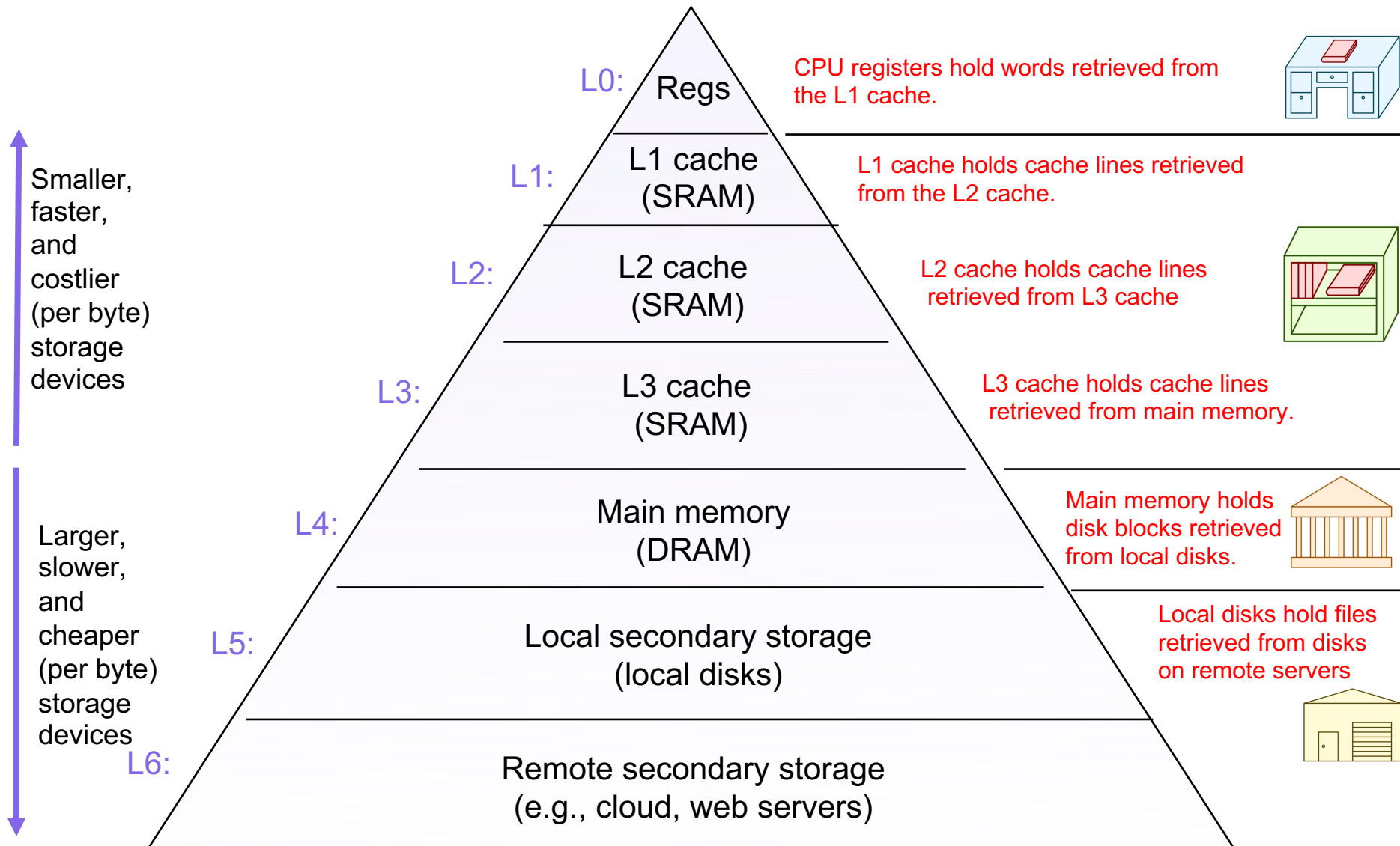


Lecture 23: System I/O

CS 105

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



1950s
IBM 350
5 MB



2021
WD Red
10 TB

- Solid State Disks (Flash Memory)

- Storage that rarely becomes corrupted
- Capacity at moderate cost (50x magnetic disk)
- Block-level random access
- Good performance for random reads
- Comparatively lower performance for random writes



2021
MacBook
512GB

File Systems 101

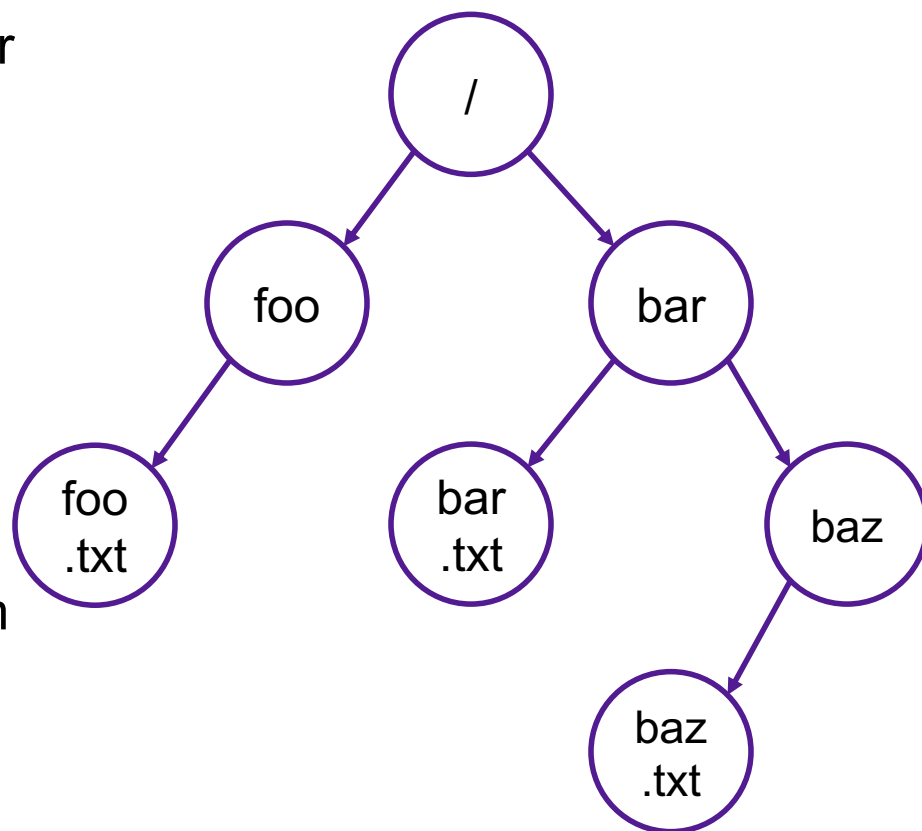
- Long-term information storage goals
 - Should be able to store large amounts of information
 - Information must survive processes, power failures, etc.
 - Processes must be able to find information
 - Needs to support concurrent accesses by multiple processes
- Solution: the file system abstraction
 - Interface that provides operations involving
 - Files
 - Directories (a special kind of file)

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 comprises 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

- 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 (stored as part of a process's data)



Exercise 1: Path Names

I'll create a file named `example1.txt` in the directory `data`, which is in the root directory.

You must

1. Specify an absolute path to the file `example1.txt`
2. Specify a relative path to the file `example1.txt` from your home directory
3. Create a file named `example2.txt` in your home directory.
4. Specify an absolute path to the file `example2.txt`
5. 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 ~`

Hint: the name of your home directory is your username

Exercise 1: Path Names

I'll create a file named `example1.txt` in the directory `data`, which is in the root directory.

You must

1. Specify an absolute path to the file `example1.txt`
 2. Specify a relative path to the file `example1.txt` from your home directory
- `/data/example1.txt` `../../data/example1.txt`
3. Create a file named `example2.txt` in your home directory.
 4. Specify an absolute path to the file `example2.txt`
 5. Specify a relative path to the file `example2.txt` from your home directory

`/home/ajcd2020/example1.txt` `./example1.txt`

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

Hint: the name of your home directory is your username

Basic File System Operations

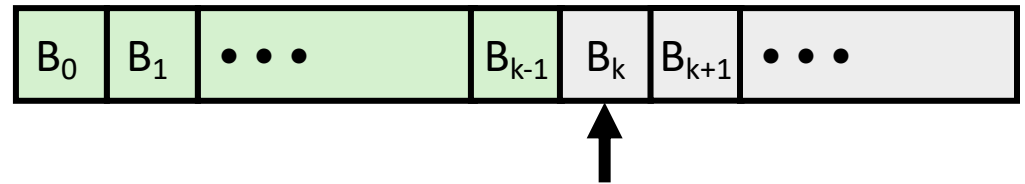
- Create a file
- Delete a file
- Write to a file
- Read from a file
- Seek to somewhere in a file

How should we (the OS) implement this?

Unix I/O Interface

- Elegant mapping of files to devices allows kernel to export simple interface:

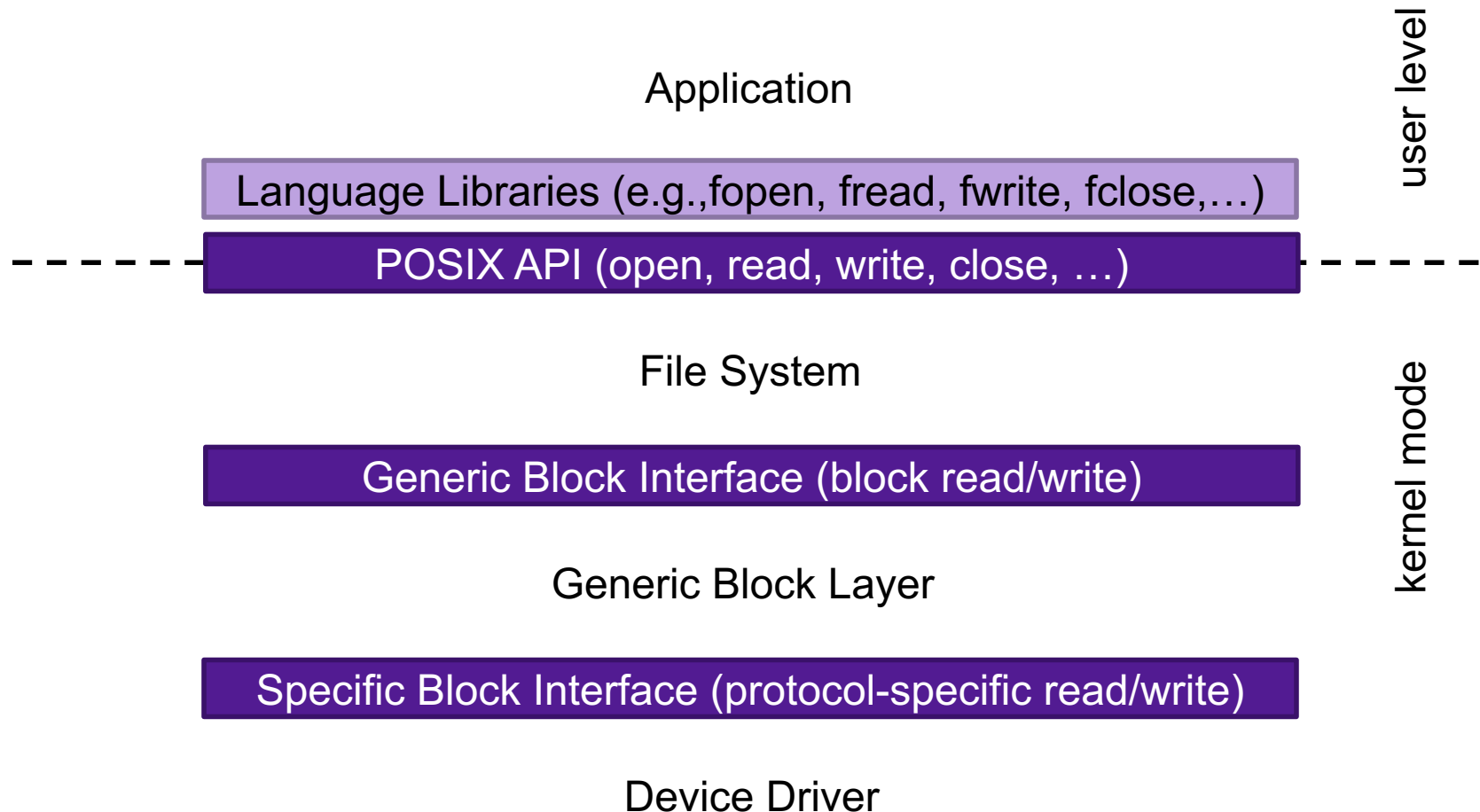
- Opening a file
 - `open()` and `close()`



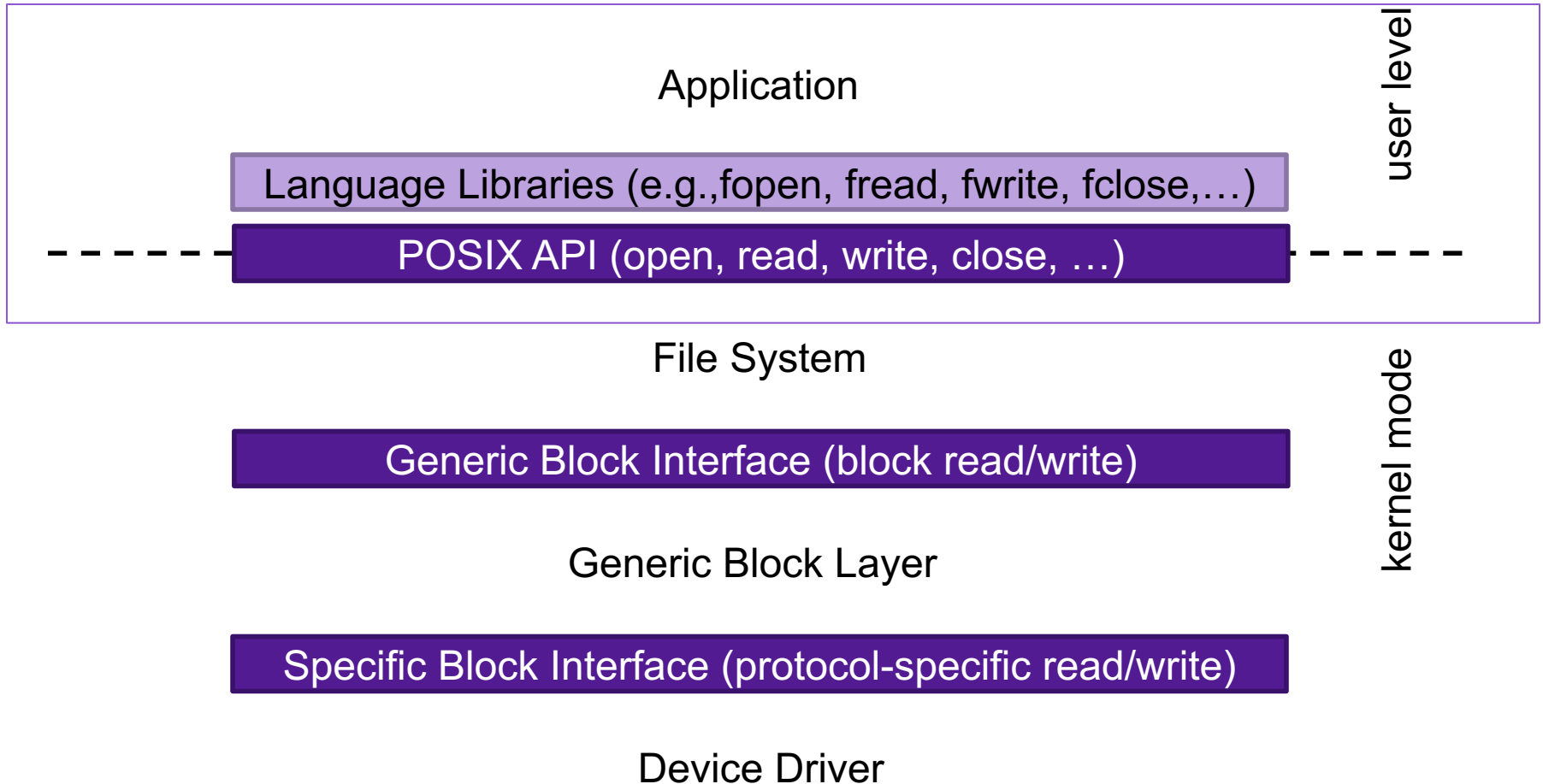
Current file position = k

- 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



The File System Stack



```
strace cat sync.c > tmp.txt
```

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);
}
```

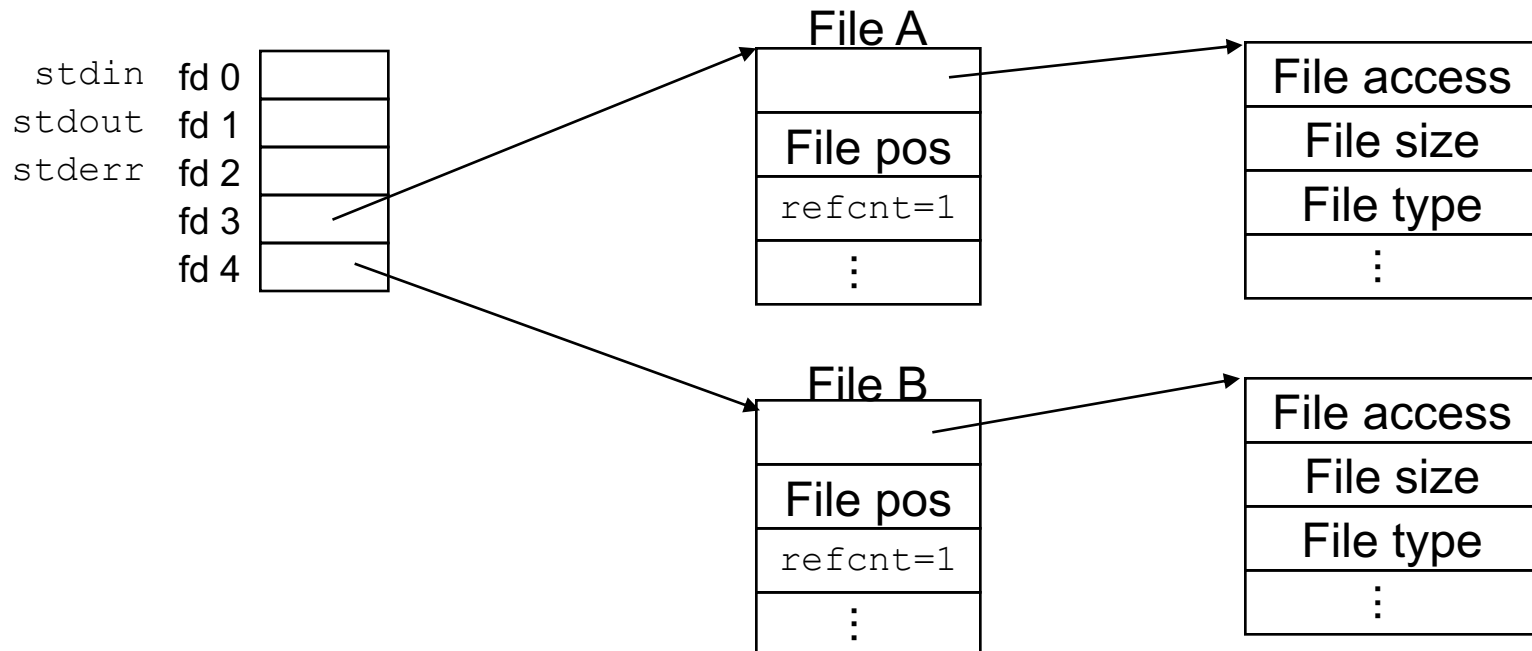
- Returns a small identifying integer **file descriptor**
 - `fd == -1` indicates that an error occurred
- Each process created by a Linux shell begins life with three open files associated with a terminal:
 - 0: standard input (stdin)
 - 1: standard output (stdout)
 - 2: standard error (stderr)

Kernel Data Structures

Descriptor table
(table created on fork(),
one table
per process)

Open file table
(entry created on open,
shared by
all processes)

v-node table
(one per file,
shared by
all processes)



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

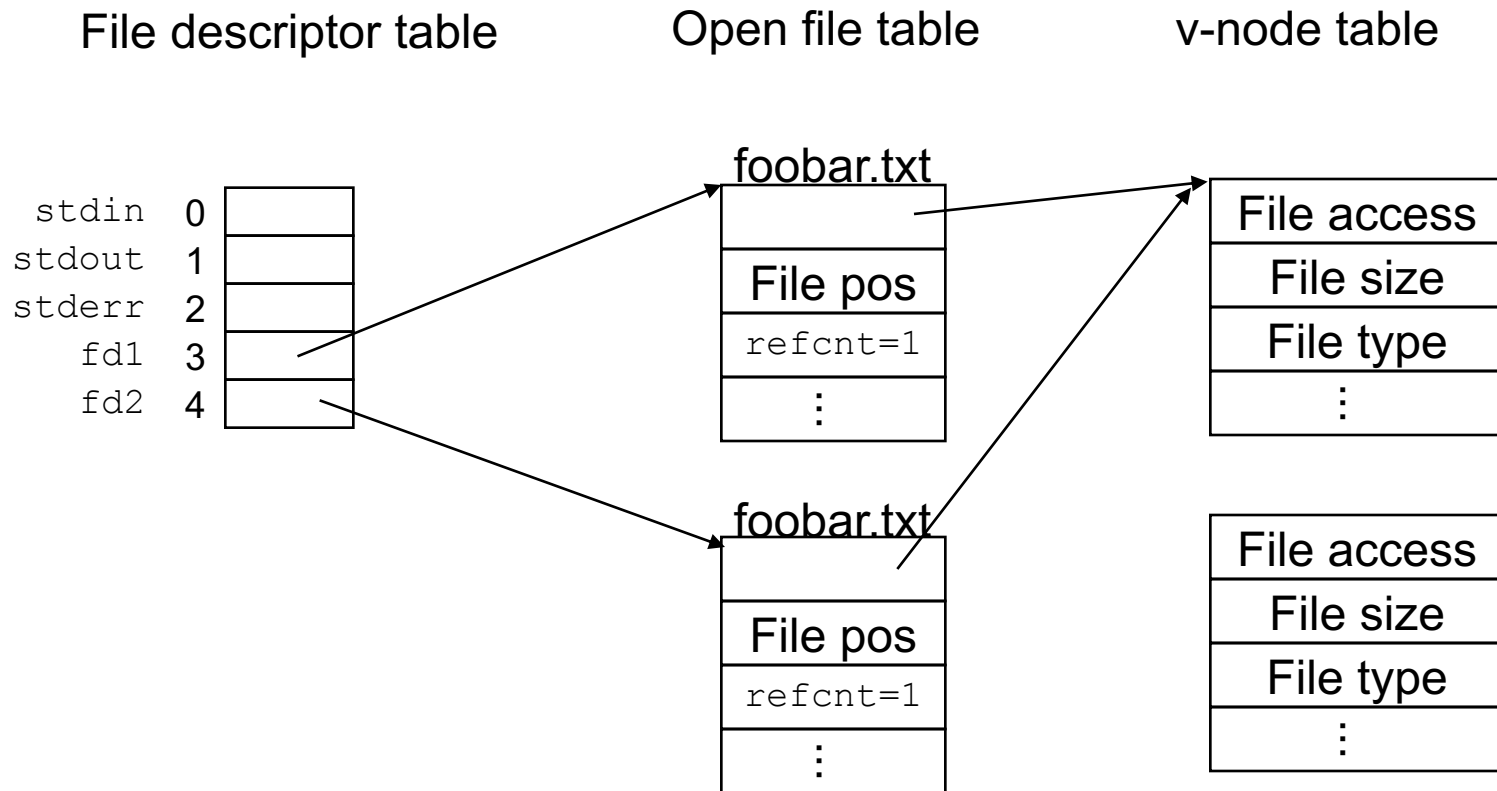
- Stream data is stored in a kernel buffer and returned to the application on request
- Enables same system call interface to handle both streaming reads (e.G., Keyboard) and block reads (e.G., Disk)

Exercise 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 = open("foobar.txt", O_RDONLY);  
    int fd2 = open("foobar.txt", O_RDONLY);  
  
    char c;  
    read(fd1, &c, 1);  
    read(fd2, &c, 1);  
  
    printf("c = %c\n", c);  
  
    return 0;  
}
```

Exercise 2: Reading and Writing



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	

File A

File pos
refcnt=2
⋮

File access
File size
File type
⋮

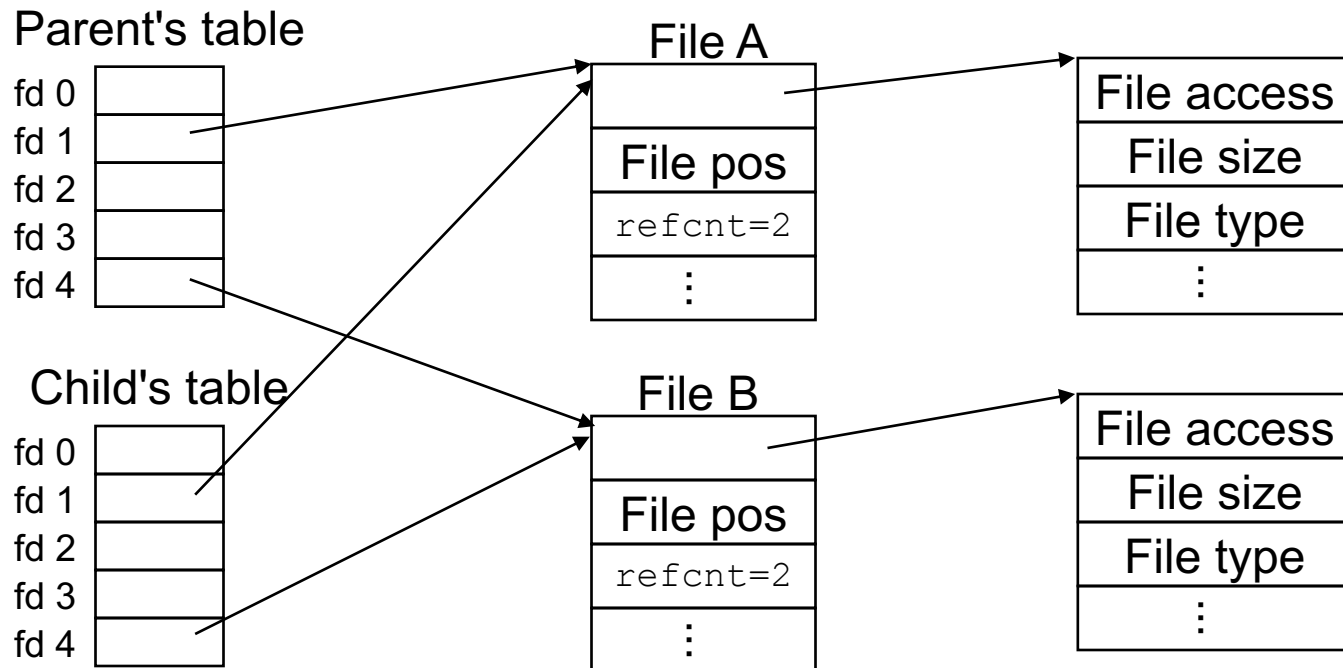
Child's table

fd 0	
fd 1	
fd 2	
fd 3	
fd 4	

File B

File pos
refcnt=2
⋮

File access
File size
File type
⋮



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){
    char c;

    int 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 3: 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	

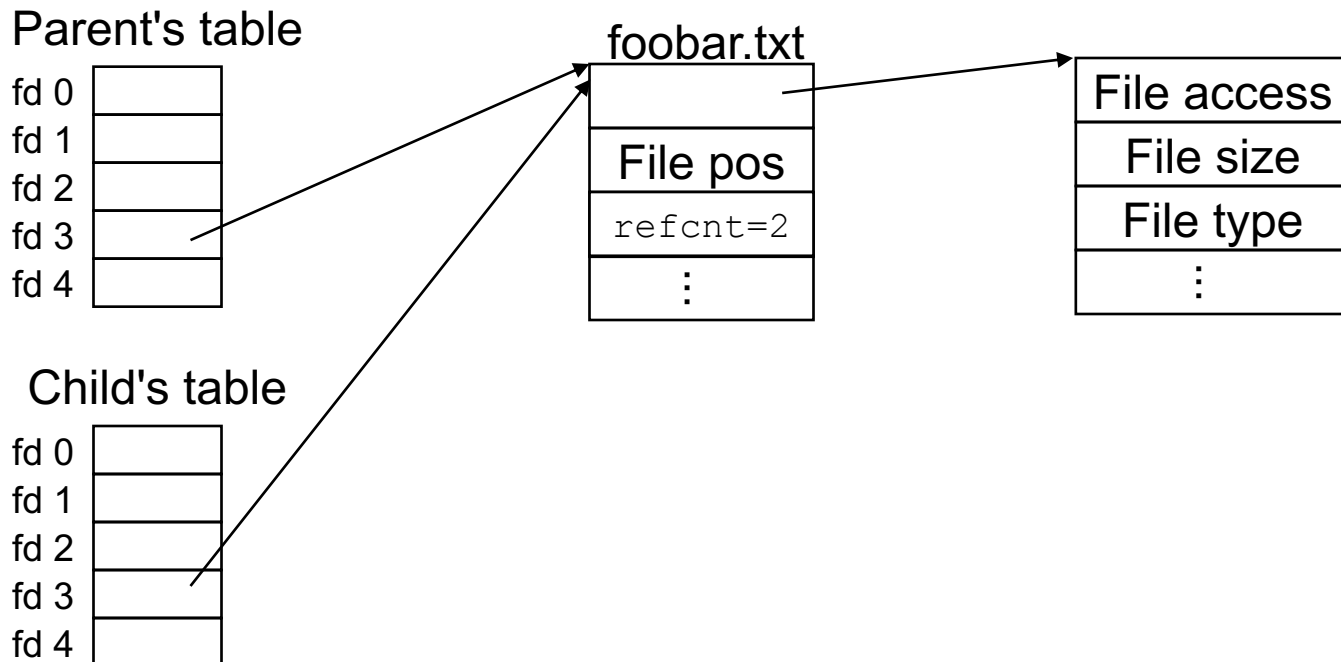
Child's table

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)

fd 0	
fd 1	
fd 2	
fd 3	
fd 4	

Open file table
(shared by
all processes)

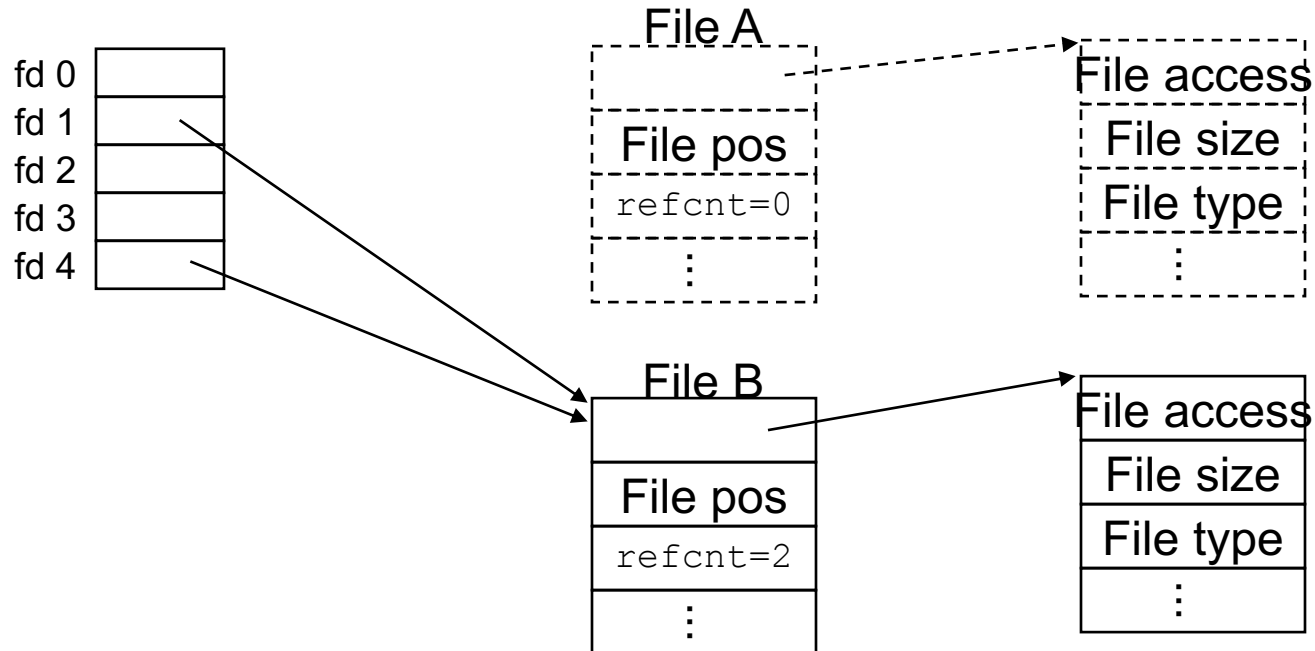
File A
File pos
refcnt=0
⋮

v-node table
(shared by
all processes)

File access
File size
File type
⋮

File B
File pos
refcnt=2
⋮

File access
File size
File type
⋮



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() {
    char c;

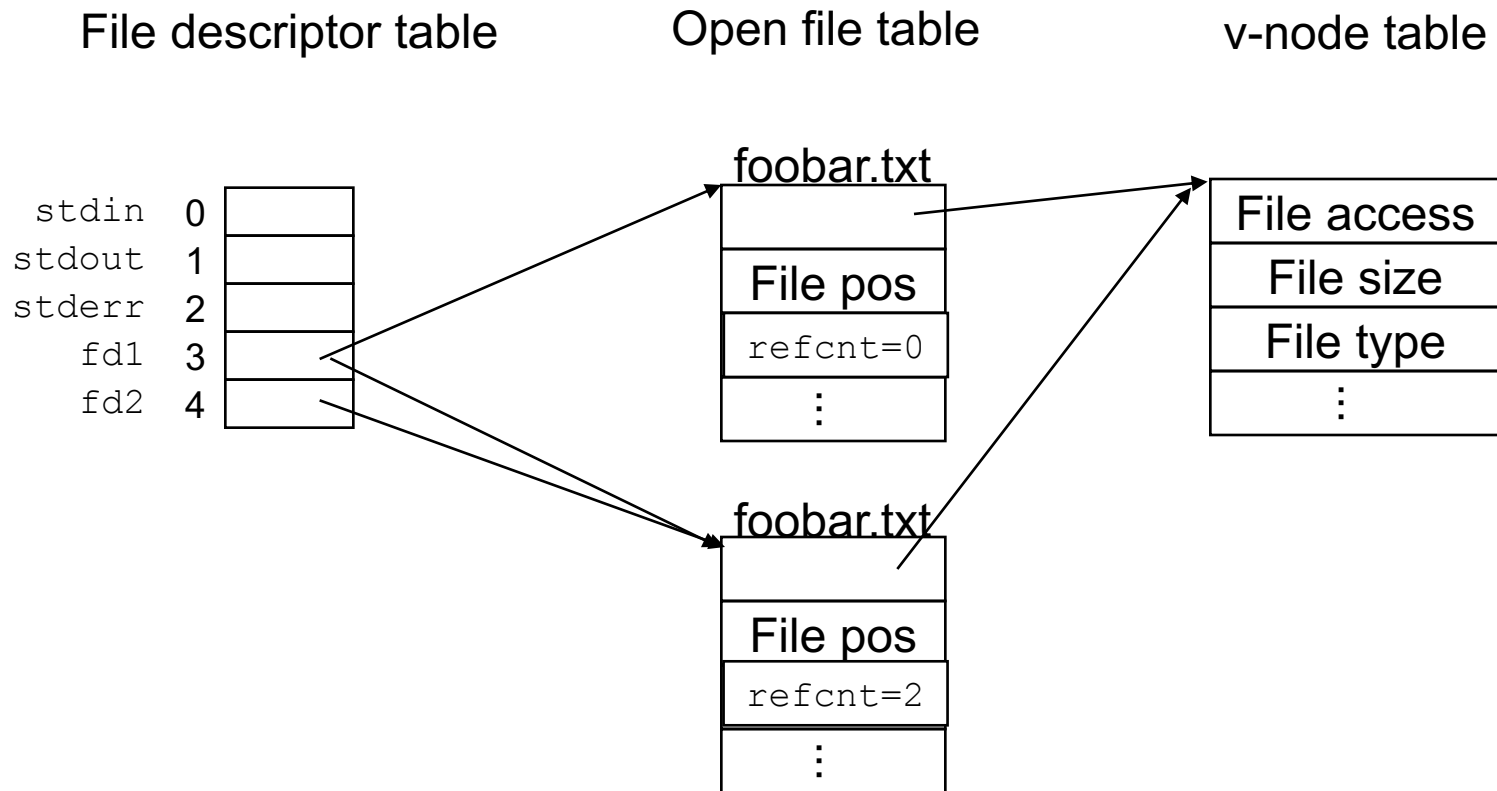
    int fd1 = open("foobar.txt", O_RDONLY);
    int fd2 = open("foobar.txt", O_RDONLY);

    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