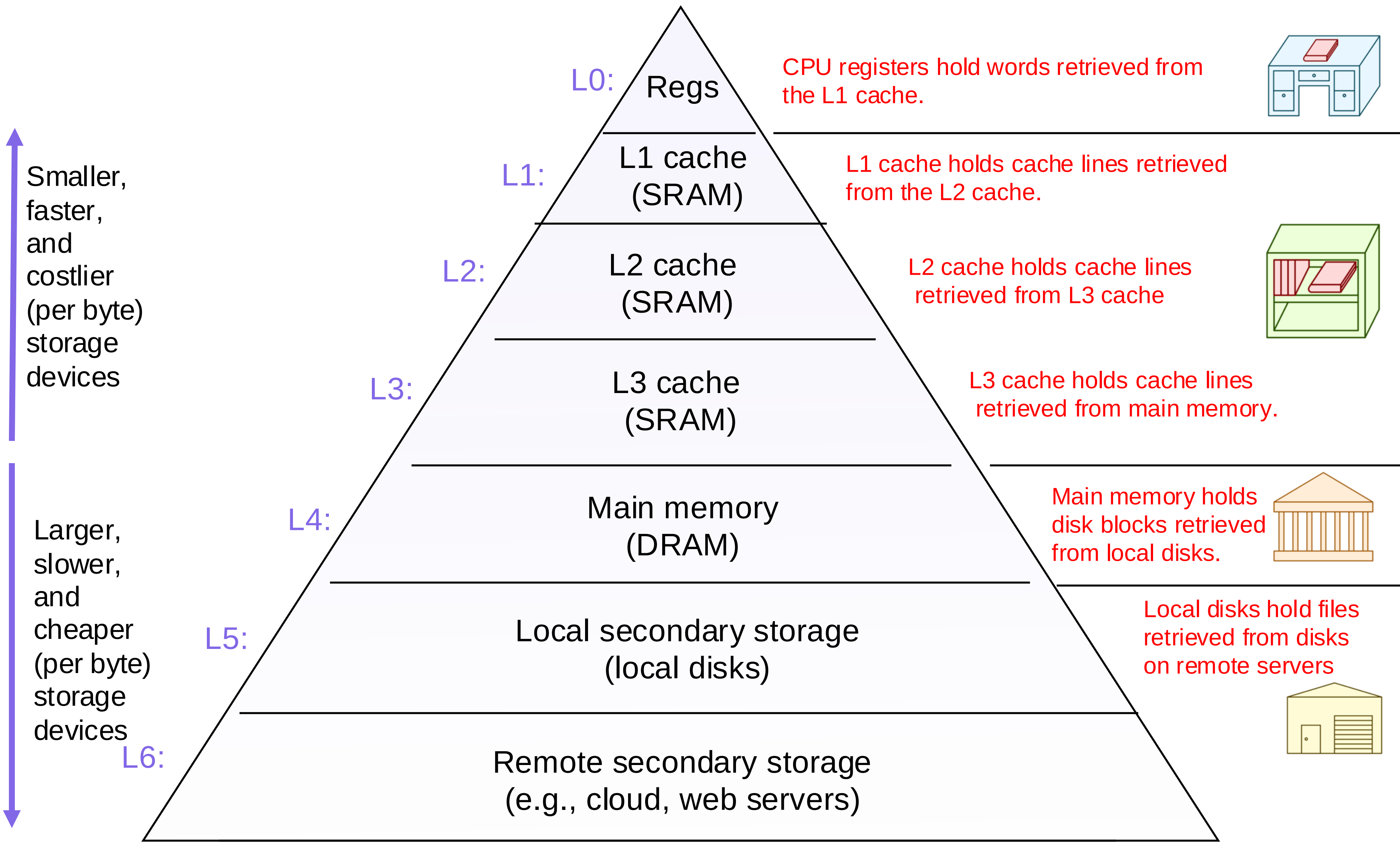
Lecture 21: System I/O

CS 105

Fall 2024

Memory Hierarchy



Storage Devices

- Magnetic Disks
  - Storage that rarely becomes corrupted
  - Large capacity at low cost
  - Block-level random access
  - Slow performance for random access
  - Better performance for streaming access
- Solid State Disks (Flash Memory)
  - Storage that rarely becomes corrupted
  - Capacity at moderate cost (50x magnetic)
  - Block-level random access
  - Good performance for random reads
  - Not-as-good performance for random writes

1950s
IBM 350
5 MB

2021
WD Red
10 TB

2024
MacBook
1TB

File Systems 101

- Long-term information storage goals
  - should be able to store large amounts of information
  - information must survive processes, power failures, etc.
  - processes must be able to find information
  - needs to support concurrent accesses by multiple processes
- Solution: the File System Abstraction
  - interface that provides operations involving files

The File System Abstraction

- interface that provides operations on data stored long-term on disk

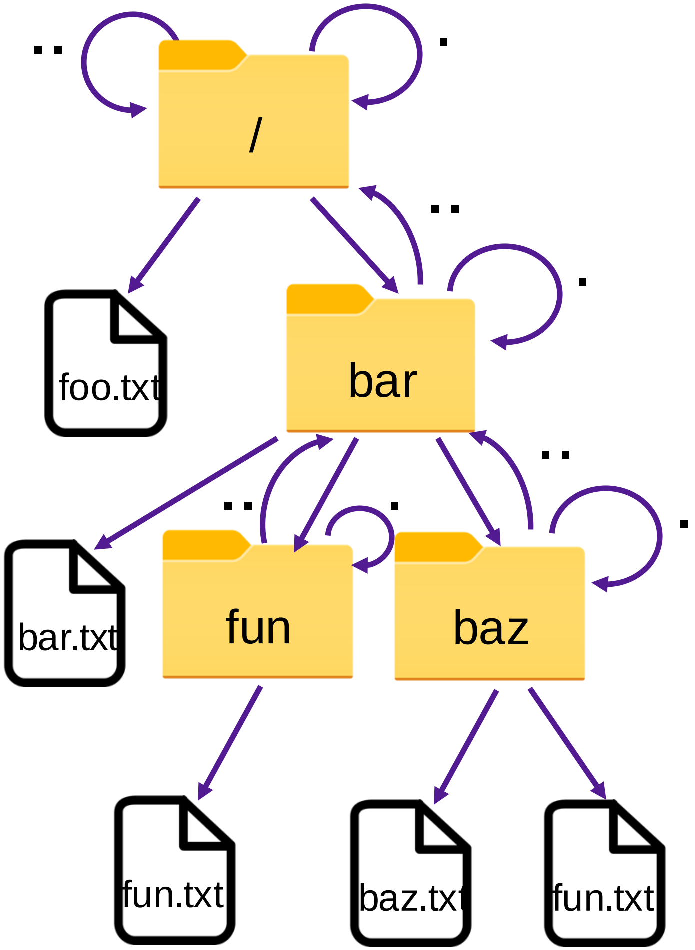
- a **file** is a named sequence of stored bytes
  - name is defined on creation
  - processes use name to subsequently access that file

- a file is comprised of two parts:
  - **data**: information a user or application puts in a file
    - an array of untyped bytes
  - **metadata**: information added and managed by the OS
    - e.g., size, owner, security info, modification time

- two types of files
  - **normal files**: data is an arbitrary sequence of bytes
  - **directories**: a special type of file that provides mappings from human-readable names to low-level names (i.e., file numbers)

Path Names

- A file system has a root directory "/"
- Directories contain other files (including subdirectories)
- Each UNIX directory also contains 2 special entries
  - "." = this directory
  - ".." = parent directory
- Each path from root is a name for a leaf
  - /foo.txt
  - /bar/baz/baz.txt
- **Absolute paths:** path of file from the root directory
- **Relative paths:** path from current working directory



Exercise 1: Path Names

I’ve created a file named `example1.txt` in the directory `cs105`, which is located in the root directory.

1. Specify an absolute path to the file `example1.txt`
2. Specify a relative path to the file `example1.txt` from your home directory (`/home/abcd2047/`).

I've created a file named `example2.txt` in my home directory (`/home/ebac2018/`).

3. Specify an absolute path to the file `example2.txt`
4. Specify a relative path to the file `example2.txt` from your home directory

Hint: you can always get back to your home directory with `cd ~`

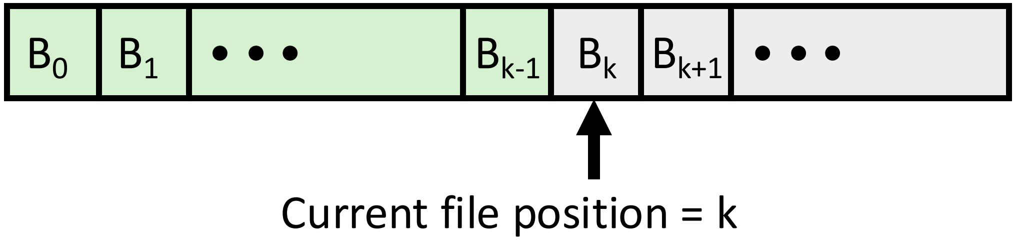
Basic File System Operations

1.
2.
3.
4.
5.

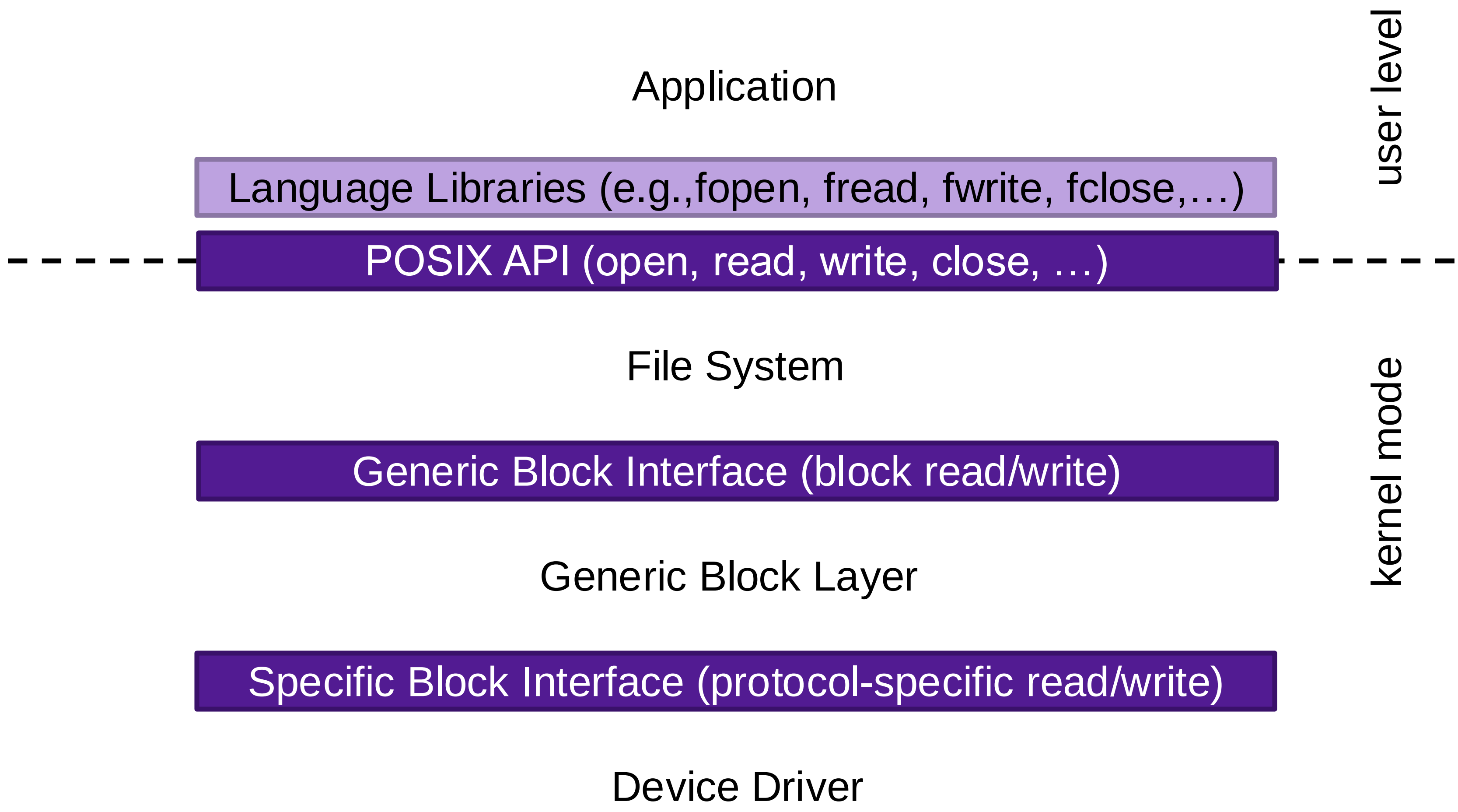
How should we implement this?

Unix I/O Interface

- 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



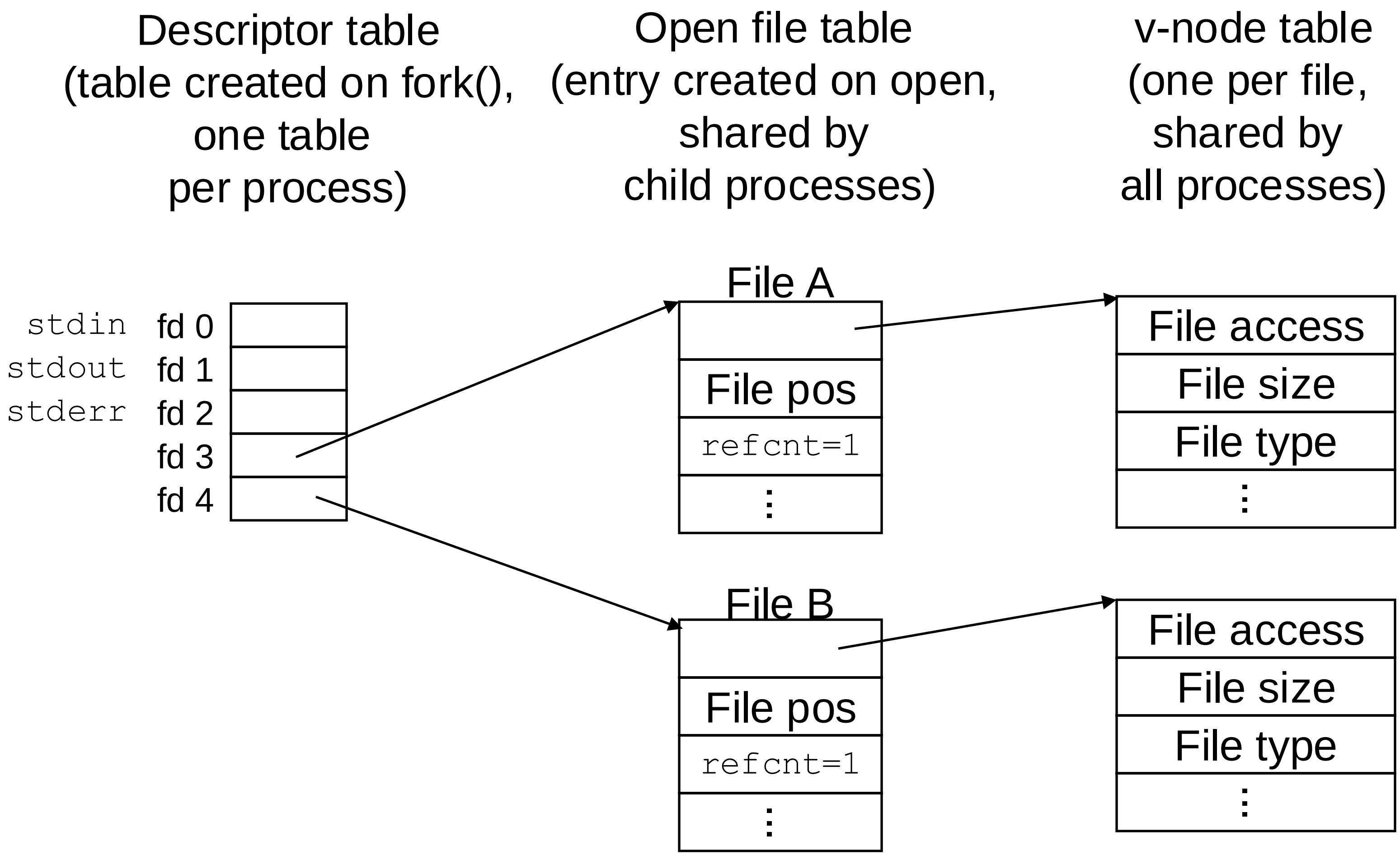
Opening Files

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

```
int fd;   /* file descriptor */
fd = open("/etc/hosts", O_RDONLY);
if (fd < 0) {
   perror("open failed");
   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



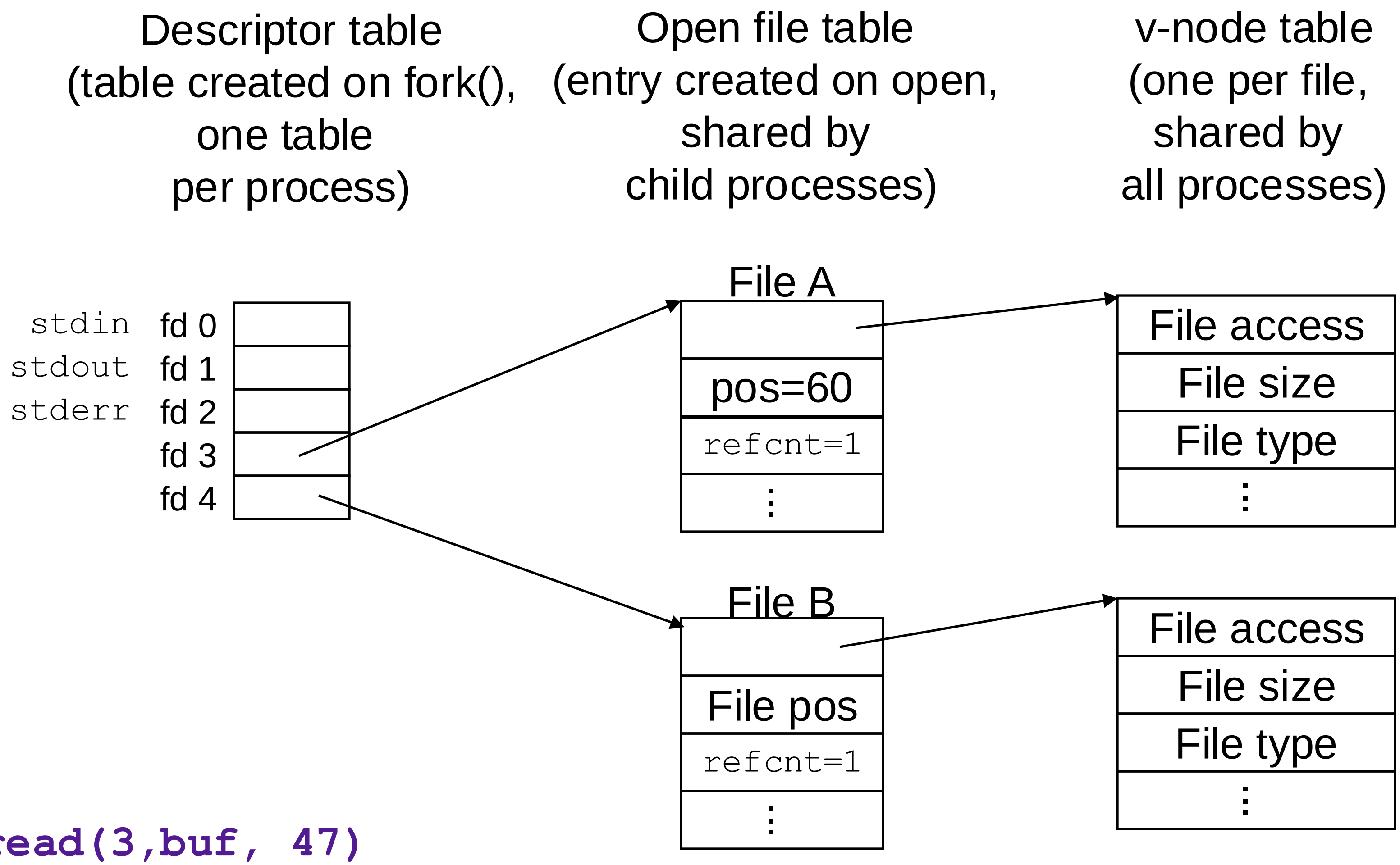
Reading Files

- 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 */
nbytes = read(fd, buf, sizeof(buf));
if (nbytes < 0) {
   perror("read error");
   exit(1);
}
```

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

Kernel Data Structures



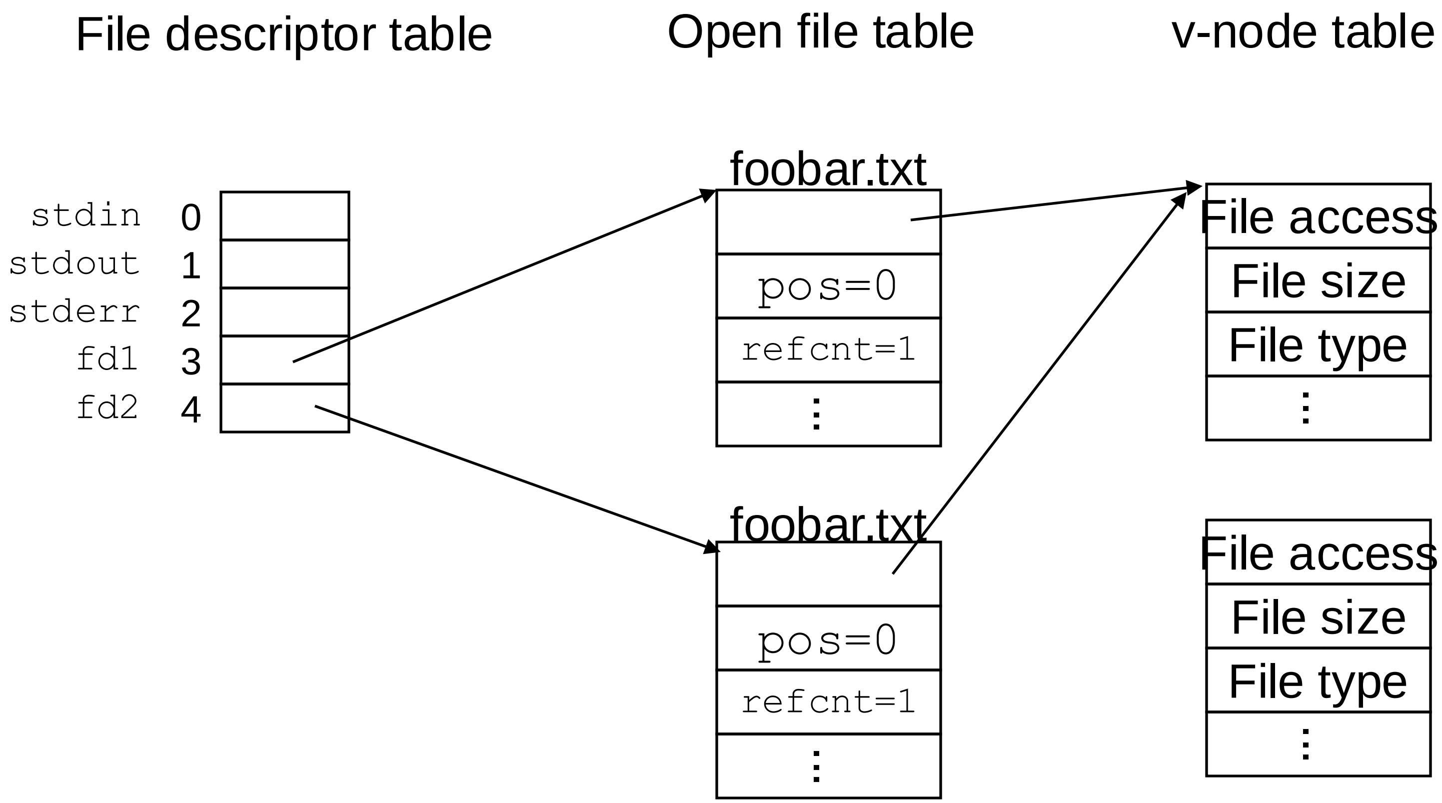

```
read(3,buf, 47)
read(3,buf, 13)
```

Exercise 2: Reading and Writing

- Assume the file `foobar.txt` consists of the six ASCII characters `foobar`. 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 2: Reading and Writing



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 */
nbytes = write(fd, buf, sizeof(buf));
if (nbytes < 0) {
   perror("write error");
   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)

Closing Files

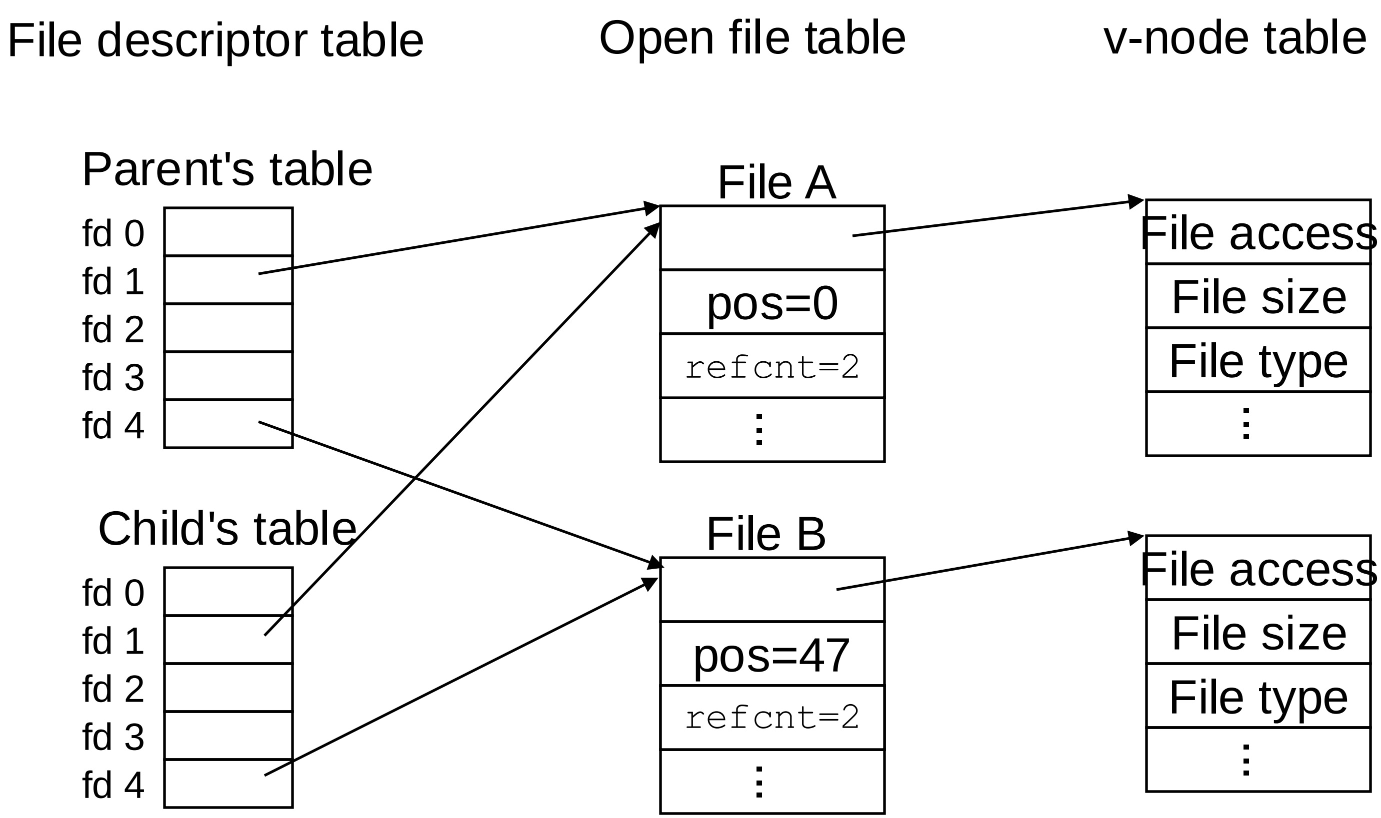
- Closing a file informs the kernel that you are finished accessing that file

```
int fd;      /* file descriptor */
int retval; /* return value */
retval = close(fd);
if (retval < 0) {
   perror("close error");
   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()`

Processes and Files

- A child process inherits all file descriptors from its parent

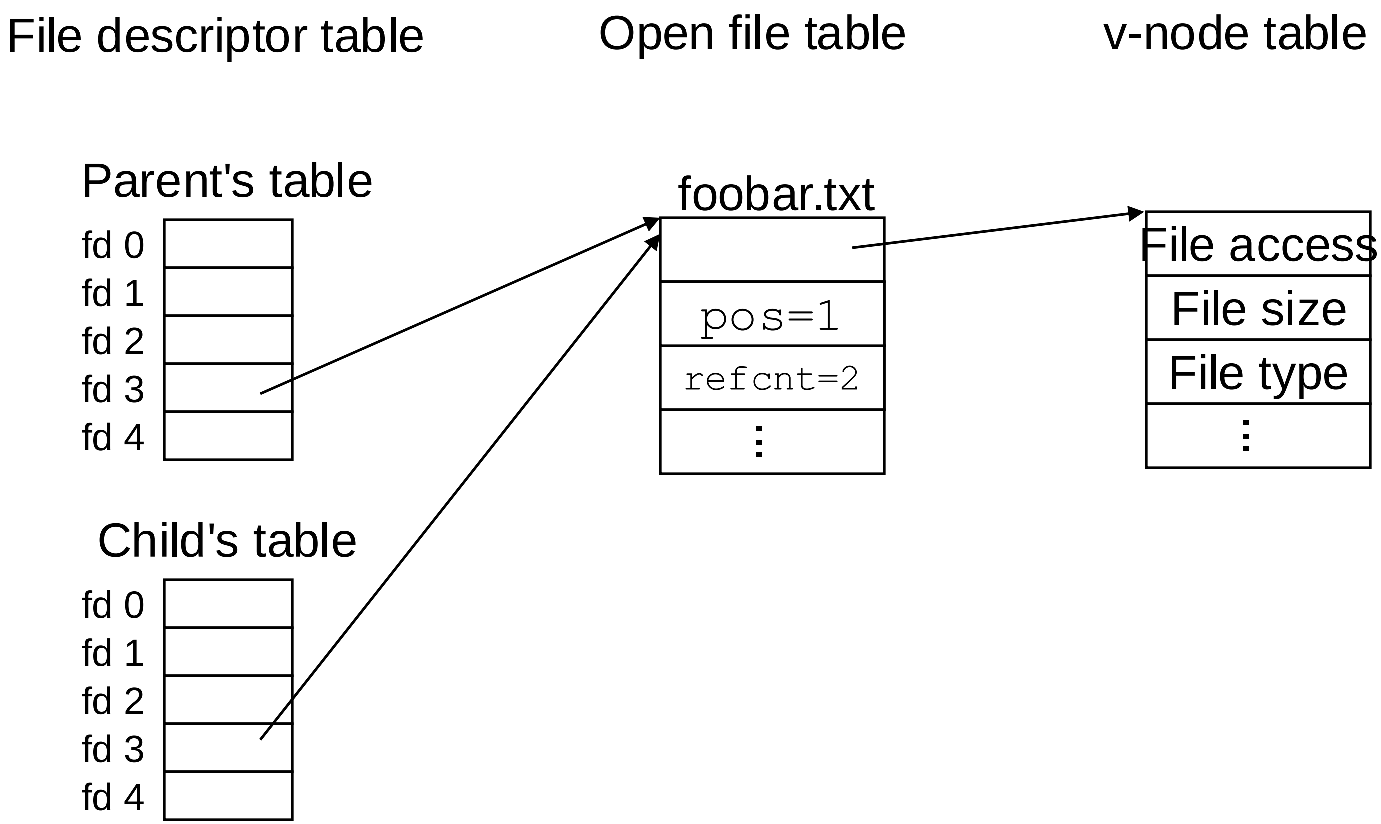


Exercise 3: Processes and Files

- 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 fd;
    char c;

    fd = open("foobar.txt", O_RDONLY);
    if(fork() == 0){ // if child process
        read(fd, &c, 1);
        return 0;
    } else {         // if parent process
        wait();      // wait for child to complete
        read(fd, &c, 1);
        printf("c = %c\n", c);
        return 0;
    }
}
```

Exercise 3: Processes and Files



I/O Redirection

- Examples of I/O redirection
  - a program can read input from a file: ./hex2raw < exploit.txt
  - a program can send output to a file: ./hex2raw > exploit-raw.txt
  - output of one program can be input to another: cat exploit.txt | ./hex2raw | ./ctarget -q

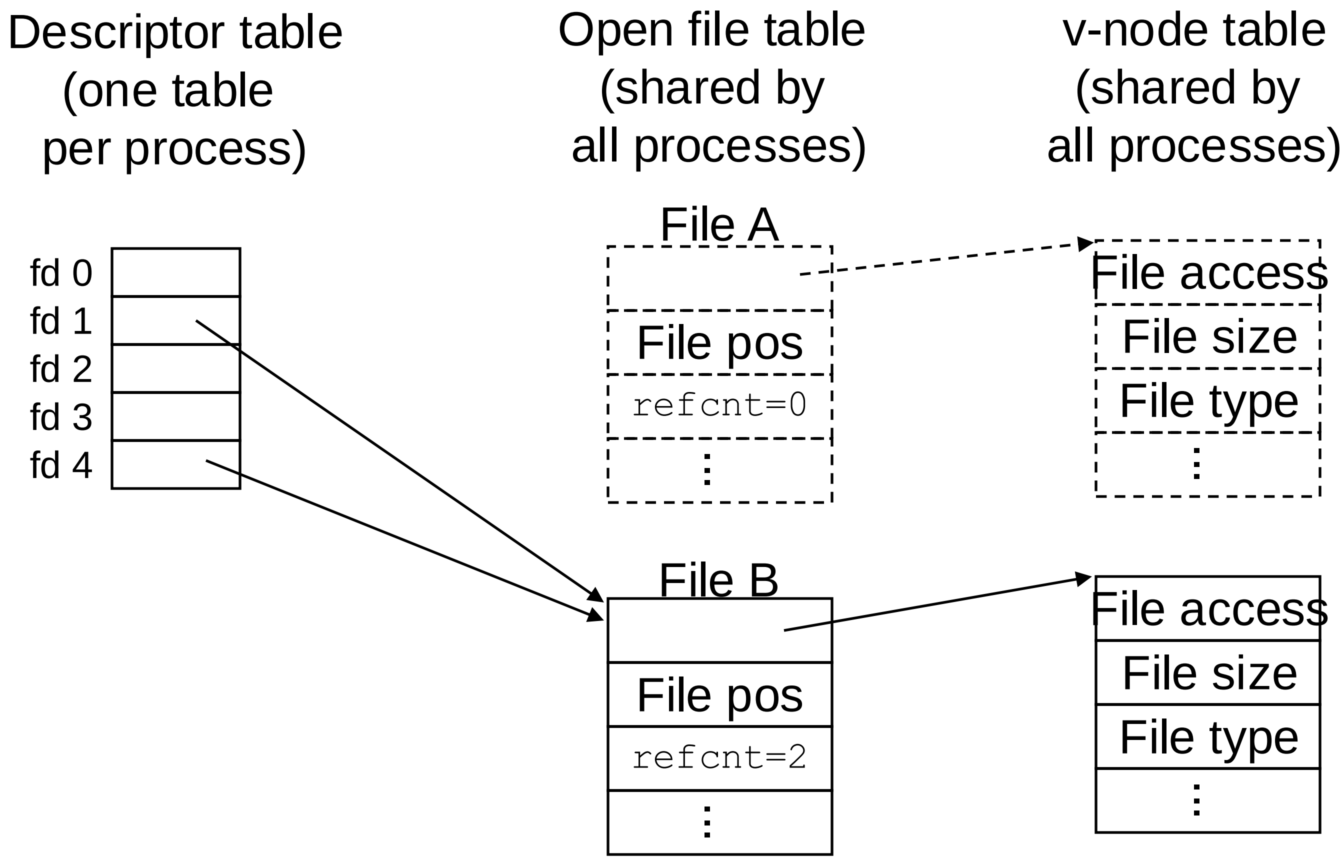
- I/O redirection uses a function called dup2

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

  - changes `newfd` to point to same open file table entry as `oldfd`

- returns file descriptor if OK, -1 on error

I/O Redirection



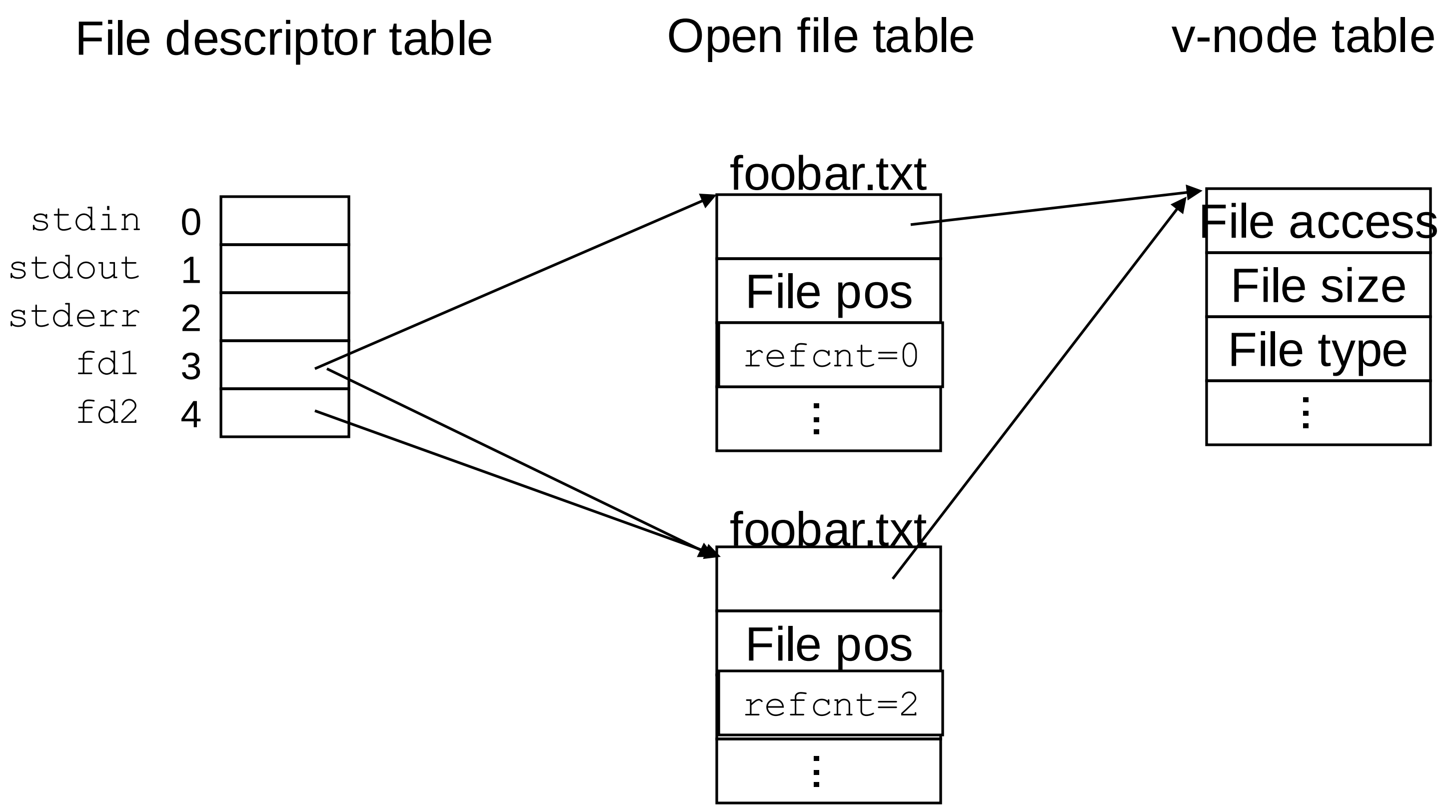
`dup2(4,1)`

Exercise 4: 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 4: 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