CS202 Review Session 3 for Lab2 (`ls`)

Yuxia Zhan - Fall 2025

Clarification

The lab's webpage is self-contained. These review sessions are just supplements. Which means, You can absolutely finish the labs without attending these review sessions, without watching the review sessions' Zoom recordings, or without reading the review sessions' notes. Actually, if you prefer more *original* experience - kinda like solving a puzzle without looking at the picture on the box, you might want to stick only to the lab webpage.

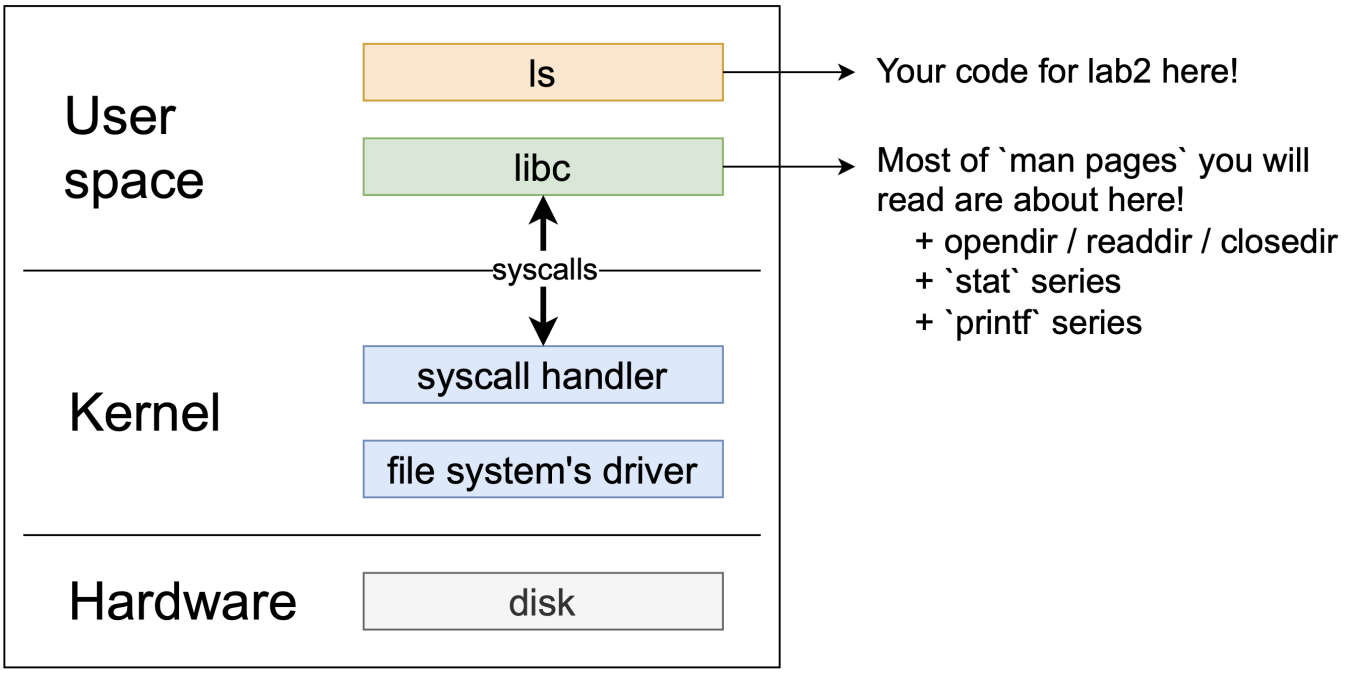
But, if you have already read through the lab's webpage, and still feel confused or unsure about how to begin, then these review sessions will help you bootstrap your lab.

Overview

- Motivation - Learn how to develop a command-line tool
  - Software stack
  - Workflow
- Step 1 - Figure out the requirements
  - Linux's abstraction for files and directories
- Step 2 - Develop the functionality using syscalls
  - Parse the flags and arguments
  - Implement basic `ls` functionality
  - Support `-l` option
  - Support more options
  - Pretty-print and error handling
- Step 3 - Test and debug

Motivation - Learn how to develop a command-line tool

The reason why Lab 2 will be helpful is that you will learn how to write a command-line tool for Linux, or put it more generally, how to write user-space software for Linux.



As introduced in class, the OS kernel functions as the middle layer between the hardware and the users' softwares. User-space software can interact with kernel using syscalls. These syscalls are exposed to C programs through the standard C library (so called **libc**).

From a high-level perspective, developing a command-line tool involves the following workflow:

1. Figuring out what features you want (or are required) to develop. In this lab, the requirements are provided on the lab's webpage. You need to read it in its entirety and understand them carefully.
2. Developing the tool using syscalls. In this lab, only a small amount (10 ~ 20) of them will be used. In real projects, you might use way more than that.
3. Testing your tool. In this lab, we provide testing scripts, some of them are in the lab's repository, while others will be used to grade your code. In reality, you might need to design your own test strategy.

In this lab, you will learn all of the steps by implementing a simple but extremely useful command-line tool, called the `ls`.

Step 1 - Figure out the requirements

Roughly speaking, Lab2 requires you to implement your own version of `ls`, with almost the same (though NOT identical!) functionality as the system-provided ls.

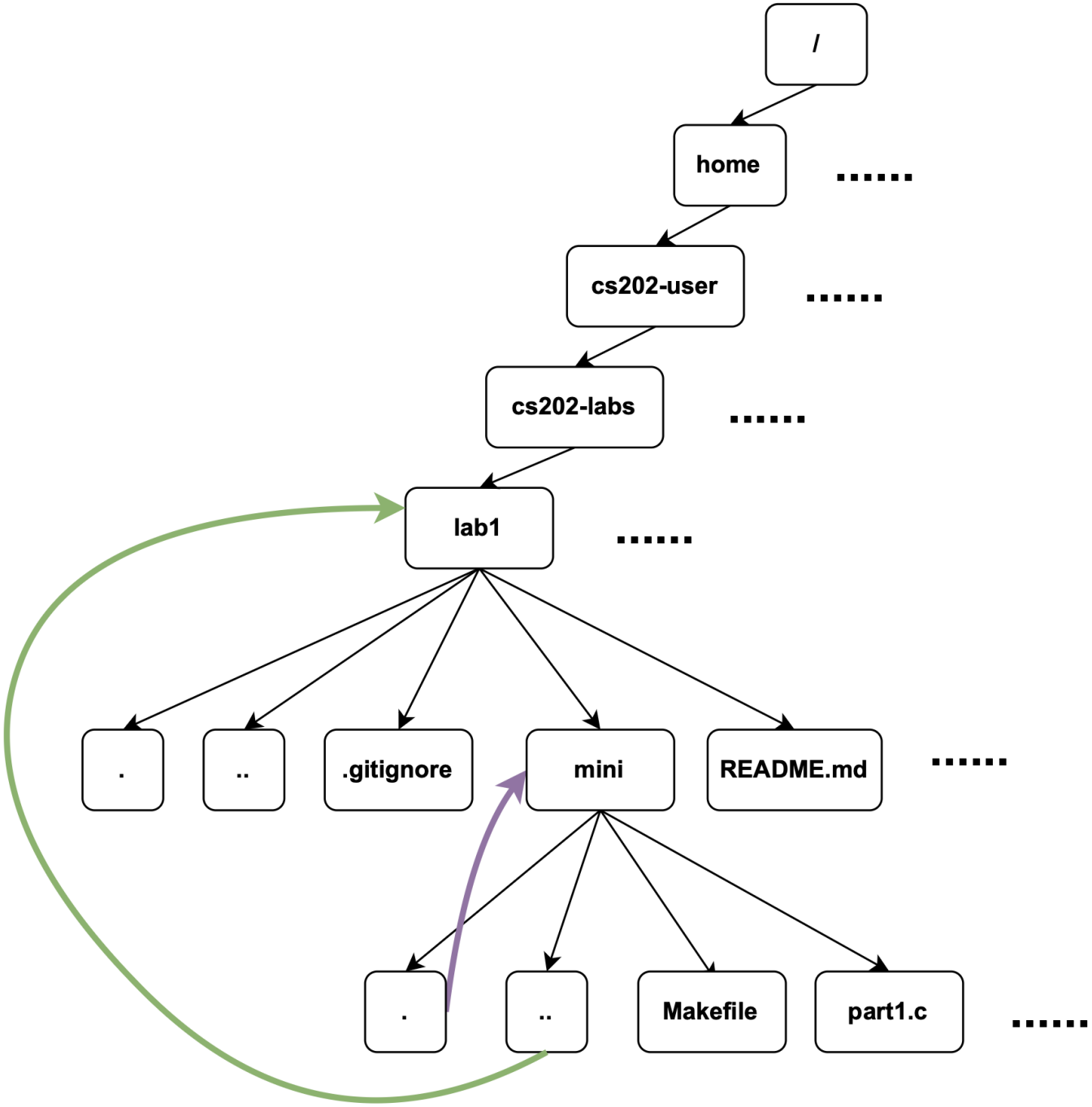
What `ls` does is **ls**ting files and subdirectories under a directory. If requested, it can also print out detailed information about each file or subdirectory. To get familiar with what `ls` should do, you can open a terminal and run `ls` (the system-provided `ls`) using different options.

- Usually, if it's your first time to use a command-line tool, you can try to type in `man <program>` to see its manual, and you will find their usage there.
- `ls -1` print each entry in a line.
- `ls -1a` hidden files (i.e., those nodes with names starting with `.`). They are `hidden` mainly for neatness and convenience, not for security or other others. It's more of a historical convention than

anything else.
    - `.` stands for current directory
    - `..` stands for parent directory
- `ls -1l` the metadata of each entry (entry could be a file or a directory).
- `ls -1R` recursively list all files (including those inside subdirectories).

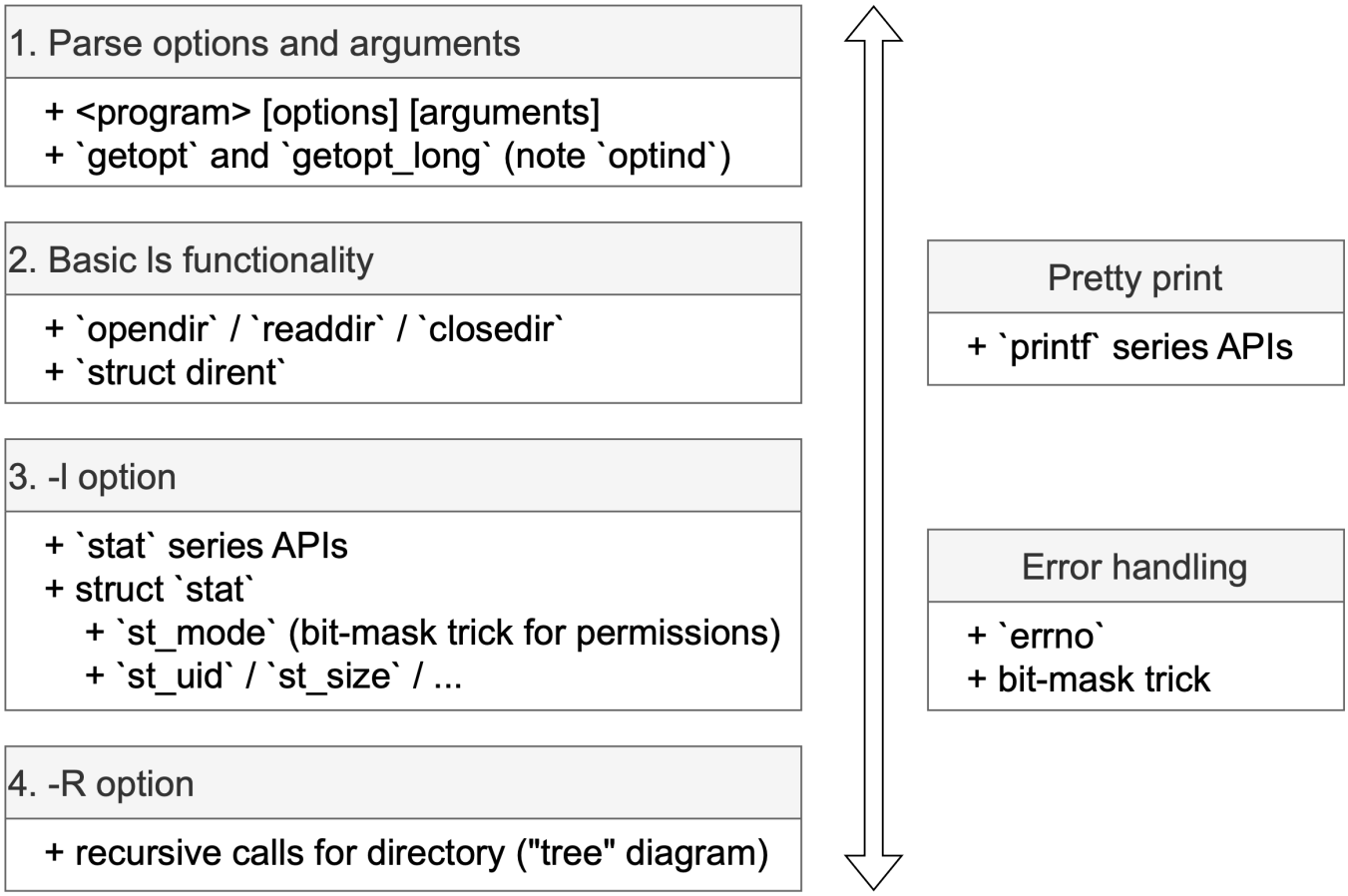
So as you can see, the Linux's filesystem can be thought as a tree structure, the tree's root is the `/` (i.e., the root directory). And each entry under the directory is its child node. And for those nodes that itself is a directory, it can be expanded to a tree as well.

Each node has two features: **metadata** and **data**, where metadata includes file type, access permissions, owner, size, modification time, etc.



Step 2 - Develop the functionality

A good thing of carefully reading the lab page is that, you can not only know the requirements, but also get some hints at how to fulfill these requirements. For example, I found the suggested sequence is particularly helpful for clarifying questions like "what am I doing, what resources should I look at, and what should I do next".



You can add more items as you need to the mental map. This kind of mental map would become even more important in later labs, since the code structure of those labs are more complicated than lab2, where you basically only have to write code in the `main.c` file.

Parse options and arguments

Your tool must parse the options (flags) and arguments that the user provides through the command-line. Essentially, this means transforming a command string into a few `bool` flags or `char*` variables in your C program.

Doing this from scratch is painful because options can appear separately (like `-a -l -R`) or combined (like `-al -R`). Fortunately, `libc` provides two helpful APIs to make your life easier: `getopt` and `getopt_long`, By the way, these APIs are declared in `<unistd.h>` and different from `read` or `write`, they don't invoke any syscalls.

Command-line usually has the following format, all of them are `char*` objects of `argv` (the input of the `main` function).

```
<program> [options] [arguments]
```

What `getopt` and `getopt_long` do is to help you parse `[options]` part. `getopt` can only parse short options like `-l, -a, -R`; and `getopt_long` can parse both short and long options like `--help` and `--hack`.

- `argc`: number of input arguments (from main)
- `argv`: vector of input arguments (from main)
- `optstring`: the options (flags) that we want to identify
- `optind`: a macro tracks the "next argument (in argv)" to process
- During the while loop, `getop()` will scan the `argv` (starting from `optind` position)
    - Here we meet the C idiom for "assign then compare" again.
    - if `getopt()` finds a matching option (i.e., a character you pre-defined it in `optstring`, and it appears in the `argv`), it will
        - return that option (a `char`)
        - update the `optind` (so that next calling `getopt()` can resume the scan)
    - otherwise, it will return -1.
- `getopt_long` is quite similar, you need to construct the `long_options` for long options that you want to identify.

Implement basic `ls` functionality

Once parsing is done, implement the basic functionality: listing files in a directory. To do so, you will need APIs like `opendir`, `readdir`, and `closedir`. These are wrappers for syscalls that can open a directory, read its entries (its child nodes), and close this directory once finishing the reading.

- `opendir()` to get the `DIR* dirp`;
- Iterate the directory's entries (`struct dirent* dp`) using `readdir()`.
    - struct dirent contains the filename and other fields.
    - You can use `ctrl + ]` in vim or `F12` in VS code to jump to its definition to see more details.
- Don't forget to `closedir()` to prevent memory leakage.

Support `-l` option

For the `-l` flag, you will not only figure out how to iterate through the directory's entries, but also follow the directory entry to locate the corresponding child node and read the metadata of this file. Most of the metadata of a file can be fetched using the `stat` series of APIs (see `man 3 stat`).

You can find helper functions (e.g., `uname_for_uid`, `group_for_gid`, `date_string`) for formatting most of these metadata in the `main.c` file, but you do have to format the access permission on your own. These permissions are stored in `st_mode` (which is essentially `uint16` type) of each file's `stat` structure. And you are expected to convert this `st_mode` into human-readable strings (i.e., `r` for readable, `w` for writable, and `x` for executable, see `man 7 inode`).

You can find the follow line, for example:

```
S_IRUSR     00400   owner has read permission
```

You might notice the `00400`, it's an octal number in C (see C23: 6.4.4.2 Integer constants), which will start with `0`.

To understand what this means, you need to know the **bit-mask**, a broadly used trick in programming.

Suppose each bit of the original value (e.g., `st_mode`) represents a flag (e.g., "owner has or has no read permission"). Using bit-mask is just a fancy way to select certain bit from a value. It's much like extracting a char from a string using `string[idx]`, but in a bit number world.

Specifically, a bit-mask is needed for each position and its value is with only one bit (in the same position) set to 1 while all other bits are 0. To extract the bit in that position, you will need to perform a bitwise operation (you should think about which operation is it) with the value and the bit-mask.

Let's take a look at the following example. Each of these a b c d is a single bit, and we want to extract the bit b.

```
                    value = a b c d
                 bit-mask = 0 1 0 0
----------------------------------------
  using some bit-op, result = 0 b 0 0
```

This is the simple example when we only want to extract one bit from a 4-bit values, bit-mask can actually extract multiple bits at a time. You can follow the similar idea to think about how to do that.

Support more options

- For `-R` option, recall the tree structure of Linux's file system shown above. Recursive traversal is quite suitable for handling such structure. You need to be careful with the arguments details while recursing. For example, you will have to tell if a `dirent` stands for a normal file or a directory.
- For `-n` option, you need to think of how to suppress the print while tracking the counting.

Pretty-print and error handling

Throughout all former steps, you will learn to construct a long string (assuming you already gathered the necessary information that need to be put in this string) in a formatted way. There are a series of `printf` APIs for this purpose, e.g., `printf`, `snprintf`, etc.

- format
- specifier like `%d` and `%s`

Regarding error handling. A good error handling system can let the program fail gracefully even when users don't know how to use it properly and clearly tell users what went wrong. Thanks to Linux's error handling system, we can directly read `errno` to know what's wrong under-the-hood, and map that into our own error

indicator as needed. Here you will use the bit-mask trick again: each bit represents the presence or absence of a specific error.

Step 3 - Test and debug

Testing and debugging are interwoven with development. After you implement some functionality, you will need to test if your implementation works.

Just as mentioned earlier, we provides a testing framework for you.

You can find the test cases in `test.bats`. Every time you run `make test`, if everything goes well, this script will be executed, and it will:

- run `./mktest.sh` to create a directory in `/tmp` (a temporary directory) with files and sub-directories. You can think of it as a "testbed".
- Demo with building this testbed.

```
./mktest.sh /tmp/testbed
ls -1Rl /tmp/testbed
```

- run each individual test case (`@test`) on the testbed, and compare the output of your `ls` with the expected outputs.

By the way, shell scripts can help us automate commands. Technically, you can type in each command line by line, and they would do the exact same thing. But it's just move convenient and much faster to put them all into a shell script and you can run everything with a single command - just run the script.

For most of the test cases, the expected outputs is the system-provided `ls` output (not as is but after some processing). For example, `system-provided ls -l` will output a line `total xxx`, while for lab2, your ls doesn't have to do that. You can find the shell command that does these processing in `@test`.

```
ls -1l --color=never --file-type . |
grep -v "^total"|                           # ignore the line that starts
with 'total'
sed 's/@$//g' |                             # remove the trailing `@` for
xattrs (not showing up in testbed though)
awk '{print $1,$2,$3,$4,$5,$6,$7,$8,$9}' |  # fetch each line's first 9
blocks (split by ' ')
sort                                        # sort the results
```

And "comparing" here means "compare, in a character-by-character manner, your `ls`'s output stream with system-provided `ls`'s output stream". This test framework uses a tool named `diff` to do the char-by-char comparing. You will need to understand the `diff` output format to identify where your code needs changes.

`diff A.txt B.txt`: to change from A to B, A needs to:

- [line]'a'[range] : (after [line] of A), add those lines from B's [range].
- [range]'d'[line] : deleted A's [range] lines ([line] here is to indicate why we need to delete A's those lines, and the reason is that B doesn't have those lines after B's [line])
- [rangeA]'c'[rangeB] : change A's [rangeA] lines to those from B's [rangeB].

For debugging with more granularity (line-by-line), you can use `gdb`. The *debugging* section of this lab webpage includes helpful hints to make your debugging experience more smooth.

Appendix - modified example code of `getopt`

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

// example code modified based on `man 3 getopt`

int
main(int argc, char *argv[])
{
    int n_flag, t_flag;
    char* t_argument;

    int identified_opt;

    t_argument = 0;
    n_flag = 0;
    t_flag = 0;

    while ((identified_opt = getopt(argc, argv, "nt:")) != -1) {
        switch (identified_opt) {
            case 'n':
                n_flag = 1;
                break;
            case 't':
                t_argument = optarg;
                t_flag = 1;
                break;
            default: /* '?' for unrecognized option */
                printf("Error. Usage: %s [-t <t_argument>] [-n] name\n",
argv[0]);
                exit(EXIT_FAILURE);
        }
    }

    printf("n_flag=%d; t_flag=%d; t_argument=%s; current optind=%d\n",
            n_flag, t_flag, t_argument, optind);

    if (optind >= argc) {
        printf("Error. Expected `name` argument after options\n");
        exit(EXIT_FAILURE);
    }
```

```
    printf("Get `name` argument = %s\n", argv[optind]);

    /* Other code omitted */

    exit(EXIT_SUCCESS);
}
```

Appendix - example code of `printf`

```
#include <stdio.h>

#define MAX_LEN 256

int
main(int argc, char* argv[]) {
    char name[] = "AAA";

    printf("printf \t\t\t: Hello from %s.\n", name);

    char greetings[MAX_LEN];
    snprintf(greetings, MAX_LEN, "Hello from %s.\n", name);

    printf("snprintf + printf \t: %s", greetings);

    fprintf(stdout, "snprintf + fprintf \t: %s", greetings);

    return 0;
}
```

Appendix - example of `diff`

```
# A.txt
Hello
This is AAA.
Hi
How are you?
aaaa

# B.txt
Hi
Hello world.
This is BBB.
How are you?
bbb
cccc
```

```
# compare A.txt and B.txt using `diff`
$ diff A.txt B.txt
1,2d0
< Hello
< This is AAA.
3a2,3
> Hello world.
> This is BBB.
5c5,6
< aaaa
---
> bbb
> cccc
```