F25

Caching I/O gearup!

Overview

- What this project is
- How to plan your design
- How to get started writing code
- How to test and debug

Background: working with files

Syscall: ask operating system (OS) to do some operation (e.g., open a file)

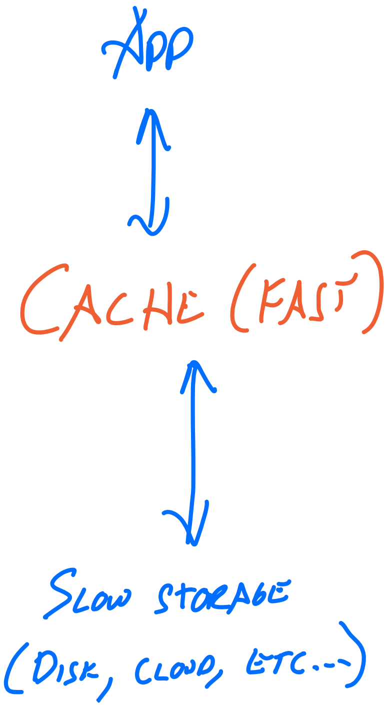
```
int fd = open("file.txt", ...);

while(1) {
    char buffer[BUFFER_SIZE
    memset(&buffer, 0, BUFFER_SIZE);


    int bytes_read = read(fd, buffer, BUFFER_SIZE);

     // . . .
}
```

System calls are expensive (read: slow!)

Goal: how can we build libraries to make programs that use files faster?

How? caching!

The general idea*:* *a cache is a* small amount of fast storage used to speed up slower storage

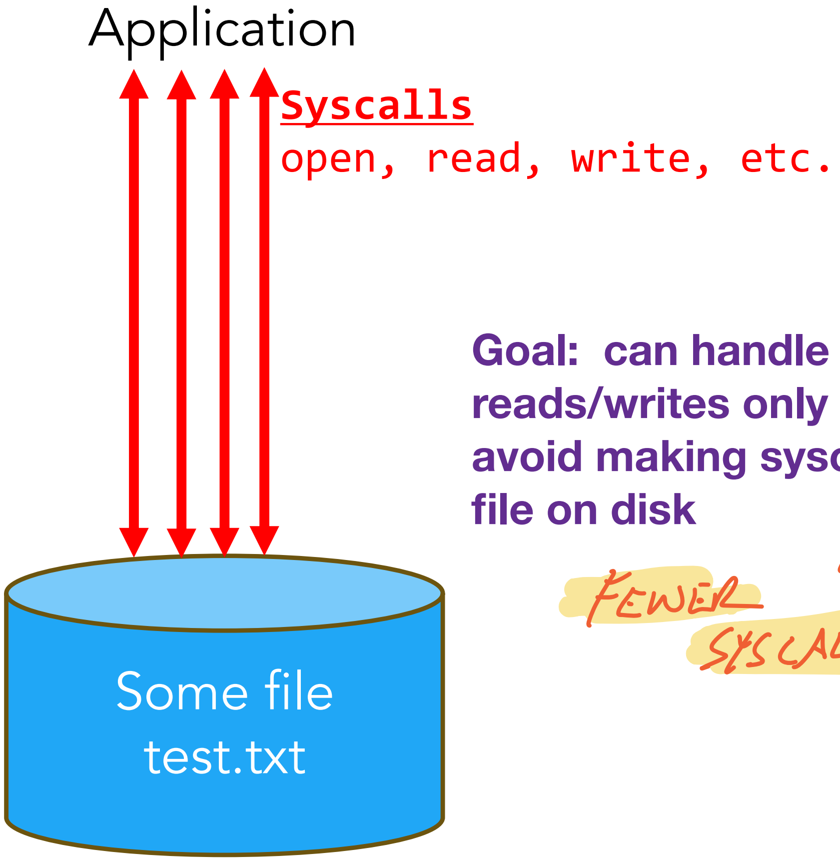


Caching appears in many forms

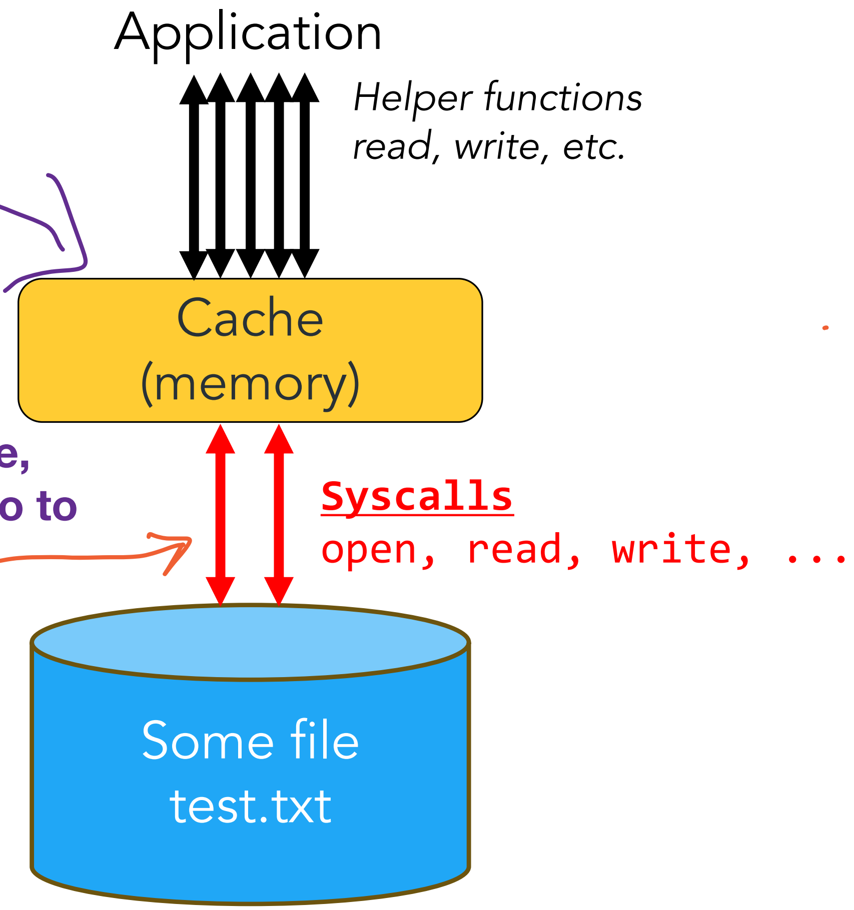
- CPU cache (hardware on CPU <=> DRAM
- Your web browser (files on your computer <=> internet)
- ...
- File I/O caching (this project) (memory <=> files)

=> Many ways to implement caching (at different layers of abstraction!)

No caching (naïve version)



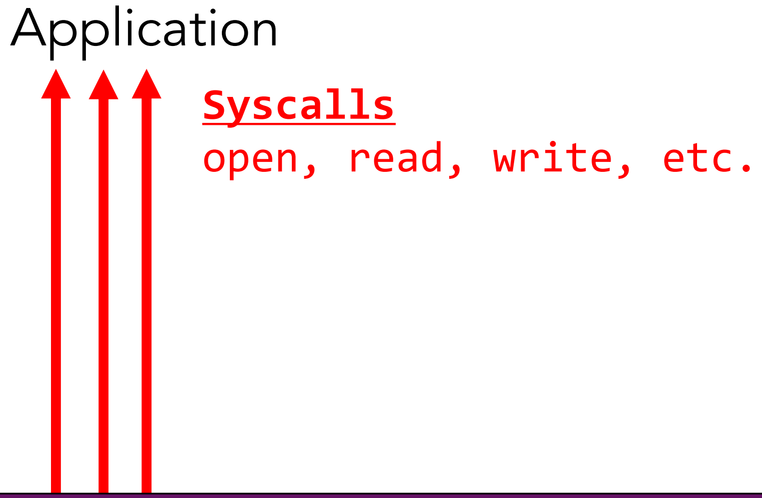
With caching



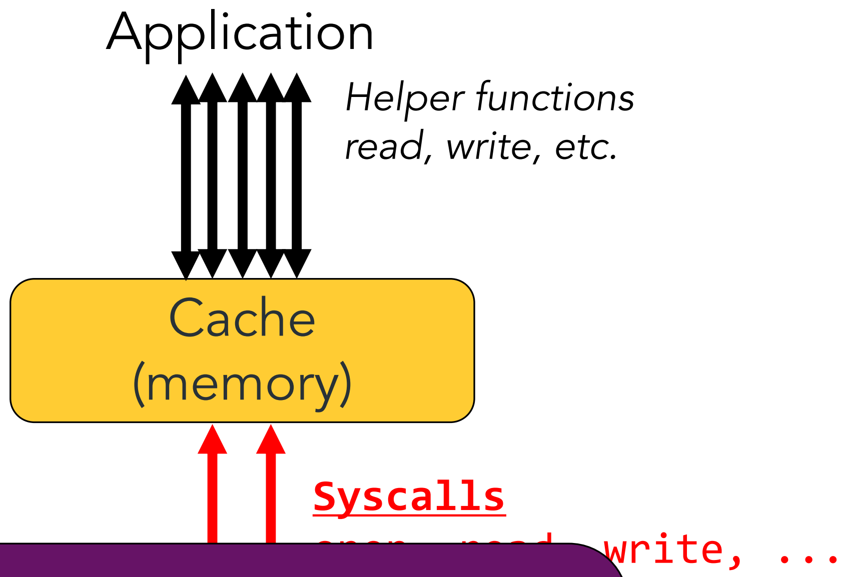
Goal: can handle some reads/writes only in cache, avoid making syscall to go to file on disk

FEWER SYSCALLS!

No caching (naïve version)



With caching



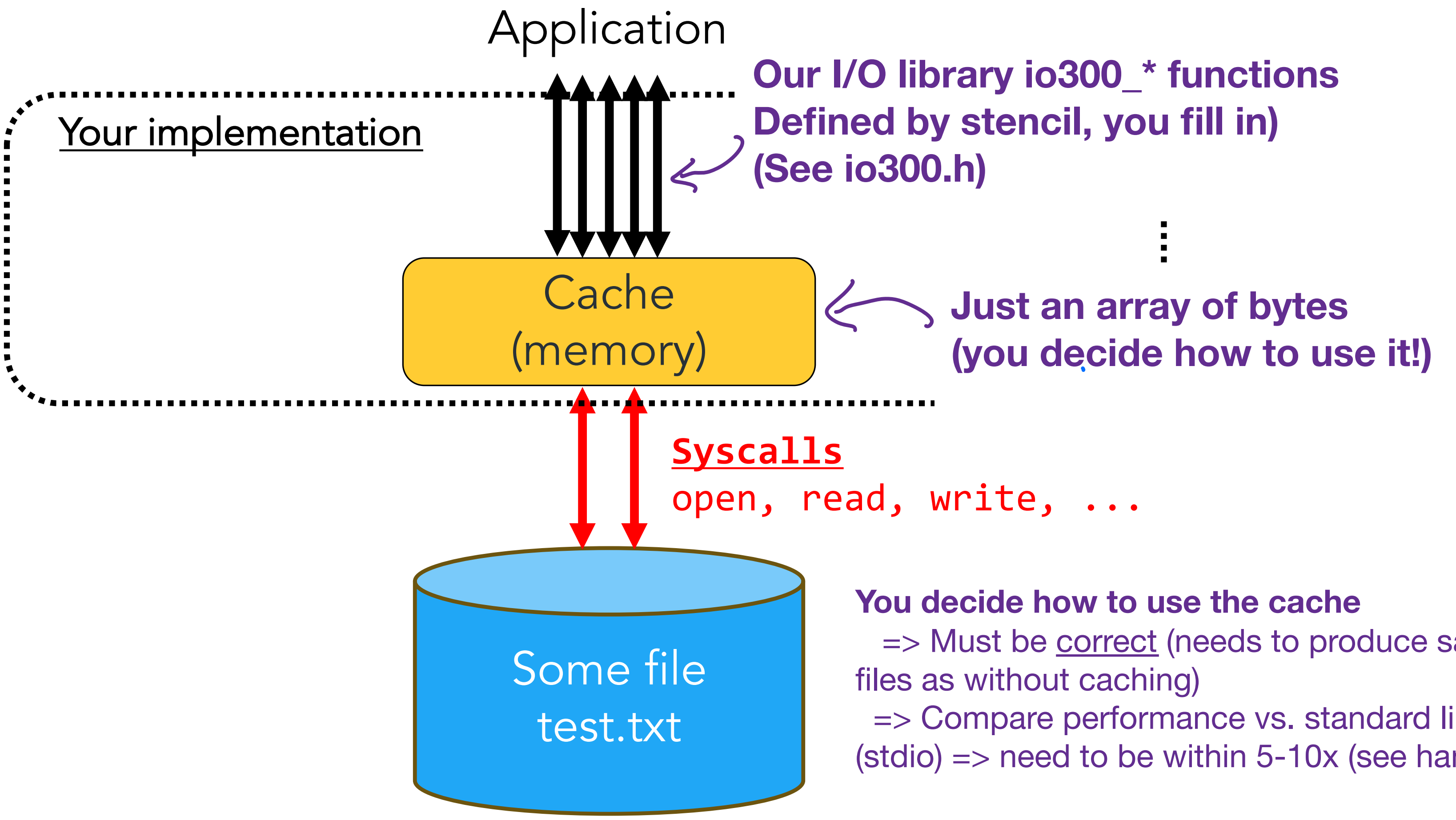
Idea: use cache to keep some file data in memory

=> Design: use some *heuristics* about how files are commonly accessed to make fewer syscalls

=> In terms of correctness, works the same way as no caching

=> However: fewer syscalls => better performance! 🚀

How you will do this



You decide how to use the cache

=> Must be correct (needs to produce same files as without caching)

=> Compare performance vs. standard library (stdio) => need to be within 5-10x (see handout)

Baselines

The stencil contains "implementations" of our I/O library (impl directory):

- **naive (impl/naive.c): always make the syscall**
  => no caching, super slow
- **stdio (impl/stdio.c): standard library version (fread, ...)**
  => our performance baseline
- **student (impl/student.c): Your version!**
  => Starts out just like naive version

Demo!

See recording!

The API (io300.h)

This is just a high-level overview of our top-level I/O functions. See io300.h and impl/student.c for the full function signatures, and more details!

```
f = open(path, mode)
close()

io300_read(f, buffer, count)
io300_write(f, buffer, count)

// Same as read/write, but only work with one byte at a time
char c = io300_readc(f)
io300_writec(f, ch)

io300_seek(f) // Move to specific position in file
```

The stencil also recommends some helpers (more on this later)!

How to think about the cache (generally)

Opening a file returns a struct io300_file: this contains the cache and any metadata about that file:

```
struct io300_file* f = io300_open(path, ...)
```

The cache, and any metadata about this file lives inside this struct. The stencil version io300_open calls open() and sets up some parameters for you, and you'll fill in the rest based on your design:

```
// From impl/student.h
struct io300_file {
    int fd;
    char* cache;

    // Your metadata goes here!
     . . .
}
```

File descriptor for this file (use for making syscalls)

✨**the cache**✨**: just an array of bytes of size CACHE_SIZE (a constant)**
- io300_open will set this up using malloc()
- Our tests will compile your code with different values for CACHE_SIZE (for debugging locally, it will use a value of 8, some tests will set it higher)

Fill in other parameters here based on your design!

About the cache (and common misconceptions)

- There is exactly one cache per open file. There aren't any "global" data structures that store information about multiple files
- The cache is just an array of bytes => you can load any bytes of the file into it, based on what you decide for your design

    => *Common misconception: the cache in this project is NOT like the CPU cache we talked about in lecture for alignment: there are no restrictions on loading in fixed "blocks" of data. (Though the "multislot cache" option for extra credit is a bit similar to this.)*

Example: naïve version

(No caching)

```
io300_readc() {
    read(fd, &c, 1); // System call
}
```

file.txt

"hi there!\n"

OS READ/WRITE HEAD STARTS HERE

```
f = io300_open("file.txt", …)

readc(f)
readc(f)
readc(f)
readc(f)
```

What will readc return each time, and how many syscalls will be made?

Example: naïve version

(No caching)

file.txt

```
"hi there!\n"
```

```
f = io300_open("file.txt", …)

readc(f) => read(fd, …, 1) => 'h'
readc(f) => read(fd, …, 1) => 'i'
readc(f) => read(fd, …, 1) => ' '
readc(f) => read(fd, …, 1) => 't'
```

```
io300_readc() {
    read(fd, &c, 1); // System call
}
```

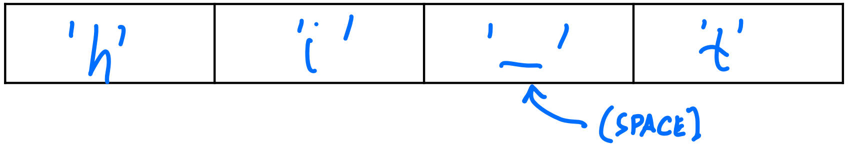
Naïve version: makes a syscall for every single operation!
=> How can we use the cache to do better??

Example 1: with caching

Assume cache = 4 bytes

file.txt

```
"hi there!\n"
```




```
f = io300_open("file.txt", …)
    "fetch" => read(fd, ...., 4);
readc(f)  => 'h'
readc(f)  => 'i'
readc(f)  => ' '
readc(f)  => 't'
```

One strategy: What if we "fetch" the whole cache into the file when we open it

=> **This is called *prefetching***: load data ahead of time because we think we'll use it later!

This is faster: since the data is in the cache, readc can return it without making a syscall!!
But what metadata do you need in order for readc to return the right thing each time?
=> Need some metadata to keep track of the next byte to be read/written

Example 1.5: with caching

Assume cache = 4 bytes

file.txt

"hi there!\n"

h	i	' '	t

```
f = io300_open("file.txt", …)
      // fetch data into cache! => read(fd, …, 4)

readc(f) => 'h'
readc(f) => 'i'
readc(f) => ' '
readc(f) => 't'
readc(f)
```

What if we do another readc after this?
What should happen now????

Example 1.5: with caching

Assume cache = 4 bytes

file.txt

"hi there!\n"

h 'h'	i 'e'	' ' 'r'	t 'e'

```
f = io300_open("file.txt", …)
      // fetch data into cache! => read(fd, …, 4)

readc(f) => 'h'
readc(f) => 'i'
readc(f) => ' '
readc(f) => 't'
readc(f)
   // fetch data into cache! => read(fd, …, 4) => 'h'
```

One strategy: probably going to read more of the file after this... how about fetching the next 4B into the cache?

=> More prefetching!

What happens to your metadata after fetching again?

=> Will need to keep it updated!

Ideas/Hints (SO FAR)

For metadata:

Byte that you will read/write next

... (more hints later) ...

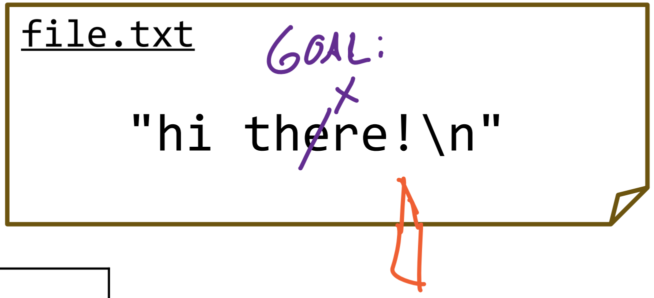
Helpers:

"Fetch" : load some amount of data into the cache (based on your metadata, etc.)

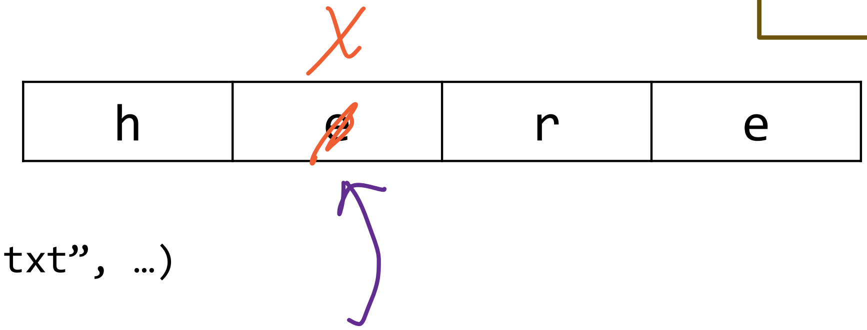
=> calls `read()`

... (more hints later) ...

Example 2: writing



Assume cache = 4 bytes




```
f2 = io300_open("file.txt", …)
readc(f) => 'h'
readc(f) => 'i'
readc(f) => ' '
readc(f) => 't'
readc(f) => 'h'
writec(f, 'x')
```

One strategy: when writing, make changes to cache, then "flush" changes to file when necessary

What should happen here? writec should set the next character in the file to 'x', which is the first 'e' (based on the sequence of readc calls already made)
(Note that this is different from the current position of the OS read/write head, which is at a different byte ('!') because of how we did the prefetching.)

Example 2: writing

Assume cache = 4 bytes

file.txt

"hi thxxx!\n"

h	x	x	x

Now what happens as we write more to the file?

```
f2 = io300_open("file.txt", …)
readc(f) => 'h'
readc(f) => 'i'
readc(f) => ' '
readc(f) => 't'
readc(f) => 'h'
writec(f, 'x')
writec(f, 'x')
writec(f, 'x')
writec(f, 'x')
 => Need to "flush" changes from cache to disk
     write(fd, ..., 4);
```

UPDATE CACHE, BUT DON'T NEED TO CHANGE FILE YET!

Cache was modified, so need to "flush" changes in memory before fetching again
=> Need to keep track of if cache was modified! (often called "dirty")

Ideas/Hints

For metadata, need some way to keep track of…

- Next byte to read/write
- Whether not cache was modified
- …

You will need more metadata than this!
=> Consider as you work on the rest of the design phase!

Helpers:

- fetch: Fill the cache with next N bytes from file => calls read()
- flush: Write cache to disk => calls write()

Planning your design

Example from handout

```
char buffer[5];
io300_file* testFile = io300_open("testfiles/tiny.txt", "tiny!");
ssize_t r = io300_read(testFile, buffer, 5);
ssize_t w = io300_write(testFile, "aaa", 3);
r = io300_read(testFile, buffer, 2);
ssize_t s = io300_seek(testFile, 12);
w = io300_write(testFile, "aaa", 3);
r = io300_readc(testFile);
io300_close(testFile);
```

Try this out similarly to what you've seen here

=> Think about what *should* happen in file

=> What metadata you will need

=> Need to make sure file is correct!

=> We provide a handy worksheet

Hints here are only a starting point!
=> See handout for more guidance!

Getting started

Do design part, bring to section
You can start writing code as soon as you feel comfortable

Phase 1: Recommend starting with readc/writec
=> Then, consider add seek

Phase 2: read/write/seek
=> Same as readc/writec, but working with multiple bytes
=> Don't implement by calling readc/writec (won't pass the performance tests)
=> You will want to use memcpy here (see handout for details)

Starting your implementation

- You can start writing code as soon as you feel comfortable
- Phase 1: readc/writec
  - Leave other functions intact until you're ready
  - Once working, consider interactions with seek
- Phase 2: read/write (+seek)
  - Same idea as readc/writec, but read/write multiple bytes
  - Use memcpy to copy data to/from cache
  - DO NOT use readc/writec as helpers (won't give you credit!)

Super helpful tool: strace

strace is a tool to show what system calls your program makes

=> Use this instead of writing complicated print statements!

See the recording for a demo, as well as guidance in the handout!

strace should become your friend!!!

Getting started with testing

- Correctness tests (make check)
  - Regression tests (make check-regression)

    => very small examples (some of which you'll write!)
  - Fuzz tests (make check-fuzz) => run lots of random operations on files
  - End-to-end tests (make check-e2e) => tests more components

- Performance tests (make perf)
  - Compares your implementation vs. stdio

Demo!

See recording for examples of how to run the tests!
(More guidance also in the handout)

Regression tests (make check-regression)

Idea: super small tests, easy to check by hand!

=> Should be good for starting out!

=> We provide a few, but we ask you to write some on your own, based on what is meaningful for your design

See the handout for a step-by-step guide for getting started!

Regression tests (make check-regression)

rtest001.c:

```
int main() {
    assert(CACHE_SIZE == 8);
    struct io300_file* f = create_file_from_string(TEST_FILE, "hello world");


    // Do some readc operations
    assert(io300_readc(f) == 'h');
    assert(io300_readc(f) == 'e');
    assert(io300_readc(f) == 'l');

    // Close the file
    io300_close(f);
}
```

=> Compare with examples you make by hand!
=> Ideally: minimal examples to test as few features as possible
(e.g., read enough bytes to fetch, read to EOF, etc.)
⇒ Once you write a test, you can keep using it to check for *regressions*
(ie, when new things break old stuff)

Fuzz* tests (make check-fuzz)

*(*Fun fact: This is a software engineering term: in general, fuzz tests try lots of random things to look for edge cases that break stuff!)*

Test program: io300_test

Each test shows the command to run it manually.
run io300_test --help for more options

```
RANDOM SEED FOR THIS RUN: 1398752801
======= (1) BASIC FUNCTIONALITY TESTS =======
1. readc
-> ./io300_test readc --seed 1398752801 -n 8192 --file-size 4096 --max-size 8192
PASSED!
2. writec
-> ./io300_test writec --seed 1398752801 -n 8192 --file-size 4096 --max-size 8192
PASSED!
3. readc/writec
-> ./io300_test readc writec --seed 1398752801 -n 8192 --file-size 4096 --max-size 8192
```

"Call readc and writec 8192 times on a random file" of size 4096…
=> Performs random operations, checks for correctness a
=> When encountering a problem, run on your own!

Best way to debug: run the test manually
=> Use the same seed value

Getting started: run io300_test yourself

To start testing on your own:

```
-> ./io300_test readc --seed 1234 -n 100 -i test_files/tiny.txt

...
```

A good way to get started is to run io300_test on a sample file (see the test_files directory)

This example asks io300_test to call readc 100 times on test_files/tiny.txt, and will verify that your readc returns the correct results each time. This is a good way to make sure one function at a time is correct!

As you get errors, compare with what you expect to see in the file to help debug.

When debugging tests write/writec, add the option --no-cleanup to make io300_test save the output file for you to inspect (see handout for more info when you get to this)

End-to-end tests

Small programs that use a combination of io300_* functions

Find in `test_programs` directory!

byte_cat: Copy bytes one at a time from input to output file (readc/writec)

block_cat: For some block size N, use read/write to read in an input file and write to output file

reverse_block/byte_cat: Do the same thing, but iterate through the file backwards (why might this be slower

... and more! Take a look at the programs to see what they do! (Helpful notes in comments)

=> Recommended way to debug: run gdb on the test with a sample file (see handout for details)

byte_cat.c

```c
for (int i = 0; i < filesize; i++) {
        int ch = io300_readc(in);
        if (ch == -1) {
             // . . .
            break;
        }
        if (io300_writec(out, ch) == -1) {
            // . . .
            break;
        }
    }
```

READ FROM INPUT FILE

WRITE TO OUTPUT FILE

Performance tests (make perf)

⚠️ WARNING: Performance can vary significantly between systems. Be sure to check your results on the grading server, which could be very different from your system! See the handout for details.

- Compares your program against stdio
- Your implementation needs to be within…
  - 10x for byte_cat tests
  - 5x for block_cat tests

- Warning: print statements will slow things down
  - See handout for how to deal with this
- Need to pass relevant correctness test first!

=> Focus on building a correct implementation first!
(We won't give credit on performance unless the relevant parts of your implementation are correct)

Performance tests (make perf)

⚠️ WARNING: Performance can vary significantly between systems. Be sure to check your results on the grading server, which could be very different from your system! See the handout for details.

Example output:

```
performance result: byte_cat: stdio=0.05s, student=0.11s, ratio=2.20
performance result: reverse_byte_cat: stdio=0.16s, student=0.18s, ratio=1.12
performance result: block_cat: stdio=0.01s, student=0.05s, ratio=5.00
performance result: reverse_block_cat: stdio=0.02s, student=0.10s, ratio=5.00
performance result: random_block_cat: stdio=0.02s, student=0.09s, ratio=4.50
performance result: stride_cat: stdio=0.27s, student=1.48s, ratio=5.48
======= PERFORMANCE RESULTS =======
byte_cat: 2.2x stdio's runtime
reverse_byte_cat: 1.12x stdio's runtime
block_cat: 5.0x stdio's runtime
reverse_block_cat: 5.0x stdio's runtime
random_block_cat: 4.5x stdio's runtime
stride_cat: 5.48x stdio's runtime
```

A note on testing

The way tests work is pretty different than other projects…

- Many tests use the same functions
- <u>One bug can cause failures on a lot of tests => this doesn't mean you're doing a bad job!</u>

When you get stuck: think back to your design
⇒ What should happen
⇒ What is happening now? (gdb, strace, print statements, etc.)
  ⇒ How do these differ?

You got this!!!!!

If you get stuck, we're always here to help! :)

Timeline

- Design plan: bring to section this week (Thurs-Sat, Oct 16-18)
  - Get feedback from your peers!
- Final deadline: Friday, October 24

Good luck!! You got this!!!