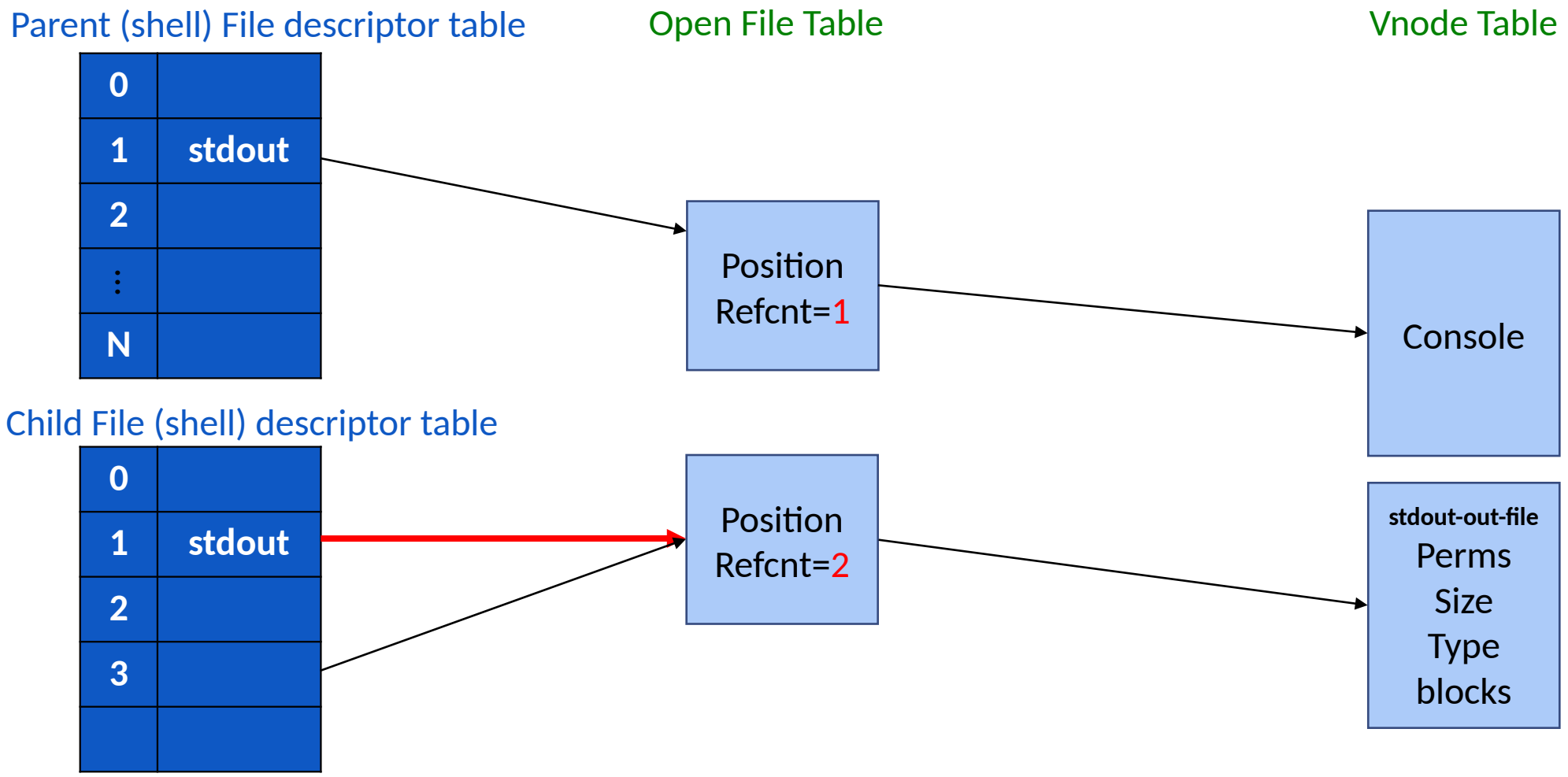
Redirecting a command in the shell (3)



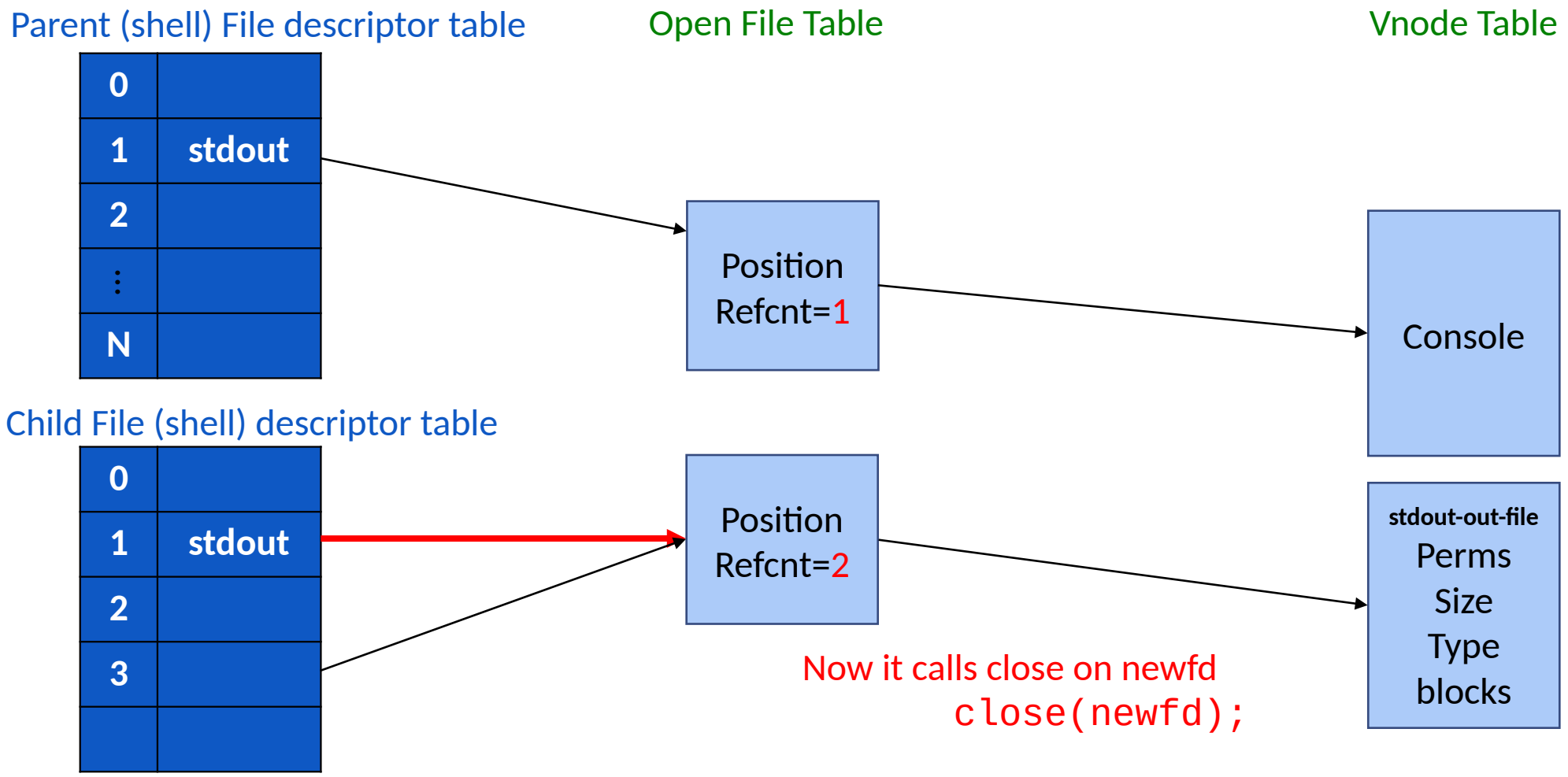
Redirecting a command in the shell (4)



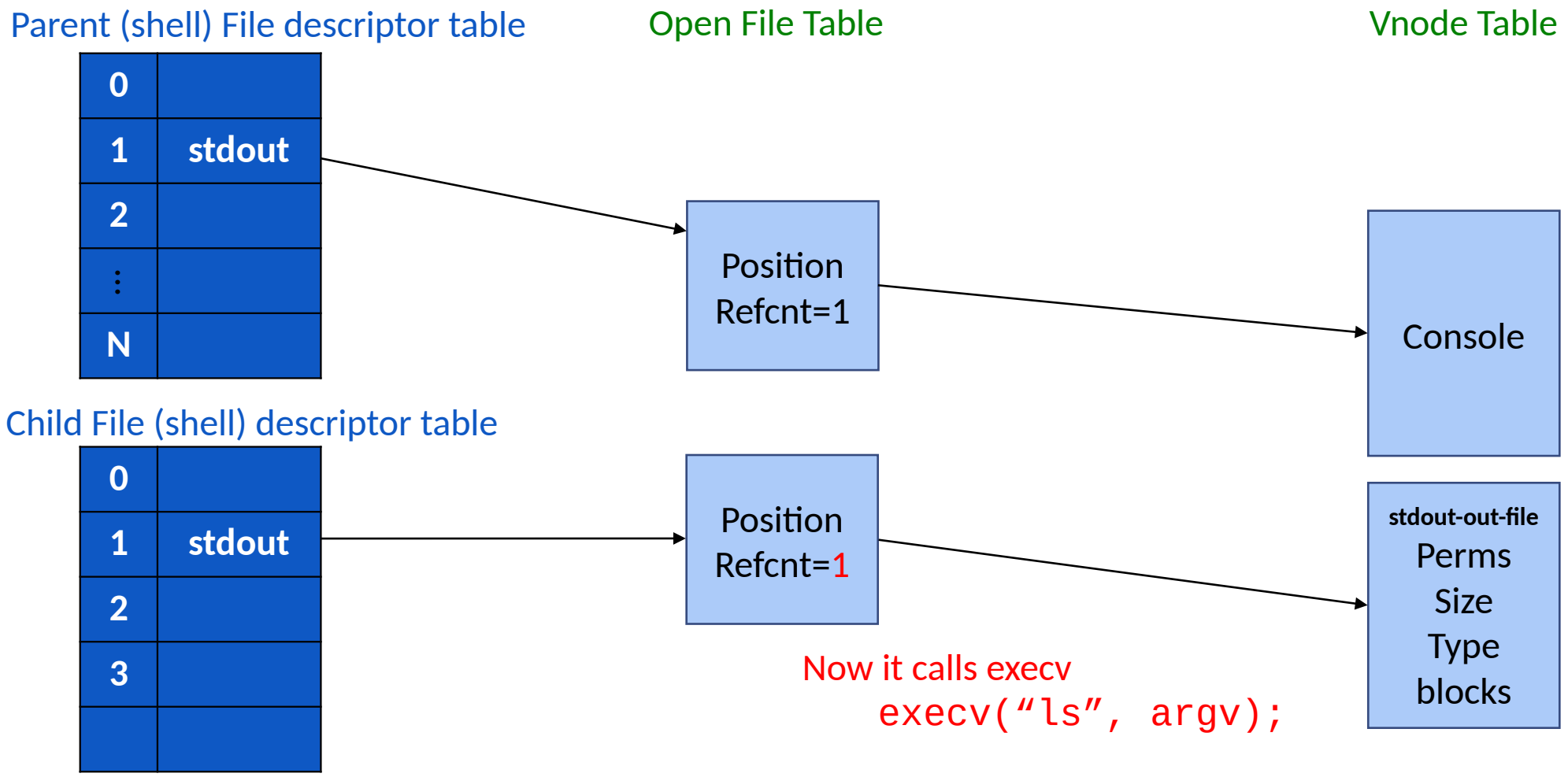
Redirecting a command in the shell (4)



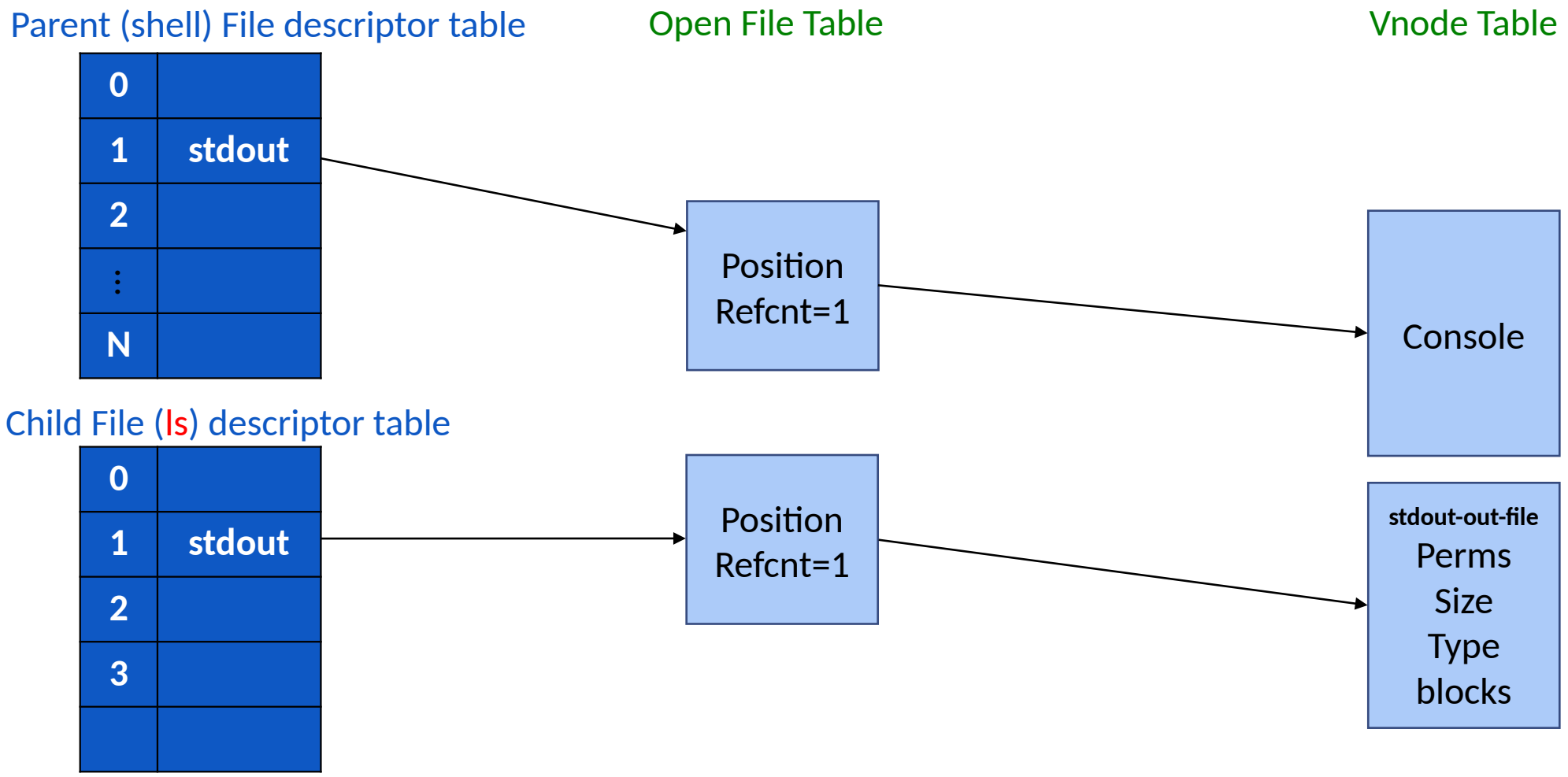
Redirecting a command in the shell (4)



Redirecting a command in the shell (5)



Redirecting a command in the shell (6)



Other fun facts about file descriptors and dup

- Redirect both standard out & standard error:
  - mycmd >& stdout-and-stderr-file.out
- Redirect standard in:
  - mycmd < input-file.txt
- Redirect standard out, but also let it display on the screen
  - mycmd | tee stdout-file.out
- Redirect standard out and append to an existing file
  - mycmd >> stdout-file.out