I/O and Binary Files

Introduction to Computer Systems, Fall 2024

Instructors: Joel Ramirez Travis McGaha

Head TAs: Adam Gorka Daniel Gearhardt
Ash Fujiyama Emily Shen

TAs:

Ahmed Abdellah
Angie Cao
August Fu
Caroline Begg
Cathy Cao
Claire Lu
Eric Sungwon Lee
Ethan Weisberg
Garrett O'Malley Kirsch
Hassan Rizwan
Iain Li
Jerry Wang
Juan Lopez
Keith Mathe
Maya Huizar
Meghana Vasireddy
Perrie Quek
Sidharth Roy
Sydnie-Shea Cohen
Vivi Li
Yousef AlRabiah

pollev.com/cis2400

How are you? What is your mood looking like this week?

Logistics

- Check-in06: Due EOD today incase you missed it.
- HW08 (Decoder) Due Friday 11/08 @ 11:59 pm
- **Assignments will very likely take increasingly longer to complete. Please please please try to not let the work accumulate. Pretty please.**

Let’s be real for a moment

Lecture Outline

- **File I/O**
- Binary files & Endianness
- Office Hours

Files Revisted

- Files are very ***simple*** objects – they consist of a ***sequence of bytes***

- We could spend two weeks on files alone.

We're not going to do that.

- *Functionality:*
  - ***Open***
  - ***Read/Write***
  - ***Close***
- *Versatile:*
  - ***Across different machines, Files can represent a myriad of things.***

How do we interact with Files?

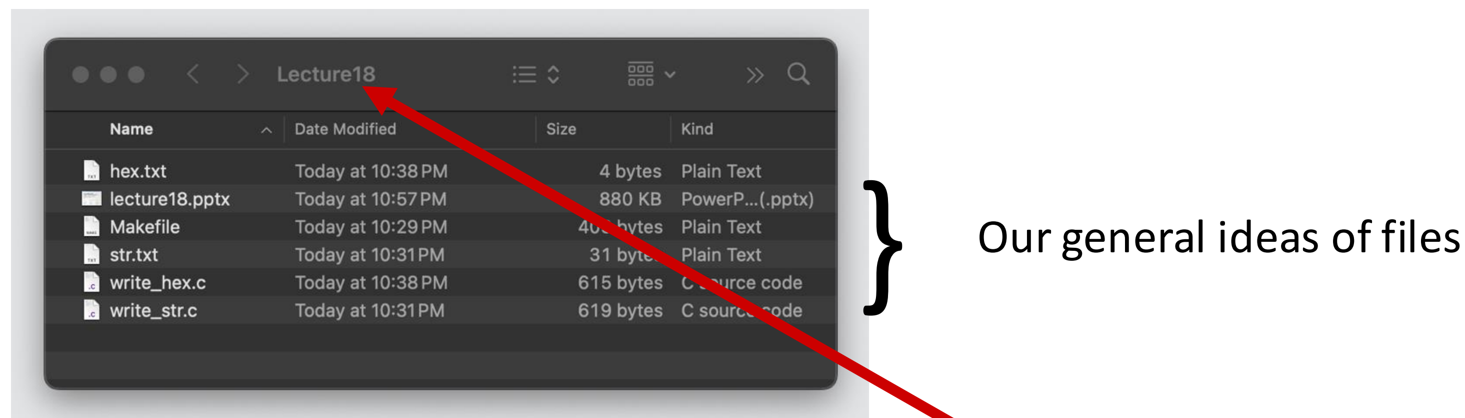
- In Unix and Unix like systems, ***File **** are pointers

- You can use these to refer to ***files*** that you've opened.
- Some files are already open for you when you run your program.

The terminal itself is treated as a file
  - you can "read from" and "write to" it
- Three file handlers your programs will always have
  - stdin : standard-input (terminal)
  - stdout : standard-output (terminal, for output)
  - stderr : standard-error (terminal, for error message

How do we interact with Files?

- ***File **** are pointers in Unix and Unix like systems
- You can use these to refer to ***files*** that you've opened in ***c***



Our general ideas of files

However, in Unix based systems, ***everything is just a file***

even directories are treated as files

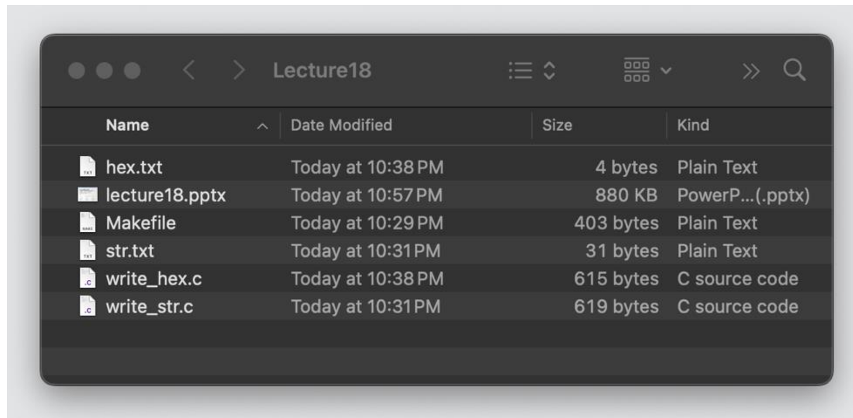
Keyboards

Mice

Sockets

How do we interact with Files?

- ***We'll keep it to these types of files in this course***



You've already been interacting with files

(maybe not programs yet though)

But that will change.

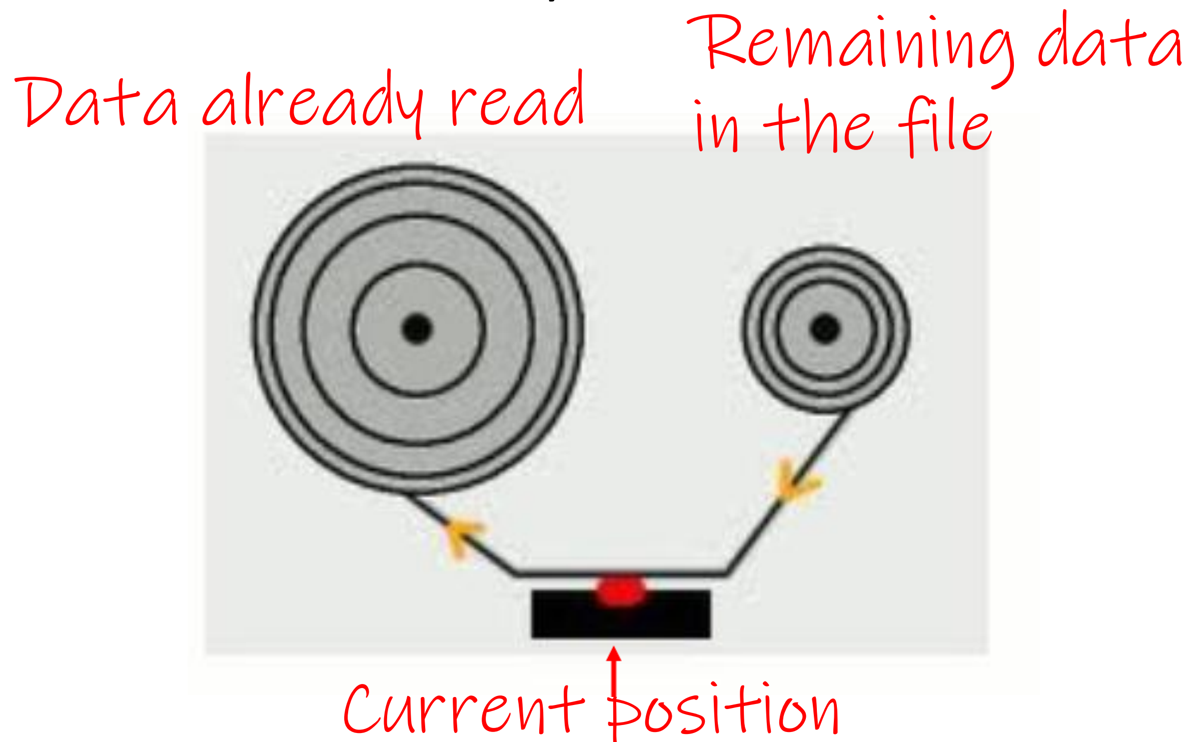
File Interface Metaphor: Tape Drives

- Programs usually interact with files following a similar file interface:
- Functions that model a sequential access device like magnetic tape drives

I remember these…

File Interface Metaphor: Tape Drives

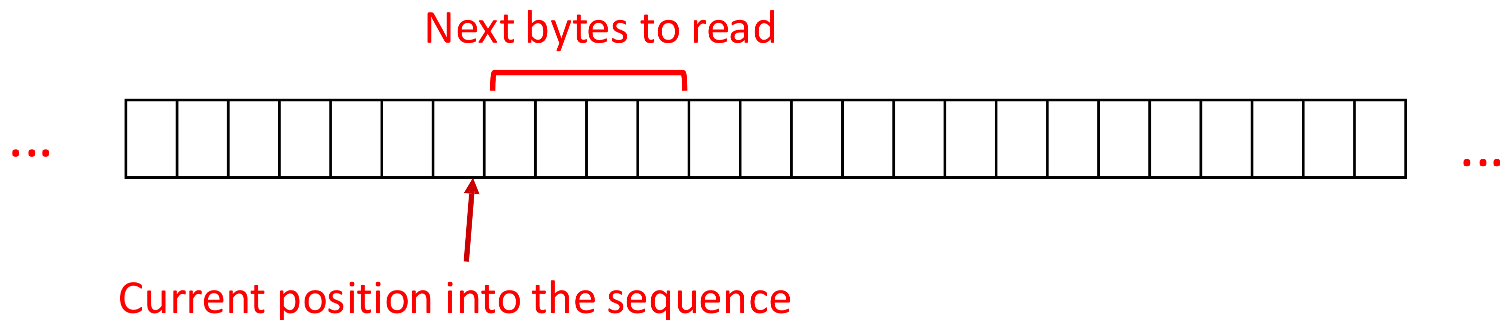
- Open a file for reading or writing
  - (usually starting at the beginning of the file)
- Read/Write the file
  - Each read/write advances the number of bytes read or written
- Rewind: start at beginning again
- others



File Interface Metaphor: Streams

- Another (more modern) abstraction is to think of I/O in terms of "streams"

- Stream:
  - A **sequence** of bytes that flows **to** and **from** a device
  - We do not have access to whole file at once (some files are too big to fit inside of memory easily)



This ends up working sort of like an iterator over the file.
Where we can read current data, and/or insert new data

C Stream Functions (1 of 3)

- Some stream functions (complete list in `stdio.h`):

Returns NULL on error

Do we create a new file if it doesn't exist?
Are we reading the file?
Are we writing the file?

```
FILE* fopen(filename, mode);
```

- Opens a stream to the specified file in specified file access mode

a **FILE*** returned by fopen

```
int fclose(stream);
```

- Closes the specified stream (and file)

```
int fprintf(stream, format, ...);
```

- Writes a formatted C string like `printf(...);` but for files

```
int fscanf(stream, format, ...);
```

- Reads data and stores data matching the format string

C Stream Functions (2 of 3)

- Some stream functions (complete list in `stdio.h`):

Pointer to the start of elements in memory to write to file

Size of an element

Number of elements

FILE*

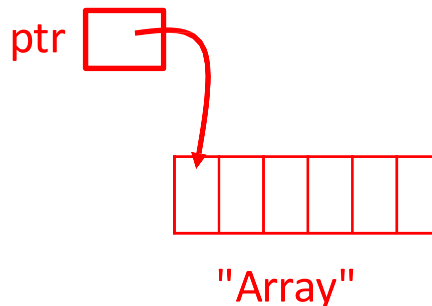
```
size_t fwrite(ptr, size, count, stream);
```

  - Writes an array of *count* elements of *size* bytes from *ptr* to *stream*

```
size_t fread(ptr, size, count, stream);
```

  - Reads an array of *count* elements of *size* bytes from *stream* to *ptr*

Returns number of elements actually read/written



C Stream Functions (3 of 3)

- Some stream functions (complete list in `stdio.h`):

```
int fgetc(FILE *stream);
```

- Reads one character from stream (one byte)

```
int fputc(FILE *stream);
```

- Writes one character from stream (one byte)

```
char* fgets(char* str, int n, FILE* stream);
```

- Reads a string from the stream into the string **`str`**. Reads N characters or until a newline character (or end of file).

C Stream Error Checking/Handling

- Some error functions (complete list in `stdio.h`):

```
int ferror(FILE *stream);
```

  - Checks if the error indicator associated with the specified stream is set

```
int clearerr(FILE *stream);
```

  - Resets error and EOF indicators for the specified stream

```
int perror(char *s);
```

  - Prints `message` followed by an error message related to `errno` to `stderr`

Global variable

Extra information

Terminal input/output

- C defines three file streams for terminal input/output
  - Defined in <stdio.h>
  - Opened at program start by default

  - **`stdin`**: standard input (console)
  - **`stdout`**: standard output (console, for normal output)
  - **`stderr`**: standard error (console, for error output)

- The following are equivalent:

```
printf("Hello World!\n");
```

```
fprintf(stdout, "Hello World!\n");
```

Demo: copy.c

Let's create a program that can make a copy of a file

- Things to do when dealing with C stream I/O:
  - Eventually we will hit the end of file, need to handle that
  - Must ask for an amount of bytes/elements to be read.
    Best practice is to request for a chunk of bytes/elements at a time (e.g. 1024 or so)

Other Functions

- Many other functions not covered in lecture (not enough time). Feel free to look up others and use them

- Some examples:
  - `int feof(FILE* f);`
    - check for end of file
  - `void rewind(FILE *f);`
    - start back at the beginning of file
  - `long ftell(FILE* f);`
    - gives the current position into the file
  - `int fseek(FILE* f, long offset, int whence);`
    - Reposition where we are in the file

Lecture Outline

- File I/O
- **Binary files & Endianness**
- Office Hours

Binary files & Serialization

- So far this lecture has focused we are working with files that hold text (characters)

- Binary files also exist where data isn't stored as 'characters'. (.obj files are an example)

- Some data/data-structures make more sense to be stored in binary through a process called **serialization**.

Serialization is the process of converting data into a sequence of bytes that can later be used to accurately reconstruct the original data.

Endianness

- In many architectures, there is one byte at each address location
  - ***For multi-byte data, how do we order it in memory?***
  - Data should be kept together, but what order should it be in?
  - Example, store the 4-byte (32-bit) int:
    0x A1 B2 C3 D4

Each byte has its own address

Most significant Byte

Least significant Byte

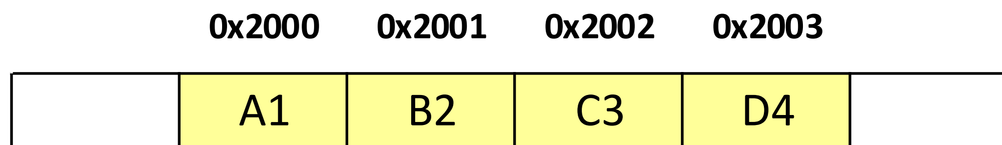
- The order of the bytes in memory is called endianness
  - Big endian vs little endian

Endianness

- Consider our example 0x A1 B2 C3 D4

Most significant Byte   Least significant Byte

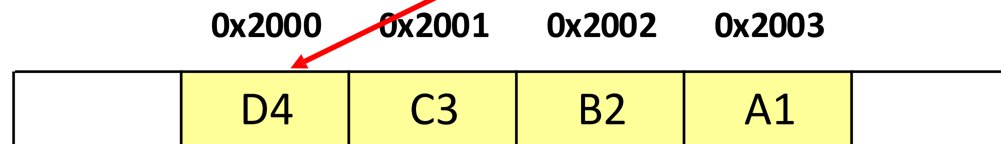
- Big endian
  - Least significant byte has highest address
  - Looks the most like what we would read
  - The standard for storing information on files/the network



Least significant Byte

- Little Endian
  - Least significant byte has lowest address
  - Your computer is probably LE

Note how the hex digits within a byte are still in the same order



pollev.com/cis2400

- If we have the following int which is four bytes. on a big-endian machine, how would this be stored in memory?

```
int num = 0xCADEDADA;
```

0x0 ... 0x3

A.

CA	DE	DA	DA

B.

DA	DA	DE	CA

C.

AC	ED	AD	AD

D.

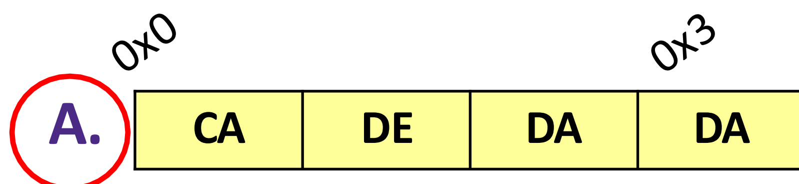
AD	AD	ED	AC

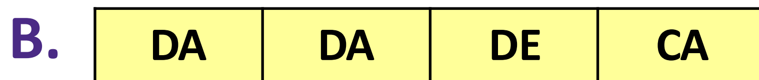
E. I'm not sure

pollev.com/cis2400

- If we have the following int which is four bytes. on a big-endian machine, how would this be stored in memory?

```
int num = 0xCADEDADA;
```





E. I’m not sure

Endianness: Why it matters

- Since machines may store things in different byte orderings, it causes problems when they share files or communicate over the network.
- A standard ordering is used for storing binary data, big endian (often called Network ordering).
- Need to make sure that we reassemble objects correctly based on byte ordering
  - Note: Re-assemblers are invaluable in network protocols

Endianness functions

- There are some functions out there that convert byte orderings
  - `htons()` -> Host to Network short (16 bits)
    - Converts from Host byte ordering to network byte ordering
  - `ntohs()` -> Network to Host short (16 bits)
    - Converts from network byte ordering to host byte ordering
- "Network byte order" is big endian. Your "host" machine Is little endian
- More info in <arpa/inet.h>
  - Variants also exist for 32 bit and 64 bit conversion

Endianness

Today, classifying machines as strictly Little Endian or Big Endian is not always straightforward.

Some machines can switch their Endianness dynamically, depending on the needs of the application or the operating environment.

ARM processors: Many ARM processors can operate in either Little Endian or Big Endian mode.

PowerPC processors: Commonly used in embedded systems, PowerPC processors can also switch between Endianness modes based on software instructions.

Demo: *Hex Dump (xxd)*

- **`write_hex vs write_string`**

- Let's see how integers are serialized (written) to a file compared to strings!

That’s it! Next half of lecture is OH

Feel free to stay if you need help with any of the material.

We hope you can take care of yourself as much as possible these next couple of weeks.