

Lecture 23: System I/O

CS 105

April 27, 2020

The File System Abstraction

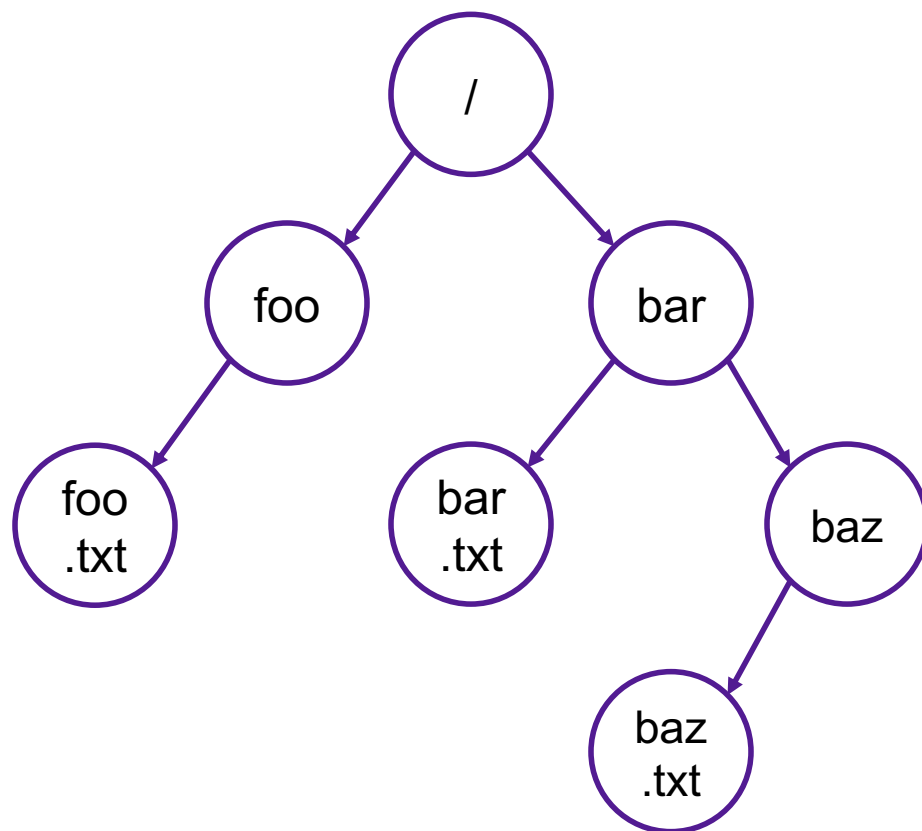
- interface that provides operations on data stored long-term on disk
- a **file** is a named sequence of stored bytes
 - name is defined on creation
 - processes use name to subsequently access that file
- two types of files
 - normal files (an arbitrary sequence of bytes)
 - directories (a special type of file)
- a file is comprised of two parts:
 - **data**: information a user or application puts in a file
 - an array of untyped bytes
 - implemented as an array of fixed-size blocks
 - **metadata**: information added and managed by the OS
 - e.g., size, owner, security info, modification time

File Names

- Each file has a unique low-level name
 - distinct from location; processes don't care where on disk a file is stored
 - file system provides mapping from low-level names to storage location
- Each file has one or more human-readable names
 - file system provides mapping from human-readable names to low-level names
- Naming conventions
 - up to 255 characters long
 - case sensitive (UNIX) or not case sensitive (Windows)
 - extensions not enforced (UNIX) or associated with meaning (Windows)

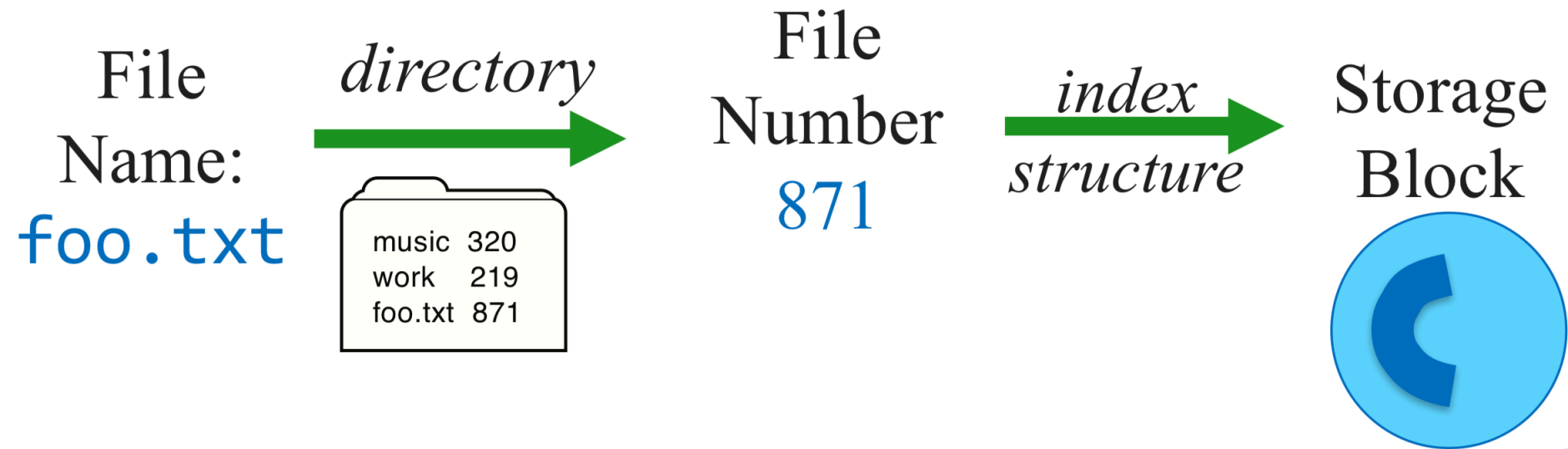
Path Names

- Each path from root is a name for a leaf
 - /foo/foo.txt
 - /bar/baz/baz.txt
- Each UNIX directory contains 2 special entries
 - "." = this directory
 - ".." = parent directory
- **Absolute paths:** path of file from the root directory
- **Relative paths:** path from current working directory



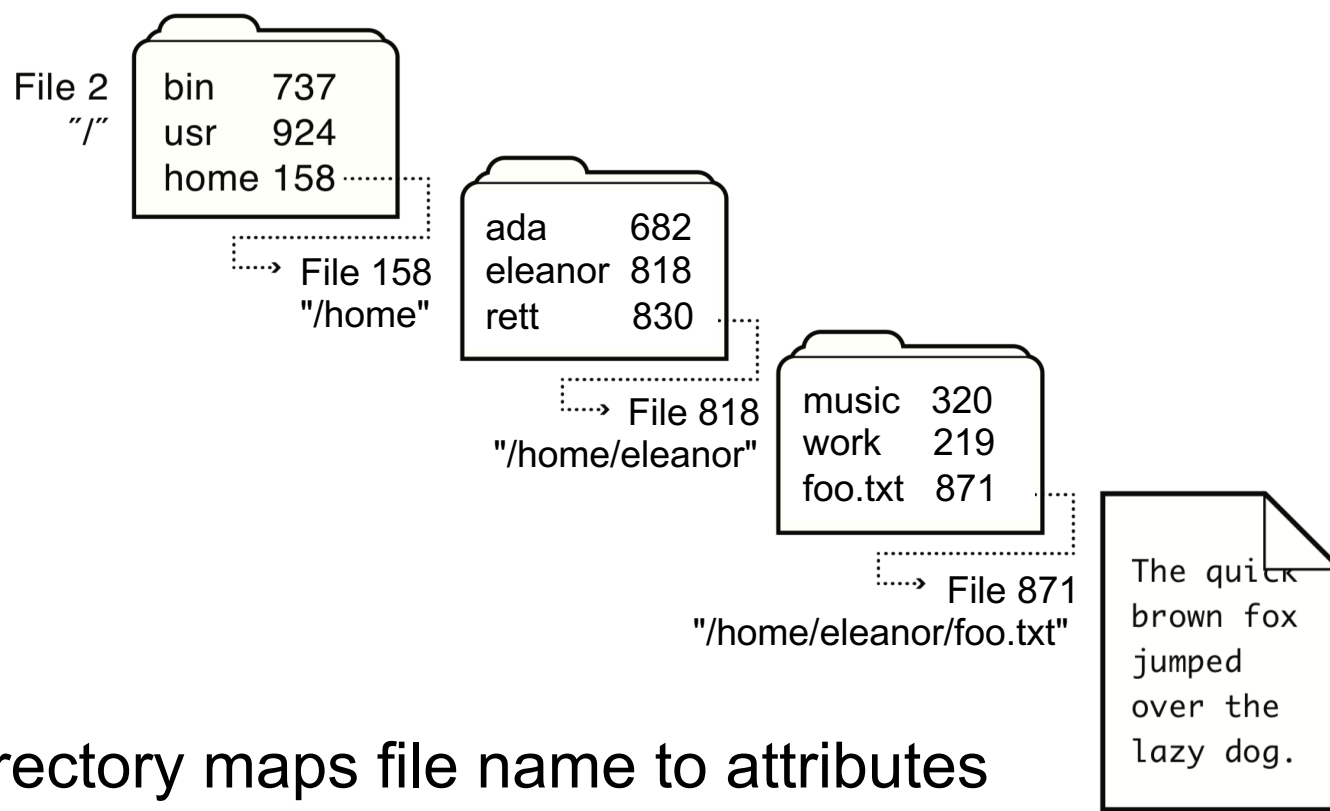
Directories

- a **directory** is a file that provides mappings from human-readable names to low-level names (i.e., file numbers):
 - a list of human-readable names
 - a mapping from each name to a specific underlying file or directory



Directories

- OS uses path name to find directories and files



- Directory maps file name to attributes and locations

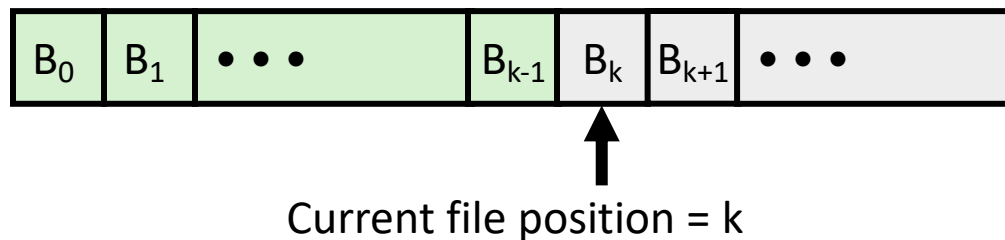
Basic File System Operations

- Create a file
- Delete a file
- Write to a file
- Read from a file
- Seek to somewhere in a file

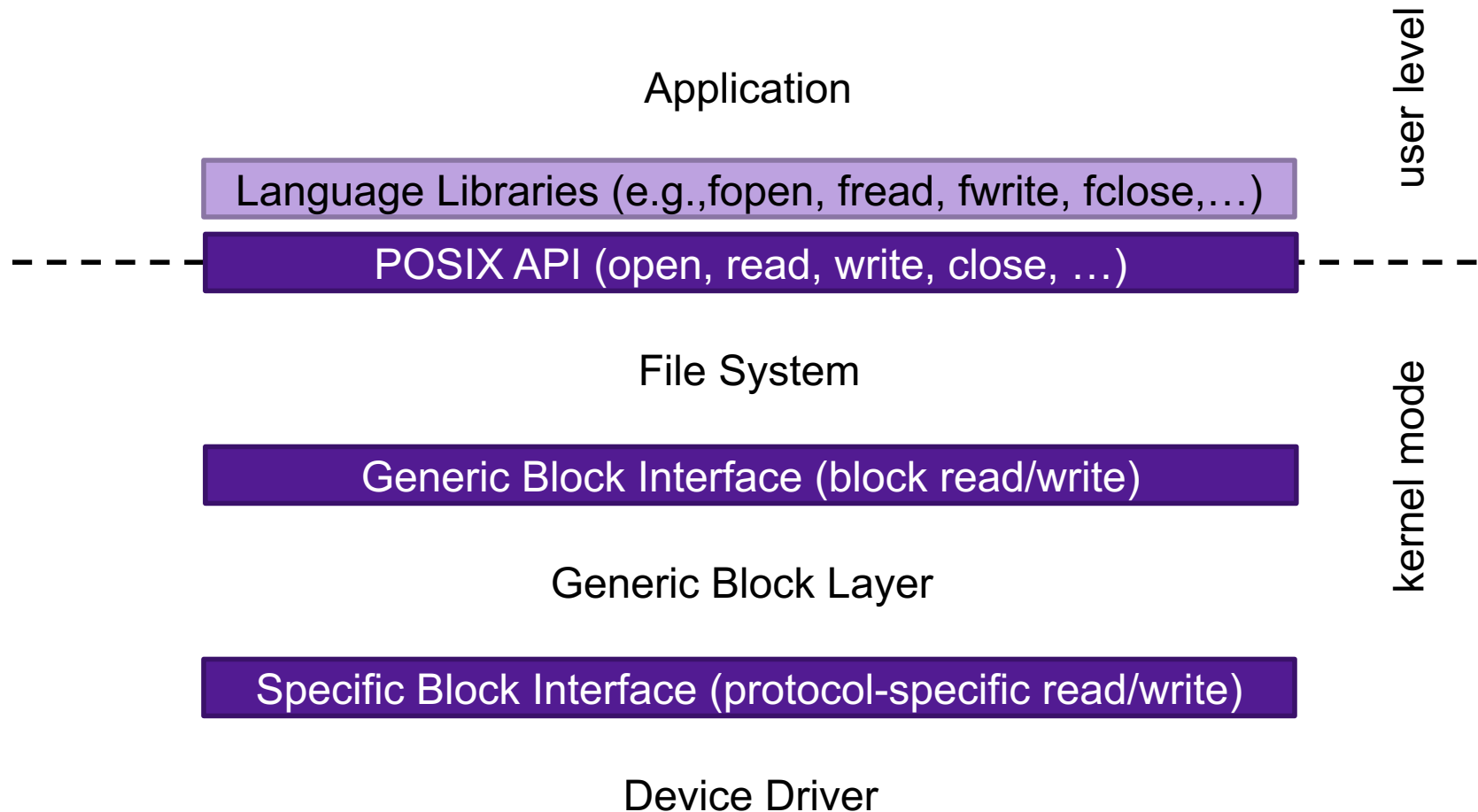
How should we implement this?

Unix I/O Interface

- Elegant mapping of files to devices allows kernel to export simple interface:
 - Opening a file
 - `open()` and `close()`
 - Reading and writing a file
 - `read()` and `write()`
 - Changing the current **file position** (seek)
 - indicates next offset into file to read or write
 - `lseek()`



The File System Stack



Opening Files

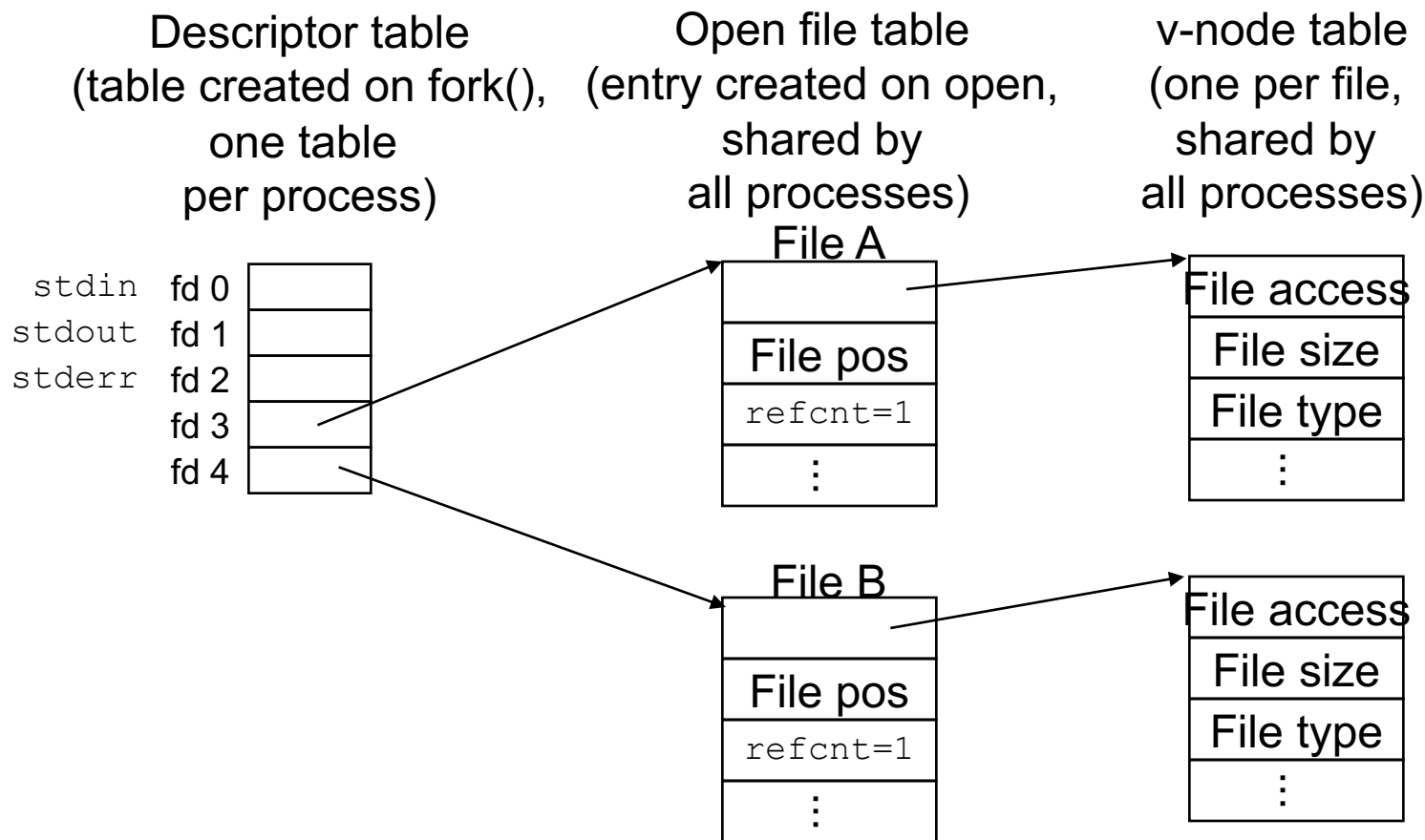
- Opening a file informs the kernel that you are getting ready to access that file

```
int fd;    /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
    perror("open");
    exit(1);
}
```

- Returns a small identifying integer **file descriptor**
 - `fd == -1` indicates that an error occurred
- Each process created by a Linux shell begins life with three open files associated with a terminal:
 - 0: standard input (stdin)
 - 1: standard output (stdout)
 - 2: standard error (stderr)

Kernel Data Structures



Closing Files

- Closing a file informs the kernel that you are finished accessing that file

```
int fd;      /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
    perror("close");
    exit(1);
}
```

- Closing an already closed file is a recipe for disaster in threaded programs
- Moral: Always check return codes, even for seemingly benign functions such as `close()`

Reading Files

- Reading a file copies bytes from the current file position to memory, and then updates file position

```
char buf[512];
int fd;      /* file descriptor */
int nbytes;  /* number of bytes read */

/* Open file fd ... */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
    perror("read");
    exit(1);
}
```

- Returns number of bytes read from file `fd` into `buf`
 - Return type `ssize_t` is signed integer
 - `nbytes < 0` indicates that an error occurred
 - **Short counts** (`nbytes < sizeof(buf)`) are possible and are not errors!

Writing Files

- Writing a file copies bytes from memory to the current file position, and then updates current file position

```
char buf[512];
int fd;          /* file descriptor */
int nbytes;     /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
    perror("write");
    exit(1);
}
```

- Returns number of bytes written from `buf` to file `fd`
 - `nbytes < 0` indicates that an error occurred
 - As with reads, short counts are possible and are not errors!

On Short Counts

- Short counts can occur in these situations:
 - Encountering (end-of-file) EOF on reads
 - Reading text lines from a terminal
- Short counts never occur in these situations:
 - Reading from disk files (except for EOF)
 - Writing to disk files
- Best practice is to always allow for short counts.

Buffered Reads/Writes

- stream data is stored in a kernel buffer and returned to the application on request
- enables same system call interface to handle both streaming reads (e.g., keyboard) and block reads (e.g., disk)

Exercise 1: Reading and Writing

- What gets printed when the following program is run?

```
int main(int argc, char ** argv){
    int fd1, fd2;
    char c;

    fd1 = open("foobar.txt", O_RDONLY);
    fd2 = open("foobar.txt", O_RDONLY);

    read(fd1, &c, 1);
    read(fd2, &c, 1);

    printf("c = %c\n", c);

    return 0;
}
```

Exercise 1: Reading and Writing

File descriptor table

stdin	0	
stdout	1	
stderr	2	
fd1	3	
fd2	4	

Open file table

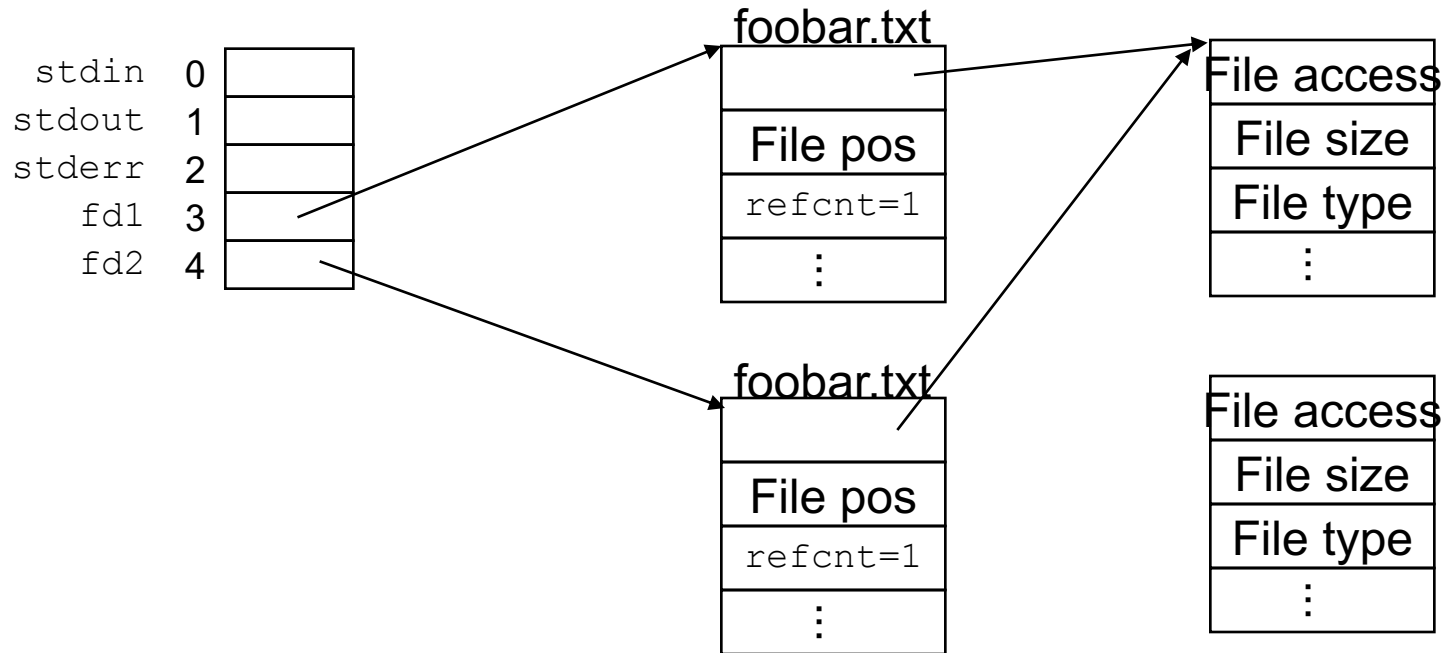
foobar.txt
File pos
refcnt=1
:

foobar.txt
File pos
refcnt=1
:

v-node table

File access
File size
File type
:

File access
File size
File type
:



Processes and Files

- A child process inherits all file descriptors from its parent

File descriptor table

Open file table

v-node table

Parent's table

fd 0	
fd 1	
fd 2	
fd 3	
fd 4	

Child's table

fd 0	
fd 1	
fd 2	
fd 3	
fd 4	

File A

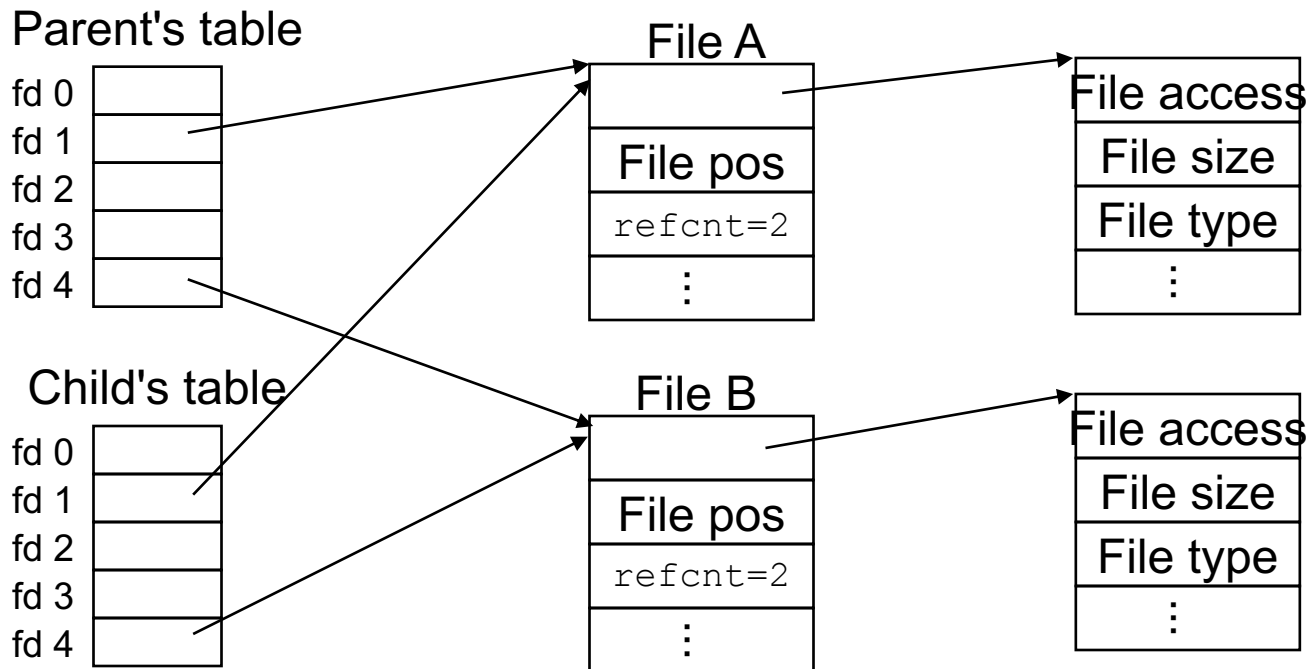
File pos
refcnt=2
⋮

File B

File pos
refcnt=2
⋮

File access
File size
File type
⋮

File access
File size
File type
⋮



Exercise 2: Processes and Files

- Suppose the file `foobar.txt` consists of the six ASCII characters `foobar`. What is printed when the following program is run?

```
int main(int argc, char ** argv){
    int fd1;
    char c;

    fd1 = open("foobar.txt", O_RDONLY);
    if(fork() == 0){
        read(fd, &c, 1);
        return 0;
    } else {
        wait();
        read(fd, &c, 1);
        printf("c = %c\n", c);
        return 0;
    }
}
```

Exercise 2: Processes and Files

File descriptor table

Open file table

v-node table

Parent's table

fd 0	
fd 1	
fd 2	
fd 3	
fd 4	

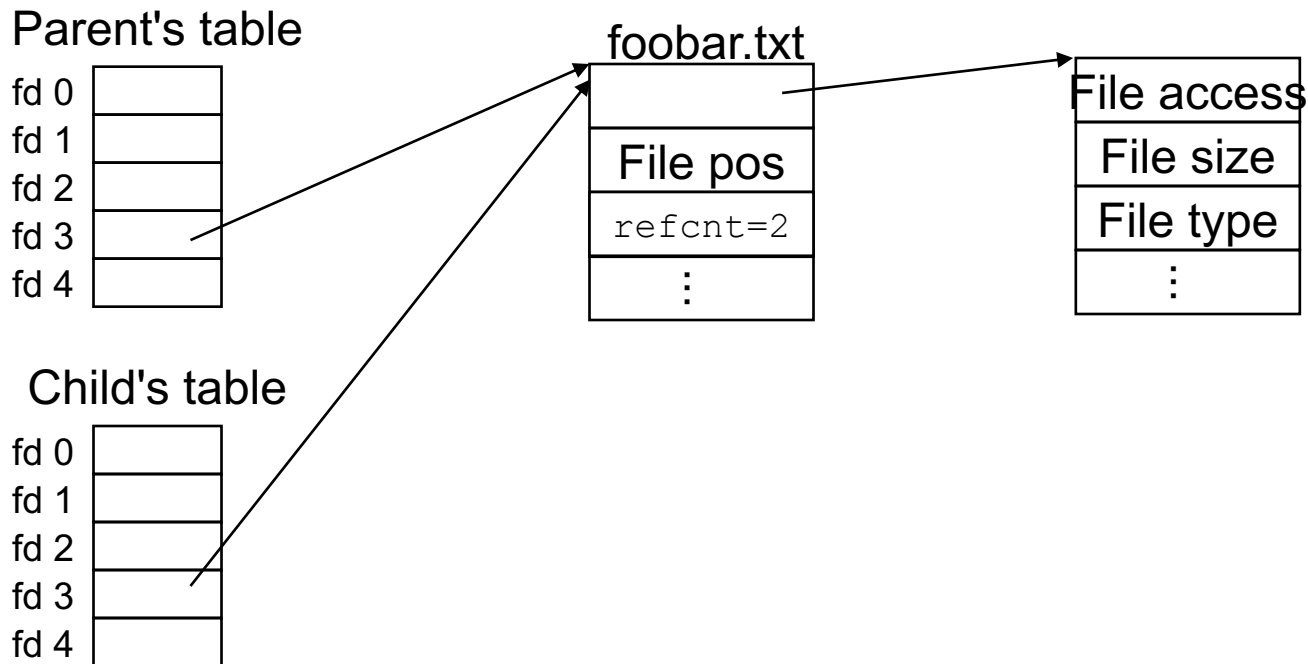
Child's table

fd 0	
fd 1	
fd 2	
fd 3	
fd 4	

foobar.txt

File pos
refcnt=2
⋮

File access
File size
File type
⋮



I/O Redirection

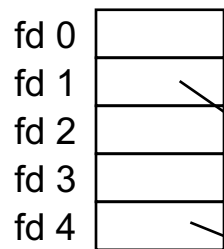
- Examples of I/O redirection
 - a program can send output to a file: `./ringbuf 4 > testout.txt`
 - a program can read input from a file: `./ringbuf 4 < testin.txt`
 - output of one program can be input to another: `cpp file.c | cparse | cgen | as > file.o`
- I/O redirection uses a function called `dup2`

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

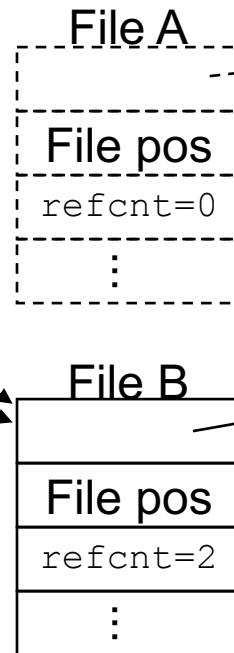
- returns file descriptor if OK, -1 on error

I/O Redirection

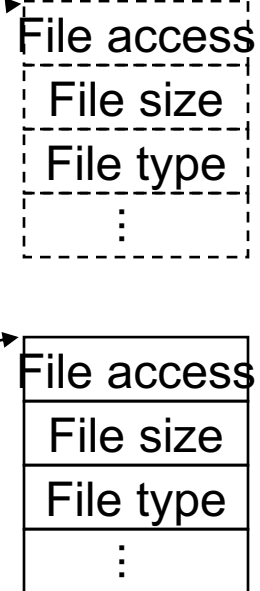
Descriptor table
(one table
per process)



Open file table
(shared by
all processes)



v-node table
(shared by
all processes)



Exercise 3: I/O Redirection

- Suppose the file `foobar.txt` consists of the six ASCII characters `foobar`. What is printed when the following program is run?

```
int main(){
    int fd1, fd2;
    char c;

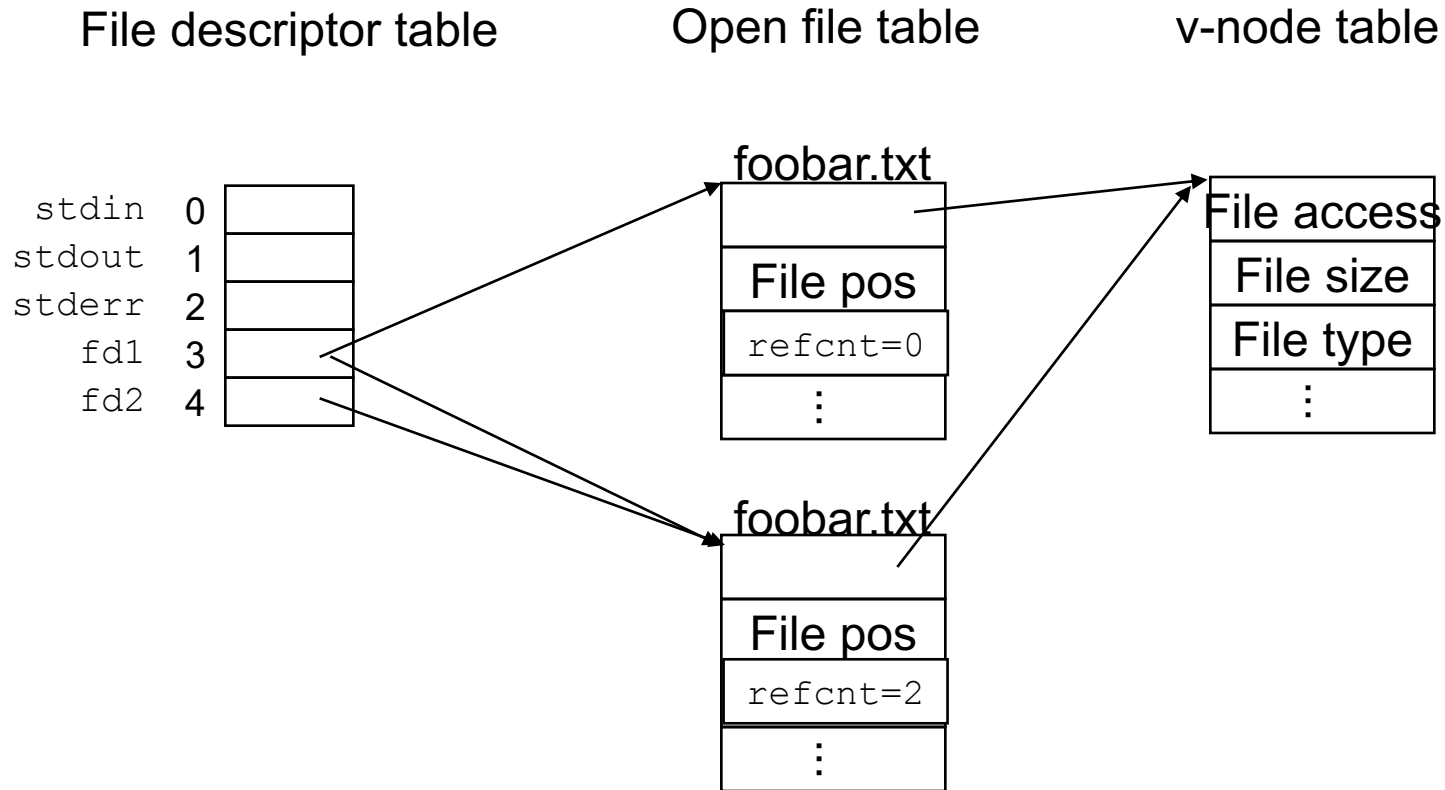
    fd1 = open("foobar.txt", O_RDONLY);
    fd2 = open("foobar.txt", O_RDONLY);

    read(fd2, &c, 1);
    dup2(fd2, fd1);
    read(fd1, &c, 1);

    printf("c = %c\n", c);

    return 0;
}
```


Exercise 3: I/O Redirect



System I/O as a Uniform Interface

- Operating systems use the System I/O commands as an interface for all I/O devices
- The commands to read and write to an open file descriptor are the same no matter what type of "file" it is
- Types of files include
 - file
 - keyboard
 - screen
 - pipe
 - device
 - network

Exercise 4: Feedback

1. Rate how well you think this recorded lecture worked
 1. Better than an in-person class
 2. About as well as an in-person class
 3. Less well than an in-person class, but you still learned something
 4. Total waste of time, you didn't learn anything
2. How much time did you spend on this video lecture?
3. Do you have any comments or suggestions for future classes?