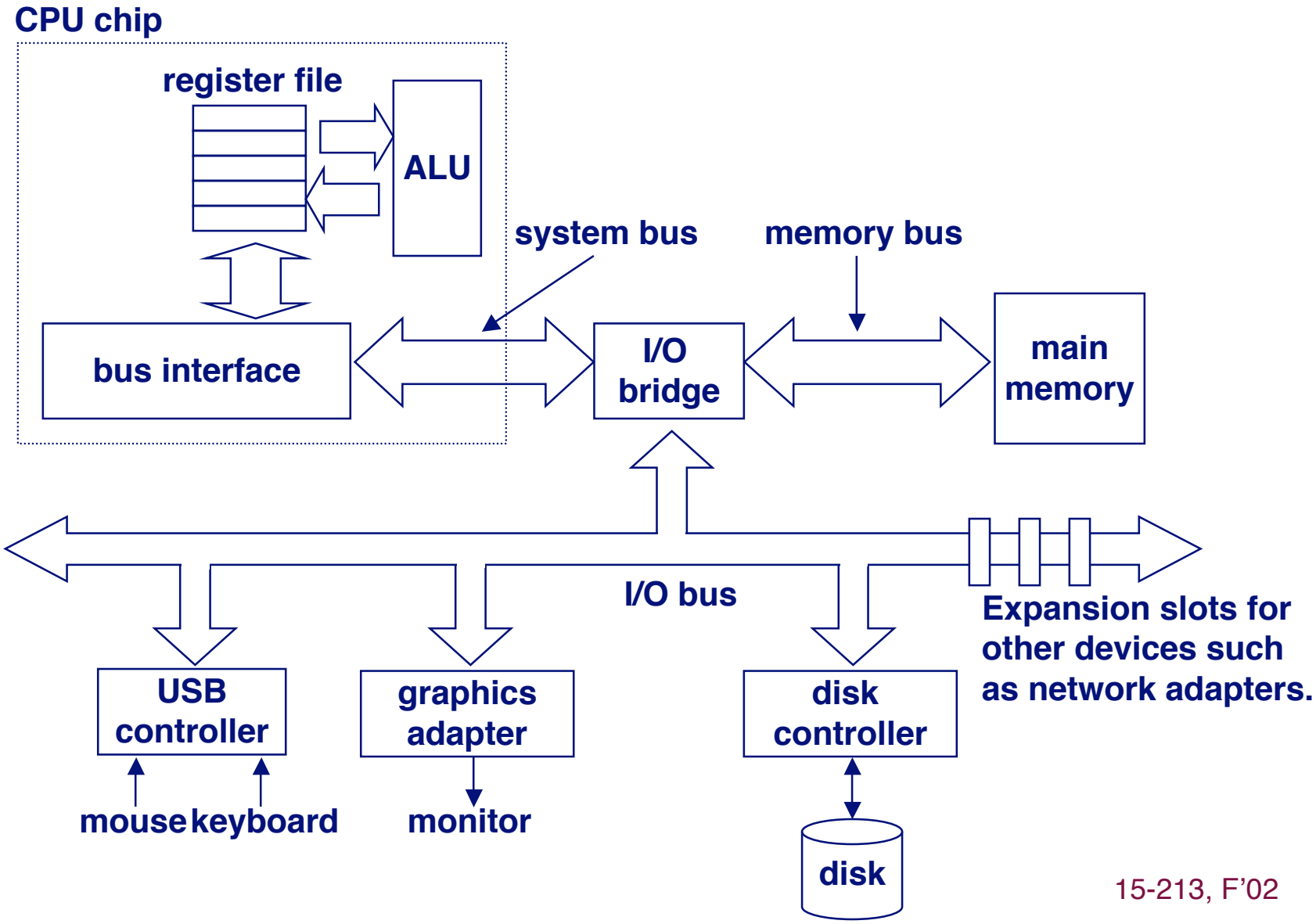
CS 201

Files and I/O

Gerson Robboy
Portland State University

A Typical Hardware System



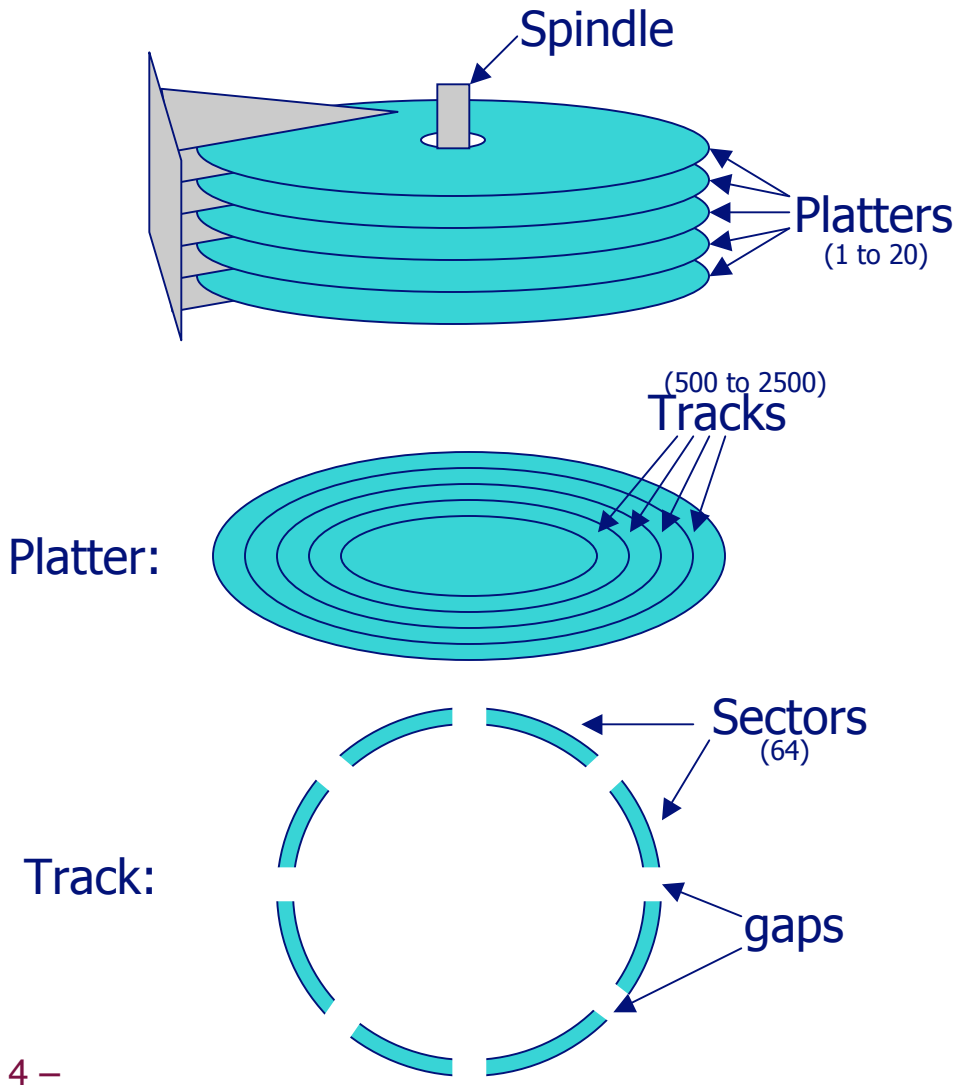
Some typical hardware components

- Hard disk
- CD-ROM
- Printer
- Screen
- Keyboard
- Mouse
- Floppy disk
- Zip drive
- Network controller
- Modem

Each of these has a detailed, unique interface

- Not standardized between different manufacturers

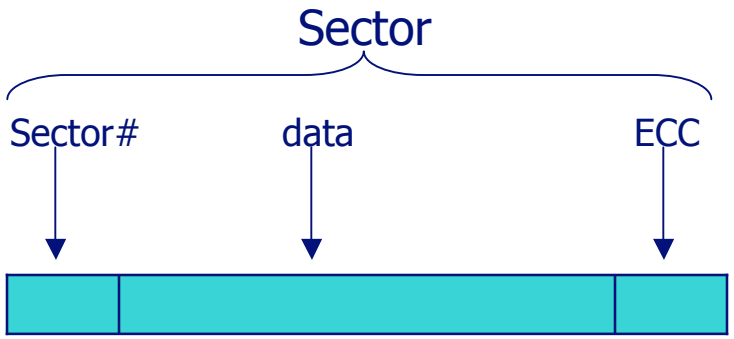
Hard Disk DrivePhysical Layer



We shall stack 1 to 20 metal discs, each side covered with magnetic recording material, and bind them by a central spindle.

They will be spun at 7,200 RPMs.

Seek time, hierarchy of controllers, transfer rate, …



As programmers, we don't want to see all that!

- We want a simple, portable abstraction for I/O
- Device drivers in the operating system take care of the device interfaces
- As application programmers, we want all devices to look alike

And here's another problem

- We don't want to manage data allocation on a disk
  - Nor do we want to know about platters, sectors, tracks, …
- The O. S. file manager allows us to deal with files, not disks.

- What is a file, anyway?
  - A collection of data managed by the O. S.
  - We can open it by name
    - We don't care "where" it is, only its name.
  - We don't have to allocate space for it.
  - We don't have to know what the underlying medium looks like.

Can devices and files be handled by one single abstraction?

Unix Files

A *file* is a named sequence of *m* bytes:

- $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$

We open it by name. Then we see it as a stream of bytes.

- **At the O. S. level, there is no internal structure to a file**

Unix Files - a brief digression

If a file is a sequence of bytes, does that mean files always contain strings of characters?

The declaration of *read* according to K&R:

```
int read(int fd, char *buf, int n);
```

Does “char *buf” mean buf contains ascii characters?

- If not, then why declare it as *char ** ?

Declaration of read in newer versions of Linux:

```
ssize_t read(int fd, void *buf, size_t count);
```

Unix Files

Unix (Bell Labs, late 1960s) introduced two profound innovations regarding files:

- **Almost everything is a file.**
  - One abstraction for accessing most external things, including I/O devices and data stored in files on a disk.
- **A file is a stream of bytes, with no other structure.**
  - No “records”
  - Higher levels of structure are an application concept, not an operating system concept.
  - Libraries are available if you want structured files.

Internally, Unix has two kinds of Files

Regular files

- Data maintained on a storage device such as a hard disk
- The O. S. file manager organizes and retrieves the data

Special files

- All I/O devices are represented as files:
  - `/dev/sda2` (`/usr` disk partition)
  - `/dev/tty2` (terminal)
- Even the kernel is represented as a file:
  - `/dev/kmem` (kernel memory image)
  - `/proc` (kernel data structures for processes)

To us as programmers, they're all just files.

Well, OK, a few more types have crept in

Regular file

- File system on a disk, managed by the O. S.

Directory

- A file that contains the names and locations of other files.
- Normally, user programs don't open and read directories. The "ls" utility is an exception.

Character special and block special files

- Terminals (character special) and disks (block special)

FIFO (named pipe)

- A file type used for interprocess comunication

Socket

- A file type used for network communication between processes

Unix I/O

Key Unix idea: All input and output is handled in a consistent and uniform way.

Basic Unix I/O operations (system calls):

- **Opening and closing files**
  - `open()` and `close()`
- **Changing the *current file position* (seeking)**
  - `lseek()`
- **Reading and writing a file**
  - `read()` and `write()`

Opening Files

```
#include <syscalls.h>
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` ➔ **an error occurred**

Each process created by a Unix shell begins life with three open files (normally associated with a terminal):

- **0: standard input**
- **1: standard output**
- **2: standard error**

Opening Files

The second argument is a constant defined in syscalls.h

There are a zillion values for it. Some important ones:

- O_CREAT: Create a new file if it doesn't exist
- O_RDRW: Open for reading and writing
- O_RDONLY: Open for reading only
- O_WRONLY: Open for writing only
- O_TRUNC: Truncate the file to length zero
  - If the file existed and had data, that data will disappear for all users.

Opening files that don't already exist

```
#include <syscalls.h>
int fd;   /* file descriptor */

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

The third argument sets permissions

- In this example, read, write, and execute for user
- The argument is required if creating a new file, else ignored

What if you leave this argument out?

Opening Files

Why do you have to check for errors after opening a file?

What kinds of things can make it fail?

Writing Files

```
#include <syscalls.h>
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.**

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

if ((fd = open("some_file", O_RDWR)) < 0) {
   perror("open");
   exit(1);
}
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

What is the current file position?

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);
}
```

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

- `nbytes < 0` ➔ an error occurred.
- *short counts* (`nbytes < sizeof(buf)`) are possible and are not errors!

Questions

In Unix, using the *read()* system call, if you read past the end of a file, what do you get?

How do you know you've gone past the end of the file?

What if you make system calls and you don't include syscalls.h or stdio.h?

- Will the compiler give you an error message?
- Will your code be correct?

Seeking

lseek() sets the position in the file where the next read or write will occur.

```
#include <syscalls.h>
char buf[512];
int fd;         /* file descriptor */
int offset, nbytes;


. . .
/* seek to offset in the file */
if (lseek(fd, offset, SEEK_SET) < 0){
       /* handle error */
}
/* Then write from buf to file at that offset */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

Seeking

The last argument is a constant defined in syscalls.h.

Values:

- SEEK_SET: The offset is set to *offset* bytes
- SEEK_CUR: The offset is set to its current location plus *offset* bytes
- SEEK_END: The offset is set to the end of the file plus *offset* bytes

What does SEEK_END do to the file?

- Does it change the size of the file?
- What if you write out past the end of the file, leaving a gap?
  - What is the size of the file after you write?
  - What happens if you read within the gap?

Exercise

Suppose a file called “fpfile” contains an array of floating point values (of type *float* in C).

Write a few lines of code to open the file, seek to the fifth element of the array of floats, and read that one element into a variable in memory.

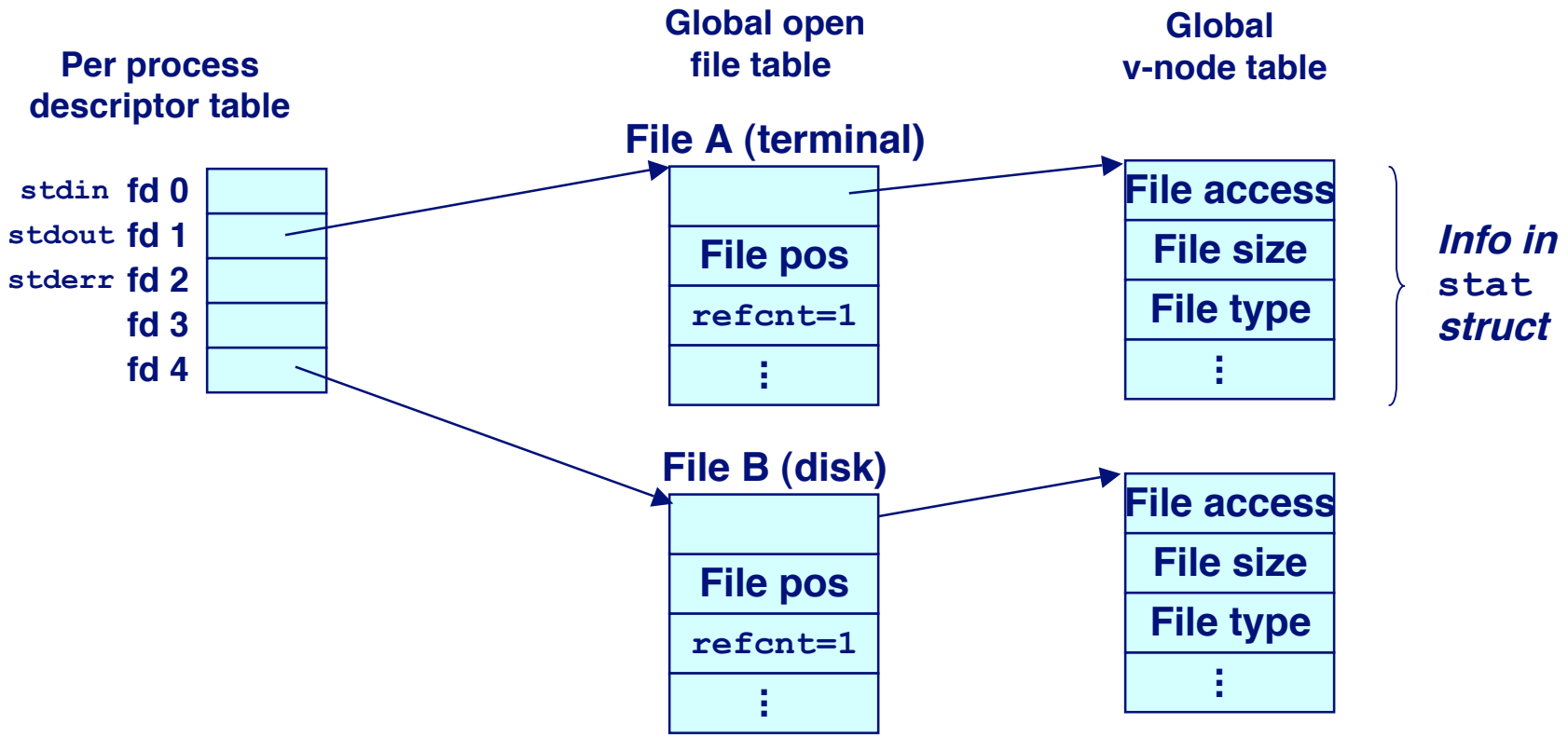
Never mind about main() or #include…

Just write a few lines out of the middle of an imaginary program.

Don’t forget to check for errors!

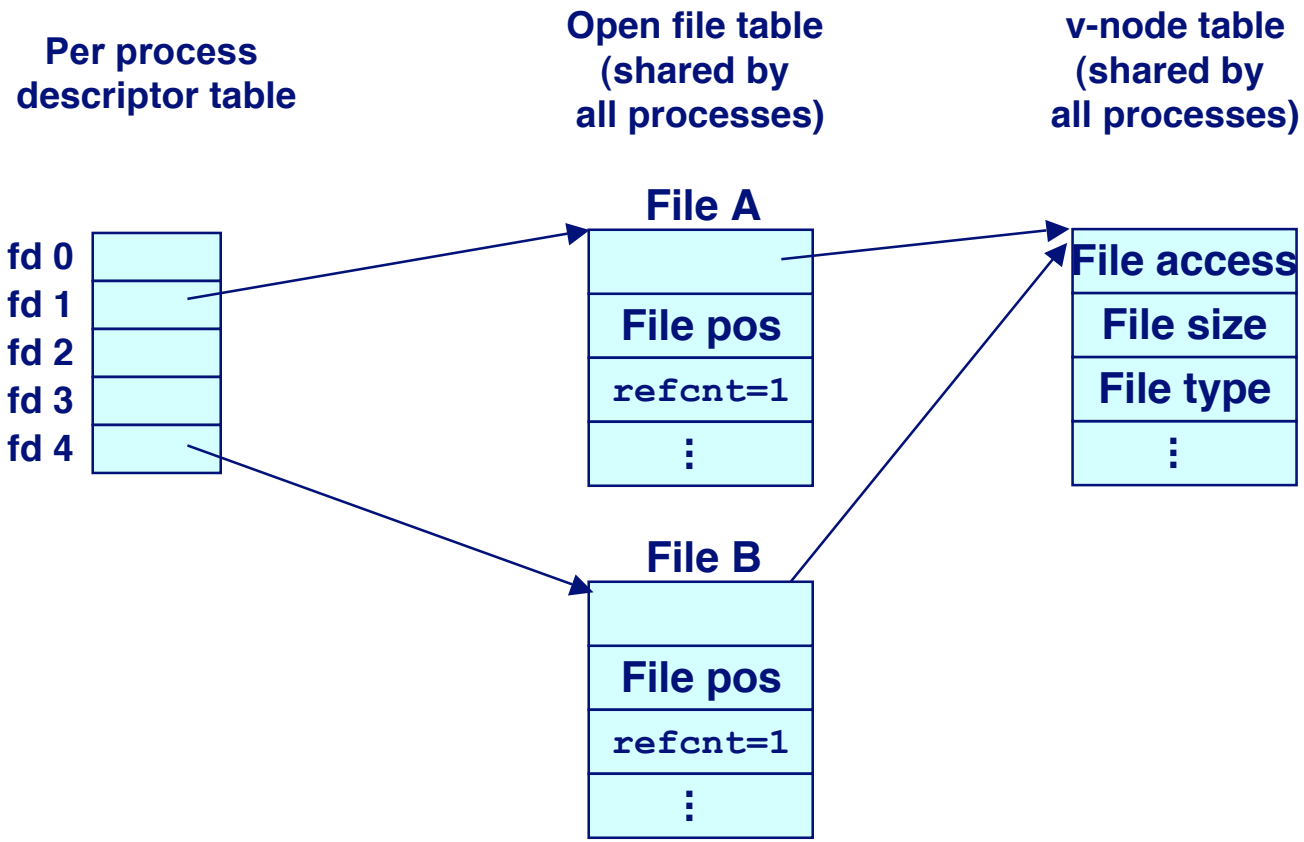
Open File Representation

Two descriptors referencing two distinct open disk files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file.



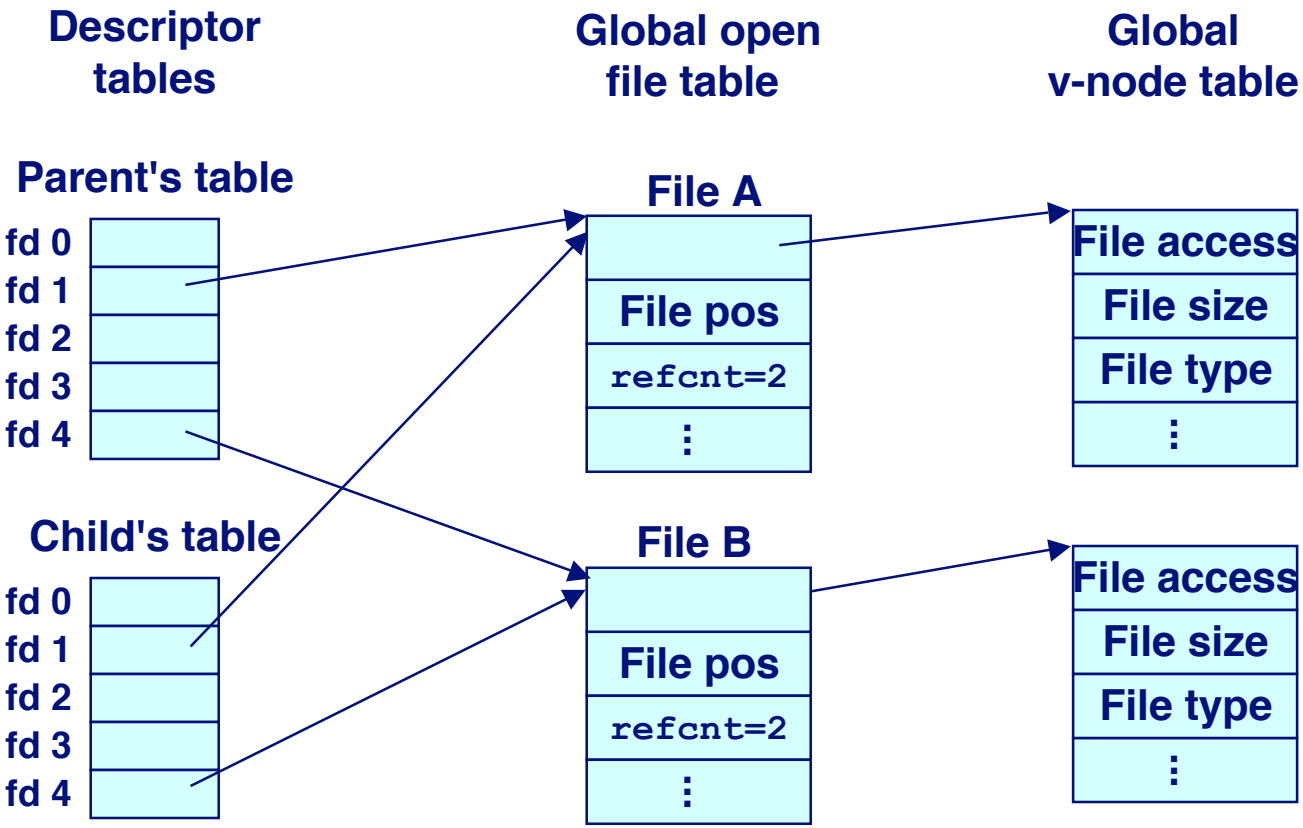
File Sharing

Calling open twice with the same filename argument



File Sharing Between Processes

A child process inherits its parent's open files. Here is the situation immediately after a fork



Exercise

Suppose a disk file called "foobar" contains 6 characters "foobar" What is the output of this program?

```
#include <syscalls.h>
main()
{
    int fd;         /* file descriptor */
    char c;

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

Exercise

Suppose a disk file called "foobar" contains 6 characters "foobar" What is the output of this program?

```
#include <syscalls.h>
main()
{
    int fd;         /* file descriptor */
    char c;

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

dup2 - duplicate an open file descriptor

```
#include <unistd.h>

    int dup2(int fildes, int fildes2);
```

The dup2() function causes the file descriptor fildes2 to refer to the same file as fildes. The fildes argument is a file descriptor referring to an open file, and fildes2 is a non-negative integer less than the current value for the maximum number of open file descriptors allowed the calling process. If fildes2 already refers to an open file, not fildes, it is closed first.

- **Do not panic. Continue breathing normally. We will explore what this means**

Exercise

Write some code to redirect the standard input to a file called “infile.”

That is, the process executing this code will get its standard input from “infile” instead of wherever it came from before.

Sharing Files

Files are inherently shared.

- If process x writes data to a file and process y has the file open, then process y can read the data

Sharing can be avoided

- locks on files
- file permissions

Sharing Files, continued

Suppose two processes write different data to the same offset in the same file? What happens?

This is perfectly legal.

- **It's exactly analogous to shared memory.**
- **If you don't want this to happen, it's up to you (the application programmer) to avoid it.**

The data in a file is *consistent* in this sense:

All processes that read the file will see the same data

File Metadata

The `stat` and `fstat` functions give us information about files.

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection and file type */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

Example of Accessing File Metadata

```
/* statcheck.c - Querying and manipulating a file's meta data */
#include "csapp.h"

int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

    Stat(argv[1], &stat);
    if (S_ISREG(stat.st_mode)) /* file type*/
        type = "regular";
    else if (S_ISDIR(stat.st_mode))
        type = "directory";
    else
        type = "other";
    if ((stat.st_mode & S_IRUSR)) /* OK to read?*/
        readok = "yes";
    else
        readok = "no";

    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
```

```
bass> ./statcheck statcheck.c
type: regular, read: yes
bass> chmod 000 statcheck.c
bass> ./statcheck statcheck.c
type: regular, read: no
```

What about files that aren't ascii strings?

Here's a program to copy a file containing ascii strings from stdin to stdout:

```
#include <stdio.h>
#define MAXLINE (32 * 1024)
#define MAXBUF (4096)
main()
{
  int n;
  char buf[MAXLINE];
  while (fgets(buf, MAXLINE-1, stdin) != 0){
   fputs(buf, stdout);
  }
}
```

Exercise: Change the program to copy MAXBUF bytes at a time.

Hint: these function declarations may come in handy:

```
size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream);

size_t fwrite(void *ptr, size_t size, size_t nobj, FILE *stream);
```