HARVARD
School of Engineering
and Applied Sciences

Files and I/O

CS61, Lecture 22

Prof. Stephen Chong

November 17, 2011

Announcements

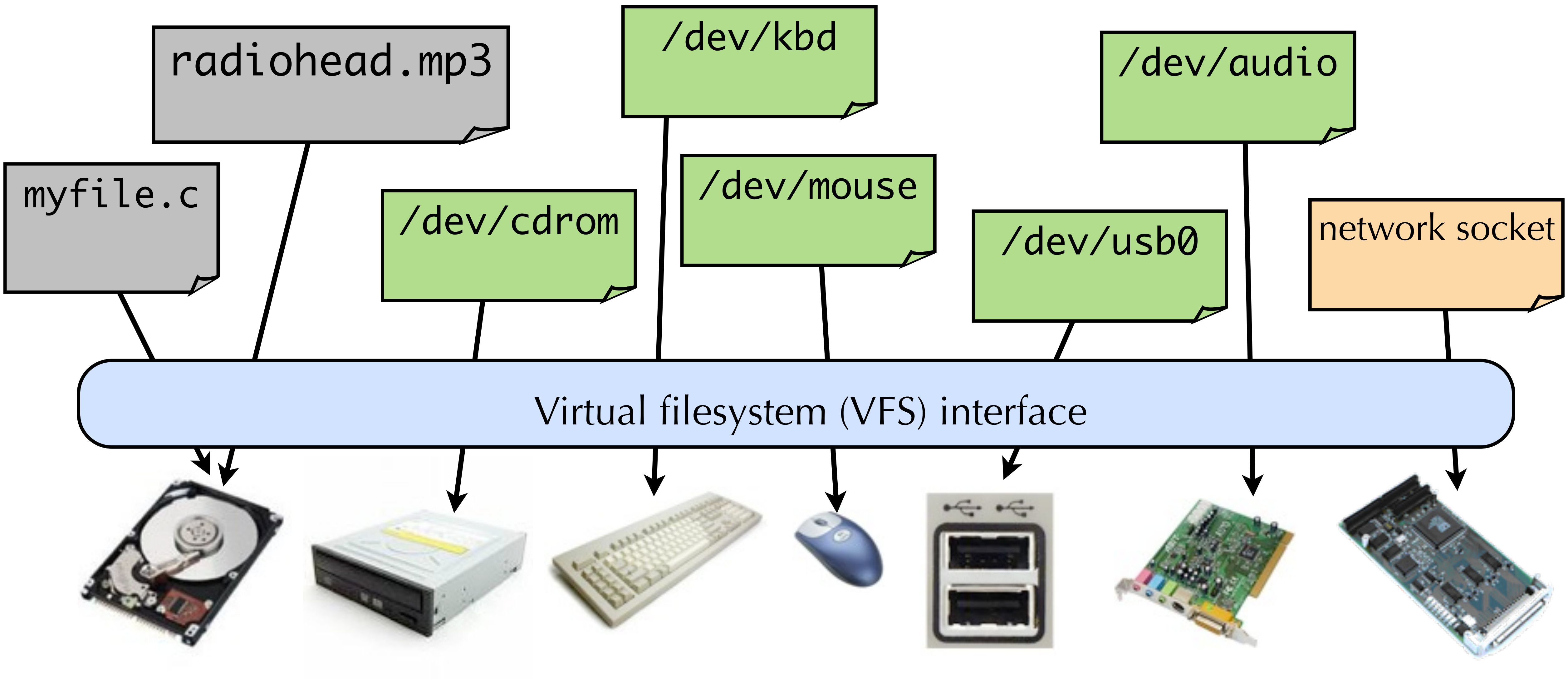
- Assignment 5 (Bank) due today
- New late day request procedure
  - Fill out the form at http://tinyurl.com/CS61-fa11-latedays to request or change late days
- Assignment 6 (Shell) will be released later today
  - Due Tuesday Dec 6
- Thanksgiving: no class Thursday November 24
  - But there **will** be sections next week
- Final exam
  - In class on Thursday Dec 1
  - May cover material from entire course (up to lecture 23: Network programming)
  - Will focus on material not covered in midterm (lecture 15 onwards)
  - Will release practice exams soon

Today

- The UNIX file abstraction
- UNIX low-level I/O interfaces
- Robust I/O
- Buffered I/O
- Standard I/O
- Accessing metadata and directories
- Fun with filehandles
- Pipes
- Summary

The UNIX File Abstraction

- In UNIX, the **file** is the basic abstraction used for I/O
  - Used to access disks, CDs, DVDs, USB and serial devices, network sockets, even memory!



The UNIX File Abstraction

- A file appears to the application as an **ordered sequence of bytes**.
  - No internal structure (such as records, header, footer, etc.)
  - Of course, most files do have such structure, but OS doesn't need to know about it.
- Basic operations on files:
- `int open(char *filename, int flags)`
  - Opens the given file, using the (optional) flags, and returns a **filehandle**
  - The filehandle is an integer that can be used for all future operations on the file.
- `int close(int filehandle)`
  - Closes the file, releasing the filehandle (which may be reused by a future open() call)
- `size_t read(int filehandle, char *buf, size_t num)`
  - Reads up to num bytes from the file into buf and returns number of bytes read
- `size_t write(int filehandle, char *buf, size_t num)`
  - Writes up to num bytes to the file from buf and returns number of bytes written

Virtual Filesystem Interface

- VFS maps open, close, read, write, etc. operations onto corresponding hardware operations
  - Makes "everything look like a file"
  - Program can read from a disk file, DVD, audio card, or serial device in the same way!

```
$ cat greetings.txt
G'day world!

$ cat /dev/disk0
¬&ÿÿÿîÿÿÿ/"¥ UªEFPART\LWb¼/"¥""¥Ç*OÞ{ìQ€€hrm(s*ÁøÒ°K É>É;å|üOXwOD('@EFI
system....

$ cat /dev/random
ù9ÁþœiëÕÝ0ô"¾²)'vhVi¾¤{ v•½³\6ÃsÚ¥´HË–©O"dÉÇ©œ+Óüaµö,t¢/
b˜h×¶§_ZQ{kQ1Â8"á–Aœã´&î£--Oy& ¥bé=@Ö±V„§'¦{ô¯ic·Jn_·ÇÄ[›ùS
ç'›i„!ó,Ñ=ký( x°7áLêÅò[Á°áv(ôXC?–Å@9VÖÔåÊùö-ÿíxÈ*ÙýÌ¢IPÁ½rý¸ë½#Ò‰gônu%ÅN
$ž;ËÌ€Ã@yÅ:Rp|` @8'_ÆN¥˜°äÕ-ÚxØp•ge6Žš-¦'ØovzB×óh5ß´$-€0Î"š.i-/ð€½
¥5×"àGj°–Ás
®q«Û¿ÄþiµëK<,^<Kšœ~ÊC÷ë9#…$ÕÆ¶l&}µÔÃåçì–ø£PÛØ    0p´è#ÇÞ°Va³œ'
```

Reading Files

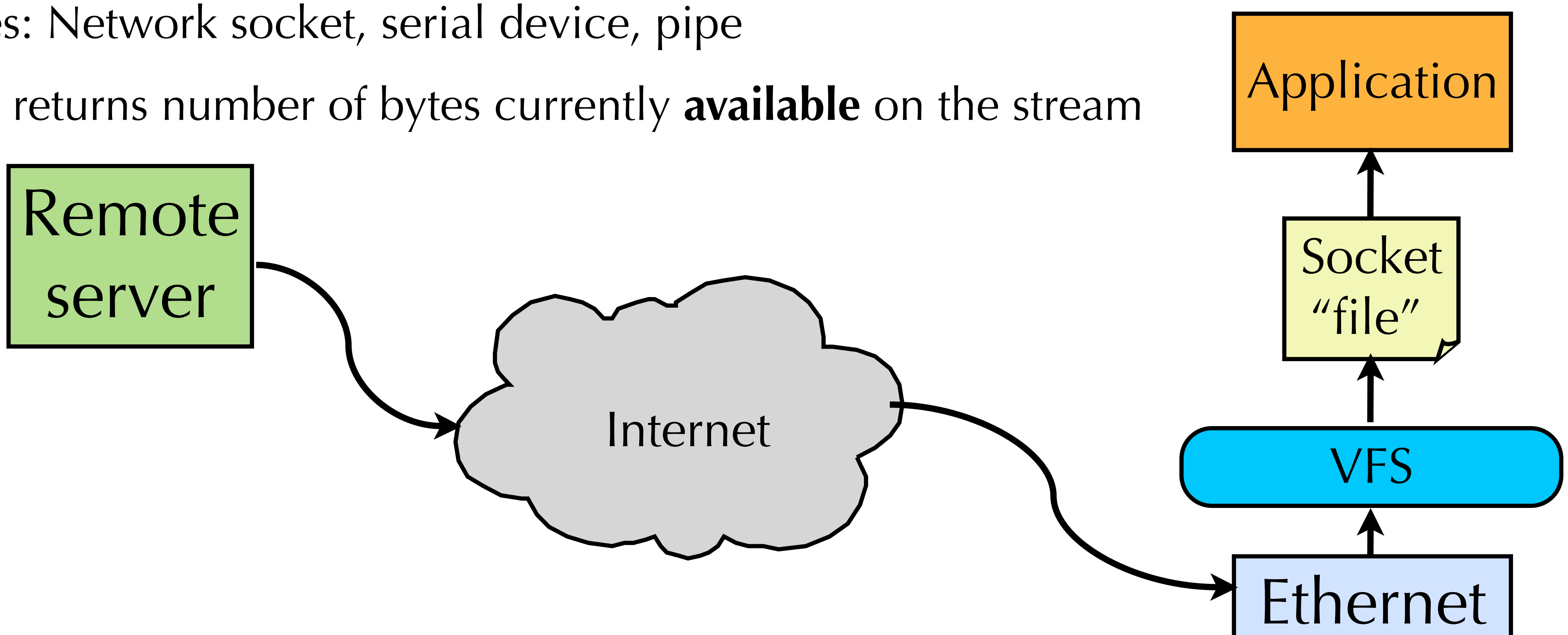
- Reading a file copies bytes from the current file position to memory, and then updates file position.

```
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!!

Why would you get a short read?

- Under what conditions could `read` return fewer than number of requested bytes?
- 1) Reading up to the end of a file
  - If file is 100 bytes in size, and you try to read 512 bytes, you'll only get 100...
- 2) Reading from a **stream**
  - In UNIX, byte streams are still treated as “files”
  - Examples: Network socket, serial device, pipe
  - `read()` returns number of bytes currently **available** on the stream



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!

stdin, stdout, stderr

- In UNIX, every process has three “special” files already open:
  - standard input (**stdin**) – filehandle 0
  - standard output (**stdout**) – filehandle 1
  - standard error (**stderr**) – filehandle 2
- By default, stdin and stdout are connected to the **terminal** device of the process.
  - Originally, terminals were physically connected to the computer by a serial line
  - These days, we use “virtual terminals” using ssh

VT100 terminal

Unix I/O Example

- Copying standard input to standard output one byte at a time.

```
#define STDIN_FILENO 0
#define STDOUT_FILENO 1

int main(void)
{
    char c;

    while (read(STDIN_FILENO, &c, 1) != 0) {
       if (write(STDOUT_FILENO, &c, 1) < 0) {
           /* Error! */
           exit(1);
       }
    }
    exit(0);
}
```

What's wrong with this code ?

Always check return codes!

```
while (read(STDIN_FILENO, &c, 1) != 0) {
   if (write(STDOUT_FILENO, &c, 1) < 0) {
       /* Error! */
       exit(1);
   }
}
exit(0);
```

```
while (read(STDIN_FILENO, &c, 1) > 0) {
   if (write(STDOUT_FILENO, &c, 1) < 0) {
       /* Error! */
       exit(1);
   }
}
exit(0);
```

- Wrappers can help immensely!

```
ssize_t Read(int fd, void *buf, size_t count) {
  ssize_t n = read(fd, buf, count);
  if (n < 0) exit(1);
  return n;
}
```

  - Textbook uses this pattern, standard functions with an initial capital are wrappers that check error conditions.

Today

- The UNIX file abstraction
- UNIX low-level I/O interfaces
- Robust I/O
- Buffered I/O
- Standard I/O
- Accessing metadata and directories
- Fun with filehandles
- Pipes
- Summary

UNIX I/O is a pain.

- `read()` and `write()` don't guarantee you read or write as much as you're asking for.
  - Can get short counts in both cases.
- Both `read()` and `write()` can be interrupted by a signal
  - Example: Hitting Ctrl-C at a terminal sends a “SIGINTR” signal
  - Have to deal with the special case in your code.
- Must check for errors each time you do an I/O.
  - Makes your code messy and harder to read.
- Solution: Wrappers for UNIX I/O routines to make your life simpler.
  - The **RIO** (Robust I/O) package is one example.
  - Download from the CS61 “Resources” page

RIO Input and Output

- `ssize_t rio_readn(int fh, char *buf, size_t num);`
    - Same interface as `read()`, but with different behavior.
    - Always reads `num` bytes, unless error (`-1`) or end-of-file.
    - When reading from a stream, won't return until `num` bytes read (or EOF).
    - Returns number of bytes actually read, or `-1` if error.
- `ssize_t rio_writen(int fh, char *buf, size_t num);`
    - Always writes `num` bytes.
    - Returns `num`, or -1 if error.

Implementation of rio_readn

```
/*
 * rio_readn - robustly read n bytes (unbuffered)
 */
ssize_t rio_readn(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nread;
    char *bufp = usrbuf;

    while (nleft > 0) {
        if ((nread = read(fd, bufp, nleft)) < 0) {
            if (errno == EINTR) /* interrupted by signal */
                nread = 0;      /* retry the read() */
            else
                return -1;      /* error */
        }
        else if (nread == 0)
            break;              /* EOF */
        nleft -= nread;
        bufp += nread;
    }
    return (n - nleft);         /* return >= 0 */
}
```

UNIX I/O is slow.

- `read()` and `write()` are **system calls**: Require calling into the OS for each I/O operation.
  - Turns out that system calls have very high overhead: 1000s of clock cycles.
  - *n* calls to `read(fd, &s, 1)` costs about *n* times calling `read(fd, &s, n)`
- Solution: **Buffering**
  - Call `read()` once to fill in a whole buffer full of data
  - Application can then grab bytes directly from the buffer
  - When the buffer starts to run empty, call `read()` again to fill it up
- Likewise, you can buffer writes...
  - Fill up a buffer full of data you'd like to write
  - Call `write()` once on the whole buffer, rather than a bunch of individual calls.
- Buffering amortizes the cost of `read()` and `write()` across multiple I/O operations.

Today

- The UNIX file abstraction
- UNIX low-level I/O interfaces
- Robust I/O
- Buffered I/O
- Standard I/O
- Accessing metadata and directories
- Fun with filehandles
- Pipes
- Summary

Standard I/O library

- The C standard library (`libc.a`) contains a collection of higher-level **standard I/O** functions
  - Like RIO, are wrappers to the lower-level UNIX system calls.
  - In addition to other features, these routines perform buffering.
  - These routines are described in `<stdio.h>`
- Examples of standard I/O functions:
  - Opening and closing files (`fopen` and `fclose`)
  - Reading and writing bytes (`fread` and `fwrite`)
  - Reading and writing text lines (`fgets` and `fputs`)
  - Formatted reading and writing (`fscanf` and `fprintf`)

Standard I/O file access

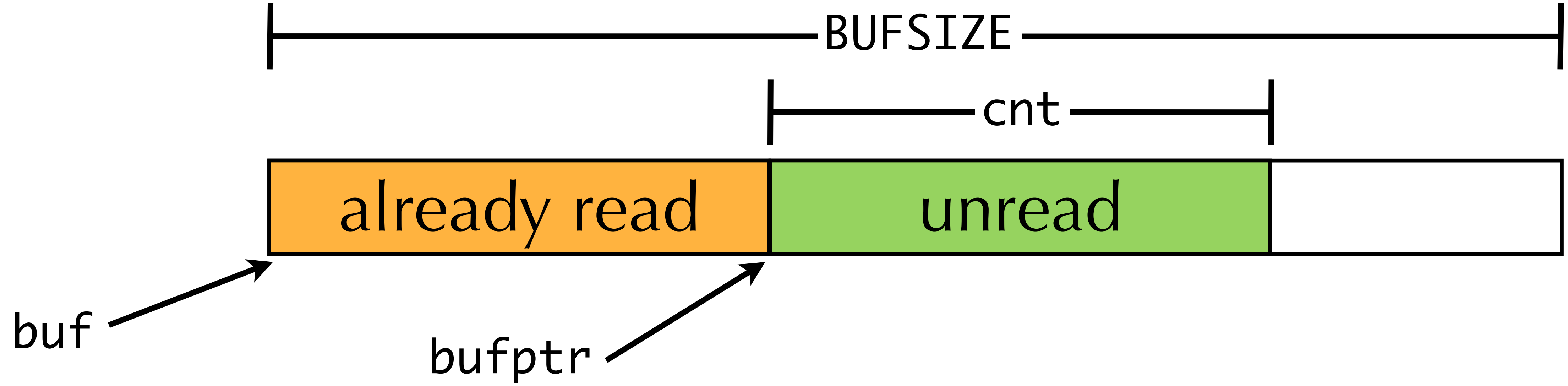
- `FILE*` represents a file in the stdio routines.

```
#include <stdio.h>

void myfunc() {

  FILE* myfile;
  myfile = fopen("somefile.txt", "w");
  if (myfile == NULL) {
     printf("Cannot open somefile.txt!\n");
     exit(1);
  }
  fprintf(myfile, "This is wicked awesome.\n");
  fclose(myfile);

}
```

Buffered I/O implementation

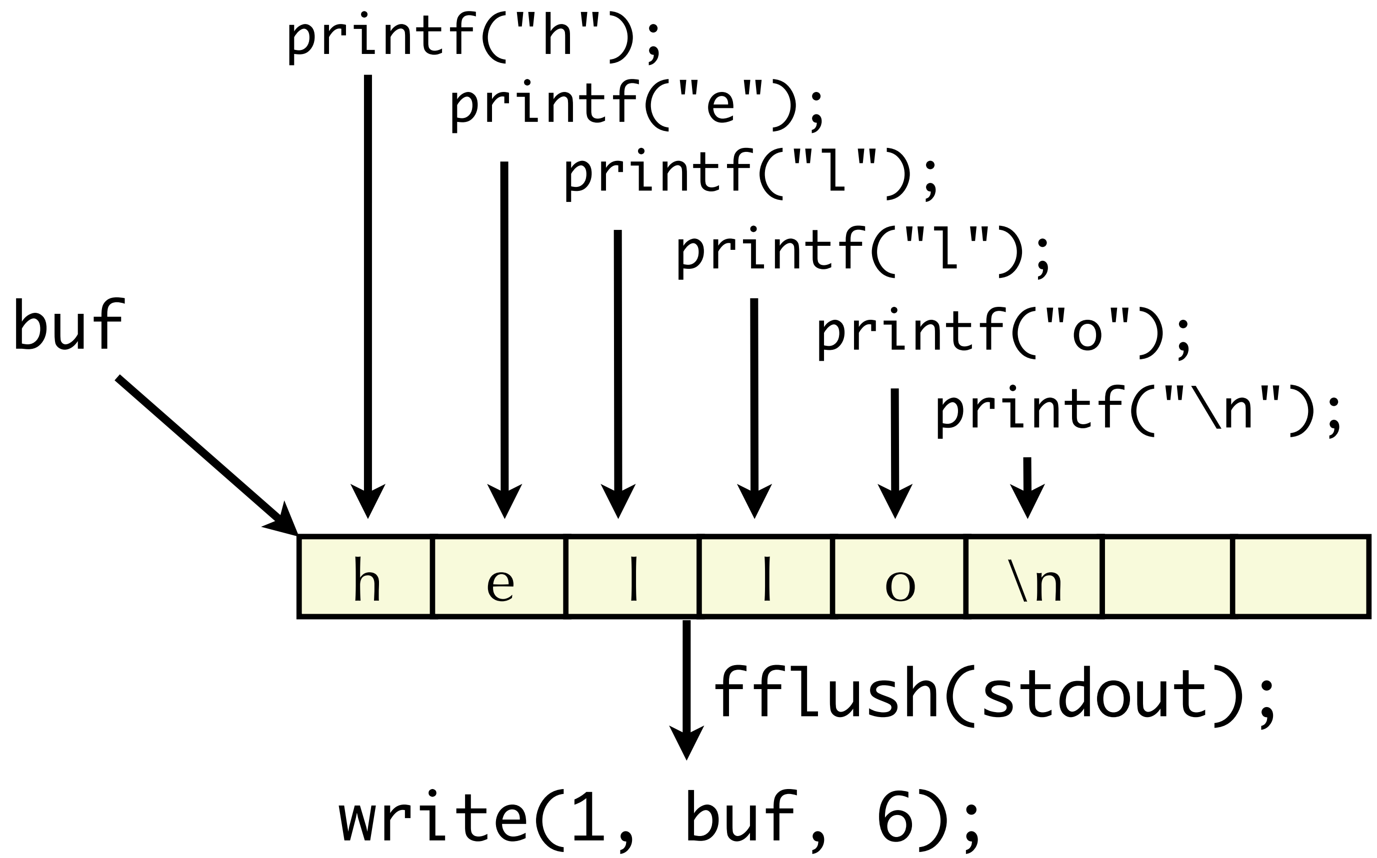
- `FILE*` maintains a buffer to hold bytes that have been read from file but not yet read by user code

```
typedef struct {
    int fd;                /* descriptor for this internal buf */
    int cnt;               /* unread bytes in internal buf */
    char *bufptr;          /* next unread byte in internal buf */
    char buf[BUFSIZE];     /* internal buffer */
} FILE;
```



Buffering in Standard I/O

- stdio routines only call `read()` or `write()` when necessary
  - When buffer is empty on a `fread()` or `fscanf()`
  - When buffer is full on a `fwrite()` or `fprintf()`
  - When application calls `fflush()` to explicitly flush buffer to OS



Today

- The UNIX file abstraction
- UNIX low-level I/O interfaces
- Robust I/O
- Buffered I/O
- Standard I/O
- Accessing metadata and directories
- Fun with filehandles
- Pipes
- Summary

Accessing file metadata

- Use the `stat()` and `fstat()` system calls to access metadata about files
  - Owner, size, permissions, etc.

```
/* Metadata returned by the stat and fstat system calls */
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 file created */
};
```

Accessing Directories

- Directories are just files, but have a special format understood by the OS.
  - Should not attempt to directly modify a directory – in fact, the OS won't let you open a directory for writing! (`open()` syscall returns an error.)
- Rather, use `opendir()` and `readdir()` calls
  - struct `dirent` contains info about each entry in the directory

```
#include <dirent.h>

void myfunc() {
  DIR *directory;
  struct dirent *de;
  ...
  if (!(directory = opendir(“mydir”)))
      error("Failed to open mydir");
  ...
  while (0 != (de = readdir(directory))) {
      printf("File name %s\n", de->d_name);
  }
  ...
  closedir(directory);
}
```

Modifying directories

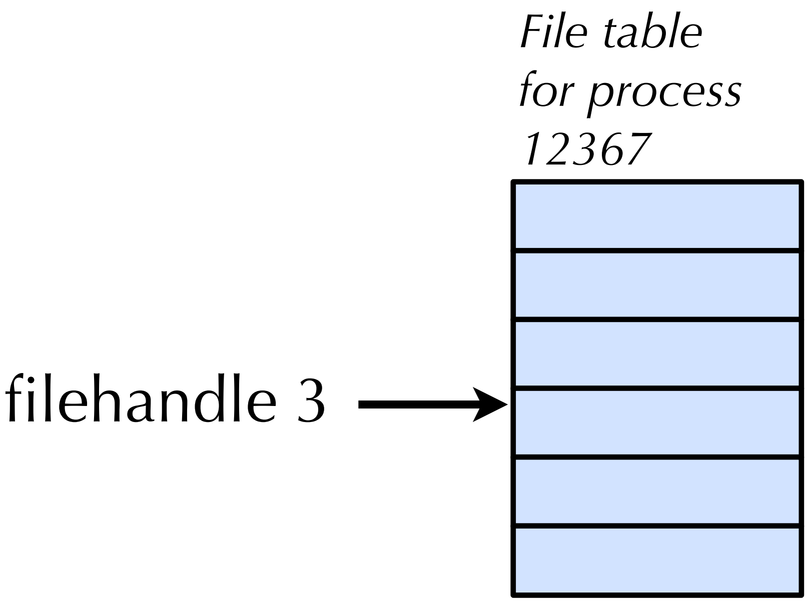
- If we're not allowed to write to a directory, how do we make changes to one?
- Answer: You don't! (At least not directly.)
- Rather, OS modifies directory entries when you...
  - Create a file (using `open()` or `creat()` system calls)
  - Delete a file (using `unlink()` system call)
  - Create a symbolic link (using `symlink()` system call)
  - Rename files (using `link()` system call)
  - Create or delete a directory (using `mkdir()` and `rmdir()` system calls)
- All of this is necessary to ensure that directories have the right format, and always contain the correct information.

Today

- The UNIX file abstraction
- UNIX low-level I/O interfaces
- Robust I/O
- Buffered I/O
- Standard I/O
- Accessing metadata and directories
- Fun with filehandles
- Pipes
- Summary

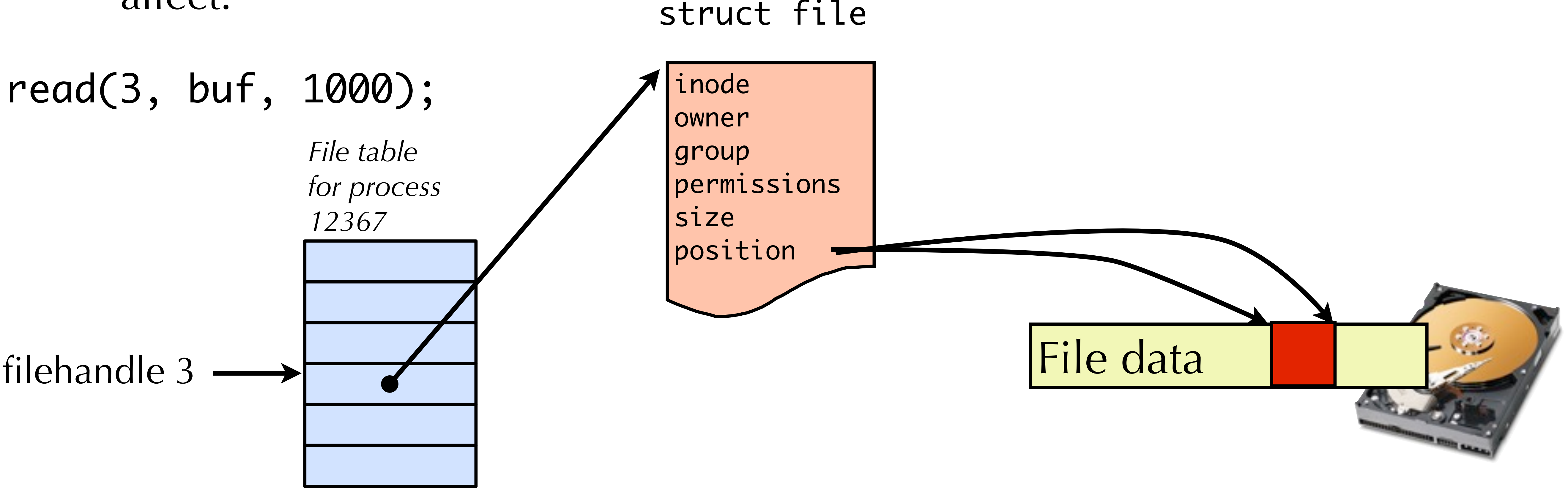
Filehandles

- A **filehandle** (a.k.a. **file descriptor**) is a reference to an open file.
  - The OS maintains a list of open files for each process.
  - The filehandle is just an index into this list.



Filehandles

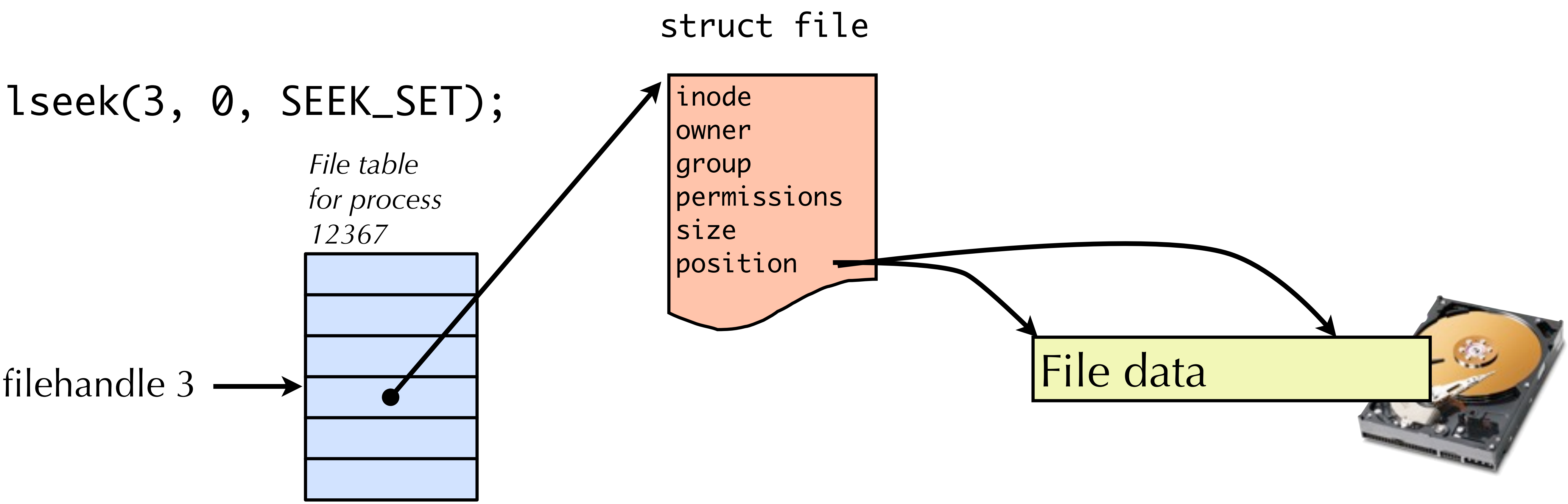
- The OS maintains an internal `struct file` for each open file.
  - The struct file includes the current **position** into the file.
  - This indicates the offset that the next `read()` or `write()` operation will affect.



Filehandles

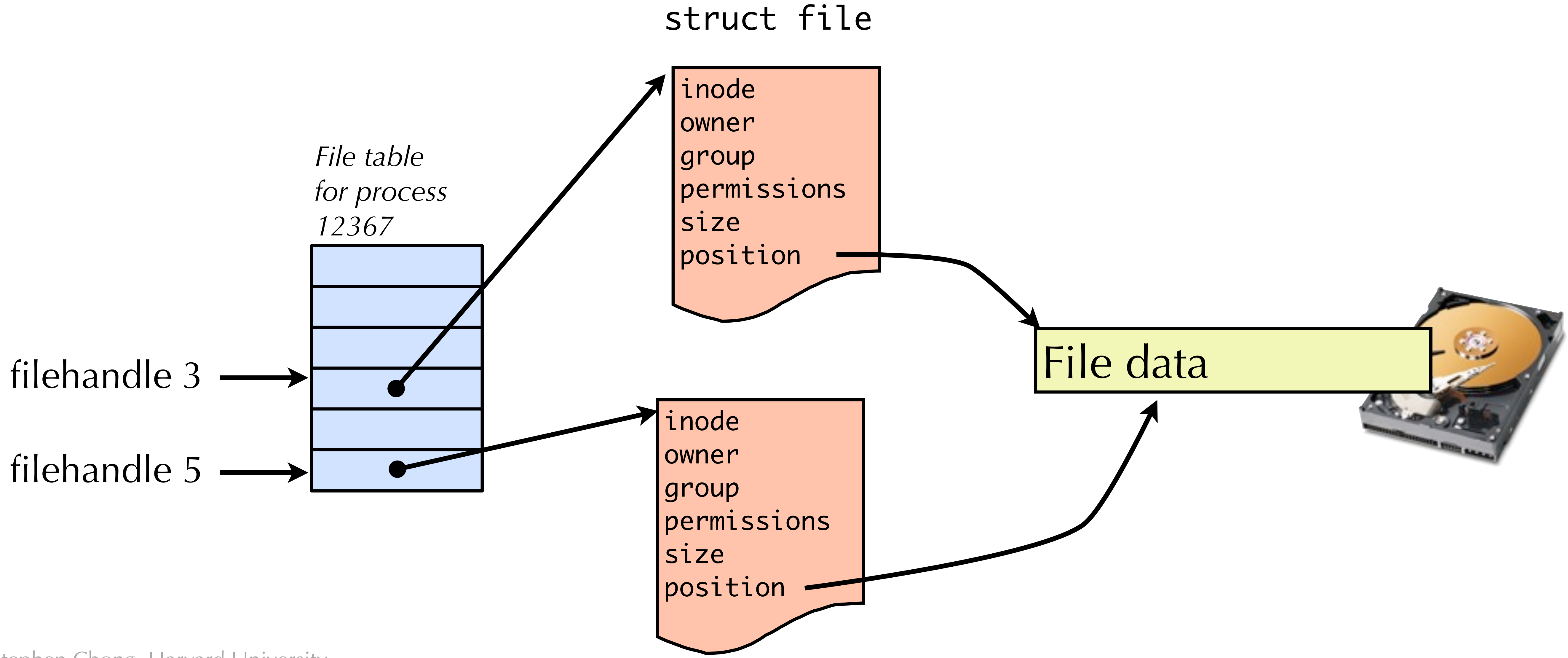
- Can also change the position using the `lseek()` system call.

```
lseek(3, 0, SEEK_SET);
```



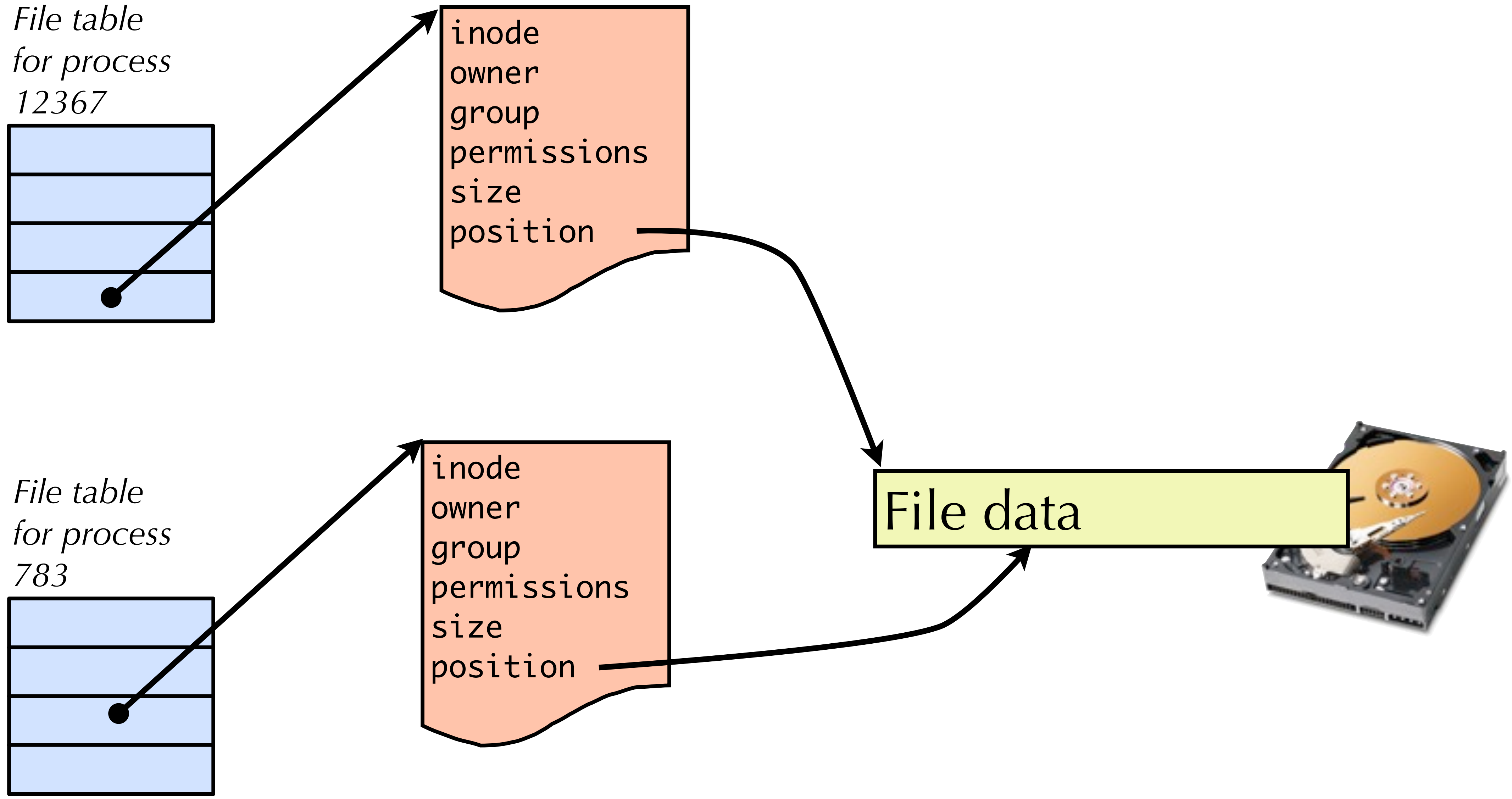
Filehandles

- Each instance of the open file has its own position.
  - Can read and write at different offsets into the file independently.
  - Different processes can also open the same file at the same time.



Processes and files

- Processes may have the same file open
  - But will have different file tables, and file table entries



Shell redirection

- The shell allows stdin, stdout, and stderr to be redirected (say, to or from a file).

```
$ ./myprogram > somefile.txt
```

Connects stdout of "myprogram" to somefile.txt

```
$ ./myprogram < input.txt > somefile.txt
```

Connects stdin to input.txt and stdout to somefile.txt

```
$ ./myprogram 2> errors.txt
```

Connects stderr to errors.txt

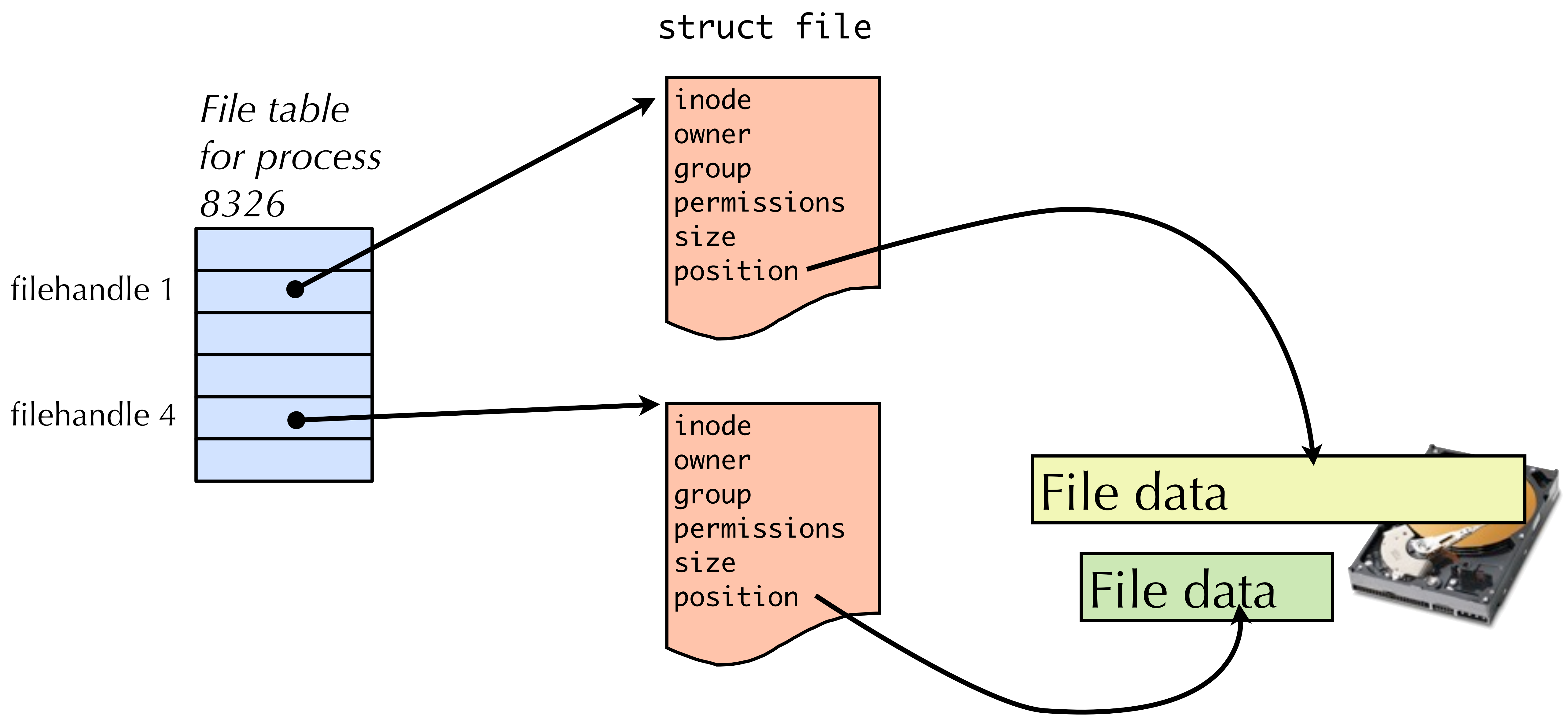
- The shell simply opens the file, making sure the file handle is 0, 1, or 2, as appropriate.
  - Problem: `open()` decides what the file handle number is.
  - How do we coerce the filehandle to be 0, 1, or 2?

Shell redirection

- The `dup2(int old_fd, int new_fd)` system call duplicates an open file descriptor, allowing you to specify the file descriptor you want.

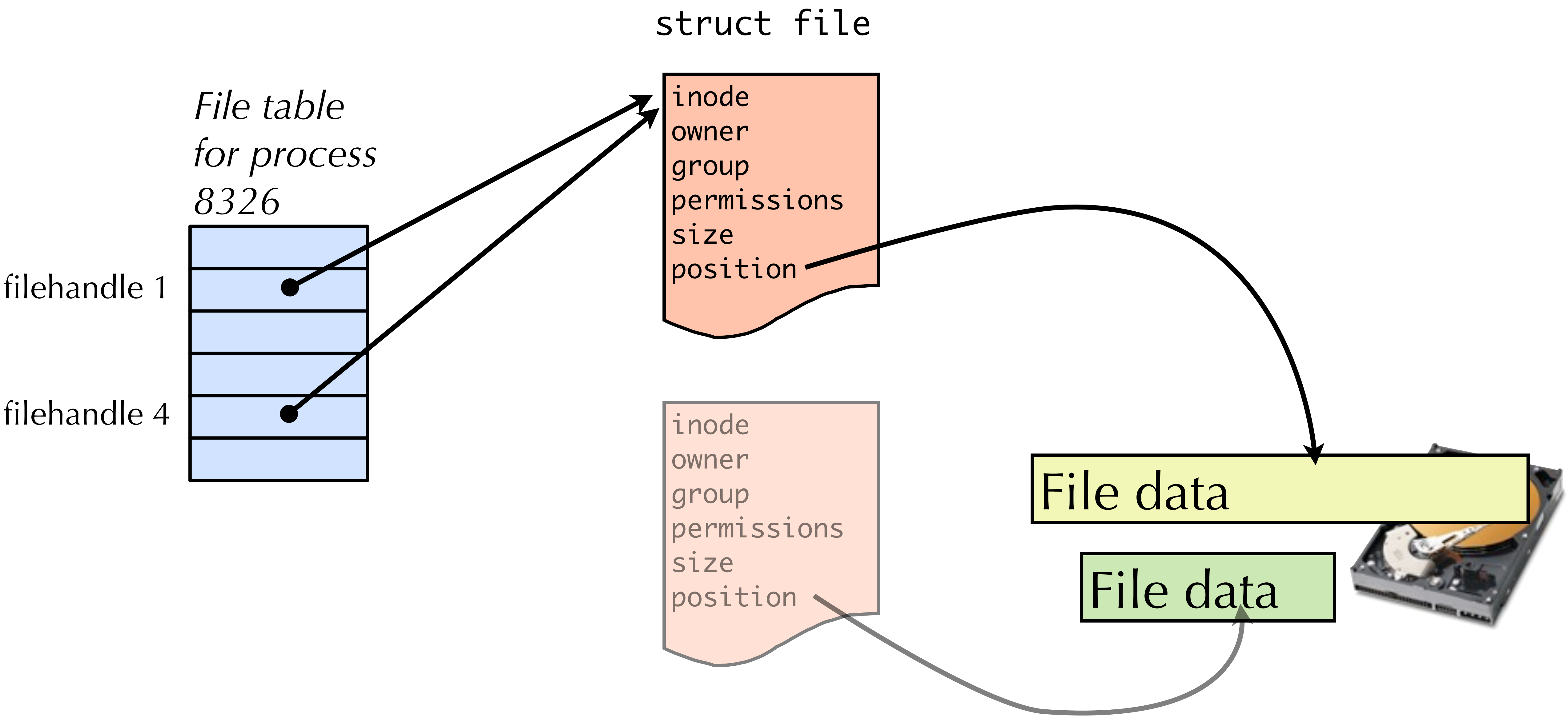
```
/* Redirect stdout to somefile.txt */
fd = open("somefile.txt", O_WRONLY);

/* This will close whatever filehandle 1 used to be, and
 * copy the filehandler fd to filehandler 1 */
dup2(fd, 1);
```

dup2 in action




```
dup2(fd1, fd4)
```

dup2 in action

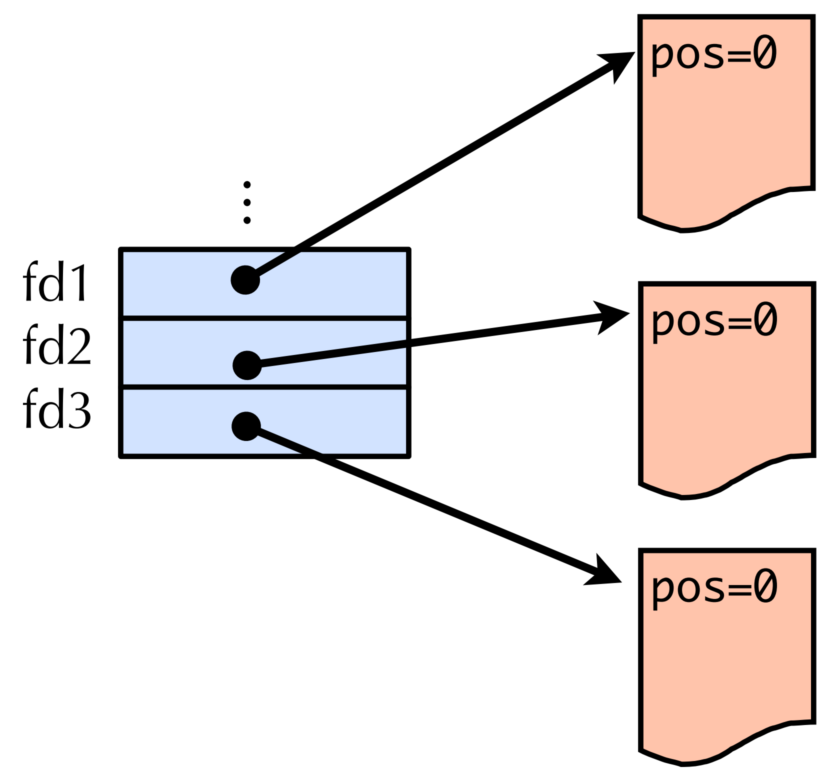



```
dup2(fd1, fd4)
```

Fun with File Descriptors (1)

- What would this program print for file containing "abcde"?

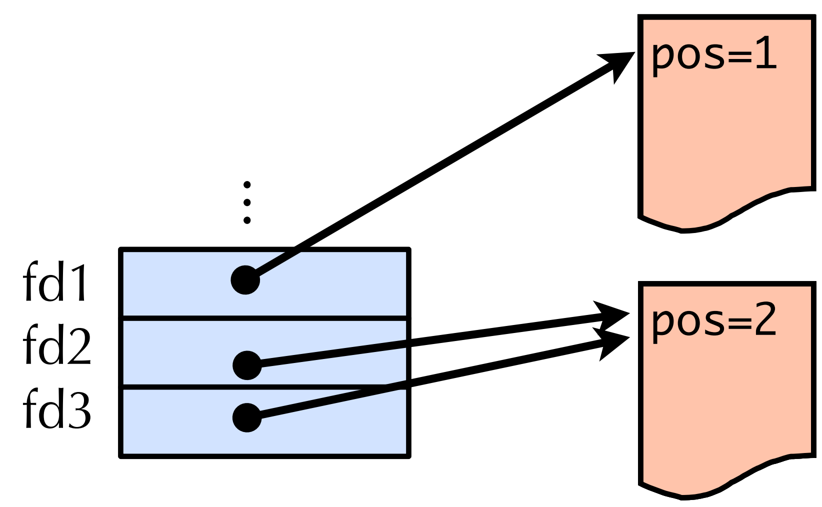
```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char c1, c2, c3;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    fd2 = Open(fname, O_RDONLY, 0);
    fd3 = Open(fname, O_RDONLY, 0);
    Dup2(fd2, fd3);
    Read(fd1, &c1, 1);
    Read(fd2, &c2, 1);
    Read(fd3, &c3, 1);
    printf("c1 = %c,
            c2 = %c,
            c3 = %c\n", c1, c2, c3);
    return 0;
}
```



Fun with File Descriptors (1)

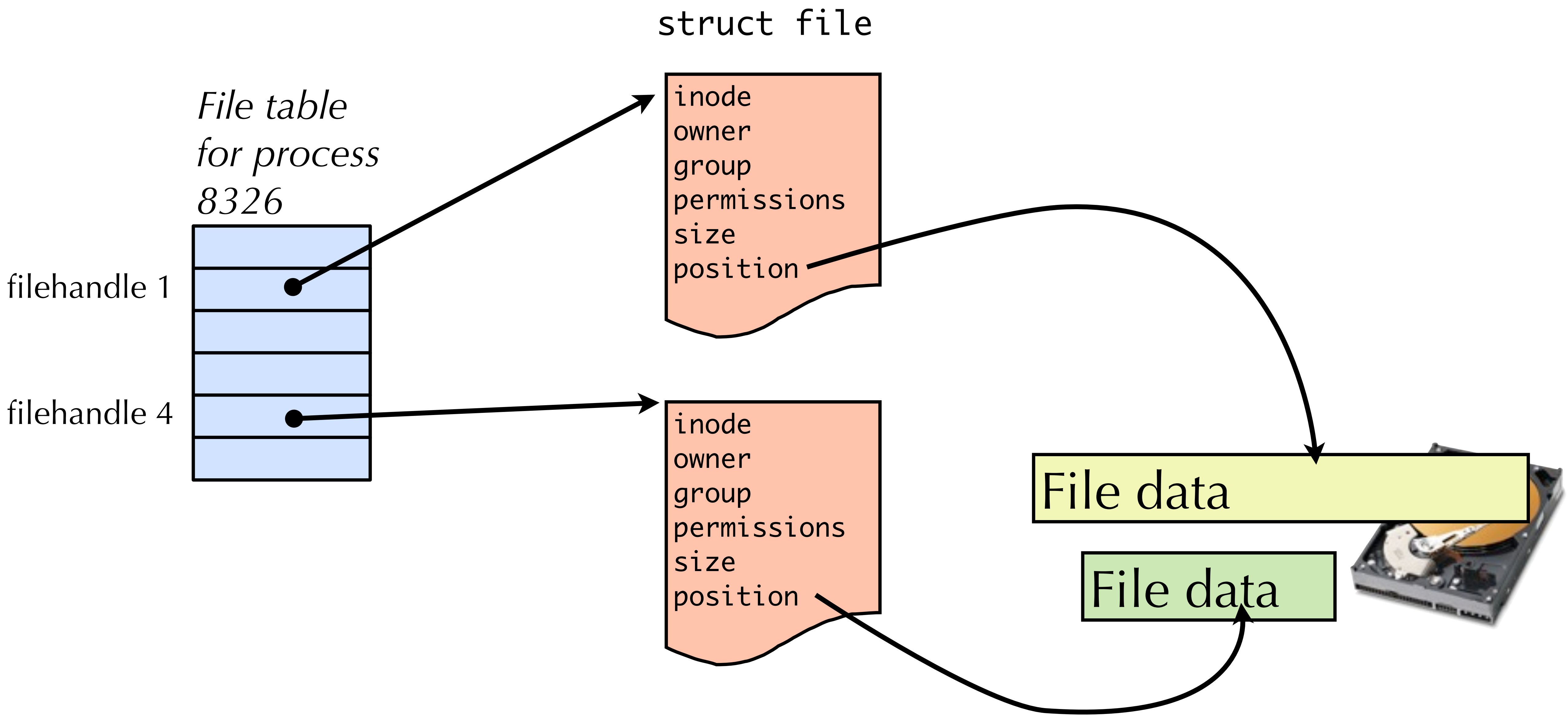
- What would this program print for file containing "abcde"?

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char c1, c2, c3;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    fd2 = Open(fname, O_RDONLY, 0);
    fd3 = Open(fname, O_RDONLY, 0);
    Dup2(fd2, fd3);
    Read(fd1, &c1, 1);
    Read(fd2, &c2, 1);
    Read(fd3, &c3, 1);
    printf("c1 = %c\n
            c2 = %c\n
            c3 = %c\n", c1, c2, c3);
    return 0;
}
```



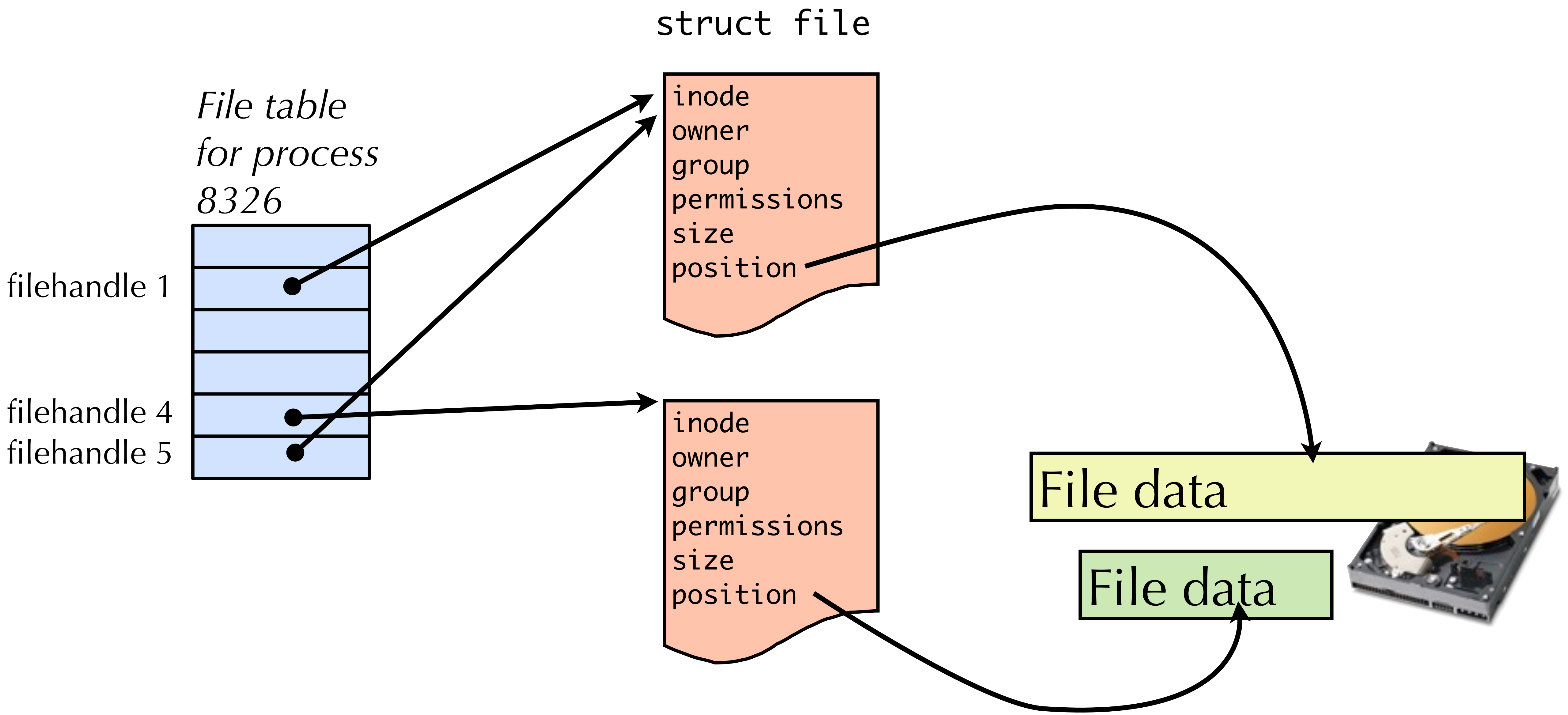

```
c1 = a
c2 = a
c3 = b
```

dup in action



`int newfd = dup(fd1)`

dup in action




```
int newfd = dup(fd1)
```

Fun with File Descriptors (2)

- What would be contents of resulting file?

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char *fname = argv[1];
    fd1 = Open(fname, O_CREAT|O_TRUNC|O_RDWR, S_IRUSR|S_IWUSR);
    Write(fd1, "pqrs", 4);
    fd3 = Open(fname, O_APPEND|O_WRONLY, 0);
    Write(fd3, "jklmn", 5);
    fd2 = dup(fd1);  /* Returns new descriptor mapped to same file */
    Write(fd2, "wxyz", 4);
    Write(fd3, "ef", 2);
    return 0;
}
```

Today

- The UNIX file abstraction
- UNIX low-level I/O interfaces
- Robust I/O
- Buffered I/O
- Standard I/O
- Accessing metadata and directories
- Fun with filehandles
- Pipes
- Summary

Pipes

- UNIX provides several mechanisms for inter-process communication (IPC)
  - Shared memory regions
  - Sockets (also used for communication over a network).
  - Pipes
- A **pipe** is a pair of file descriptors for communication between two processes.
  - One process can write data to one "end" of the pipe
  - The other process can read data from the other "end" of the pipe.
- `int pipe(int filedes[2]);`
  - `filedes[1]` is the write end of the pipe; `filedes[0]` is the read end of the pipe.

Using pipes

- But how do we get two processes to use a pipe?
- Idea: Parent process first creates the pipe, then forks the child
    - Since parent and child share open files, they can communicate.
- This is exactly what the UNIX shell does for you when you "pipe" the output of one command into another.

```
$ ./myprog | grep 'somestring'
```

- Shell creates the pipe, forks both "myprog" and "grep", and uses dup2() to wire the ends of the pipe into stdout and stdin of each process.
- Somewhat more complex example:

```
$ ./myprog | grep 'somestring' | sort | uniq | more
```

Pipe example

```
main() {
   char inbuf[BUFSIZE];
   int p[2], j, pid;

   /* open pipe */
   if(pipe(p) == -1) {    perror("pipe call error");
        exit(1);
   }

   switch(pid = fork()){
   case -1: perror("error: fork failed");
            exit(2);

   case 0:  /* if child then write down pipe */
         close(p[0]);  /* first close the read end of the pipe */
         write(p[1], “Hello there.”, 12);
         write(p[1], “This is a message.”, 18);
            write(p[1], “How are you?”, 12);
         break;

   default:   /* parent reads pipe */
         close(p[1]);  /* first close the write end of the pipe */
         read(p[0], inbuf, BUFSIZE);  /* What is wrong here?? */
         printf("Parent read: %s\n", inbuf);
         wait(NULL);
   }
   exit(0);
}
```

Summary

- Unix I/O
  - System calls
  - `read()`, `write()`, etc.
- Robust I/O package (RIO)
  - Provides some buffering around Unix I/O
  - (Developed by the textbook authors)
- Standard I/O
  - `fopen()`, `fclose()`, `fread()`, `fwrite()`, etc.
  - Standard way to perform I/O for files, terminals

Pros and Cons of Unix I/O

- Pros
  - Unix I/O is the most general and lowest overhead form of I/O.
    - All other I/O packages are implemented using Unix I/O functions.
  - Unix I/O provides functions for accessing file metadata.
- Cons
  - Dealing with short counts is tricky and error prone.
  - Efficient reading of text lines requires some form of buffering, also tricky and error prone.
    - Both of these issues are addressed by the standard I/O and RIO packages.

Pros and Cons of Standard I/O

- Pros:
  - Buffering increases efficiency by decreasing the number of read and write system calls.
  - Short counts are handled automatically.
- Cons:
  - Provides no functions for accessing file metadata
  - Standard I/O is not appropriate for input and output on network sockets
  - There are poorly documented restrictions on streams that interact badly with restrictions on sockets

Choosing I/O Functions

- General rule: Use the highest-level I/O functions you can.
  - Many C programmers are able to do all of their work using the standard I/O functions.
- When to use standard I/O?
  - When working with disk or terminal files.
- When to use raw Unix I/O
  - When you need to fetch file metadata.
  - In rare cases when you need absolute highest performance.
- When to use RIO?
  - When you are reading and writing network sockets or pipes.
  - *Never use standard I/O or raw Unix I/O on sockets or pipes.*

For Further Information

- The Unix bible:
    - W. Richard Stevens & Stephen A. Rago, *Advanced Programming in the Unix Environment*, 2nd Edition, Addison Wesley, 2005.
        - Updated from Stevens' 1993 book
- Stevens was arguably the best technical writer ever.
    - Produced authoritative works in:
        - Unix programming
        - TCP/IP
        - Unix network programming
        - Unix IPC programming.