CS310 Operating Systems

Lecture 36 : File System - 4

Ravi Mittal

IIT Goa

Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - Book: Modern Operating Systems by Andrew Tanenbaum and Herbert Bos,
    - Chapter 4
  - Book: Linux System Programming: talking directly to the kernel and C library, by Robert Love
  - Book: Computer Systems, A programming Perspective, Bryant and O'Hallaron
  - Class presentation: University of California, Berkeley, CS162

Read the following:

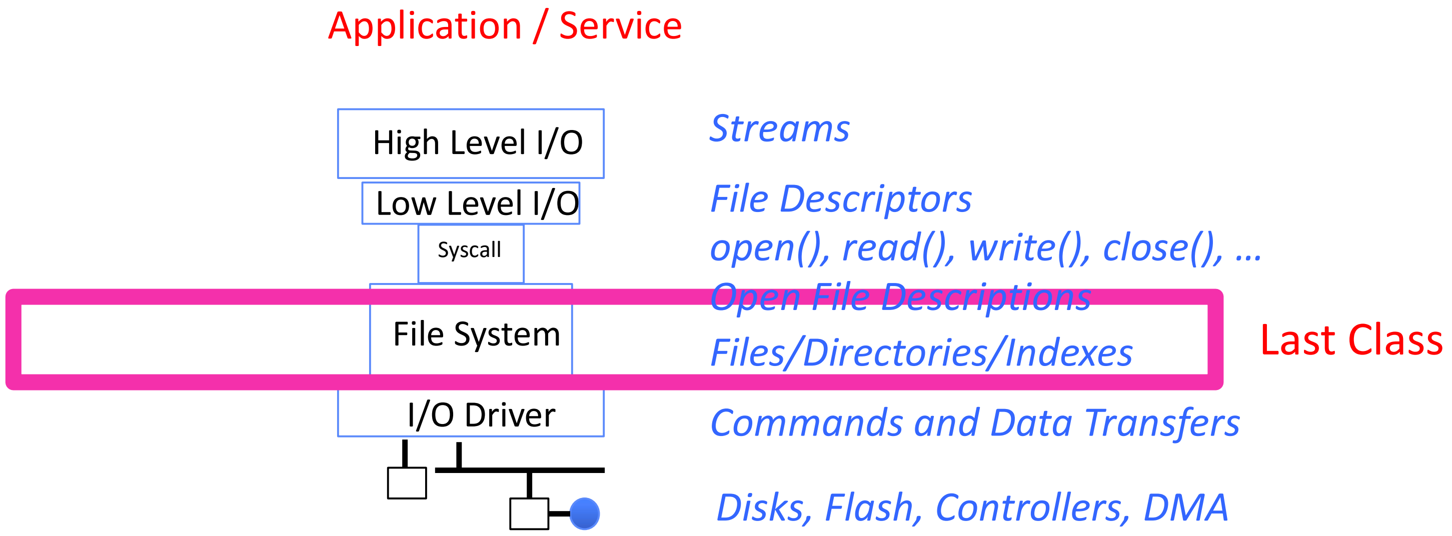
- Book: Modern Operating Systems, by Andrew Tanenbaum and Herbert Bos
  - Chapter 4
- Book: Linux System Programming: talking directly to the kernel and C library, by Robert Love

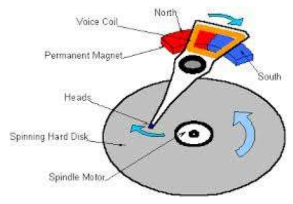
We will study..

- Where are we (File Systems)?
- C Library APIs
- Buffered I/O (File System): Introduction
- Buffered I/O File Operations

Where are we ?

I/O and Storage Layers





I/O and Storage Layers

Application / Service

We will study

High Level I/O — *Streams*

Low Level I/O — *File Descriptors*

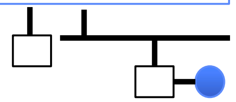
Syscall — *open(), read(), write(), close(), …*

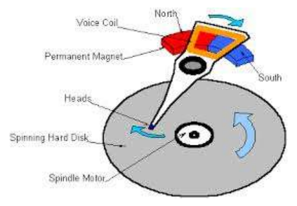
Open File Descriptions

File System — *Files/Directories/Indexes*

I/O Driver — *Commands and Data Transfers*

Disks, Flash, Controllers, DMA





File Access

- Sequential access
  - Read from the beginning
  - Can’t skip around
  - Corresponds to magnetic tape
- Random access
  - Start from anywhere
  - Example: disks
  - Necessary for many applications
    - Database systems

The File Abstraction

- High-Level File I/O: Streams
- Low-Level File I/O: File Descriptors ⟸ **We know**

The File Abstraction

- High-Level File I/O: Streams ⇐ **Today**
- Low-Level File I/O: File Descriptors

C Library API

Buffered I/O - Introduction

C library High level APSs vs Linux Syscalls

- A C library function implementation
  - C interface the library provides to programmers to access kernel related functions
- If a C programmers directly uses file syscalls, he/she needs to read the documentation
- Each syscall has a number of arguments – makes programming complicated
  - A user, most of the time, may not use all arguments
  - User space programs are expected to find out arguments for each syscall by inspecting the documentation

C library High level APIs vs Linux Syscalls

- Example: `fopen()` is a library function which provides buffered I/O services for opening a file while `open()` is a system call that provides non-buffered I/O services

- Standard library file operations don't directly use *file descriptors*
- In standard I/O parlance, an open file is called a ***stream***

```
stream = fopen("/etc/manifest", "r
```

- However, Syscalls use *file descriptors*

```
fd = open
("/home/kidd/madagascar", O_RDONLY);
```

Adapted from: Linux programming: Robert Love

User-Buffered I/O

- The *block* is an abstraction representing the smallest unit of storage on a filesystem
- Inside the kernel, all filesystem operations occur in terms of blocks
- No I/O operation may execute on an amount of data less than the block size
- If you only want to read a byte, you'll have to read a whole block
  - Even to read and modify a byte, you will have to perform operations on whole block
- Partial block operations are inefficient
- User applications don't work this way..
  - They use abstractions: byte, strings, etc - independent of block size
- Example: Reading a single byte 1024 times vs reading a single 1024 bytes once

User Buffered I/O

- To improve performance, data can be buffered internally by delaying writes and reading ahead
- In practice, blocks are usually 512, 1,024, 2,048, 4,096, or 8,192 bytes in size
- Buffers are usually multiple of block sizes: 4096 or 8192 bytes

User-Buffered I/O

- Buffering is done in the user space
  - Transparently in a library
- Writing
  - As data is written, it is stored in a buffer inside the program's address space
  - When the written data size reaches a set size – buffer size – entire buffer is written out (to disk) in a single write operation
    - Which means it's written to the underlying file descriptor
- Reading
  - Data is read (from disk) using buffer-sized block aligned chunks
  - Application's various sized read requests are served out from this buffer – say one byte at a time
  - When buffer is empty, another block-aligned chunk is read in
- Overall less system calls

User-Buffered I/O

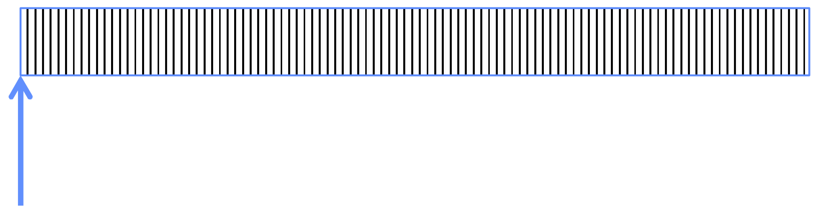
- You can design and implement user buffering by hand in your own program
  - Many mission critical applications do this
- Vast majority of programs use
  - Popular standard I/O library (as a part of standard C library)
  - *iostream library* (as a part of *standard C++ library*)

Buffered I/O – File Operations

Standard I/O library – File Operations

- Standard I/O routines
  - Do not operate directly on file descriptors
  - Use their own unique identifier, known as the *file pointer*
- Inside C library, file pointer maps to a file descriptor
- File pointer is represented by a pointer to the FILE typedef
  - That is defined in <stdio.h>
- In standard I/O parlance, an open file is called a *stream*
- Streams may be opened for reading or writing or both

Opening a file



```
#include <stdio.h>
FILE *fopen( const char *path, const char *mode );
```

This function opens the file path with the behavior given by mode and associates a new stream with it.

Modes

Mode Text	Descriptions
r	Open existing file for reading. The stream is positioned at the start of the file
w	Open for writing. If the file exists, it is truncated to zero length. If the file doesn’t exist it is created. Stream is positioned at the beginning of the file
a	Open for appending; created if does not exist. The stream is positioned at the end of the file
r+	Open existing file for reading & writing. Stream is positioned at the start of the file
w+	Open for reading & writing; truncated to zero if exists, create otherwise. The stream is positioned at the beginning of the file
a+	Open for reading & writing. Created if does not exist. Read from beginning, write as append. Read from the beginning and write from the end.

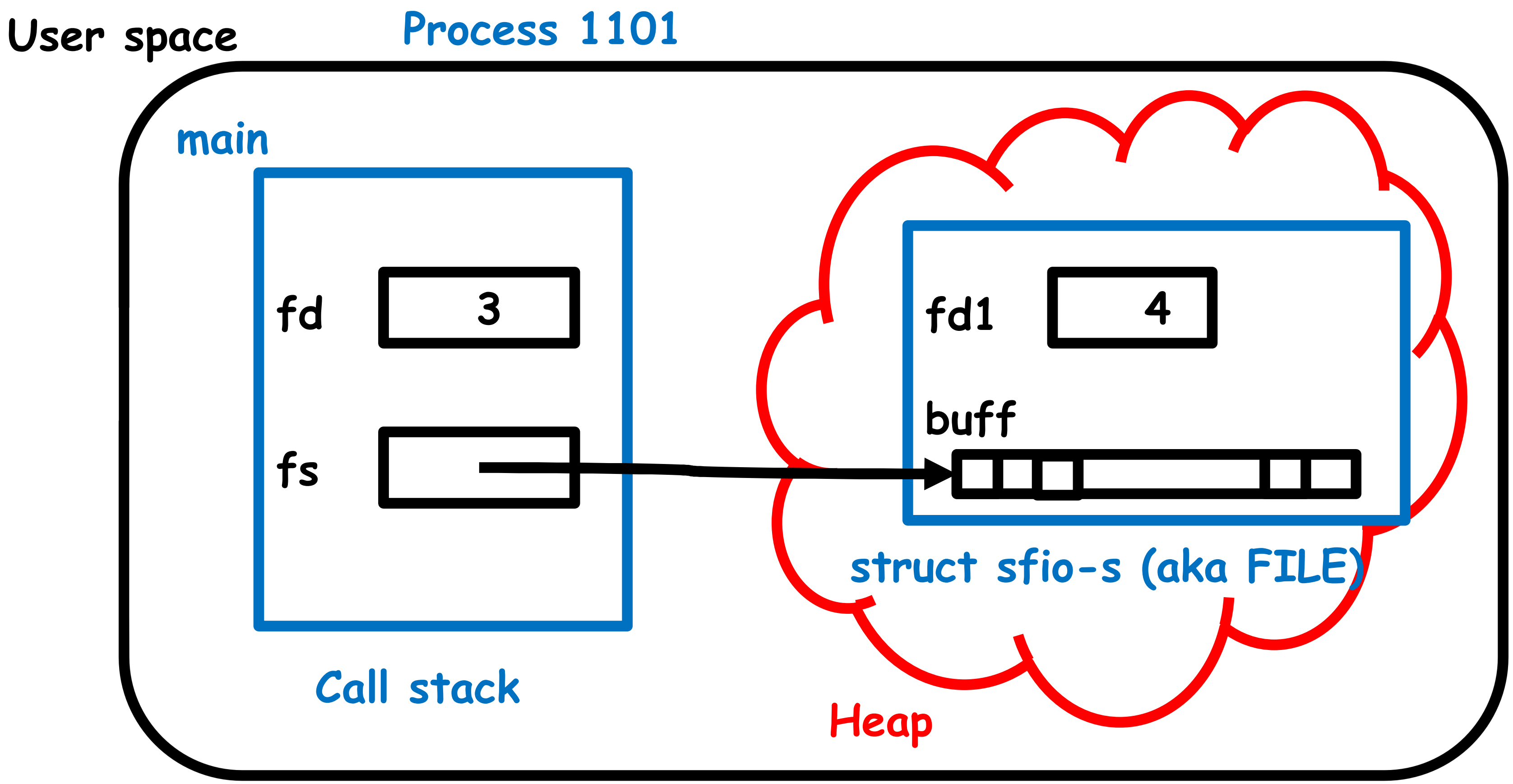
Opening a file

```
FILE *stream
stream = fopen("/home/ravi/exam.txt", "r");
If (!stream)
       /*error */
```

- Upon success, `fopen()` returns a valid FILE pointer
  - On failure it returns NULL and sets errno appropriately
  - FILE* represents a stream which is assigned to a variable – file pointer stream

read() syscall vs fread() API - User space

```
Int main(){
      int fd = open(“foo.txt”, “O_RDONLY”);          // for I/O syscall
     FILE *fs = fopen(‘bar.txt”, “w”);               // for C lib I/O
```



Closing Streams

- The fclose() function closes a given stream:

```
#include <stdio.h>
int fclose (FILE *stream);
```

  - Any buffered and not-yet-written data is first flushed
  - On success, fclose() returns 0
  - On failure, it returns EOF and sets errno appropriately
- fcloseall() function closes all streams associated with the current process

C API Standard Streams

- Three predefined streams are opened implicitly when a program is executed
  - `FILE *stdin` – normal source of input, can be redirected
  - `FILE *stdout` – normal source of output, can be redirected
  - `FILE *stderr` – diagnostics and errors, can be redirected

- STDIN / STDOUT enable composition in Unix
- All can be redirected
  - `cat hello.txt | grep “World!”`
  - **`cat`**’s **`stdout`** goes to **`grep`**’s **`stdin`**

Reading Files – Multiple Functions

- Once file is open, we can read input from it
- We use one of three functions:
  - `fgetc, fgets, fread`
- `fgetc`
  - Useful when you want to read one character (eg letter) at a time

    ```
    int fgetc(FILE * stream);
    ```
  - This function reads the next character from stream
  - returns it as an unsigned `char` cast to an `int`
  - The return value of fgetc() must be stored in an `int`
  - EOF is returned when end of file or error
  - Reading the character advances the current position in the stream

Reading an entire line

- The function fgets() reads a string from a given stream:

```
#include <stdio.h>
char * fgets (char *str, int size, FILE *stream);
```

  - This function reads up to *one less* than size bytes from stream
  - Stores the results in `str`
  - A null character (\0) is stored in the buffer after the last byte read in
  - Reading stops after an EOF or a newline character is reached

Reading Binary Data

- Some developers want to read binary data such as C structures
- fread()

```
#include <stdio.h>
size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
```

- Read up to nr elements of data, each of size bytes into the buffer pointed at by buf

Writing to a stream

- Writing a single character

```
#include <stdio.h>
int fputc (int c, FILE *stream);
```

- Writing a string of characters

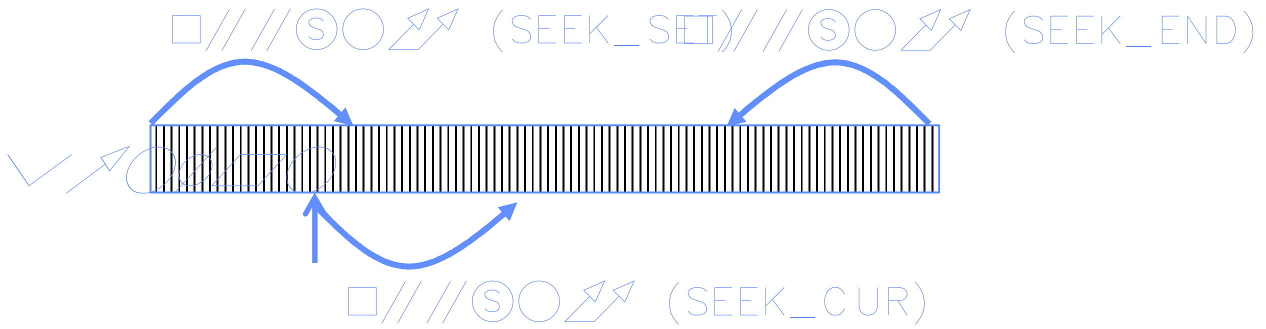
```
#include <stdio.h>
int fputs (const char *str, FILE *stream);
```

- Writing Binary Data

```
#include <stdio.h>
size_t fwrite (void *buf, size_t size,
size_t nr, FILE *stream);
```

Random Access

```
int fseek(FILE *stream, long int offset, int whence);
```

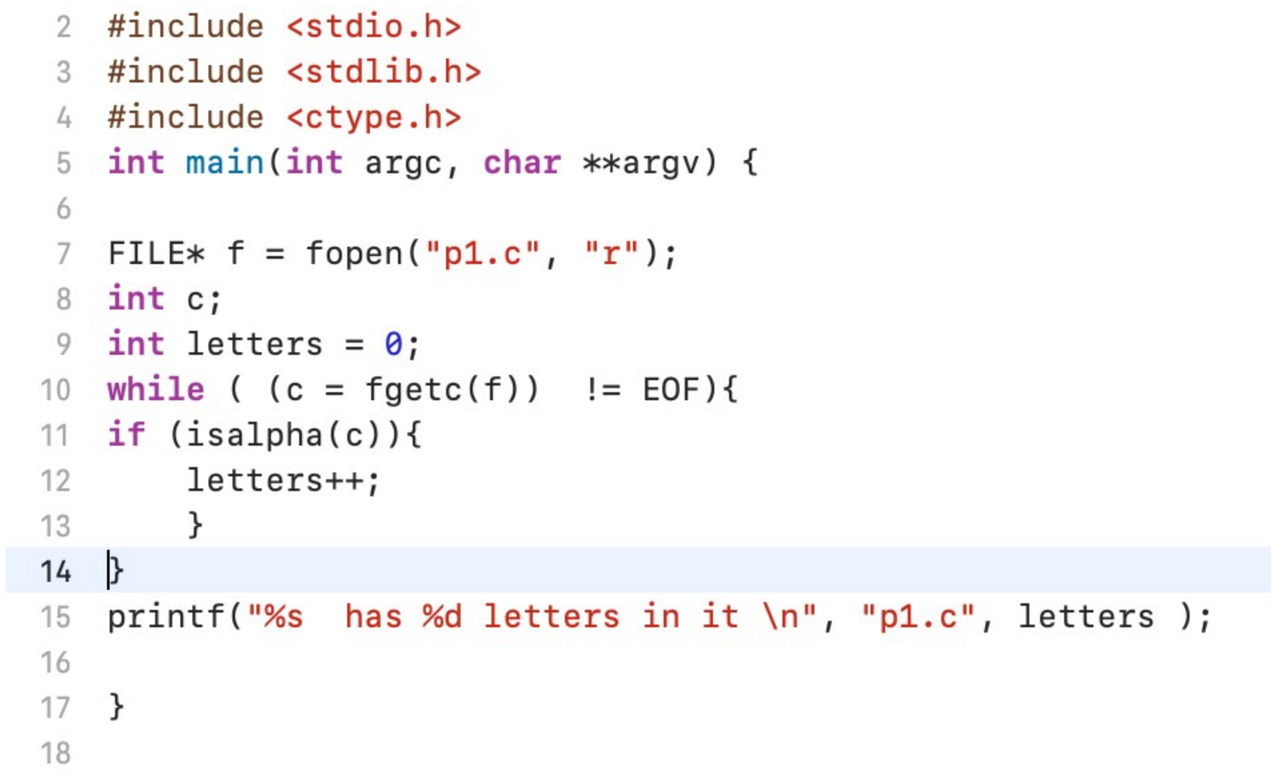


- Sets file position pointer to a specific position
- *Stream:* pointer returned by fopen
- *Offset:* The position to seek to, relative to one of the positions specified by *whence*
- *whence:* The position from which to apply the offset; 3 positions
    - **`SEEK_SET`** – seek starts at beginning of file
    - **`SEEK_CUR`** – seek starts at current location in file
    - **`SEEK_END`** – seek starts at end of file

Many more – Stream Ops

```
// formatted
int fprintf(FILE *restrict stream, const char *restrict format,
              ...);
int fscanf(FILE *restrict stream, const char *restrict format,
              ...);
```

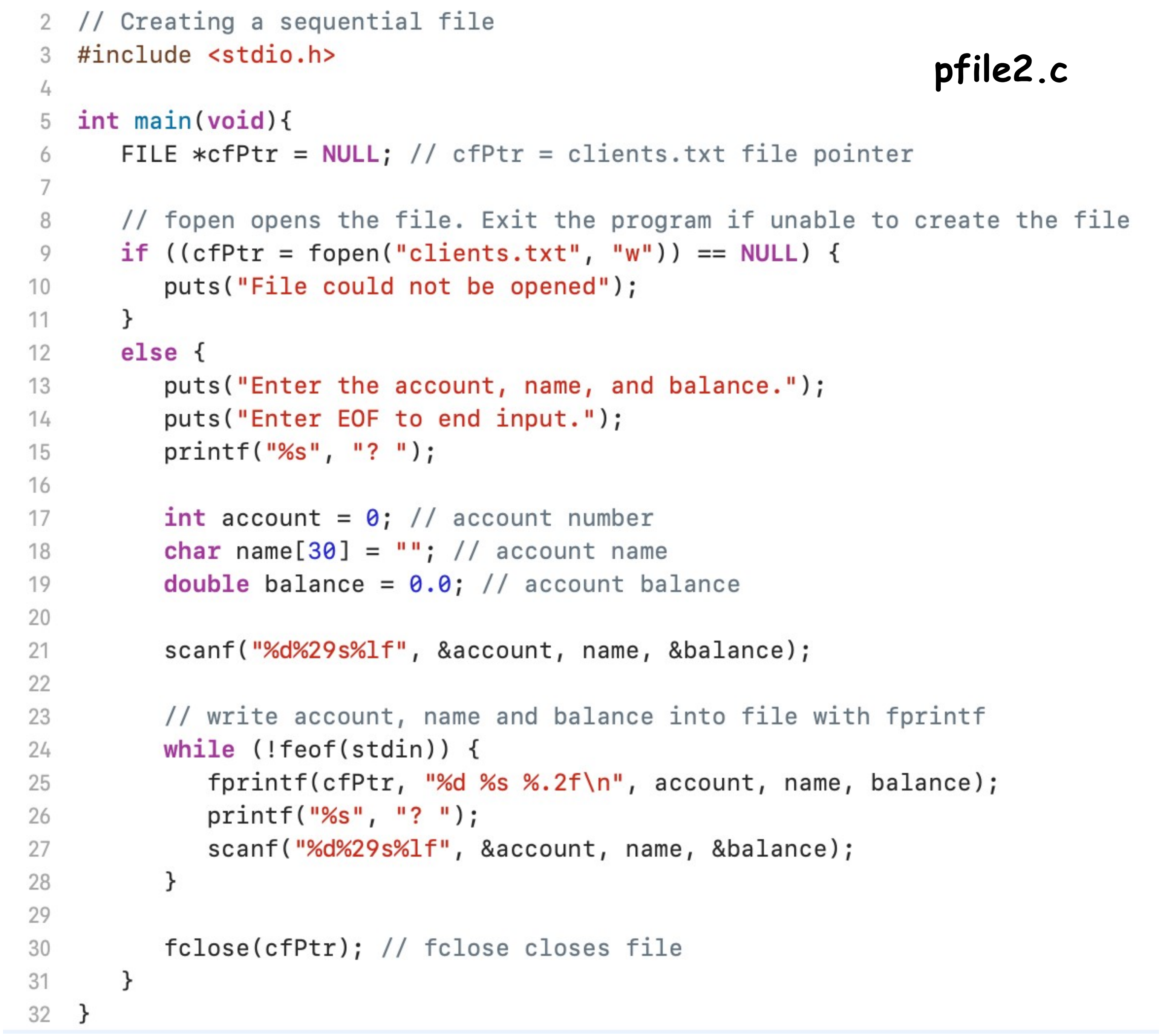
A sample program – character oriented streaming



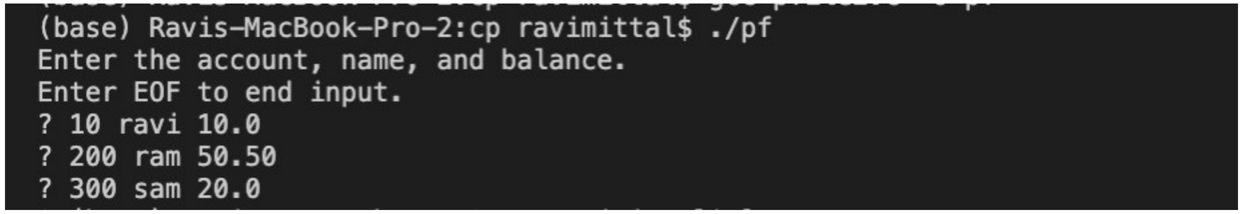
```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int main(int argc, char **argv) {

FILE* f = fopen("p1.c", "r");
int c;
int letters = 0;
while ( (c = fgetc(f))  != EOF){
if (isalpha(c)){
    letters++;
    }
}
printf("%s  has %d letters in it \n", "p1.c", letters );

}

```

Program using Syscalls

pfile2.c

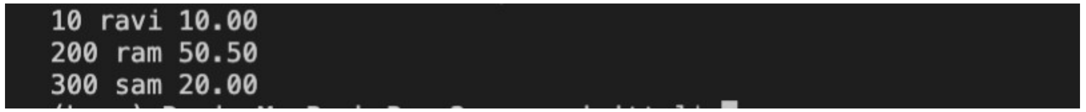


```c
// Creating a sequential file
#include <stdio.h>

int main(void){
   FILE *cfPtr = NULL; // cfPtr = clients.txt file pointer

   // fopen opens the file. Exit the program if unable to create the file
   if ((cfPtr = fopen("clients.txt", "w")) == NULL) {
      puts("File could not be opened");
   }
   else {
      puts("Enter the account, name, and balance.");
      puts("Enter EOF to end input.");
      printf("%s", "? ");

      int account = 0; // account number
      char name[30] = ""; // account name
      double balance = 0.0; // account balance

      scanf("%d%29s%lf", &account, name, &balance);

      // write account, name and balance into file with fprintf
      while (!feof(stdin)) {
         fprintf(cfPtr, "%d %s %.2f\n", account, name, balance);
         printf("%s", "? ");
         scanf("%d%29s%lf", &account, name, &balance);
      }

      fclose(cfPtr); // fclose closes file
   }
}
```



```
(base) Ravis-MacBook-Pro-2:cp ravimittal$ ./pf
Enter the account, name, and balance.
Enter EOF to end input.
? 10 ravi 10.0
? 200 ram 50.50
? 300 sam 20.0
```

$ cat clients.txt



```
10 ravi 10.00
200 ram 50.50
300 sam 20.00
```

EOF character
Linux/MAC OS : <Ctrl> d
Windows: <Ctrl> z enter

Lecture Summary

- Standard I/O is a user-buffering library provided as part of the standard C library
- Buffered file operations are useful when
  - You issue many system calls
  - Performance is crucial
  - Your access patterns are character- or line-based
  - You want interfaces to make such access easy without issuing extraneous system calls
  - You prefer a higher-level interface to the low-level Linux system calls